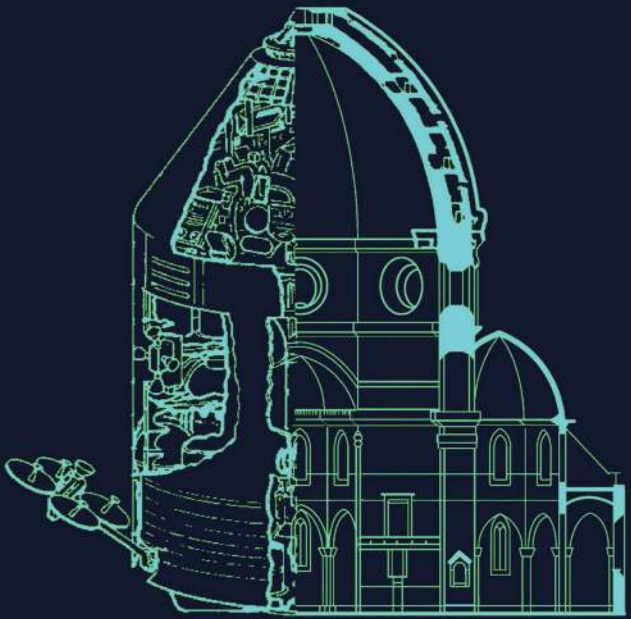
FOURTH EDITION

THE ART OF SYSTEMS ARCHITECTING



MARK W. MAIER
AND EBERHARDT RECHTIN

CRC Press
Taylor & Francis Group

The Art of Systems Architecting

The Art of Systems Architecting, Fourth Edition, provides structured heuristics to improve the least structured, most art-like elements of systems design. It offers unique techniques to bridge the difference between scientific engineering and qualitative design along with comprehensive methods for combining architectural design with digital engineering. This book illustrates how to go from model-based systems architecture to model-based systems engineering and includes case studies of good and bad architectural decision-making in major systems.

Changes to this edition include materials on architecture processes, architecture description frameworks, and integration with model-based systems engineering (MBSE) and digital engineering. The publication of the ANSI/IEEE 1471 and ISO/IEC 42010 standards on architecture description has provided common vocabulary and organizing methods for documenting architectures. This edition provides a practical application of these standards in architecting and integrating their concepts with a simple process framework. The rise of MBSE and digital engineering tools is in the process of revolutionizing the development of complex systems. The emphasis has been on detailed design descriptions and powerful analysis methods (for example, digital twins). Architects can make effective use of these methods and tools as well, and this new edition provides an integrated set of heuristics and modeling methods to do so. There are many other improvements and additions included to bring this textbook up to date.

This book can be used as a reference book for engineers and managers involved in creating new systems, people responsible for developing mandated architecture descriptions, software architects, system architects, and systems engineers, or as a textbook in graduate engineering courses.

Exercises are interspersed throughout the text, with some designed for self-testing and understanding and others intended to provide opportunities for long-term study and further exploration of the subject.

The Art of Systems Architecting

Fourth Edition

Mark W. Maier and Eberhardt Rechtin

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

Fourth edition published 2025
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton FL 33431

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

First edition published by CRC Press 1996
Second edition published by CRC Press 2000
Third edition published by CRC Press 2009

Library of Congress Cataloging-in-Publication Data
Names: Rechtin, Eberhardt, author. | Maier, Mark W. (Mark Warner), 1961-author.
Title: The art of systems architecting / Mark W. Maier and Eberhardt Rechtin.
Description: Fourth edition. | Boca Raton, FL : CRC Press, 2025. |
Earliest edition by Eberhardt Rechtin, Mark W. Maier. |
Includes bibliographical references and index.
Identifiers: LCCN 2024057175 (print) | LCCN 2024057176 (ebook) |
ISBN 9781032774381 (hbk) | ISBN 9781032775258 (pbk) | ISBN 9781003483595 (ebk)
Subjects: LCSH: Systems engineering. | System design.
Classification: LCC TA168 .R368 2025 (print) | LCC TA168 (ebook) |
DDC 620.001/171—dc23/eng/20250304
LC record available at https://lccn.loc.gov/2024057175
LC ebook record available at https://lccn.loc.gov/2024057176

ISBN: 978-1-032-77438-1 (hbk)
ISBN: 978-1-032-77525-8 (pbk)
ISBN: 978-1-003-48359-5 (ebk)

DOI: 10.1201/9781003483595

Typeset in Times
by codeMantra

To Eberhardt Rechtin,
who opened new vistas to
so many of us, and inspired
us to go out and find more.

Mark W. Maier

Contents

Preface

THE CONTINUING DEVELOPMENT OF SYSTEMS ARCHITECTING

> Architecting, the planning and building of structures, is as old as human societies—and as modern as the exploration of the solar system.

So began this book's original 1991 predecessor (Rechtin 1991). The earlier work was based on the premise that architectural methods, similar to those in the centuries-long tradition in civil works, were being used, albeit unknowingly, to create and build complex aerospace, electronic, software, command, control, and manufacturing systems. If so, then other powerful ideas from that traditions—such as qualitative reasoning and the relationships between client, architect, and builder—could be found and exploited in today's engineering fields. Over the ensuing 30 plus years and three previous editions of this book, the original premise has found wide support. The use of architectural concepts has become common in systems engineering discussions. Heuristics and patterns, concepts drawn from architectural practice, are widely recognized as tools worth developing. Continuing experience has confirmed a fundamental insight: the decisions made very nearly in a system's conceptual lifetime have inordinate impact on value, cost, and risk (NRC 2008).

The years since the 1991 book have also brought recognition of other important factors in architectural thinking. These have been incorporated in the editions of this book, a process that continues to this fourth edition. Among the most important are:

- Focusing on early phase decisions. This can be expressed in the concept that architecture is a set of decisions, the set of decisions largely determining value, cost, and risk of a system.
- The importance of the interplay between technical decisions on the system's structure and the structure of the program that builds the system. The choice to build incrementally, or in the proto-flight pattern, or any other of the basic templates carries important technical consequences.
- Architecture is the technical embodiment of strategy. We can't know if the architecture is well chosen unless we know if the strategy is well chosen. This is equally true in business-driven systems as in politically driven systems.
- The language we use to express architecture (normally a modeling language) has a deep influence on what kinds of architectures we can express. This becomes especially important in software-dominated systems, and more and more systems are software dominated.

ARCHITECTING IN THE SYSTEMS WORLD

A strong motivation for expanding the architecting process into new fields has been the retrospective observation that the success or failure of today's widely publicized (and unpublicized) systems often seems preordained—that is, traceable to their beginnings. Some system development projects start doomed, and no downstream engineering efforts are likely to rescue them. Other projects seem fated for success almost in spite of poor downstream decisions. The initial concept is so "right" that its success is inevitable, even if not necessarily with the first group that tries to execute it. This is not a new realization. It was just as apparent to the ancient Egyptians, Greeks, and Romans who originated classical architecting in response to it. The difference between their times and now is in the extraordinary complexity and technological capability of what could then and now be built.

Today's architecting must handle systems of types unknown until the early 21st century, for example, systems that are very high quality, real time, closed loop, reconfigurable, interactive, software intensive, and, for all practical purposes, autonomous. New domains like proliferated satellite networks, sensor webs, autonomous vehicles (and swarms of vehicles), personalized health services, and joint service command and control are calling for new architectures—and for architects specializing in those domains. Their needs and lessons learned are in turn leading to new architecting concepts and tools and to the acknowledgment of a new formalism—and evolving profession—called systems architecting, a combination of the principles and concepts of both systems and architecting. However, for all the new complexity, many of the roots of success and failure are nearly constant over time. By examining a series of case studies, interwoven with a discussion of the particular domains to which they belong, we can see how relatively timeless principles (for example, technical and operational coupled revolution, and strategic consistency) largely govern success and failure.

The reasons behind the general acknowledgment of architecting in the new systems world are traceable to that remarkable period immediately after the end of the Cold War in the early-1990s. Abruptly, by historical standards, a 50-year period of continuity ended. During the same period, there was a dramatic upsurge in the use of smart, real-time systems, both civilian and military, that required much more than straightforward refinements of established system forms. Long-range management strategies and design rules, based on years of continuity, came under challenge.

That challenge was not short lived; instead, it has resorted itself repeatedly in the years between editions of this book. It is now apparent that the new era of global transportation, global communications, global competition, and global security turmoil is not only different in type and direction; it is unique technologically and politically. It is a time of restructuring and invention, of architecting new products and processes, and of new ways of thinking about how systems are created and built.

Long-standing assumptions and methods are under challenge. For example, for many engineers, architectures were a given; automobiles, airplanes, and even spacecraft had the same architectural forms for decades. Where was there room for architectural revolution? Global competition and a surge of innovation soon provided an answer. Architecturally different systems were capturing markets. Consumer product lines and defense systems are well-reported examples. Other questions remained: How can software architectures be created that evolve as fast as their supporting technologies? How deeply should a systems architect go into the details of all the system's subsystems? What are the relationships between the architectures of systems and the human organizations that design, build, support, and use them?

DISTINGUISHING BETWEEN ARCHITECTING, ENGINEERING, AND PROJECT MANAGEMENT

Because it is the most asked by engineers introduced to architecting, the first issue to address is the distinction between architecting and engineering in general—that is, regardless of engineering discipline. Although civil engineers and civil architects, even after centuries of debate, have not answered that question in the abstract, they have answered it in practice. Generally speaking, engineering deals primarily with measurables using analytic tools derived from mathematics and the hard sciences. Engineering is largely a deductive process. Architecting deals largely with unmeasurables using nonquantitative tools and guidelines based on practical lessons learned; that is, architecting is an inductive process. Architecting embraces the world of the user/sponsor/client, with all the ambiguity and imprecision that may entail. Architecting seeks to communicate across the gap from the user/sponsor/client to the engineer/developer, and architecting is complete (at least its initial phase) when a system is well enough defined to engage developers. An engineer, at least in common development environments, can work with user/client surrogates, like requirements and derived models. An architect must always be engaged with the user/client and deeply understand the mission for which a system is to be a "solution."

At a more detailed level, engineering is concerned more with quantifiable costs, architecting more with qualitative worth. Engineering aims for technical optimization, architecting for client satisfaction. Engineering is more of a science, and architecting is more of an art. Although the border between them is often fuzzy, the distinction at the end is clear.

In brief, the practical distinction between engineering and architecting is in the problems faced and the tools used to tackle them. This same distinction appears to apply whether the engineering branch involved is civil, mechanical, chemical, electrical, electronic, aerospace, software, or systems. Both architecting and engineering can be found in every one of the established disciplines and in the multidisciplinary whole-system contexts. Multidisciplinary systems are engineered as much as any product in a disciplinary environment. Architecting and engineering are roles, distinguished by their characteristics. They represent two edges of a continuum of systems practice. Individual engineers often fill roles across the continuum at various points in their careers or on different systems. The characteristics of the roles, and that these roles are on a continuum, are shown in Table P.1.

As the table indicates, architecting is characterized by dealing with ill-structured situations, situations where neither goals nor means are known with much certainty. In systems engineering terms, the requirements for the system have not been stated more than vaguely, and the architect cannot appeal to the client for a resolution, as the client has engaged the architect precisely to assist and advise in such a resolution. The architect engages in a joint exploration of requirements and design, in contrast to the classic engineering approach of seeking an optimal design solution to a clearly defined set of objectives.

Because the situation is ill structured, the goal cannot be optimization. The architect seeks satisfactory and feasible problem-solution pairs. Good architecture

TABLE P.1
Characteristics of the Roles in the Architecting and Engineering Continuum

Characteristic	Architecting	Architecting and Engineering	Engineering
Situation/goals	Ill-structured	Constrained	Understood
	Satisfaction	Compliance	Optimization
Methods	Heuristics	$\longleftrightarrow$	Equations
	Synthesis	$\longleftrightarrow$	Analysis
	Art and science	Art **and** science	**Science** and art
Interfaces	Focus on "mis-fits"	Critical	Completeness
System integrity maintained through	"Single mind"	Clear objectives	Disciplined methodology and process
Management issues	Working for client	Working with client	Working for builder
	Conceptualization and certification	Whole waterfall	Meeting project requirements
	Confidentiality	Conflict of interest	Profit versus cost

and good engineering are both the products of art and science, and a mixture of analysis and heuristics. However, the weight will fall on heuristics and "art" during architecting.

An "ill-structured" problem is a problem where the statement of the problem depends on the statement of the solution. In other words, knowing what you can do changes your mind about what you want to do. A solution that appears correct based on an initial understanding of the problem may be revealed as wholly inadequate with more experience. Architecting embraces ill-structured problems. A basic tenet of architecting is to assume that one will face ill-structured problems and to configure one's processes to allow for it.

One way to clearly see the distinction between architecting and engineering is in the approach to interfaces and system integrity. When a complex system is built (say one involving 10,000 person-years of effort), only absolute consistency and completeness of interface descriptions and disciplined methodology and process will suffice. Even on relatively simple systems, minor mistakes in details can have inordinate consequences. When a system is physically assembled, it matters little whether an interface is high tech or low tech; if it is not exactly correct, the system does not work. In contrast, during architecting, it is necessary only to identify the interfaces that cannot work—the mis-fits. Mis-fits must be eliminated during architecting, and then, interfaces should be resolved in order of criticality and risk as development proceeds into engineering.

Table P.1 has proven useful and popular in explaining architecting in educational settings. It effectively highlights different roles taken by different people. More than once a classroom session that included a detailed discussion of the table and some case-study work has concluded with a student commenting "Now I know what architecting is, why it is important, and that I don't want to do it." Those students recognized that doing the left column work in Table P.1 was essential and had to be done if engineering is to be done well, but they don't want to do it. We find that a good outcome. The required ratio of architects to engineers is very low in most organizations. Most organizations will need a lot more good engineers than they will need good architects.

One important point is that the table represents management in the classical paradigm of how architecting is done, not necessarily how it actually is done. Classically, architecting is performed by a third party working for the client. In practice, the situation is more complex as the architecting might be done by the builder before a client is found, might be mixed into a competitive procurement, or might be done by the client. These variations are taken up in chapters to come.

As for project management, architecting clearly exists within the larger project cycle. If we examine the development of systems very holistically, looking from the earliest to the latest phases, we see architecting existing within that large picture. But, at a practical level, what is usually taught as project management has a narrower focus, as does what is usually taught as systems engineering. The narrower focus assumes that definite requirements (in the unambiguous, orthogonal, measurable, and testable senses) exist and can be documented, that budgets and schedules exist and must be managed, and that specific end points are defined through contracts or other agreements. For a given organization (a contract developer, a government project office), that narrower focus may be all that matters and may encompass billions of dollars. Often, by the time that narrower focus has been arrived at, the architecting is over. Often, by the time that narrower focus has been arrived at, the project is already doomed to failure or well on its way to success.

Table P.1 implies an important distinction in architecting as currently practiced. The table, and this book, emphasizes architecting as decision-making. Architecting has been accomplished when the fundamental structural decisions about a system and how it will be developed have been made, regardless of what sort of architecture description document has been produced. In contrast, many "architecture" projects currently being conducted are description-centric. Their basis is producing an architecture framework-compliant description document about a system or system-of-systems that typically already exists. These are sometimes called "as-is" or "baseline" architecture documents. This book has relatively little to say about such projects, although we cover standards and methods associated with architecture in some detail in Chapters 10 and 11. The authors' emphasis, and the emphasis of this book, is on the structural decisions that underlie the "as-is" system and identifying and making the structural decisions for the next system. The methods of this book have been usefully applied to making an assessment of the as-is decisions and reevaluating those decisions, but our focus remains on building something new.

ARCHITECTING AS ART AND SCIENCE

Systems architecting is the subject of this book, and the art of it in particular, because, being the most interdisciplinary, its tools can be most easily applied in the other branches. Good architecting is not just an art, and virtually, all architects of high-technology systems, in the authors' experience, have strong science backgrounds. But the science needed for systems architecting already is the subject of many publications, but few address the art systematically and in depth. The overriding objective of this book is to bring the reader a toolbox of techniques for handling ill-structured, architectural problems that are different from the engineering methods already taught well and widely published.

It is important in understanding the subject of this book to clarify certain expressions. The word "architecture" in the context of civil works can mean a structure, a process, or a profession; in this text, it refers only to the structure, although we will often consider "structures" that are quite abstract. The word "architecting" refers only to the process. Architecting is an invented word to describe how architectures are created, much as engineering describes how "engines" and other artifacts are created. In another, subtler, distinction from conventional usage, an "architect" is meant here to be an individual engaged in the process of architecting, regardless of domain, job title, or employer. By definition and practice, from time to time, an architect may perform engineering and an engineer may perform architecting—whatever it takes to get the job done.

Clearly, both processes involve elements of the other. Architecting requires top-level quantitative analysis to determine feasibility and quantitative measures to certify readiness for use. Engineering can and occasionally does require the creation of architecturally different alternatives to resolve otherwise intractable design problems. Good engineers are armed with an array of heuristics to guide tasks ranging from structuring a mathematical analysis to debugging a piece of electronic hardware. For complex systems, both engineering and architecting are essential. In practice, it is usually necessary to draw a sharp line between them only when that sharp line is imposed by business or legal requirements.

CRITERIA FOR MATURE AND EFFECTIVE SYSTEMS ARCHITECTING

An increasingly important need of project managers and clients is for criteria to judge the maturity and effectiveness of systems architecting in their projects—criteria analogous to those developed for software development by Carnegie Mellon's Software Engineering Institute. Based upon experience to date, criteria for systems architecting appear to be, in rough order of attainment:

- A recognition by clients and others of the need to architect complex systems.
- An accepted discipline to perform that function, in particular, the existence of architectural methods, standards, and organizations.
- A recognized separation of value judgments and technical decisions between client, architect, and builder.
- A recognition that architecture is an art as well as a science, in particular, the development and use of nonanalytic as well as analytic techniques.
- The effective utilization of an educated professional cadre—that is, of master's level, if not doctorate level, individuals and teams engaged in the process of systems-level architecting.

By those criteria, systems architecting is in its adolescence, a time of challenge, opportunity, and controversy. History and the needs of global competition would seem to indicate adulthood is close at hand.

THE ARCHITECTURE OF THIS BOOK

The original purpose of this book was to restate and extend into the future the retrospective architecting paradigm of Rechtin (1991). While it was originally intended, and used, as a companion to the first book rather than a replacement, it is intended to stand on its own. Over the editions, and as the earlier book has gone in and out of print, we have paid further attention to making this a standalone book. There are topics in Rechtin (1991) that are not covered in this book, however, and the reader may wish to refer to the earlier book for a detailed treatment. Those topics particularly include the definition and characterization of ultraquality, architecting in the face of purposeful opposition, and the role of economics in systems architecting.

An essential part of both retrospective and extended paradigms is the recognition that systems architecting is part art and part science. Part I of this book further defines what we mean by "the art" and extends the central role of heuristics with different perspectives for identifying and using them. We open Part I in this edition with a case study of the DARPA Grand Challenge in the early 2000s that jumpstarted the field of autonomous vehicles. We invite the reader to put themselves in the shoes of a team tasked to develop a vehicle to perform a task that nobody knew how to do and to consider the early choices every such team had to make. The choices must encompass both technical and programmatic issues and are fundamentally architectural. Part II introduces five important domains that contribute to the understanding of that art. We buttress the retrospective lessons of the original book by providing some detailed stories on some of the case studies that motivated the original work and use those case studies to introduce each chapter in Part II. Part III helps bridge the space between the science and the art of architecting. It develops the core architecting process of modeling and representation and does so from the perspective of model-based systems engineering (MBSE) and digital engineering (DE). Part III is the most changed from previous editions and the most influenced by recent developments in standardized modeling notations and the widespread availability of powerful tools. Part IV concentrates on architecting as a profession: its relationship to business strategy and activities; the political process and its part in system design; and the professionalization of the field through education, research, and development of a body of knowledge.

The architecture of Part II deserves an explanation. Without one, the reader may inadvertently skip some of the domains—builder-architected systems, manufacturing systems, social systems, software systems, and collaborative systems—because they are outside the reader's immediate field of interest. These chapters, instead, recognize the diverse origins of heuristics and diverse means of illustrating and exploiting them. Heuristics often first surface in a specialized domain where they address an especially prominent problem. Then, by abstraction or analogy, they are carried over to others and become generic. Such is certainly the case in the selected domains. In these chapters, the usual format of stating a heuristic and then illustrating it in several domains is reversed. Instead, it is stated, but in generic terms, in the domain where it is most apparent. Readers are encouraged to scan all the chapters of Part II. The chapters may even suggest domains, other than the reader's, where the reader's experience can be valuable in these times of vocational change. References are provided for further exploration. Moreover, the case studies that open each chapter are chosen to illustrate a wide variety of lessons in the practice of architecting. These lessons are not restricted to the domain of the case study alone. For professionals already in one of the domains, the description of each is from an architectural perspective, looking for those essentials that yield generic heuristics and providing in return other generic ones that might help better understand those essentials. In any case, the chapters most emphatically are not intended to advise specialists about their specialties.

Architecting is inherently a multidimensional subject, difficult to describe in the linear, word-follows-word, format of a book. Consequently, it is occasionally necessary to repeat the same concept in several places, internally and between books. A good example is the concept of systems. Architecting can also be organized around several different themes or threads. Alternative organizations would be around the elements of the system development process or the formation of strategy. This book is organized by fundamentals, tools, tasks, domains, models, and vocation. Readers are encouraged to choose their own personal theme as they go along. It will help tie systems architecting to their own needs.

Exercises are interspersed in the text, designed for self-test of understanding and critiquing the material just presented. If the reader disagrees, then the disagreement should be countered with examples and lessons learned—the basic way that mathematically unprovable statements are accepted or denied. Most of the exercises are thought problems, with no correct answers. Read them, and if the response is intuitively obvious, charge straight ahead. Otherwise, pause and reflect a bit. A useful insight may have been missed. Other exercises are intended to provide opportunities for long-term study and further exploration of the subject. That is, they are roughly the equivalent of a master's thesis.

Notes and references are organized by chapter. Heuristics by tradition are boldfaced when they appear alone, with an appended list of them completing the text. Figure slides are also available on the Routledge website (www.routledge.com/9781032774381).

CHANGES SINCE THE THIRD EDITION

The changes to this, the fourth edition, grow out of several inter-related experiences and threads. These changes are broken down into three areas: Material to clarify teaching points, integration of MBSE and digital engineering, and the emergence of key standards and practices.

First, we have made extensive use of the third edition in teaching the subject to professional and academic audiences. From the student feedback in those courses, we have a number of areas to change and clarify. The addition of the DARPA Grand Challenge case study in Part I came about as we found it worked exceptionally well to clarify the scope and intent of architecting versus engineering and project management. It also helped highlight, even to those engineers with the most narrow analytical focus, the role of judgment-based design and that in so many real cases there just aren't a perfect set of requirements waiting to be discovered if only one asks the right sponsor.

We also make more extensive use of the APM-ASAM process for organizing the work of architecting. While it would be a rare real-world project that could follow APM-ASAM step-by-step with little adaptation, it is still very useful for getting started, or having something concrete to begin to organize the mess of differing considerations one has to deal with in reality.

Second, Part III has been rewritten to build on now widely accepted MBSE tools and approaches. In the earlier editions, we appealed to the use of "integrated modeling" as a tool for architecture description. We no longer have to refer to a spectrum of tools and methods, hoping the reader will be able to pick from the pieces they need in their environment. MBSE notations have large settled down (though there are still competitors), and there are high-capability commercial modeling tools. We describe these notations and tools and map them directly to APM-ASAM elements.

Third, one of the reasons the integrated modeling and notation conflicts have settled down is the emergence of standards. At the time of the third edition, we could cite the ANSI/IEEE 1471 standard (Maier et al. 2004) and its early ISO derivative as the emerging preferred approach to architecture description. The ISO/IEC/IEEE 42010 standard on architecture description has now been through multiple revisions and is widely accepted (if not necessary as widely implemented in full). There are now companion standards in the 42000 series that cover other architectural aspects. Ideas that were new when this book was first published in the 1990s are now standard practices.

Standardizing architecture descriptions is like standardizing blueprint standards. It is very useful to the practitioner but doesn't result in better buildings. Architecture standardization is more like the development of building code. One lesson there is it works best when technology and architecture are stable, as it is in civil construction. There should be no surprise that there isn't much in the way of similar standardization in other areas as the rate of innovation has only increased since the third edition was published.

Changes to standards and the general increase in maturity of concepts are recognized in changes in Parts III and IV. Here, we have rewritten or replaced material that was speculative 15 years ago to being standard today. Much of the material on politics and business is timeless, however, as it reflects the slowly changing on-the-ground realities of how decisions are made on funding large systems.

It is still true that architecture can be seen as the physical (or technical) embodiment of strategy. Conversely, architecture without strategy is, essentially by definition, incoherent. Many of the common problems encountered in attempting to improve architecting practices are linked directly to problems in organizational strategy. Moreover, this linkage provides fertile ground for looking at intellectual links with other engineering-related subjects, such as decision theory.

READERSHIP AND USAGE OF THIS BOOK

This book is written for present and future systems architects; for experienced engineers interested in expanding their expertise beyond a single field; and for thoughtful individuals concerned with creating, building, or using complex systems. It is intended either for simple reading, for reference in professional practice, or in classroom settings. From experience with its predecessor, the book can be used as a reference work for graduate studies, for senior capstone courses in engineering and architecture, for executive training programs, and for the further education of consultants and systems acquisition and integration specialists, and as background for legislative staffs.

This book continues to be a basic text for courses in systems architecture and engineering at several universities and in several extended professional courses. Best practice in using this book in such courses appears to be to combine it with selected case studies and an extended case exercise. Because architecting is about having skills, not about having rote knowledge, it can only be demonstrated in the doing. The author's courses have been built around course-long case exercises, normally chosen in the student's individual field. In public courses, such as at universities, the case studies presented here are appropriate for use. The source materials are reasonably available, and students can expand on what is presented here and create their own interpretations. In professional education settings, it is preferable to replace the case studies in class with case studies drawn directly from the student's home organizations.

Everything in this book represents the opinions of the authors and does not represent the official position of any organization they have been associated with or their customers. All errors are the responsibility of the authors.

REFERENCES

Maier, M. W., et al. (2004). "ANSI/IEEE 1471 and systems engineering." *Systems Engineering* **7**(3): 257–270.

NRC (2008). *Pre-milestone A and Early-phase Systems Engineering: A Retrospective Review and Benefits for Future Air Force Systems Acquisition*. Washington, DC: National Academies Press.

Rechtin, E. (1991). *Systems Architecting: Creating and Building Complex Systems*, Englewood Cliffs, NJ: Prentice Hall.

Acknowledgments

Eberhardt Rechtin, who originated and motivated so much of the thinking here, passed away in 2006. Although no longer with us, his spirit and words pervade this book. The first edition of this book was formulated while he taught at the University of Southern California (USC). He treasured his interactions with his students there and believed that the work was enormously improved through the process of teaching them. At least a dozen of them deserve special recognition for their unique insights and penetrating commentary: Michael Asato, Kenneth Cureton, Susan Dawes, Norman P. Geis, Douglas R. King, Kathrin Kjos, Jonathan Losk, Ray Madachy, Archie W. Mills, Jerry Olivieri, Tom Pieronek, and Marilee Wheaton. The quick understanding and extension of the architecting process by all the students has been a joy to behold and a privilege to acknowledge.

Several members of the USC faculty were instrumental in finding a place for this work and the associated program. In particular, there was Associate Dean Richard Miller; Associate Dean Elliot Axelband, who originally requested this book and directed the USC Master's Program in Systems Engineering and Architecture; and two members of the School of Engineering staff, Margery Berti and Mary Froehlig, who architected the Master of Science in Systems Architecture and Engineering out of an experimental course and a flexible array of multidisciplinary courses at USC. Particular thanks go to Charles Weber, who greatly encouraged Eb Rechtin in creating the program, and encouraged his then-graduate student, Mark W. Maier to take the first class offered in systems architecting as part of his Ph.D. in Electrical Engineering Systems. Brenda Forman, then of USC, now retired from the Lockheed Martin Corporation and the author of Chapter 12, accepted the challenge of creating a unique course on the "facts of life" in the national political process and how they affect—indeed often determine—architecting and engineering design.

Our former colleagues at The Aerospace Corporation have been instrumental in the later development of the ideas that have gone into this book.

Mark W. Maier taught many versions of this material under the auspices of the Aerospace Systems Architecting Program and its derivatives. That program was dependent on the support of Mal Depont, William Hiatt, Dave Evans, and Bruce Gardner of the Aerospace Institute and several key sponsors in the U.S. Government. The program in turn had many other collaborators, including Kevin Kreitman, Andrea Amram, Glenn Buchan, and James Martin. Of great importance to the quality of the presentation was the extensive editing and organization of the materials in the Aerospace Systems Architecting Program by Bonnie Johnston and Margaret Maher.

Manuscripts may be written by authors but publishing them is a profession and contribution unto itself requiring judgment, encouragement, tact, and a considerable willingness to take risk. For all of these, we thank Norm Stanton, a senior editor of Taylor & Francis Group/CRC Press and editor of the first edition of this book, who has understood and supported the field beginning with the publication of Frederick

Brooks' classic architecting book, *The Mythical Man-Month*; and Cindy Carelli for her support of subsequent editions of this book.

Of course, a particular acknowledgment is due to the Rechtin and Maier families for the inspiration and support they have provided over the years, and their continuing support in revising this book.

Mark W. Maier

About the Authors

Dr. Mark W. Maier is a technical fellow at The Aerospace Corporation, Electronics and Sensors Division, an architect-engineering firm specializing in space systems for the United States Government. His specialty is systems architecture. He teaches and consults in that subject for The Aerospace Corporation, its clients, and corporations throughout the United States and Europe. He has done pioneering work in the field, especially in collaborative systems, socio-technical systems, and modeling, as well as significant research in advanced sensors. He is a senior member of the IEEE and one of the lead authors of ANSI/IEEE 1471 Recommended Practice for Architecture Description, which has become the basis for further international standardization. Dr. Maier received his B.S. degree in Engineering and Applied Science and an M.S. in Electrical Engineering from Caltech in 1983 and 1984, respectively. He joined the Hughes Aircraft Company in El Segundo, CA, upon graduating from Caltech in 1983. At Hughes, he was a section head responsible for signal processing algorithm design and systems engineering. While at Hughes, he was awarded a Howard Hughes Doctoral Fellowship, on which he completed a Ph.D. in Electrical Engineering, specializing in radar signal processing, at USC. At USC, he began his collaboration with Dr. Eberhardt Rechtin, who started the first systems architecture academic program. In 1992, he joined the faculty of the University of Alabama in Huntsville in the Electrical and Computer Engineering Department. As an associate professor at UAH, he carried out research in system architecture, stereo image compression, and radar signal processing. He has published several dozen journal articles and conference papers in these fields.

Dr. Eberhardt Rechtin received his B.S. and Ph.D. degrees from the California Institute of Technology (Caltech) in 1946 and 1950, respectively. He joined Caltech's Jet Propulsion Laboratory (JPL) in 1948 as an engineer, leaving it in 1967 as Assistant Director. At JPL, he was the chief architect and director of the NASA/JPL Deep Space Network. In 1967, he joined the Office of the Secretary of Defense as the director of the Advanced Research Projects Agency (ARPA) and later as assistant secretary of Defense for Telecommunications. He left the Department of Defense in 1973 to become chief engineer of Hewlett-Packard. He was elected as the president and CEO of The Aerospace Corporation in 1977, retiring in 1987. After retirement from Aerospace, he joined the faculty of the University of Southern California (USC) as a professor with joint appointments in Electrical Engineering-Systems, Industrial & Systems Engineering, and Aerospace Engineering. During his time at USC, he founded their graduate program in Systems Architecting. Dr. Rechtin was extensively honored for his engineering accomplishments. He was a member of the National Academy of Engineering; a fellow of the IEEE, AIAA, and the AAAS; and an Academician of the International Academy of Astronautics. He was further honored by the IEEE with its Alexander Graham Bell Award, by the Department of Defense with the Distinguished Public Service Award, by NASA with its Medal for

Exceptional Scientific Achievement, by the AIAA with its von Karman Lectureship, by Caltech with its Distinguished Alumni award, and by NEC with its C&C Prize. His published works include *Systems Architecting: Creating and Building Complex Systems*, Prentice Hall, 1991; *The Art of Systems Architecting* with Mark W. Maier; and *Why Eagles Can't Swim: The Systems Architecting of Organizations*. Dr. Rechtin passed away in 2006.

Part I

Introduction

A BRIEF REVIEW OF CLASSICAL ARCHITECTING METHODS

Architecting: The Art and Science of Designing and Building Systems[1]

Building on the paradigm of civil architecting, we ask what have been traditional approaches to how to create architecture? What methods are identified with architectural practice that are distinct from (or overlap with) engineering practice? The four most important methods are referred to as normative, rational, participative, and heuristic (Lang 1987, Rowe 1991)[2] (Table PI.1). As might be expected, like architecting itself, they contain both science and art. The science is largely contained in the first two, normative and rational, and the art in the last two, participative and heuristic.

The normative technique is solution based; it prescribes architecture as it "should be"—that is, as given in handbooks, civil codes, and pronouncements by acknowledged masters. The normative method proceeds through a set of hard rules to follow. Success is defined by following the rules. Building codes are examples of the normative method. You assess success on building codes by whether or not your design and construction are compliant.

While normative success might be defined by following the rules, the rules had to come from somewhere and need to be strongly linked to desired outcomes (say that the building survives harsh conditions). As science and engineering developed, we had an increasing ability to move from desired outcome to design by rigorous analysis. This leads us to the rational method, scientific and mathematical principles to be followed in arriving at a solution to a stated problem. Both the normative and rational methods are deductive, allow for easy certification of a system, are well understood,

TABLE PI.1
Four Architecting Methodologies

Normative (solution based)	Examples: Building codes and communications standards
Rational (method based)	Examples: Systems analysis and engineering
Participative (stakeholder based)	Examples: Concurrent engineering and brainstorming
Heuristic (lessons learned)	Examples: All the really important mistakes are made on the first day, partition for maximum cohesion and minimal coupling

and are widely taught in academia and industry. Moreover, the best normative rules are discovered through engineering science (think of modern building codes)—truly a formidable set of positives.

Although science-based methods are absolutely necessary parts of architecting, they are not the focus of this book. They are already well treated in numerous architectural and engineering texts. Most people who are serious practitioners of systems architecting, or who aspire to be serious practitioners, come from an engineering and science background. They already realize the necessity of applying scientific and quantitative thinking to the design of complex systems and already have a substantial analytical toolkit. Equally necessary, and the focus of this part of the book, is the art, or practice, needed to complement the science.

In contrast with science-based methodologies, the art or practice of architecting—like the practices of medicine, law, and business—is largely nonanalytic, inductive, difficult to certify, less understood, and, at least until recently, seldom taught formally in either academia or industry. It is a process of insights, vision, intuitions, judgment calls, and even "taste" (Spinrad 1988). It is key to creating truly new types of systems for new and often unprecedented applications. Here are some of the reasons.

For unprecedented systems, past data are of limited use. When building something truly, new rules from the past are unlikely to be of much use. Even for precedented systems, analysis can be overwhelmed by too many unknowns, too many stakeholders, too many possibilities, and too little time for data gathering and analysis to be practical. To cap it off, many of the most important factors are not measurable. Perceptions of worth, safety, affordability, political acceptance, environmental impact, public health, and even national security provide a limited basis for numerical analyses—even if they were not highly variable and uncertain. Of course, any of those characteristics can be modeled quantitatively. But the models are not the perceptions. Value is judged by people, not models, even if the people use models to be informed. Yet, if the system is to be successful, these perceptions must be accommodated from the first, top-level, conceptual model down through its derivatives.

The art of architecting, therefore, complements its science where science is weakest: in dealing with immeasurables, in reducing past experience and wisdom to practice, in conceptualization, in inspirationally putting disparate things together, in providing "sanity checks," and in warning of likely but unprovable trouble ahead. Terms like reasonable assumptions, guidelines, indicators, elegant design, and beautiful performance are not out of place in this art, nor are lemon, disaster, snafu, or loser. These terms are hardly quantifiable but are as real in impact as any science.

The participative methodology recognizes the complexities created by multiple stakeholders. Its objective is consensus. As a notable example, designers and manufacturers need to agree on a multiplicity of details if an end product is to be manufactured easily, quickly, and profitably. In simple but common cases, only the client, architect, and contractor have to be in agreement. But as systems become more complex, many more participants have to agree as well.

The heuristics methodology is based on "common sense"—that is, on what is sensible in a given context. Contextual sense comes from collective experience stated in a simple and concise manner as possible. Put another way, the "common sense" we want is inducted, preferably from wide and diverse experience. These statements are called heuristics, the subject of Chapter 2, and are of special importance to architecting because they provide guides through the rocks and shoals of intractable, "wicked" system problems. *Simplify!* is the first and probably the most important of them. They exist in the hundreds if not thousands in architecting and engineering, yet they are some of the most practical and pragmatic tools in the architect's kit of tools.

DIFFERENT METHODS FOR DIFFERENT PHASES OF ARCHITECTING

The nature of classical architecting changes as the project moves from phase to phase. In the earliest stages of a project, it is a structuring of an unstructured mix of dreams, hopes, needs, and technical possibilities when what is most needed has been called an inspired synthesizing of feasible technologies. It is a time for the art of architecting. Later, architecting becomes an integration of, and mediation among, competing subsystems and interests—a time for rational and normative methodology. Eventually, there comes certification to all that the system is suitable for use, when it may take all the art and science to that point to declare the system as built is complete and ready for use. If all that was done very successfully, there will be many cycles of evolution after, and eventually, the process must repeat (perhaps by those who were already successful, quite possibly by others seeking to unseat them) as something new replaces what has become old.

Not surprisingly, architecting is often individualistic, and the end results reflect it. As Frederick P. Brooks put it in 1983 (Brooks Jr 1995) and Robert Spinrad stated in 1987 (Spinrad 1987), the greatest architectures are the product of a single mind—or of a very small, carefully structured team—to which should be added in all fairness: a responsible and patient client, a dedicated builder, and talented designers and engineers.

NOTES

1 *Webster's II, New Riverside University Dictionary.* Boston, MA: Riverside 1984. As adapted for systems by substitution of "building systems" for "erecting buildings."

2 They are adapted for systems architecting in Rechtin, E. (1991). *Systems Architecting: Creating and Building Complex Systems* (pp. 14–22). Englewood Cliffs, NJ: Prentice Hall.

REFERENCES

Brooks Jr, F. P. (1995). *The Mythical Man-Month (Anniversary ed.)*, Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Lang, J. (1987). *Creating Architectural Theory*, New York: van NostrandRheinhold.

Rowe, P. G. (1991). *Design Thinking*, Cambridge, MA: MIT Press.

Spinrad, R. J. (1987).*Lecture at the University of Southern California*, Los Angeles, CA: University of Southern California.

Spinrad, R. J. (1988).*Lecture at the University of Southern California*, Los Angeles, CA: University of Southern California.

Case Study 1

The DARPA Grand Challenge

To Build Something New

ARCHITECTING BY EXAMPLE

Architecting and architecture span a wide range of scenarios, concerns, and methods. Architecting can apply to fresh-start projects, legacy replacements, and incremental revisions. It spans from stakeholder analysis of vaguely stated needs and constraints to (potentially) detailed physical engineering concerns. Before we build a more formal framework for what systems architecting is and how to do it, we need to get a feel for what it entails and how it differs from other engineering, program management, or analytical perspectives. Some readers may already have a good grasp of this and have personal examples readily at hand. But others will not and some kind of unifying example has proven very useful with many audiences. To provide this introductory example, consider the case of the Defense Advanced Research Projects Agency (DARPA) Grand Challenge.

By way of background, the US DARPA was established in 1958 to accelerate the infusion of advanced research results into militarily valuable systems. The signature DARPA approach is to identify an area where basic research has revealed the potential for important new capability, then sponsor the development of an application demonstration in a prototype system, with an opportunity for transition to the military services. The exact form of this has varied from technology area to technology area and over time, but it has repeatedly resulted in exceptional new US military capabilities. Famously, DARPA has played a major role in the early development of stealth technology, weather satellites, GPS, drones, and the Internet (via the predecessor ARPAnet).

In 2003, DARPA announced the DARPA Grand Challenge for autonomous vehicles. The motivation was to accelerate the progress in developing militarily relevant autonomous ground vehicles. Ultimately, this might include autonomous fighting vehicles. The more immediate goal was to enable autonomous supply vehicles, whether large trucks operate behind the lines or smaller vehicles operate close to the front. A second goal was to enable autonomous operation of scout and mine-clearing vehicles. This was heavily motivated by experience in Iraq and Afghanistan, where vehicle operators were broadly exposed to improvised explosive device threat and insurgent attacks throughout the battle space.

This program had a very unusual structure. Instead of typical contracted research and demonstration activity, it would be an all-comers race with a million-dollar winner-take-all prize. Competitors would bring their vehicles to a designated site in the desert between California and Nevada on a set date in 2004. A few hours before start time, they would be given a set of GPS waypoints defining an approximately 120-mile course of desert roads or off-road terrain. They would start one at a time and attempt to traverse the full course using solely on-board sensors and control mechanisms with no human supervision or intervention (save a kill signal in case of on-course problems). There are a number of scholarly accounts of the event (Behringer et al. 2004, Ozguner et al. 2004, Seetharaman et al. 2006, Thrun et al. 2006) among others. There is a well-produced documentary video (NOVA 2006) that the reader may want to review in conjunction with reading this case study.

When DARPA made the announcement, there was a background of many years of autonomous vehicle research, but nothing like the proposed race had ever been accomplished. The general opinion was that the set task was infeasible with current technology, and no competitors would succeed. When the race was held in March 2004, the skeptics were seemly proven right. Most vehicles crashed or otherwise failed within sight of the start line. The best performer only made 7.5 miles of the course before stranding itself on a rock. DARPA's response was to double the prize money and schedule another race a bit more than a year away.

Interestingly, the second race attracted a far larger group of competitors in spite (or because of?) the failures of the first. So many teams developed vehicles that DARPA had to hold a qualifying event first to select a workable number of vehicles to race. Performance was dramatically improved, with most vehicles in the second race exceeding the best-performing vehicle of the first race, and four vehicles completing the full course with three under the 10-hour time limit. The success of the second Grand Challenge race set off a wave of commercial activity in the autonomous vehicle area. There was considerable optimism that true autonomous vehicles would soon occupy public roads. However, the actual result has been more mixed. The DARPA Grand Challenge was indisputedly successful at jumpstarting serious development and large investments in the field outside the U.S. Department of Defense (DoD). The actual path of working applications has been more circuitous. In some areas, like autonomous ocean vehicles and air vehicles, there are many working military applications. In civilian land transport, there are still few deployed applications, at least at the time of this writing (2024), though there is a vigorous, and sometimes controversial, prototyping process.

Our objective is not to assess the success of the DARPA program, but it is to consider what it means to do architecting, to create architecture, to *build something new*, using the Grand Challenge as an example. We invite the readers to imagine themselves as members of a team shortly after the Challenge announcement in 2003 considering entering the race and trying to structure the first design activities.

Day 1 of the New Project

To make the situation a little more concrete, we will assume that the team members are all part of the School of Engineering at a major university. The Dean of Engineering is excited about the challenge and wants to explore the university participating. The Dean has allocated some discretionary money to get things started, but it is obvious that far more money will be necessary to build and field a competitive vehicle (or may be vehicles?). Several professors have research programs in areas relevant to the challenge (vehicle design, sensing, control theory, and robotics), and the School has significant laboratory facilities where vehicles for things like the "Formula SAE" (SAE 2024) have been built. The school has significant corporate partners in research and education programs, some clearly relevant to the challenge.

While there are many resources that are clearly relevant, all people are aware of one over-riding consideration, nobody, anywhere, knows how to design an autonomous vehicle that will assuredly perform the Grand Challenge task. Nobody, anywhere, has built a fully autonomous vehicle that has gone the distances and speeds required to meet the Challenge's minimum requirements (120 miles in under 10 hours fully autonomously). Not only is originality required, but if someone put forth a solution that would succeed, there is no way to know that without getting out to the field and trying it, either before the race or simply in the race. If instead you imagine yourself in the place of a team taking on the second running of the race, the observation that "nobody knows how to do this" remains, now reinforced by the failure of the vehicles in the first race.

A conventional systems engineering approach seems attractive, for a moment. The usual first question asked would be "what are the requirements?" It is not hard to define the operational requirements, there are hardly any.

- Bring your vehicle to the race location
- Receive the course waypoints when available and load them into your vehicle
- Drive the course as defined by the waypoints, sensing the environment and adapting to the course as needed to avoid being stuck
- Respond to a kill signal and stop

On the physical side, the only requirements of significance were maximum size and weight constraints.

Going from these operational requirements to functions is superficially easy. You can outline functions for sensing GPS position, sensing the environment, moving in the environment, adjusting course, and so forth. The trouble is that going below that level runs into big problems immediately. Driving straight line waypoint-to-waypoint will presumably not be feasible (lots of mountainous terrain and big rocks out there). What kind of obstacles do we have to detect and characterize? How sophisticated do our route adaptation algorithms have to be to find a feasible course? We do not know in advance, we have to define some kind of environment space. If we make the problem too hard, we will probably go too slow compared to a vehicle that makes different assumptions. If we assume too easy, we will never make it around the course. Choosing a driving speed is a subtle trade between going too slow (and losing) and going too fast (and upping the crash probability). To really know that trade, we would need a lot of data on autonomous vehicle operation, but there is not any data (unless we collect some, making our effort larger).

If that process does not bog down, some in the group will point out the complexity of the goals. Is the goal to win the race and get the money? It will take more than a million dollars (probably) to build a competitive vehicle. For sure, sponsorship money will be required. In a university environment, if we get lots of sponsorship and build a lot of interesting technology and educate our students, do we "win" regardless of whether or not we win the race? Do we even need to finish, if we succeed in sponsorship, technology innovation, and education? DARPA's goals and our goals are not necessarily the same. As a team, we may have significant differences in our own goals.

ARCHITECTURE DECISIONS: A SELECTIVE OUTLINE

One way to start sorting out this confusion, and something we will emphasize throughout this book, is to look at the design process as a sequence of decisions. Some are identifiable, front-and-center, from the beginning. Some are contingent, and the situation changes the decisions change. Some are stakeholder view dependent. One heuristic is that architecture decisions are the decisions that drive value, cost, and risk. One observation about early-phase architecting and engineering is that a small set of decisions drive most of the value, cost, and risk. Identifying, understanding, and controlling those decisions is a key to success, at least to the extent that success is measured by value, cost, and risk.

We would not try to do a complete analysis of architecture decisions for this case, but we will walk through a few, leading to identifying three component architecting problems in the next section.

- Should we participate? If we do participate, what are our objectives?

  If we do participate, we will expend our team's time and other resources on DARPA's challenge and not on other things. To assess if this has good return on investment for us, we need to understand our objectives. In the example of the university team, a university has a distinct mission, different from DARPA's, different from a corporation that might participate. It might be that the vehicle concept that best maximizes our objectives will not be the concept that maximizes somebody else's objective.
- Given that we participate, should we put design constraints on ourselves (in service of our objectives)? If so, what design constraints should we consider?

  For example, a team can focus on sensors and control, while constraining the vehicle to be an off-the-shelf off-road vehicle, or a team can customize the vehicle. If you have large resources and are focused on winning the race jointly optimizing the sensors, the processing, and the physical vehicle makes sense. But resources are always limited and winning might be just only one objective among many. Trying to do everything well might cost you the opportunity to do a more valuable subset very well.
- Top-level technical decisions about our vehicle and our program.

  Here we get into choosing a vehicle, choosing sensors, and choosing sensor fusion approaches. Here is also where we have to design our program for building the vehicle and consider how the vehicle design and the program design interfere with each other.

THREE ARCHITECTING PROBLEMS

To structure this case further, let us consider three early-phase decision problems about the architecture of the hypothetical vehicle/program. As an exercise, the reader should consider each of these problems in turn and try to develop arguments for and against each possible course of action. Also consider if there are other courses of action the team could take. Finally, consider if you were faced with this problem in 2003 (not 2023). How would you resolve it? When you think how to resolve it, remember that in the real situation in 2003, the race day was fixed and immovable. So, any decision to gather more data and decide later burns up some clock time, time that could have been used to do something else later in the cycle.

The three problems are as follows:

1. The vehicle selection problem
2. The sensor selection problem
3. The build-test-adjust schedule problem

The Vehicle Selection Problem

To participate in the race, you have to build a vehicle. Selection of the vehicle is obviously a key decision. Suppose in early discussions, there are two approaches identified for the vehicle.

Group 1: Off-The-Shelf Vehicle

The first group advocates for starting with an off-the-shelf vehicle, an off-road capable sport-utility vehicle or truck, minimally modified to support autonomous driving. This would be obtained from a dealer or directly from a manufacturer and then minimally modified. The only mechanical modifications to the vehicle would be those essential for autonomous operation: electro-mechanical actuation of steering, shifting, braking, and acceleration. They argue that the challenge is not in the vehicle, since a very wide variety of vehicles can easily be driven 120 miles in less than 10 hours on poor roads or even off-road. Existing vehicles have already solved the mechanical problems. Selecting an off-the-shelf vehicle would allow the team to focus solely on the sensor and control problems, perceived by this group to be the heart of the challenge. Using an off-the-shelf vehicle would take advantage of production-level reliability and allow the team to put all resources on sensor, electronic, and software specialties.

Group 2: Custom, Tailored Vehicle

The second group advocates for opening the design space to include building a vehicle that exploits not having to support a human driver to allow form factors and structures that could operate more robustly in the race environment. By opening the design space to more robust designs able to tolerate conditions, perhaps created by poor-quality autonomous driving, that would stop an off-the-shelf vehicle. More robust mechanical design could tolerate poorer sensor performance and driving control. Taking this course of action would change the team makeup, and skills and resources required, but would allow incorporation of additional areas of expertise and possible design innovations.

Assessment Task

The team has to pick a vehicle approach. If they go with Group 1's approach, they will need to pick an off-the-shelf vehicle and then design around its limitations. If sensors and controllers cannot be designed that operate within the vehicles performance envelope, they will fail. They will not need an extensive mechanical engineering or construction effort as that will be effectively outsourced to the vehicle manufacturer. If they go with Group 2's approach, they will need to build a more comprehensive development team, but they will gain a larger design tradespace. That could be a significant advantage or perhaps a disadvantage if the larger design tradespace greatly complicates the process in the limited time available.

How should the team decide between these two courses of action, especially considering the finite time available and that time spent deeply analyzing the issue is time no longer available to engineer the system? What considerations or objectives should be controlling on the decision? Are there widely accepted heuristics, guidelines, or principles that could help make this decision? What practical, very rapid (week or two), low-cost actions could they take to gather better information to improve the decision quality rather than just deciding immediately? With particular regard to Group 2's idea, if you could win the race with a mechanically innovative vehicle that tolerated very poor autonomous control, would that be a "win" for the broad objectives the stakeholders were trying to achieve?

The Sensor Selection Problem

DARPA will provide a set of GPS waypoints for the course, but it is understood that the vehicle will need some set of sensors, and the associated processing, to successfully complete the course. The requirement is to drive the course, not house a particular set of sensors. Nobody has accomplished the challenge before, so the relationship between what sensors are used and how successfully it can drive the course is unknown. There are many possible sensors that could be used, either individually or in combination:

- Video cameras, either singly or in stereo pair or in multi-stereo sets (three or more arranged to provide additional stereo). Cameras could be set up with wide or narrow fields of view, with close-in focus or for distant vision, or steerable and zoomable.
- Laser range finders or imaging laser radars (LIDAR) add direct measurement of distance to imaging capabilities.
- Microwave radars to measure distance to objects.
- Inertial sensors.
- In addition, sensors could be hard mounted to the vehicle (simple, robust, but then the sensor is exposed to more severe vibration and the sensor moves with the vehicle requiring some form of compensation in processing). Or, the sensors could be mounted on a stabilized gimbal providing vibration and shock isolation and a stable platform at the cost of much greater mechanical complexity and potential reliability issues.

Putting many sensors on the vehicle will provide a diversity of data, potentially a major advantage. But with many sensors, the complexity of sensor integration will rise, probably non-linearly in the number of sensors. Imperfectly correlated multi-sensor data can be worse than any of the sensor feeds alone. Many of the complexities in using the sensor data likely will not be understood until the team has built something and tested it in a representative environment.

How should the team choose what sensors to include in their design? What strategic considerations can be used to attack this decision problem, and how is the decision problem impacted by the team's own understanding of their objectives?

The Build-Test-Adjust Schedule Problem

The third problem to consider is not specifically technical; it is choosing the architecture of the program. At the very top level, if the team is going to enter the race, they need to design, build, test, and deliver an autonomous vehicle. They have nearly full control over the details of how they do that, at least within whatever resources they can obtain. They could spend most of the time in specifying, designing, and fabricating the vehicle and only test to verify that it does what it was intended to do. They could deliberately choose a very simple design that could be fabricated quickly, take it to the test environment very early, and concentrate on refining the implementation. Whatever they choose, they have to deliver a working vehicle on the race day roughly a year away or the whole effort will be largely in vain. There are many programmatic decisions to be made, but consider two very strategic ones:

- When in the one-year time available should they target having a working vehicle that can be tested in an operationally representative environment? If they choose an early point, say 6 months in, then they will have potentially many cycles of refinement, but the overall design will have to be simple enough, and design choices more constrained, to be available that quickly. Many design options are likely to be foreclosed simply because they cannot be ready in half of the total time. If they choose a late point for full-scale testing, then the design can be more capable, but there will be little time to refine.
- During test, and during the race, the system will have to decide how fast to go. The faster it goes, the more competitive it will be (boosting the possibility of winning), but also raising the risk of crashing. If the vehicle crashes in test, it might be so damaged as to be unrecoverable (at least in time for the race). But if they do not drive faster in test, they will not have information to support driving fast in the race. How do they resolve this conundrum of pushing risk in test to perform better in the race versus being more conservative in test to maximize the likelihood of making the race, but being less competitive?

The lack of hard requirements from DARPA throws these choices almost entirely on the race teams. There is no appeal to authority; the choices have to be based on the team's own objectives for participating in the race.

ARCHITECTURE LESSONS FROM 2004 AND 2005

The reader is encouraged to work out their own answers to the challenge questions above. The teams who built vehicles for the races in 2004 and 2005 all had to come up with their own solutions, for better or worse. Since there were two races, the first in 2004 and the second in 2005, we can observe differences in how teams approached the challenge as probably reflecting some lessons learned.

First, at the macro-level, teams clearly made choices aligned with their own goals and objectives, based in part on their pre-existing organizations. Conway's Law (Conway 1968, Bailey et al. 2013) says that organizations build technical architectures that mirror their own organizational structure, and vice versa, their organizations come to resemble the things they build. More generally, they make choices that reflect their structure. Stanford's winning vehicle was built by a computer science team. They choose an almost off-the-shelf vehicle, hard mounted the sensors, and put all of their efforts into information processing, accepting all of the constraints and limitations their simple mechanical design imposed. In contrast, the second-place team from Carnegie-Mellon had a broad pre-existing base of expertise and facilities in mechanical engineering and diverse robot construction. They put far more effort into vehicle modifications and sensor gimbal systems. Both worked (the winning margin was quite narrow) and both reflected the pre-dispositions and objectives of each team.

Taking up the questions posed above, teams in the second race mostly focused on minimally modified off-the-shelf vehicles versus the more diverse array of more customized vehicles in the first (unsuccessful) race. Teams seemed to realize that it was not a mechanical challenge getting around the course, and failing to finish because of a mechanical breakdown was a waste. By choosing off-the-shelf vehicles, each team could focus on the hard parts of the challenge, doing the observation and analysis necessary to get around the course.

Teams in both races made diverse choices in sensors. Generally, a combination of cameras and LIDAR were used, and both seemed to be needed by the most successful teams. It was clear in the second race that successful teams put a much greater emphasis on getting to the field early and often and building performance up from simple to complex incrementally. Carnegie-Mellon, who had been the top performer in the first race, nevertheless re-did integration on the 2003 vehicle nearly from scratch for the second race, while simultaneously developing a second, more sophisticated vehicle. They were driving the first vehicle on cameras alone very early in the run-up to the second race and then incrementally adding additional sensors.

The emphasis on incrementalism brings in another observation on architectures, the need to consider the true, full lifecycle not just the operational period. If a vehicle is going to be run through perhaps 6 months of testing prior to the race, and then raced, the vehicle has to work reliably for far longer than the race period. The vehicle has to be fully supportable during the months of testing. It has to be possible to gather data during testing in enough depth not just to verify that a requirement was met to know exactly what the vehicle and sensors experienced, so any issue can be diagnosed and improved. This may call for features and data recording capabilities far in excess of what is needed in the operational period (the actual race).

Consider something not-so-obvious, human driveability. Most of the vehicles in the second, more successful, race were not only nearly off-the-shelf vehicles, but they retained human driveability. People could sit in the vehicle as it drove itself and take over and drive manually if desired. In the race, this capability is useless, as the vehicle cannot be touched during the race itself. But human driveability is very useful during test. A human-driveable vehicle can be driven to the test area (instead of requiring support vehicles and staff as would a pure robot). Humans can observe what it does in real time. The humans can drive the vehicle out of non-pre-planned places, or into non-pre-planned places, if appropriate. A human can drive the vehicle with the sensors in record mode, at various designated speeds, and the resulting data set is examined later. Human driveability opens up a range of operating capabilities during development that would not otherwise be available. If we consider the development period as an inherent part of the lifecycle, we may architect for it in response. If we focus solely on the operational period (the race) and neglect to consider the development period as part of the lifecycle, we will miss this.

As an aside, we should also remember that if we intend for humans to be in the vehicle during development during test, then the humans are exposed to the risks of the vehicle. That may drive additional design and operational considerations that might be at odds with the advantages discussed above. Each architectural choice generates a series of consequences, perhaps crossing from technical to programmatic to ethical considerations. Architecting is complex, we cannot escape that. Building something new is complex, architecting is a response.

Keep the DARPA Grand Challenge case in mind as we define and elaborate systems architecting in the chapters to come. The clarifying nature of the case is how the system development teams were forced, by the nature of the situation, to confront the full set of architecting challenges. In most cases, there are more constraints, which are responsible for some architectural decisions that may be obscured. We have to deal with that case-by-case basis, but this case study provides clarity on how architecting is fundamentally decision-centric, and the broad nature of those decisions.

REFERENCES

Bailey, S. E.,et al. (2013). A decade of Conway's law: A literature review from 2003–2012. *2013 3rd International Workshop on Replication in Empirical Software Engineering Research*, New York: IEEE.

Behringer, R., et al. (2004). The DARPA grand challenge-development of an autonomous vehicle. *IEEE Intelligent Vehicles Symposium*, New York: IEEE.

Conway, M. E. (1968). How do committees invent? *Datamation* **39**(12): 28–31.

NOVA (2006). "The Great Robot Race." from https: //www.pbs.org/wgbh/nova/darpa/.

Ozguner, U., K. A. Redmill, and A. Broggi (2004). Team Terra Max and the DARPA grand challenge: a general overview. *IEEE Intelligent Vehicles Symposium*, New York: IEEE.

SAE (2024). "Formula SAE." fromhttps: //www.fsaeonline.com/.

Seetharaman, G., A. Lakhotia, and E. P. Blasch (2006). "Unmanned vehicles come of age: The DARPA grand challenge." *Computer* **39**(12): 26–29.

Thrun, S., et al. (2006). "Stanley: The robot that won the DARPA grand challenge." *Journal of Field Robotics* **23**(9): 661–692.

1 Extending the Architecting Paradigm

WHAT IS ARCHITECTING?

Whenever we start to build a system, especially when that system is supposed to be something essentially new, somebody faces a poorly structured problem. Somebody is faced with simultaneously having to work through "what do we want?" and "what can we have?" realizing that they aren't independent questions. They face decisions in multiple dimensions. How do I choose a trade space? Too wide and I may never make progress. Too narrow and I may be writing off the most valuable solutions. They face structuring the program to build the system, whether all at once or incrementally. It is not enough to just go off and ask somebody what the requirements are, the requirements themselves will be part of the trade space.

The opening DARPA Grand Challenge case study shows this. Every team who considered building a vehicle faced an obvious problem: Build a vehicle that (would hopefully) win the designated race, with all the uncertainty and ambiguity of just what that race would eventually require and what the competition would consist of. But beyond the obvious problem lay the real problem. Each group had to understand for itself what they wanted to achieve, and how gathering and spending the required scarce resources (people's time, money, and facilities) would lead to their goals. It is very unlikely that any team who participated in either of the two races was solely motivated by the possibility of the win and the prize money. Everybody involved saw participation as a path to their own goals, and different groups did not have identical goals.

We call this situation at the beginning of a system development an architecting situation. We use the term with inspiration from the classical model of how an architect works, and how the practice of architecture of building has been distinguished from the practice of civil engineering.

- The architect works for clients who want a building and whose needs and goals span as wide a range of concerns as suits them. They may want specific physical properties, practical functional characteristics, aesthetics, spiritual inspiration, or embodiment of a business brand.
- Both the problem and the solution are in the trade space. Quite likely, there is a higher level, like the DARPA program office in the case study, who have constrained the problem and solution space, but there are many degrees of freedom for the team.
- The architect's job is not to design the building to the construction drawing level (at least it usually is not). The architect's job is to come up with multiple

DOI: 10.1201/9781003483595-3

alternatives and eventually produce a partial design, a partial design that will be further refined by others to the level of detailed construction plans. This partial design should encompass the decisions that determine most of the cost, value, and risk.

- The architect's products are both decisions and descriptions. The decisions are those that determine most of the value, cost, and risk. The descriptions are the physical products, whether drawings, models, or other digital objects, that document and define the decisions.
- The architect acts as the agent of the clients, primarily involved in the beginning phases (when essential decisions about what to build and how to build it are made) and again toward the end of the process when the clients have to make the determination of "is the system suitable for use?"

The historical and traditional practice and profession of architecture embraces these factors. Because it embraces these factors, we look to their practices for inspiration and hints, even if the development of autonomous vehicles or spacecraft contains many different elements than civil construction. The recorded history of classical architecting, the process of creating architectures, began in Egypt more than 4,000 years ago with the pyramids, the complexity of which had been overwhelming designers and builders alike. As systems became increasingly more ambitious, the number of interrelationships among the elements increased far faster than the number of elements. These relationships were not solely technical. Pyramids were no longer simple burial sites; they had to be demonstrations of political and religious power, secure repositories of god-like rulers and their wealth, and impressive engineering accomplishments. Each demand, in itself, would require major resources. When taken together, they generated new levels of technical, financial, political, and social complications. Complex interrelationships among the combined elements were well beyond what the engineers' and builders' tools could handle.

From that lack of tools for civil works came classical or civil architecture. Millennia later, technological advances in shipbuilding created the new and complementary fields of marine engineering and naval architecture. As the industrial revolution took hold in the 19th century and was then compounded by the science-engineering breakthroughs of the 20th century, the situation repeated again and again. Whether in military systems, transportation, communications, computing, aircraft, or other fields, the same development quandary repeated. New technical capabilities changed our understanding of what problem we wanted to solve, our changed understanding led to new operational or business concepts, which in turn demanded different technical capabilities. The value in what we built was primarily not in the individual component (even when enabled for the newest technology of the day) but in how those components combined to produce a whole whose capabilities enabled something new. Each iteration increased complexity in component count, interrelationships, and scope of consideration. Engineering tools were typically lagging at the forefront of system development.

One factor that stands out is the utility of the classical architecting paradigm. If we are to use the classical paradigm, we must understand it as it does not apply without adaptation.

THE CLASSICAL PARADIGM AND ITS EXTENSIONS

The classical paradigm applies most strongly when six factors are present (Rechtin 1994). The architectural paradigm applies more broadly, as we will see through extensions, as the six factors vary, but these are the foundation.

1. **A Systems Approach:** We are moving between problem space and solution space. Both sides are in play. We expect our understanding of the problem to flex as part of the process, and that stakeholders may rethink their needs (and operations) in light of solutions. The value is, on the whole, in the capabilities generated.
2. **Purpose Orientation:** The effort is driven by client needs. An important extension is architecting in the technology-driven scenario, but the foundational case assumes the process is driven by client needs.
3. **We Work with Models:** Our system is complex enough that architecting has to be conducted on models or abstractions. The architect does not work with the physical system itself.
4. **Ultraquality:** The system is unique or produced in numbers much lower than the inverse of the failure rate.
5. **Certification:** Again, the numbers are low, so we have to accept systems as fit for use on an individual basis. We are operating outside performance ranges where statistics are an expansive guide.
6. **Complexity Pushing us into the Need for Insight:** Complexity is high enough that analytical results can only be accepted or rejected on a judgment basis. Insight beyond straight analysis is required. Some quality factors demanded are not subject to quantitative analysis.

Complexity in Problem and Solution

Complexity is commonly cited as the root difficulty in system development today. When architects and builders are asked to explain cost overruns and schedule delays, by far the most common, and quite valid, explanation is that the system is much more complex than originally thought. Exactly what this means may not be explained. It may mean we didn't realize the difficulty of what we set out to do. The difficulties may have been magnified by how interconnected parts of the system turned out to be, how interconnected parts of the problem were, or both to each other. It may mean we misunderstood the magnitude of what we had to do in terms of the number of parts, stakeholders, or concerns.

There is a large literature on complexity in general, on complex systems specifically, on complex or "wicked" problems. That literature recognizes complexity as a mixture of size (number of elements), interconnectedness, interdependency, and irreducibility (difficulty in understanding the whole from the parts). The emergent properties of a system, a fundamental identifier of a system, are more and more removed from the element properties as the system is more complex. The more elements and interconnections, the more complex the architecture and the more difficult the system-level problems.

TABLE 1.1
Dimensions of Development Complexity

Dimension	Simple		Complex	
Sponsors	One, with money	Several, with money	One, without money	Many, speculative
Users	Same as sponsors	Aligned with sponsor	Distinct from sponsor	Unknown, conflicting
Situation-Objectives	Tame	Discoverable	Ill-structured	Wicked
Quality Required	Directly measurable	Indirect measures	Semi-measurable	One-shot and unstable
Control	Centralized with sponsor	Centralized elsewhere	Distributed	Virtual
Feasibility	Easy	Barely		No
Development scope	Few elements, few interface	Intermediate		Many elements, many interfaces
Technology Maturity	Very high	Medium	Low	Very low

For our purposes, we need a working understanding of what complexity means in the context of developing a system. This can be developing a new system from scratch, as in our DARPA Grand Challenge case example, or evolving an existing system in response to changing needs. We have found a convenient framework with eight dimensions that helps characterize development complexity and also illustrates key distinctions between architecting and engineering practices. The framework is shown in Table 1.1.

The first three rows are primarily problem-space factors, and they are built on what we are trying to achieve. The last three factors are primarily solution space, and they depend on the nature of the thing we want to build. The middle two bridge the problem and solution spaces.

Sponsors: Who wants the system will pay for it and is sponsoring the architecting-design effort. A single sponsor with full funding is the simplest; many sponsors (perhaps not known) without funding are the most difficult.

Users: Who will use any system ultimately developed? If they are also the sponsors, this is simplest. If they are unknown or conflicting with the sponsors, this is most difficult.

Situation-Objectives: A "Tame" situation is where the sponsors know and can authoritatively state the objectives/requirements, and they won't change. In a discoverable situation, they can be known with enough inquiry but are not available up-front. Ill-structured means they are likely to change when solutions are available. The best definition of "wicked" is that the rate of change of the objectives is faster than any possible development timeline, and different stakeholders with authority want incompatible things.

Quality Required: In a simple development, all desired quality factors can be directly measured, ideally from the design as well as from the actual system. Greater complexity is marked by the inability to directly measure the desired quality factors except in operation (when it may be too late), and by cases where the operation is one time and so we cannot know if the system works as intended until all opportunities to do anything about it have passed.

Control: This refers to how the system is controlled, and by whom, in operation. In the simple case, control is centralized, usually with the sponsors. In the most complex cases, the system is uncontrolled, and it exists in the environment under the virtual control of users and others without any central direction.

Feasibility: Sponsors and users have some base expectations of what the system should do. Feasibility measures the trade space we have for meeting basic expectations. If there are many concepts meeting basic expectations, the situation is simpler than if there are none.

Development Scope: This is the typical notion of complexity as size. Things with more parts, more interfaces, and more inherently dependent interfaces are more complex and thus more difficult.

Technology Maturity: This assesses the availability of component parts needed to meet basic expectations. The simple case is when an acceptable system can be built from currently in-production parts all known to work in the relevant environment. The most difficult is where some components are unavailable, even at laboratory maturity.

Complexity is a core driver of the methods needed to architect and design. Qualitatively different problem-solving techniques are required at high levels of complexity than at low ones. Purely analytical techniques, powerful for the lower complexity levels, can be overwhelmed at the higher ones. At higher levels, architecting methods, experience-based heuristics, abstraction, and integrated modeling must be called into play. More broadly than just techniques, considering multiple dimensions of complexity challenges us to simplify in any of the dimensions, not just scope. Consolidate and simplify the objectives. Focus on the things driving one or more dimensions of complexity and eliminate them. Put to one side minor issues likely to be resolved by the resolution of major ones. Draw in the boundaries of control. Model (abstract) the system at as high a level as possible and then progressively reduce the level of abstraction. A problem that is maximally complex in one dimension can be addressed by methods tuned to that dimension. Something maximally complex in every dimension is a lost cause.

The High Rate of Advances in the Computer and Information Sciences

An additional factor is the unprecedented rate of technological advances in the computational and information sciences. Software has been driving a true paradigm shift in system design for at least two decades, but the process of change continues. Software had been treated as the glue that tied hardware elements together but has become the center of system design and operation. We see it in consumer electronics, vehicles, spacecraft, and military systems. The precipitous drop in hardware costs has generated a major design shift—from "keep the computer busy" to "keep the user busy." Designers happily expend hardware resources to save redesigning either hardware or software. In automobiles, software increasingly determines the performance, quality, cost, and feel of cars and trucks. We see it in aircraft, where controls are coming to drive aerodynamic and structural design, and military system designers discuss a shift to designing the airframe around the sensors instead of designing the sensors around the airframe. Software has been the principal enabler of the movement to unoccupied (drone) aircraft systems.

One measure of this phenomenon is the proportion of development effort devoted to hardware and software for various classes of products. Anecdotal reports from a variety of firms in telecommunications and consumer electronics commonly show a reversal of the proportion from 70% hardware and 30% software common a few decades ago to 30% hardware and 70% software. This shift has created major challenges and destroyed some previously successful companies. When the cost of software development dominates total development, systems should be organized to simplify software development. But good software architectures and good hardware architectures are often quite different. Good architectures for complex software usually emphasize layered structures that cross many physically distinct hardware entities. Good software architectures also emphasize information hiding and close parallels between implementation constructs and domain concepts at the upper layers. These are in contrast to the emphasis on hierarchical decomposition, physical locality of communication, and interface transparency in good hardware architectures. Organizations find trouble when their workload moves from hardware to software-dominated, but their management and development skills no longer "fit" the systems they should support.

Whatever the source of the challenge, the approach based on the architecting paradigm has six parts, summarized earlier. We elaborate on these six elements here.

A Systems Approach

To take a systems approach means to focus on the system as a whole, specifically linking value judgments (what is desired, the problem domain) and design decisions (what is feasible to build, the solution domain). A systems approach means that the design process includes the "problem" as well as the solution. The architect seeks a joint problem–solution pair and understands that the problem statement is not fixed when the architectural process starts. At the most fundamental level, *systems are collections of different things that together produce results unachievable by the elements alone*. For example, only when all elements are connected and working together do automobiles produce transportation, human organs produce life, and spacecraft produce information. These system-produced results, or "system functions," derive almost solely from the interrelationships among the elements, a fact that largely determines the technical role and principal responsibilities of the systems architect. But here, it is not just that the problem domain is included in the process; it is that we shape the problem domain and what we address in the problem space as part of the system-level trades.

Systems are interesting because they achieve results, and achieving those results requires different things to interact. From much experience with it over the last decade, it is difficult to underestimate the importance of this specific definition of systems to what follows, literally on a word-by-word basis. Taking a systems approach means paying close attention to results, the reasons we build a system. Classically, architecture *must* be grounded in the client's/user's/customer's purpose. But a full understanding of that purpose is rarely captured in any initial problem statement or any recitation of "requirements." Architecture is not just about the structure of components. One of the essential distinguishing features of architectural design versus other sorts of engineering design is the degree to which architectural design embraces results from the perspective of the client/user/customer. The architect does not assume some particular problem formulation, as "requirements" are fixed. The architect engages in joint exploration, ideally directly with the client/user/customer, of what system attributes will yield results worth paying for.

It is the responsibility of the architect to know and concentrate on the critical few details and interfaces that really matter and not to become overloaded with the rest. Recall the distinctions made in Table P.1 of the Preface. It is a responsibility that is important not only for the architect personally but for effective relationships with the client and builder. To the extent that the architect must be concerned with component design and construction, it is those specific details that critically affect the system as a whole.

For example, a loaded question often posed by builders, project managers, and architecting students is, "How deeply should the architect delve into each discipline and each subsystem?" A graphic answer to that question is shown in Figure 1.1, exactly as sketched by Bob Spinrad in a 1987 lecture at the University of Southern California. The vertical axis is a relative measure of how deep into a discipline or subsystem an architect must delve to understand its consequences to the system as a

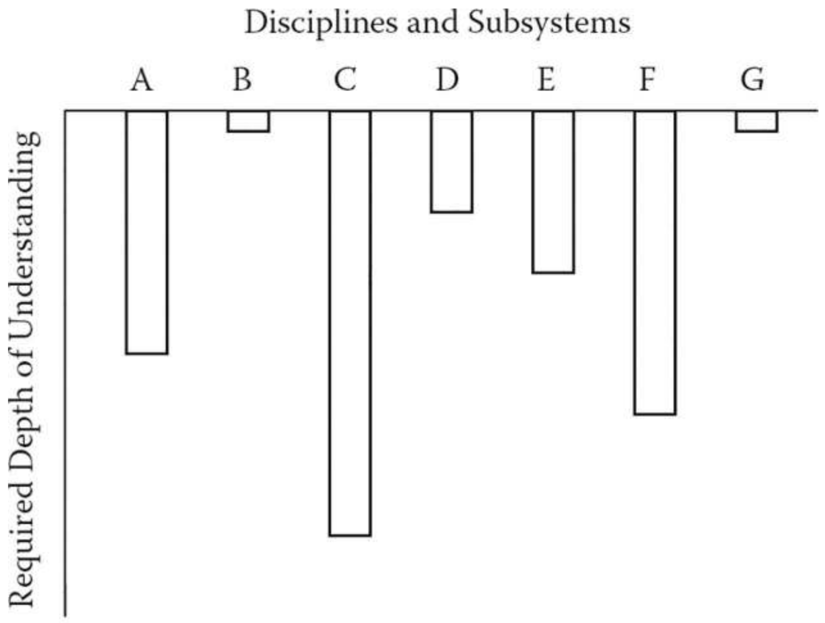


FIGURE 1.1 The architect's depth of understanding of subsystem and disciplinary details.

whole. The horizontal axis lists the disciplines, such as electronics or stress mechanics, and the subsystems, such as computers or propulsion systems. Depending upon the specific system under consideration, a great deal, or a very little depth, of understanding may be necessary.

But that leads to another question, "How can the architect possibly know before there is a detailed system design, much less before system test, what details of what subsystem are critical?" A quick answer is: only through experience; through encouraging open dialog with subsystem specialists; and by being a quick, selective, tactful, and effective student of the system and its needs. Consequently, and perhaps more than any other specialization, architecting is a continuing, day-to-day learning process. No two systems are exactly alike. Some will be unprecedented, never built before.

Exercise: Put yourself in the position of an architect asked to help a client build a system of a new type whose general nature you understand (a house, a spacecraft, a nuclear power plant, or a system in your own field) but which must considerably outperform an earlier version by a competitor. What do you expect to be the critical elements and details and in what disciplines or subsystems? What elements do you think you can safely leave to others? What do you need to learn the most about? Reminder: You will still be expected to be responsible for all aspects of the system design.

Critical details aside, the architect's greatest concerns and leverage are still, and should be, with the systems' connections and interfaces: First, because they distinguish a system from its components; second, because their addition produces unique system-level functions, a primary interest of the systems architect; and third, because subsystem specialists are likely to concentrate most on the core and least on the periphery of their subsystems, viewing the latter as (generally welcomed) external constraints on their internal design. Their concern for the system as a whole is understandably less than that of the systems architect; if not managed well, the system functions can be in jeopardy.

A Purpose Orientation

Classically, systems architecting is a process driven by a client's purpose or purposes. A purpose-driven system starts with a sponsor willing to pay and users prepared to use. A president wants to meet an international challenge by safely sending astronauts to the moon and back. Military services need nearly undetectable strike aircraft. Airlines need an aircraft that can operate profitably on identified routes not yet served.

Clearly, if a system is to succeed, it must satisfy a useful purpose at an affordable cost for an acceptable period of time. Note the explicit value judgments in these criteria: A *useful* purpose, an *affordable* cost, and an *acceptable* period of time. Everyone is the client's prerogative and responsibility, emphasizing the criticality of client participation in all phases of system acquisition. But of the three criteria, satisfying a *useful* purpose is predominant. Without it being satisfied, all others are irrelevant. Architecting therefore begins with, and is responsible for maintaining, the integrity of the system's utility or purpose.

For example, Apollo's manned mission to the moon and back had a clear purpose, an agreed cost, and a no-later-than date. It delivered on all three. Those requirements, kept up front in every design decision, determined the mission profile of using an orbiter around the moon and not an earth-orbiting space station, and on developing electronics for a lunar orbit rendezvous instead of developing an outsize propulsion system for a direct approach to the lunar surface.

As another example, NASA Headquarters, on request, gave the NASA/JPL Deep Space Network's huge ground antennas a clear set of priorities: First performance, then cost, then schedule, even though the primary missions they supported were locked into the absolute timing of planetary arrivals. As a result, the first planetary communication systems were designed with an alternate mode of operation in case the antennas were not yet ready. As it turned out, and as a direct result of the NASA risk-taking decision, the antennas were carefully designed, not rushed, and satisfied all criteria not only for the first launch but for all launches for the next 40 years or so.

The Douglas Aircraft DC-3, though originally thought by the airline (later TWA) to require three engines, was rethought by the client and the designers in terms of its underlying purpose—to make a profit by providing affordable long-distance air travel over the Rocky and Sierra Nevada mountains for paying commercial passengers. The result was the two-engine DC-3, the plane that introduced global air travel to the world.

When a system fails to achieve a useful purpose, it is doomed. When it achieves some purpose but at an unfavorable cost, its survival is in doubt, but it may survive. The purpose for which the space shuttle was conceived and sold, low-cost transport to low earth orbit was never achieved. However, its status as the sole U.S. source of manned space launch allowed its survival. The space shuttle was a tremendous technical achievement. The success of architecting is not measured by technical success but by success in mission. In a similar fashion, it has proven impossible to meet the original purpose of the space station at an acceptable cost, but its role in the U.S. manned space program and international space diplomacy has assured minimum survival. In contrast, the unacceptable cost/benefit ratios of the supersonic transport, the space-based ballistic missile defense system, and the superconducting supercollider terminated all these projects before their completion.

A purpose-driven system may be successful but evolve away from its original purpose. The F-16 fighter aircraft was designed for visual air-to-air combat but has been mostly used for ground support. The ARPANET communication network originated as a government-furnished computer-to-computer linkage in support of university research but became the foundation for the global Internet. Both are judged as successful. Why? Because, as circumstances changed, providers and users redefined the meaning of useful, affordable, and acceptable. A useful heuristic comes to mind: *Design the structure with "good bones."* It comes from the architecting of buildings, bridges, and ships, where it refers to structures that are resilient to a wide range of stresses and changes in purpose. It could just as well come from physiology and the remarkably adaptable spinal column and appendages of all vertebrates—fishes, amphibians, reptiles, birds, and mammals.

Exercise: Identify a system whose purpose is clear and unmistakable. Identify, contact, and if possible, visit its architect. Compare notes and document what you learned.

Beyond the classical architecting paradigm, not all systems are purpose-driven. Systems may be sponsored on a technology vision, lacking committed users (Chapter 3). Purpose may evolve, or even emerge, collaboratively without central sponsorship (Chapter 7). These break the classical paradigm, but the essential echo of that paradigm remains. The original motivation for these systems may be technology capabilities and a vision of what that can do, but success only comes when a sufficient purpose is found and exploited.

A Modeling Methodology

Modeling is the creation of abstractions or representations of the system to predict and analyze performance, costs, schedules, and risks and to provide guidelines for systems research, development, design, manufacture, and management. Modeling is the centerpiece of systems architecting—a mechanism of communication to clients and builders, of design management with engineers and designers, of maintaining system integrity with project management, and of learning for the architect, personally.

Examples: The balsa wood and paper scale models of a residence, the full-scale mockup of a lunar lander, the rapid prototype of a software application, the computer model of a communication network, or the mental model of a user.

Modeling is of such importance to architecting that it is the sole subject of Part III. Modeling is the fabric of architecting because architecting is at a considerable distance of abstraction from actual construction. The architect does not manipulate the actual elements of construction. The architect builds models that are passed into more detailed design processes (the left to right progression in Table P.1 of the Preface). Those processes lead, eventually, to construction drawings or the equivalent and actual system fabrication or coding.

Viewing architecting and design as a continuum of modeling refinement leads naturally to the "stopping question." Where does architecting stop and engineering or design begin? Or, when should we stop any design activity and move onto the next stage? From a modeling perspective, there is no stopping. Rather modeling is seen to progress and evolve, continually solving problems from the beginning of a system's acquisition to its final retirement. There are of course conceptual models, but there are also engineering models and subsystem models; models for simulation, prototypes, and system test; demonstration models, operational models, and mental models by the user of how the system behaves. From another perspective, careful examination of the "stopping question" leads us to a better understanding of the purpose of any particular architecting or design phase. Logically, they stop when their purpose is fulfilled.

Models are in fact created by many participants, not just by architects. These models must somehow be made consistent with overall system imperatives. It is particularly important that they be consistent with the architect's system model, a model that evolves, becoming more and more concrete and specific as the system is built. It provides a standard against which consistency can be maintained and is a powerful tool in maintaining the larger objective of system integrity. And finally, when the system is operational and a deficiency or failure appears, a model—or full-scale simulator if one exists—is brought into play to help determine the causes and cures of the problem. The more complete the model, the more accurately possible failure mechanisms can be duplicated until the only cause is identified.

Ultraquality Implementation

Ultraquality is defined as a level of quality so demanding that it is impractical to directly measure its achievement. It is a limiting case of quality driven to an extreme, a state beyond acceptable quality limits (AQLs) and statistical quality control. It requires a zero-defect approach not only to manufacturing but also to design, engineering, assembly, test, operation, maintenance, adaptation, and retirement—in effect, the complete life cycle. The concept is discussed in the literature (Juran 1988, Phadke 1995) and Chapter 8 of Rechtin (1991).

Some examples include a new-technology spacecraft with a design lifetime of at least 10 years, a nuclear power plant that will not fail catastrophically within the foreseeable future, an aircraft flight control computer system whose meantime between failures is required to be longer than the collective operating time of all of its copies (Lala and Harper 1994), and a communication network of millions of nodes, each requiring almost 100% availability. In each case, the desired level of quality cannot, even in principle, be directly measured; only the absence of the quality desired can be directly measured. Ultraquality is a recognition that the more components there are in a system, the more reliable each component must be to a point where, at the element level, defects become impractical to measure within the time and resources available. Or, in a variation on the same theme, the operational environment cannot be created during test at a level or for a duration that allows measurement at the system level. Yet, the reliability goal of the system as a whole must still be met. In effect, it reflects the reasonable demand that a system—regardless of size or complexity—should not fail to perform more than about 1% or less of the time. An intercontinental ballistic missile (ICBM) should not. A space shuttle, at least 100 times more complex, should not. An automobile should not. A passenger airliner, at least 100 times more complex, should not; as a matter of fact, we expect the airliner to fail far, far less than the family car.

> **Exercise:** Trace the histories of commercial aircraft and passenger buses over the last 50 years in terms of the number of trips that a passenger would expect to make without an accident. What does that mean to vehicle reliability as trips lengthen and become more frequent, as vehicles get larger, faster, and more complex? How were today's results achieved? What trends do you expect in the future? Did more software help or hinder vehicle safety?

The subject would be moot if it were not for the implications of this "limit state" of zero defects to design. Zero defects as a philosophy originated as long ago as World War II, largely driven by patriotism. As a motivator, the zero defects principle was a prime reason for the success of the Apollo mission to the moon.

Successfully achieving ultraquality has its own architectural implications (Lala and Harper 1994). If a manufacturing line operated with zero defects, there would be no need, indeed it would be worthless, to build elaborate instrumentation and information-processing support systems. This would reduce costs and time by a very significant amount, perhaps 30%. If an automobile had virtually no design or production defects, then sales outlets would have much less need for large service shops with their high capital and labor costs. That would further reduce enterprise costs (though it also drastically disrupts the economics of the dealer system). Extremely high quality levels are structurally or "architecturally" disruptive.

As another example, microprocessor design and development has maintained the same per-chip defect rate even as the number and complexity of operations increased by factors of thousands. The corresponding failure rate per individual operation is now so low as to be almost unmeasurable. Indeed, for personal computer applications, a microprocessor hardware failure more than once a year is already unacceptable.

Demonstrating this limit state in high quality is not a simple extension of existing quality measures, though the latter may be necessary in order to get within range of it. In the latter, there is a heuristic: [Measurable] *acceptance tests must be both complete and passable*. How then can inherently unmeasurable ultraquality be demanded or certified? The answer is a mixture of analytical and heuristic approaches, forming a set of surrogate procedures, such as zero defects programs. Measurements play an important role but are always indirect because of the immeasurability of the core quality factors of interest.

If ultraquality is not directly measurable, how can it be achieved? Given its importance, and role in key systems, there is a considerable body of literature. Some are devoted to analytical methods, analysis methods that work around the problems in direct measurements. Others are observational resulting eventually in heuristics. These heuristics (discussed originally in Chapter 8 of Rechtin (1991)) range from approaches to system design to guidance for the organization trying to achieve ultraquality.

When discussing ultraquality, it may seem odd to be discussing heuristics. After all, is not something as technologically demanding as quality beyond measure, the performance of things like heavy space boosters, not the domain of rigorous, mathematical engineering? In part, of course, it is. But experience has shown that rigorous engineering is not enough to achieve ultraquality systems. Ultraquality is achieved by a mixture of analytical and heuristic methods. The analytical side is represented by detailed failure analysis and even the employment of proof techniques in system design. In some cases, these very rigorous techniques have been essential in allowing certain types of ultraquality systems to be architected.

Flight computers are a good example of the mixture of analytical and heuristic considerations in ultraquality systems. Flight control computers for statically unstable aircraft are typically required to have a mean time between failures (where a failure is one that produces incorrect flight control commands) on the order of 10 billion hours. This is clearly an ultraquality requirement because the entire production run of a given type of flight computer will not collectively run for 10 billion hours during its operational lifetime. The requirement certainly cannot be proved by measurement and analysis. Nevertheless, aircraft administration authorities require that such a reliability requirement be certified.

Achieving the required reliability would seem to necessitate a redundant computer design as individual parts cannot reach that reliability level. The problem with redundant designs is that introducing redundancy also introduces new parts and functions, specifically the mechanisms that manage the redundancy, and must lock out the signals from redundant sections that have failed. For example, in a triple redundant system, the redundant components must be voted to take the majority position (locking out a presumptive single failure). The redundancy management components are subject to failure, and it is possible that a redundant system is actually more likely to fail than one without redundancy. Further, "fault tolerance" depends upon the fault to be tolerated. Tolerating mechanical failure is of limited value if the fault is human error.

Creating redundant computers has been greatly helped by better analysis techniques. There are proof techniques that allow pruning of the unworkable failure trees by assuming "Byzantine" failure[1] models. These techniques allow strong statements to be made about the redundancy properties of designs. The heuristic part is trying to verify the absence of "common-mode-failures," or failures in which several redundant and supposedly independent components fail at the same time for the same reason.

The Ariane 5 space launch vehicle was destroyed on its initial flight in a classic common mode failure. The software on the primary flight control computer caused the computer to crash shortly after launch. The dual redundant system then switched to the backup flight control computer, which had failed as well moments before for exactly the same reason that the primary computer failed. Ironically, the software failure was due to code leftover from the Ariane 4 that was not actually necessary for the phase of flight in which it was operating. Arguably, in the case of the Ariane 5, more rigorous proof-based techniques of the mixed software and systems design might have found and eliminated the primary failure. But, the failure is a classic example of a "common mode failure," where redundant systems are simultaneously carried away by the same reason. Greater rigor in tracing how an implemented system meets the assumptions it was built to can never eliminate the failures that are inherent in the original assumptions.

Thus, the analytical side is not enough for ultraquality. The best analysis of failure probabilities and redundancy can only verify that the system as built agrees with the model analyzed and that the model possesses desired properties. It cannot verify that the model corresponds to reality. Well-designed ultraquality systems fail, but they typically fail for reasons not anticipatable in the reliability model.

Certification

Certification is a formal statement to the client or user that the system, as built, meets the criteria both for client acceptance and for builder receipt of payment; that is, it is ready for use (to fulfill its purposes). Certification is the grade on the "final exams" of system test and evaluation. To be accepted, it must be well supported, objective, and fair to client and builder alike.

Exercise: Pick a system for which the purposes are reasonably clear. What tests would you, as a client, demand be passed for you to accept and pay for the system? What tests would you, as a builder, contract to pass in order to be paid? Whose word would each of you accept that the tests had or had not been passed? When should such questions be posed? (Hopefully, quite early, before the basic concept has been decided upon!)

Clearly, if certification is to be unchallenged, then there must be no perception of conflict of interest by whoever does the certification. The certifier must also have deep understanding of the purposes of the client and user, not just formal acquisition requirements. The builder of the system is obviously conflicted. A third party new to the process will lack the immersion in the problem domain required. This leads us to the architect, assuming the architect was in a third-party relationship between the client and the builder.

The no-conflict imperative has led to three widely accepted, professionally understood, constraints on the role of the architect (Cantry 1963):

1. **A Disciplined Avoidance of Value Judgments:** That is, intruding in questions of worth to the client; questions of what is satisfactory, what is acceptable, affordable, maintainable, reliable, and so on. Those judgments are the imperatives, rights, and responsibilities of the client. As a matter of principle, the client should judge on desirability and the architect should decide (only) on feasibility. To a client's question of "What would you do in my position?" the experienced architect responds only with further questions until the client can answer the original one. To do otherwise makes the architect an advocate and, in some sense, the "owner" of the end system, preempting the rights and responsibilities of the client. It may make the architect famous, but the client will feel used. Residences, satellites, and personal computers have all suffered from such preemption (Frank Lloyd Wright houses, early attempts at low earth-orbiting satellite constellations, and the Lisa computer, respectively).[2]
2. **A Clear Avoidance of Perceived Conflict of Interest:** Through participation in research and development, including ownership or participation in organizations that can be, or are, building the system. The most evident conflict here is the architect recommending a system element that the architect will supply and profit from. This constraint is particularly important in builder-architected systems (Chapter 3).[3]
3. **An Arms-Length Relationship with Project Management:** That is, with the management of human and financial resources other than of the architect's own staff. The primary reason for this arrangement is the overload and distraction of the architect created by the time-consuming responsibilities of project management. A second conflict, similar to that of participating in research and development, is created whenever architects give project work to themselves. If clients, for reasons of their own, nonetheless ask the architect to provide project management, it should be considered as a separate contract for a different task requiring different resources.

The first imperative, the avoidance of value judgments, is often fraught. The architect's role is as an expert. Including the concept of operation within the scope of architecture, for all of the reasons already given, puts the architect within the client's problem domain. The client is almost certain to want the architect's view of what should be done in the problem space as well as the solution space. This puts the architect in the mode of making value judgments, or at least recommending them. When dealing with unprecedented and very innovative systems somebody will have to engage in extensive program advocacy, it is very unlikely that the client comes into the process with all the necessary resources already in place. Program advocacy means advocating for value judgments.

There is no easy way to resolve this quandary. Architects and clients have to realize the social trades involved. Avoiding value judgments by the architect carries important advantages downstream in certification but complexities in the front end. Only the architect and client can judge which drawbacks can be lived with best.

Insight and Heuristics

A picture is worth a thousand words.

Chinese Proverb, 1000 B.C.

One insight is worth a thousand analyses.

Charles Sooter, April 1993

Insight, or the ability to structure a complex situation in a way that greatly increases understanding of it, is strongly guided by lessons learned from one's own or others' experiences and observations. Given enough lessons, their meaning can be codified into succinct expressions called "heuristics," a Greek term for guide. Heuristics are an essential complement to analytics, particularly in situations where analysis alone cannot provide either insights or guidelines. In many ways, they resemble what are called principles in other arts; for example, the importance of balance and proportion in a painting, a musical composition, or the ensemble of a string quartet. Whether as heuristics or principles, they encapsulate the insights that have to be attained and practiced before a masterwork can be achieved.

Both architecting and the fine arts clearly require insight and inspiration as well as extraordinary skill to reach the highest levels of achievement. Seen from this perspective, the best systems architects are indeed artists in what they do. Some are even artists in their own right. Renaissance architects like Michaelangelo and Leonardo da Vinci were also consummate artists. They not only designed cathedrals, they executed the magnificent paintings in them. The finest engineers and architects, past and present, are often musicians; Simon Ramo and Ivan Getting, famous in the missile and space field, and, respectively, violinist and pianist, are modern-day examples.

The wisdom that distinguishes the great architect from the rest is the insight and the inspiration that combined with well-chosen methods and guidelines and fortunate circumstances, creates masterworks. Unfortunately, wisdom does not come easily. As one conundrum puts it:

- Success comes from wisdom.
- Wisdom comes from experience.
- Experience comes from mistakes.

Therefore, because success comes only after many mistakes, something few clients would willingly support, one might think it is either unlikely or must follow a series of disasters.

This reasoning might well apply to an individual. But applied to the profession as a whole, it clearly does not. The required mistakes and experience and wisdom gained from them can be those of one's predecessors, not necessarily one's own. Organizations that care about successful architecting consider designing their program portfolios to generate experience. When staged experience is understood as important, staged experience can be designed into an organization.

And from that understanding comes the role of education. It is the place of education to research, document, organize, codify, and teach those lessons so that the mistakes need not be repeated as a prerequisite for future success. Chapter 2 is a start in that direction for the art of systems architecting.

THE ARCHITECTURE PARADIGM SUMMARIZED

This book uses the terms *architect, architecture,* and *architecting* with full consciousness of the "baggage" that comes with their use. Civil architecture is a well-established profession with its own professional societies, training programs, licensure, and legal status. Systems architecting borrows from its basic attributes:

1. The architect is principally an agent of the client, not the builder. Whatever organization the architect is employed by, the architect must act in the best interests of the client for whom the system is being developed. When the architect is employed by the builder's or the client's organization this requires some means of managing for independence.
2. The architect works jointly with the client and builder on problem and solution definition. System requirements are an output of architecting, not really an input. Of course, the client will provide the architect some requirements, but the architect is expected to jointly help the client determine the ultimate requirements to be used in acquiring the system. An architect who needs complete and consistent requirements to begin work, though perhaps a brilliant builder, is not an architect, or at least not one worth paying.
3. The architect's product, or "deliverable," is an architecture representation, a set of abstracted designs of the system. The designs are not (usually) ready to use to build something. They have to be refined, just as the civil architect's floor plans, elevations, and other drawings must be refined into construction drawings.
4. The architect's product is not just physical representation. As an example, the civil architect's client certainly expects a "ballpark" cost estimate as part of any architecture feasibility question. So, too, in systems architecting, where an adequate system architecture description must cover whatever aspects of physical structure, behavior, cost, performance, human organization, or other elements are needed to clarify the clients' priorities.
5. An initial architecture is a vision. An architecture description is a set of specific models. The architecture of a building is more than the blueprints, floor plans, elevations, and cost estimates; it includes elements of ulterior

motives, belief, and unstated assumptions. This distinction is especially important in creating standards. Standards for architecture, like community architectural standards, are different from blueprint standards promoted by agencies or trade associations.

Architecting takes place within the context of an acquisition process. The traditional way of viewing hardware acquisitions is known as the waterfall model. The waterfall model captures many important elements of architecting practice, but it is also important in understanding other acquisition models, particularly the spiral for software, incremental development for evolutionary designing, and collaborative assembly for networks.

The Waterfall Model of Systems Acquisition

As with products and their architectures, no process exists by itself. All processes are part of still larger ones. And all processes have subprocesses. As with the product of architecture, so also is the process of architecting a part of a still larger activity, the acquisition of useful things.

Hardware acquisition is a sequential process that includes design, engineering, manufacturing, testing, and operation. This larger process can be depicted as an expanded waterfall, Figure 1.2, adapted after Rechtin (1991) but updated to today's

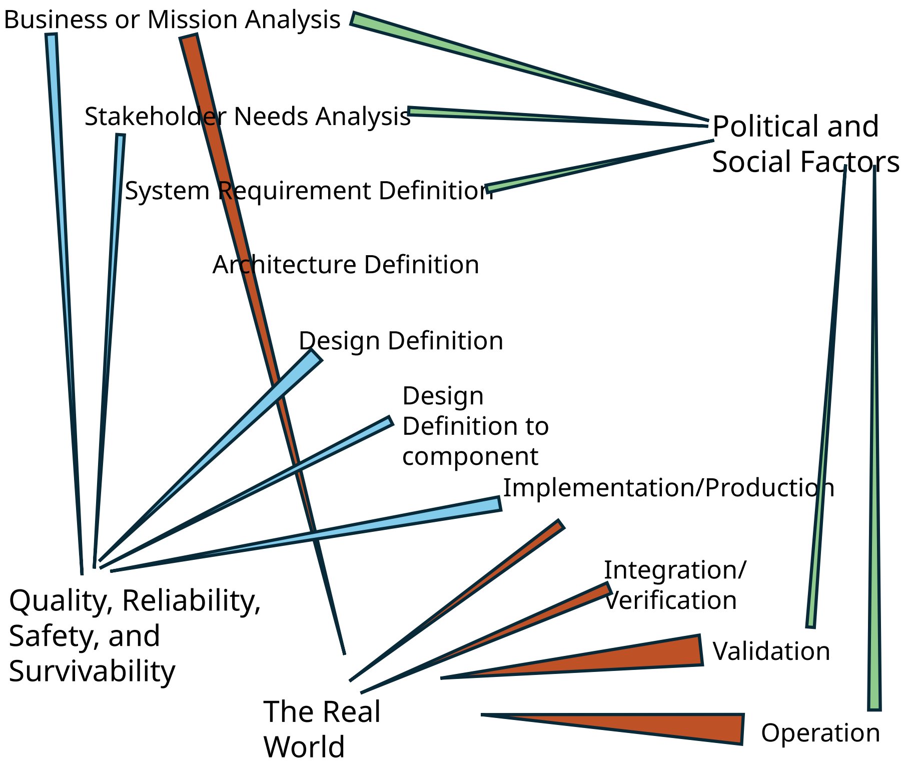

Business or Mission Analysis
Stakeholder Needs Analysis
System Requirement Definition
Architecture Definition
Design Definition
Design Definition to component
Implementation/Production
Integration/ Verification
Validation
Operation
Political and Social Factors
Quality, Reliability, Safety, and Survivability
The Real World

FIGURE 1.2 The expanded waterfall.

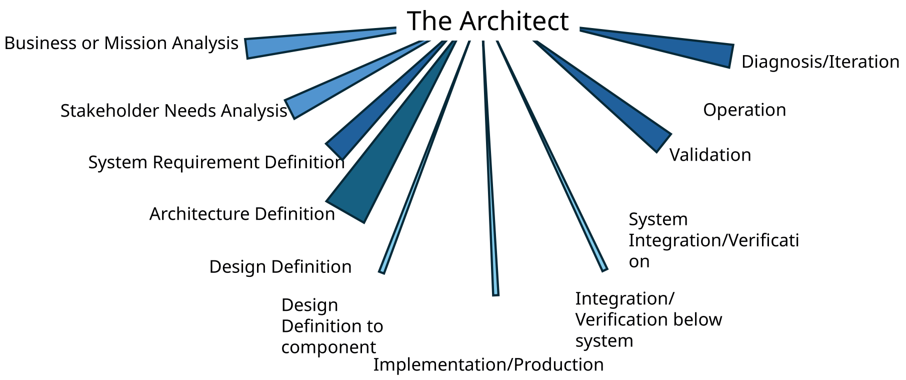

The Architect
Business or Mission Analysis
Stakeholder Needs Analysis
System Requirement Definition
Architecture Definition
Design Definition
Design Definition to component
Implementation/Production
Integration/ Verification below system
System Integration/Verificati on
Validation
Operation
Diagnosis/Iteration

FIGURE 1.3 The architect and the expanded waterfall (Adapted from Rechtin 1991).

standardized terminology from ISO (2023). The architect's functional relationship with this larger process is sketched in Figure 1.3. Managerially, the architect could be a member of the client's or the builder's organization, or of an independent architecting partnership in which perceptions of conflict of interest are to be avoided at all costs. In any case and wherever the architect is physically or managerially located, the relationships with the client and the acquisition process are essentially as shown. The strongest (thickest line) decision ties are with architecture definition and its near predecessors of requirements and needs analysis, and with diagnosis and evolution in operation. Less prominent are the monitoring ties with engineering and manufacturing. There are also important, if indirect, ties with social and political factors, the "illities" and the "real world" as indicated in Figure 1.2.

This waterfall model of systems acquisition has served hardware systems acquisition well for centuries. However, as new technologies create new, larger-scale, more complex systems of all types, others have been needed and developed. The most recent ones are due to the needs of software-intensive systems, as will be seen in Chapters 4 and 6 and in Part III. Although these models change the roles and methods of the architecting process, the basic functional relationships shown in Figure 1.3 remain much the same.

In any case, the relationships in Figure 1.3 are more complex than simple lines might suggest. As well as indicating channels for two-way communication and upward reporting, they infer the tensions to be expected between the connected elements, tensions caused by different imperatives, needs, and perceptions.

Some of competing technical factors are shown in Figure 1.4 adapted from Rechtin (1991). This figure was drawn such that directly opposing factors are located across from each other. For example, new technology (and all of its opportunities and risks) against mature technology. The trade-off is usually built on targeting new technology only where the impact on delivery value is very large and there are sufficient backup paths to reduce risk. Most of these trade-offs can be expressed in analytic terms, which certainly helps, but some cannot, as will become apparent in the social systems world of Chapter 5.

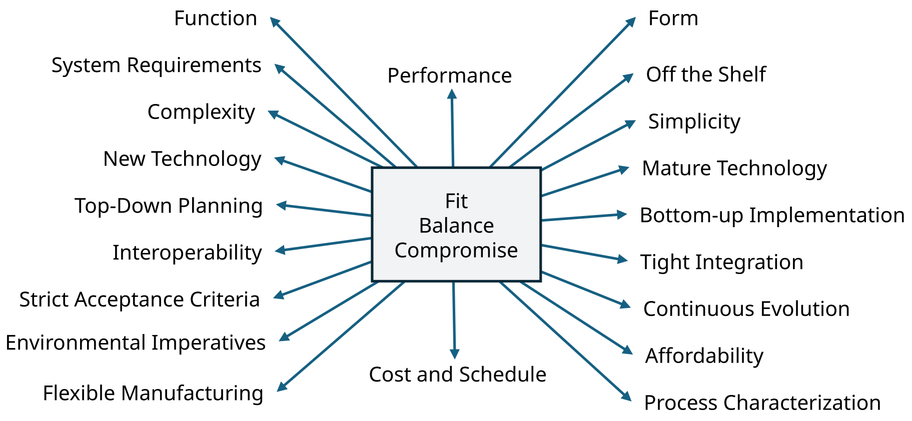


FIGURE 1.4 Tensions in systems architecting.

Exercise: Give examples from a specific system of what information, decisions, recommendations, tasks, and tensions might be expected across the lines of Figure 1.4.

Spirals, Increments, and Collaborative Assembly

The waterfall model of system development has been rightly criticized, though its core ideas remain in all alternatives. The critiques revolve primarily around the waterfall's static picture and non-accommodation for learning. In the waterfall, the presumption is that the development team can fully develop objectives and requirements in the earliest phase and that if the system is delivered compliant to those requirements the sponsors/users will find it satisfactory. The early freeze on requirements is seen as a virtue largely on the basis of the heuristic that states that the cost of removing a defect rises exponentially with the time (roughly the number of project phases elapsed) since the defect was introduced. If a defect is introduced as a faulty requirement and is not discovered until hardware is built and tested, then the cost of removal is likely to be high indeed. Hence there should be a premium on getting the requirements right and doing it early.

The counters to this perspective have been driven from the software-intensive systems perspective. Software developers have long understood that most software-intensive projects are not well suited to a sequential process but rather to a highly iterative one. The value of iteration comes in two forms. First, in many systems users will change how they operate once they have a working system in hand. In the third decade of the 21st century, the disruption of industries (or even society) by unanticipated usage of new systems is so common as to be familiar. Second, while fixing a software defect certainly gets more expensive the farther into the process it is discovered the impact is usually not nearly as bad as it is in hardware (though exceptions certainly exist). If model of automobile shifts with a hardware design defect that compromises reliability the recall and fix effort is large and expensive.

If the defect is a software defect then it may be fixed in the next over-the-air software update and may not even be apparent to the owner. But it is important to note, if the defect had major safety impact the situation could be very different. In addition, the heuristic on the cost of defect removal does not account for the cost of defect discovery. Discovering a defect introduced at any point in the process is not cost-free. Requirements defects might be very expensive to remove, but if the only realistic means of discovering them is to prototype the system then they are likewise expensive to discover.

Iterative development approaches exploit the relatively low cost to update a system to address the problems of finding defects that manifest when users change how they use the system. Instead of building the whole system in one go (the way a building or an aircraft carrier is typically built) build versions of increasing size and complexity. Each version is built on a mini-waterfall of specification, design, and test. These iterative processes have borrowed various terms, though we start from one of the earliest, the spiral. There is a strong incentive to iteratively modify software in response to user experience. As the market, or operational environment, reveals new desires, those desires are fed back into the product. One of the first formalizations of iterative development is due to Boehm and his famous spiral model. The spiral model envisions iterative development as a repeating sequence of steps. Instead of traversing a sequence of analysis, modeling, development, integration, and test just once, software may return over and over to each. The results of each are used as inputs to the next. This is depicted in Figure 1.5.

The original spiral model is intended to deliver one, hopefully stable, version of the product, the final version of which is delivered at the end of the last spiral cycle.

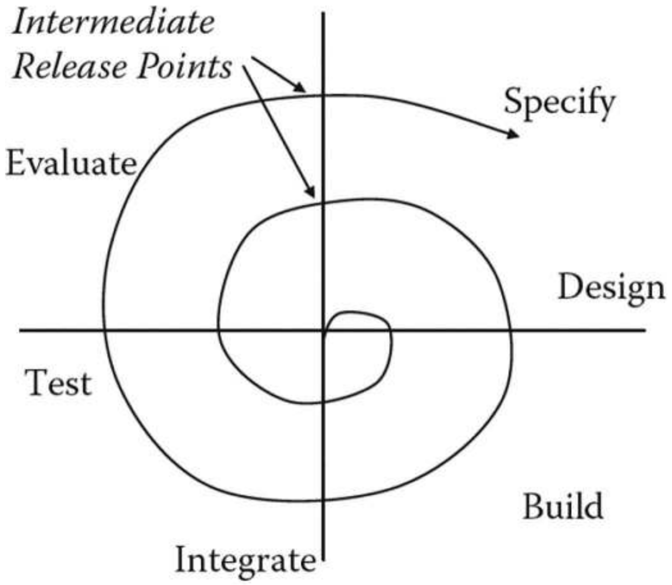


FIGURE 1.5 The "classic" spiral development model employs multiple cycles through the waterfall model's steps to reach a final release point.

The purpose of the cycles is risk control. The nominal approach is to set a target number of cycles at the beginning of development, and partition the whole time available over the target number of cycles. The objective of each cycle is to resolve the most risky thing remaining. So, for example, if user acceptance was adjudged as the most risky at the beginning of the project, the first spiral would concentrate on those parts of the system that produce the greatest elements of user experience. Even the first cycle tests would focus on increasing user acceptance. Similarly, if the most risky element was adjudged to be some internal technical performance issue, the product of the initial cycle would focus on technical feasibility.

Note that the risk spiral approach is often equally valuable for hardware-centric as for software-centric systems if the risks are identifiable and discrete and there is a relatively low cost in cycling. Suppose a system concept depended on a specific technical component (say, a laser or specialized computational element) reaching some target performance level, while the rest of the system was believed to be technically low risk but a large development effort (measured in human labor and material). A spiral approach would do a targeted prototype of the critical component while deferring the remaining efforts to a later cycle after the central component had reached the target performance level. This would be a cost-efficient approach (deferring low-risk work until after high risk was accomplished). Of course, it would not be schedule efficient since the spiral approach would likely take substantially longer than a once-through waterfall, assuming the risky portion worked the first time. The choice of development model is itself a design choice, at the level of program architecture and is itself subject to trades.

Many software products, or the continuing software portion of many product lines, are delivered over and over again. A user may buy the hardware once and expect to be offered a steady series of software upgrades that improve system functionality and performance. This alters a spiral development process (which has a definite end) to an incremental process, which has no definite end. The model is now more like a spiral spiraling out into circles, which represent the stable products to be delivered. After one circle is reached, an increment is delivered, and the process continues. Actually, the notion of incremental delivery appears in the original spiral model where the idea is that the product of spirals before the last can be an interim product release if, for example, the final product is delayed. As you shrink the cycles and emphasize the ability to revisit and adjust the feature set to be implemented you arrive at what we currently call agile development.

Finally, there are a number of systems in use today that are essentially continuously assembled, and where the assembly process is not directly controlled. The canonical example is the Internet, where the pieces evolve with only loose coupling to the other pieces. Control over development and deployment is fundamentally collaborative. Organizations, from major corporations to individual users, choose which product versions to use and when. No governing body exists (at least, not yet) that can control the evolution of the elements. The closest thing at present to a governing body, the Internet Society and its engineering arm, the Internet Engineering Task Force (IETF), can affect other behavior only through persuasion. If member organizations do not choose to support IETF standards, the IETF has no authority of compel compliance, or to block noncomplying implementations.

We call systems like this "collaborative systems." The development process is collaborative assembly. Whether or not such an uncontrolled process can continue for systems like the Internet as they become central to daily life is unknown, but the logic and heuristics of such systems now are the subject of Chapter 7. In Chapter 12, we address again different models of programs as examples of "program architecture" or patterns for designing a development program. Many strategic goals a client has to require addressing in the structure of the program rather than in the structure of the system.

The INCOSE Handbook of Systems Engineering (INCOSE 2023) uses slightly different terminology and is somewhat different in characterizing development program templates than we do here. We take up the details in depth in later chapters. The handbook templates are sequential, incremental, and evolutionary. The sequential form is essentially the waterfall and involves fixed objectives at the start and a single delivery. The incremental form presumes the full requirements are known at the beginning, but the development proceeds in learning cycles. There may or may not be a real system release in each cycle. This is essentially the classic spiral discussed above where the development spirals out to a pre-determined end point. What we call the protoflight and breadboard-brassboard-flight-production models later are essentially the incremental model. The evolutionary model allows the requirements to change and shift with each cycle. We learn in each cycle, and one thing we learn is more about what the requirements should be. In this model, there is a presumption of iterative release with the cycles. This is the spiral with no fixed endpoint and the functional incremental model.

SCOPES OF ARCHITECTING

What is the scope of systems architecting? By scope, we mean what things are inside the architect's realm of concern and responsibility and which are not. In the classic architecting paradigm (what we have discussed so far in this chapter), the client has a problem and wants to build a system in response. The system is the response to the client's needs, as constrained by the client's resources and any other outside constraints. The concern of architecting is finding a satisfactory and feasible system in response to the client's problem. A primary difference with other conceptualizations of similar situations is that architecting does not assume that the client's problem is well structured, or that the client fully understands it. It is quite likely a full understanding of the problem will have to emerge from the client–architect interaction.

As we look beyond the classic scope of the problem and system, we see several other issues of scope. First, a system is built within a "program," here defined as the collection of arrangements of funding, contracts, builders, and other elements necessary to actually build and deploy a system, whether a single-family house or the largest defense system. The program has a structure; we can say the program has an architecture. The architecture of the program will consist of strategic decisions, like is the system delivered once or many times? Is the system incrementally developed from breadboard to brassboard, or is it incrementally developed through fully deployable but reduced functionality deliveries? How is the work parceled out to different participants? We introduced this idea above with spiral and functional incremental development models and noted that the development model is itself a trade.

Who is responsible for the programmatic architectural decisions? In some cases, it may be important to integrate programmatic structure into the technical structure of the system. For example, if the system is to be partitioned over particular subsystem manufacturers, the system must possess a structure in subsystem compatible with what the suppliers can produce, and those subsystems must be specifiable in ways that allow for eventual integration. Whether or not the programmatic architectural decisions are in the scope of the system architect's responsibilities, there is a scope of programmatic architecture that somebody must carry out.

Moving upward and outward one more layer, the client presumably has an organization, even if the organization is only him- or herself. That organization may have multiple programs and be concerned with multiple systems. Those other systems may themselves form some larger structure like a system-of-systems, family-of-systems, or portfolio-of-systems (Maier 1998, 2019). The client's organization also has a structure, which many would call an architecture. The principal concerns at the organizational scope are the organization's strategic identity, or how does the organization give itself a mission? The organization exists to do something, what is that something? Is it to make money in particular markets, to advance a particular technology, or to carry out part of a military mission?

From the perspective of the system architect, it is unlikely that the strategic identity of the client's organization is in play in anything like the sense that the basic problem description is. However, the strategic identity of the client is important to the system architect. If that strategic identity is unclear, or poorly articulated, or simply unrealistic, then it will be very difficult for the client to make effective value judgments.

These scopes are illustrated in Figure 1.6. Figure 1.6 also illustrates one more issue of scope. Back at the scope of immediate concern to the system architect, both

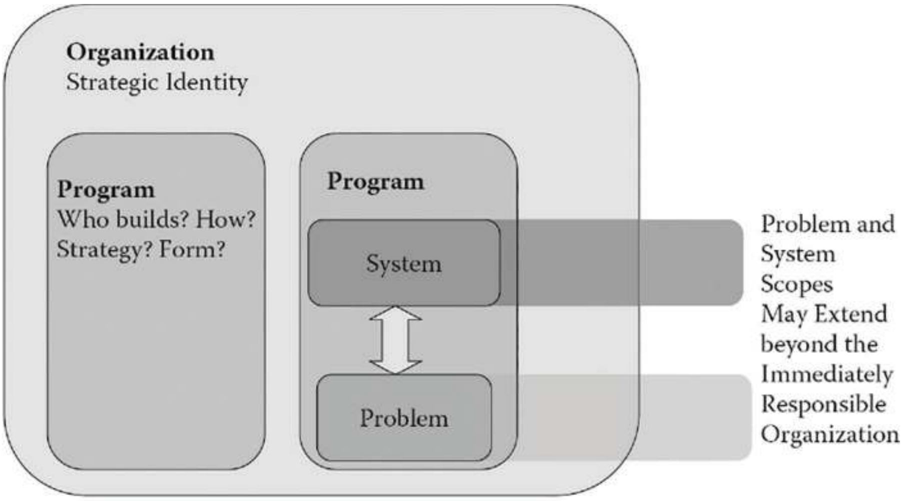

Organization
Strategic Identity
Program
Who builds? How? Strategy? Form?
Program
System
Problem
Problem and System Scopes May Extend beyond the Immediately Responsible Organization

FIGURE 1.6 The relationship between contexts for architecting. Our core concern is with the relationship between problem and system. But the structure of the development program and the identity and portfolio of the responsible organization are additional concerns.

the solution and problem may apply well outside the immediate program and the client's organization. Other clients may have the same or similar problems. A system developed for one client may apply elsewhere. Part of architecting is the one-to-one system-to-client orientation and individual customization, but this does not mean that others may not also be served by similar systems. Depending on the architect, it is quite likely that the issues of scope in problems and systems applying outside the immediate client's realm will be important. We return to these issues in detail in Chapter 12.

CONCLUSION

A system is a collection of different things that together produce results unachievable by themselves alone. The value added by systems is in the interrelationships of their elements.

Architecting is creating and building structures—that is, "structuring." *Systems* architecting is creating and building *systems*. It strives for fit, balance, and compromise among the tensions of client needs and resources, technology, and multiple stakeholder interests.

Architecting is both an art and a science—both synthesis and analysis, induction and deduction, and conceptualization and certification—using guidelines from its art and methods from its science. As a process, it is distinguished from systems engineering in its greater use of heuristic reasoning, lesser use of analytics, closer ties to the client, and particular concern with certification of readiness for use.

The foundations of systems architecting are a systems approach, a purpose orientation, a modeling methodology, ultraquality, certification, and insight. To avoid perceptions of conflict of interest, architects must avoid substituting their value judgments for those of the sponsors/users, avoid perceived conflicts of interest, and keep an arms-length relationship with project management. This is likely to be challenged in many circumstances as clients may be unprepared to make value judgments and desire advice.

A great architect must be as skilled as an engineer and as creative as an artist or the work will be incomplete. Gaining the necessary skills and insights depends heavily on lessons learned by others, a task of education to research and teach.

The role of systems architecting in the systems acquisition process depends upon the phase of that process. It is strongest during conceptualization and certification, but never absent. Omitting it at any point, as with any part of the acquisition process, leads to predictable errors of omission at that point to those connected with it.

NOTES

1 In a Byzantine failure, the failed component does the worst possible thing to the system. It is as if the component were possessed by a malign intelligence. The power of the technique is that it lends itself to certification, at least within the confines of well-defined models.

2 That said, when we break away from the classical architecting paradigm, we will see how responsibilities may change, and the freedom and risks inherent in doing so.

3 Precisely, this constraint led the Congress to mandate the formation in 1960 of a non-profit engineering company, The Aerospace Corporation, out of the for-profit TRW Corporation, a builder in the aerospace business.

REFERENCES

Cantry, D. (1963). “What architects do and how to pay them.” *Architectural Forum* **119**(September): 92–95.

INCOSE (2023). *INCOSE Systems Engineering Handbook*, Hoboken, NJ: John Wiley & Sons.

ISO (2023). *ISO/IEC/IEEE 15288:2023 Systems and Software Engineering - System Life Cycle Processes*, New York: IEEE.

Juran, J. M. (1988). *Juran on Planning for Quality*, New York: The Free Press.

Lala, J. H. and R. E. Harper (1994). “Architectural principles for safety-critical real-time applications.” *Proceedings of the IEEE* **82**(1): 25–40.

Maier, M. W. (1998). “Architecting principles for systems-of-systems.” *Systems Engineering: The Journal of the International Council on Systems Engineering* **1**(4): 267–284.

Maier, M. W. (2019). “Architecting portfolios of systems.” *Systems Engineering* **22**(4): 335–347.

Phadke, M. S. (1995). *Quality Engineering Using Robust Design*, Englewood Cliffs, NJ: Prentice Hall PTR.

Rechtin, E. (1991). *Systems Architecting: Creating and Building Complex Systems*, Englewood Cliffs, NJ: Prentice Hall.

Rechtin, E. (1994). “Foundations of systems architecting.” *Systems Engineering* **1**(1): 35–42.

2 Heuristics as Tools

INTRODUCTION: A METAPHOR

Mathematicians are still smiling over a gentle self-introduction by one of their famed members. “There are *three* kinds of mathematicians,” he said, “those that know how to count and those that don’t.” The audience waited in vain for the third kind until, with laughter and appreciation, they caught on. Either the member could not count to three—ridiculous—or he was someone who believed that there was more to mathematics than numbers, important as they were. The number theorists appreciated his acknowledgment of them. The “those that don’t” quickly recognized him as one of their own, the likes of a Gödel who, using thought processes alone, showed that no set of theorems can ever be complete.

Modifying the self-introduction only slightly to the context of this chapter: There are three kinds of people in our business, those who know how to count and those who do not—including the authors.

Those who know how to count (most engineers) approach their design problems using analysis and optimization, powerful and precise tools derived from the scientific method and calculus. Those who do not (most architects) approach their qualitative problems using guidelines, abstractions, and pragmatics generated by lessons learned from experience—that is, heuristics. As might be expected, the tools each use are different because the kinds of problems they solve are different. We routinely and accurately describe an individual as “thinking like an engineer”—or architect, or scientist, or artist. Indeed, by their tools and works you will know them.

Of course, we exaggerate to make a point. The reality is that architects often compute (must compute), and engineers use many heuristics. Both architecting and engineering are complex amalgams of art and science. To be one who uses heuristics does not mean avoiding being systematic and quantitative or being highly detail-oriented (when needed). To be analytically oriented does not mean applying the same standards of quantitative analysis anywhere and everywhere regardless of circumstances. But the complexity of integrating the art and science can wait. For now, we want to understand those things that are squarely part of the “art” of systems architecting.

This chapter, metaphorically, is about architects’ heuristic tools. As with the tools of carpenters, painters, and sculptors, there are literally hundreds of them—but only a few are needed at any one time and for a specific job at hand. To continue the metaphor, although a few tool users make their own, the best source is usually a tool supply store—whether it be for hardware, artists’ supplies, software—or heuristics. Appendix A, Heuristics for Systems-Level Architecting, is a heuristics store, organized by task, just like any good hardware store. Customers first browse, and then select a kit of tools based on the job, personal skill, and knowledge of the origin and intended use of each tool.

DOI: 10.1201/9781003483595-4

Heuristic has a Greek origin, *heuriskein,* a word meaning "to find a way" or "to guide" in the sense of piloting a boat through treacherous shoals. Architecting is a form of piloting. Its rocks and shoals are the risks and changes of technology, construction, and operational environment that characterize complex systems. Its safe harbors are client acceptance and safe, dependable, long life. Heuristics are guides along the way—channel markings, direction signs, alerts, warnings, and anchorages—tools in the larger sense. But they must be used with judgment. No two harbors are alike. The guides may not guarantee safe passage, but to ignore them may be fatal. The stakes in architecting are just as high—reputations, resources, vital services, and, yes, lives. Consonant with their origin, the heuristics in this book are intended to be trusted, time-tested guidelines for serious problem-solving.

Heuristics as so defined are narrower in scope, subject to more critical test and selection, and intended for more serious use than other guidelines, for example, conventional wisdom, aphorisms, maxims, rules of thumb, and the like. For example, a pair of mutually contradictory statements like (1) *look before you leap* and (2) *he who hesitates is lost* are hardly useful guides when encountering a cliff while running for your life. In this book, neither of these pairs would be a valid heuristic because they offer contradictory advice for the same problem.

The purpose of this chapter is therefore to help the reader—whether architect, engineer, or manager—find or develop heuristics that can be trusted, organize them according to need, and use them in practice. The first step is to understand that heuristics are abstractions of experience.

HEURISTICS AS ABSTRACTIONS OF EXPERIENCE

One of the most remarkable characteristics of the human race is its ability not only to learn but to pass on to future generations sophisticated abstractions of lessons learned from experience. Each generation knows more, learns more, plans more, tries more, and succeeds more than the previous one because it does not need to repeat the time-consuming process of re-living the prior experiences. Think of how extraordinarily efficient are such quantifiable abstractions as $F=ma$, $E=mc^2$, and $x=F(y, z, t)$; of algorithms, charts, and graphs; and of the basic principles of economics. This kind of efficiency is essential if large, lengthy, complex systems and long-lived product lines are to succeed. Few architects ever work on more than two or three complex systems in a lifetime. They have neither the time nor the opportunity to gain the experience needed to create first-rate architectures from scratch. By much the same process, qualitative heuristics, condensed and codified practical experience, came into being to complement the equations and algorithms of science and engineering in the solving of complex problems. Passed from architect to architect, from system to system, they worked. They helped satisfy a real need.

In contrast to the symbols of physics and mathematics, the format of heuristics is words expressed in the natural languages. Unavoidably, they reflect the cultures of engineering, business, exploration, and human relations in which they arose. The birth of a heuristic begins with anecdotes and stories, hundreds of them, in many fields which become parables, fables, and myths, easily remembered for the lessons they teach. Their impact, even at this early stage, can be remarkable not only on politics, religion, and business but also on the design of technical systems and services. The lessons that have endured are those that have been found to apply beyond the original context, extended thereby analogy, comparison, conjecture, and testing.[1] At their strongest they are seen as self-evident truths requiring no proof. See sources such as Pearl (1984), Klir (1985), Asato (1988), Rowe (1988), Kittay (1990).

There is an interesting human test for a good heuristic. An experienced listener, on first hearing one, will know within seconds that it fits that individual's model of the world. Without having said a word to the speaker, the listener almost invariably affirms its validity by an unconscious nod of the head, and then proceeds to recount a personal experience that strengthens it. Such is the power of the human mind.

Selecting a Personal Kit of Heuristic Tools

The art in architecting lies not in the wisdom of the heuristics, but in the wisdom of knowing which heuristics apply, a priori, to the current project.

Williams (1992)

All professions and their practitioners have their own kits of tools, physical and heuristic, selected from their own and others' experiences to match their needs and talents. But, in the case of architecting of systems prior to the late 1980s, selections were limited and, at best, difficult to acquire. Many heuristics existed, but they were mainly in the heads of practitioners. There were notable efforts in other fields. Civil architecting has a history of codified heuristics, as in Christopher Alexander's work. Alexander's concept of a pattern language was exploited in software at roughly the same time. But no efforts had been made to articulate, organize, and document a useful set for general systems architecting. The heuristics in this book were codified largely through the University of Southern California graduate course in Systems Architecting. The students and guest instructors in the course, and later program, were predominantly experienced engineers who contributed their own lessons learned throughout the West Coast aerospace, electronics, and software industries. Both as class exercises, and through the authors' writings, they have been expressed in heuristic form and organized for use by architects, educators, researchers, and students.

An initial collection of about 100 heuristics documented in an appendix of Rechtin (1991) was soon surpassed by contributions from over 200 students, reaching nearly 1,000 heuristics within 6 years. Many, of course, were variations on single, central ideas—just as there are many variations of hammers, saws, and screwdrivers—repeated in different contexts. The four most widely applicable of these heuristics were as follows, in decreasing order of popularity:

1. *Do not assume that the original statement of the problem is necessarily the best or even the right one.*

   Example: The original statement of the problem for the F-16 fighter aircraft asked for a high-supersonic capability, which is difficult and expensive to produce. Discussions with the architect, Harry Hillaker, brought out that the reason for this statement was to provide a quick exit from combat, something far better provided by a high thrust-to-weight, low supersonic design. In short, the original high-speed statement was replaced by a high acceleration one, with the added advantage of exceptional maneuverability.

2. *In partitioning, choose the elements so that they are as independent as possible; that is, choose elements with low external complexity and high internal complexity.*

   Example: One of the difficult problems in the design of microchips is the efficient use of their surface area. Much of that area is consumed by connections between components—that is, by communications rather than by processing. Carver Mead of Caltech demonstrated that a design based on minimum communications between process-intensive nodes results in much more efficient use of space, with the interesting further result that the chip "looks elegant"—a sure sign of a fine architecture and another confirmation of the heuristic:

3. *The eye is a fine architect. Believe it.Simplify. Simplify. Simplify.*

   Example: One of the best techniques for increasing reliability while decreasing cost and time is to reduce the piece part count of a device. Automotive engineers, particularly recently, have produced remarkable results by substituting single castings for multiple assemblies and by reducing the number of fasteners and their associated assembly difficulties by better placement.

   Example: Recall the dimensions of complexity in Chapter 1. Sometimes the most dramatic simplifications come from re-framing the problem, itself a heuristic of great power.

4. *Build in and maintain options as long as possible in the design and implementation of complex systems. You will need them.*

   Example: In the aircraft business, they are called "scars." In the software business, they are called "hooks." Both are planned breaks or entry points into a system that can extend the functions the system can provide. In aircraft, they are used for lengthening the fuselage to carry more passengers or freight. In software, they are used for inserting further routines or to allow integration of data with other programs.

Though these four heuristics do not make for a complete tool kit, they do provide good examples for building one. All are aimed at reducing complexity, a prime objective of systems architecting. All have been trusted in one form or another in more than one domain. All have stood the test of time for decades if not centuries.

The first step in creating a larger kit of heuristics is to determine the criteria for selection. The following were established to eliminate unsubstantiated assertions, personal opinions, corporate dogma, anecdotal speculation, mutually contradictory statements, and the like. As it turned out, they also helped generalize domain-specific heuristics into more broadly applicable statements. The strongest heuristics passed all the screens easily. The criteria were as follows:

- The heuristic must make sense in its original domain or context. To be accepted, a strong correlation, if not a direct cause and effect, must be apparent between the heuristic and the successes or failures of specific systems, products, or processes. Academically speaking, both the rationale for the heuristic and the report that provided it were subject to peer and expert review. As might be expected, a valid heuristic seldom came from a poor report.

- The general sense, if not the specific words, of the heuristic should apply beyond the original context. That is, the heuristic should be useful in solving or explaining more than the original problem from which it arose. An example is the preceding *do not assume* heuristic. Another is *Before the flight it is opinion; after the flight it is obvious.* In the latter, the word "flight" can sensibly be replaced by test, experiment, fight, election, proof, or trial. In any case, the heuristic should not be wrong or contradictory in other domains where it could lead to serious misunderstanding and error. This heuristic applies in general to ultraquality systems. When they fail, and they usually fail after all the tests are done and they are in actual use, the cause of the failure is typically a deterministic consequence of some incorrect assumptions; and we wonder how we missed such an obvious failure of our assumptions.
- The heuristic should be easily rationalized in a few minutes or on less than a page. As one of the heuristics states, *If you can't explain it in five minutes, either you don't understand it or it doesn't work* (Darcy McGinn 1992 from David Jones). With that in mind, the more obvious the heuristic is on its face, and the fewer the limitations on its use, the better. *Example*: *A model is not reality.*
- The opposite statement of the heuristic should be foolish, clearly not "common sense." For example: The opposite of *Murphy's Law—If it can fail, it will*—would be "If it can fail, it won't," which is patent nonsense.
- The heuristic's lesson, though not necessarily its most recent formulation, should have stood the test of time and earned a broad consensus. Originally this criterion was that the heuristic *itself* had stood the test of time, a criterion that would have rejected recently formulated heuristics based on retrospective understanding of older or lengthy projects. *Example*: *The beginning is the most important part of the work* (Plato 4th Century B.C.), reformulated more recently as *All the serious mistakes are made in the first day.*[2]

It is probably true that heuristics can be even more useful if they can be used in a set, like wrenches and screwdrivers, hammers and anvils, or files and vises. The taxonomy grouping in a subsequent section achieves that possibility in part.

It is also probably true that a proposed action or decision is stronger if it is consistent with several heuristics rather than only one. A set of heuristics applicable to acceptance procedures substantiates that proposition.

And it would certainly seem desirable that a heuristic, taken in a sufficiently restricted context, could be specialized into a design rule, a quantifiable, rational evaluation, or a decision algorithm. If so, heuristics of this type would be useful bridges between architecting, engineering, and design. There are many cases where we have such progressive extensions, from a fairly abstract heuristic that is broadly applicable to a set of more narrowly applicable, but directly quantifiable, design rules.

USING HEURISTICS

Virtually everybody, after brief introspection, sees that heuristics play an important role in their design and development activities. Hard rules and rigorous analysis are both inherently limited. However, even if we accept that everyone uses heuristics, it is not obvious that those heuristics can be communicated and used by others. This book takes the approach that heuristics can be effectively communicated to others. One lesson from Rechtin (1991) and previous editions of this book is that heuristics do transfer from one person to another, but not always in simple ways, and not always in the way expected by the authors. It is useful to document heuristics and teach from them, but learning styles differ.

People typically use heuristics in three ways. First, they can be used as evocative guides. They work as guides if they evoke new thoughts in the reader. Some readers have reported that they use the catalog of heuristics in the appendices at random when faced with a difficult design problem. They scan the appendix list and when one of the heuristics seems suggestive, they follow up by considering how that heuristic could describe their present situation, what solutions it might suggest, or what new questions it suggests.

The second usage is as codifications of experience. In this usage, the heuristic is like an outline heading, a guide to the detailed discussion that follows. In this case, the stories behind the heuristics can be more important than the basic statement. Taking the "all the really important mistakes are made the first day," this is shorthand for the lessons on how problem statements are often poorly formulated (and can be revised) or that strongly asserted requirements are sometimes proxies for deeper objectives that may, or may not, be identical with the asserted requirement. In this form, the heuristic is a pedagogical tool, a way of teaching lessons not well captured in other engineering teaching methods.

The third usage is the most structured. It is when heuristics are integrated into development and design description processes. This means that the heuristics are attached to the steps of an overall design process. The design process specifies a series of steps and models to be constructed. The heuristics are attached to the steps as guidelines for accomplishing those steps.

A good example of heuristic-process integration is in software. Many software development methods are built on producing a sequence of models, from relatively abstract to code in a programming language. Object-oriented methods, for example, usually begin with a set of textual requirements, build a model of abstracted classes and objects, and then refine the class/object model into code in the target programming environment. There are often intermediate steps in which the problem-domain-derived objects are augmented with objects and characteristics from the target environment. A problem in all such methods is knowing how to construct the models at each step. When should a problem domain concept be captured as an object? What object granularity of object representation is best when choosing implementation space elements? The transformation from a set of textual requirements to classes and objects is not unique, but it involves extensive judgment by the practitioner. Heuristic-augmented methods assist the practitioner by giving explicit, prescriptive heuristics for each step.

A PROCESS FRAMEWORK FOR ARCHITECTING HEURISTICS

In Part III of this book, we will present a basic process framework for system architecting, the Architecture Project Model-Applied Systems Architecting Method (APM-ASAM). The process framework will define activities repeatedly required in effective architecting and define how those activities are arranged relative to each other. We place those activities in a larger architecture project framework. This process framework is illustrated in Figure 2.1. As noted above, one method for using heuristics is to attach them to steps in a design process. By doing so, the heuristics become local guides to each aspect of the process. A complete process with step-by-step designated models and transformation heuristics is not appropriate for general systems architecting. There is simply too much variation from domain to domain, too many unique domain aspects, and too many important domain-specific tools. Even so, it is useful to recognize the basic structure of the process framework and how the heuristics relate to that framework.

It is important to distinguish between the activity cycle for an entire development program and the activity cycle for an architecture project. The goal of a development program is to build and deliver a system. The goal of an architecture project is something else. In the simplest case, the goal of the architecture project is to initiate a development program. Even in the simple case, we recognize that development programs go in fits and starts. There might be several discrete architecture projects,

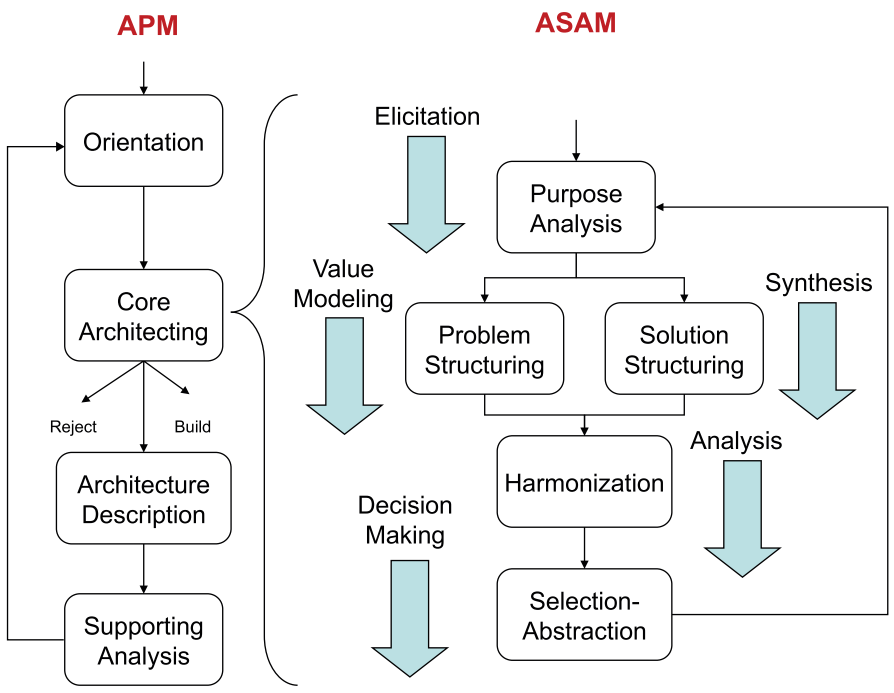


FIGURE 2.1 Activities in the APM-ASAM architecting process model.

simultaneously or sequentially developing architectural concepts for every actual development program since many, perhaps most, architecture projects may not result in a full development program. Some architecture projects do not have a specific system development as their goal, as in the architecture projects that concern collaborative systems (which we take up in Chapter 7).

The beginning of an architecting project is "orientation" or determining where you are and where you want to go. This refers both to the architecture project as well as the underlying, assumed but not yet existing, system development project. Orientation is less technical and more business or political. Its intent is to ensure that the architecture effort can proceed for at least one iterative cycle in an organized fashion. Heuristics associated with orientation relate to topics like identifying the driving characteristics of a project, finding leading stakeholders, and clarifying relationships between the architect, sponsors, and downstream users. Orientation is about scoping and planning, and so the heuristics of Appendix A and in Chapter 9 under the associated topics apply most strongly. Orientation leads to core architecting, which is defined by purpose analysis, problem structuring, solution structuring, harmonization, and selection-abstraction.

Purpose analysis is a broad-based study of why the system of interest should have value. It works from an understanding of the client strategy and expands to all stakeholders with significant power over the eventual construction, deployment, and operation of the system. Purpose analysis is an elicitation activity, and so all heuristics that relate to elicitation apply most strongly here.

Problem structuring is where we organize elements of the problem space with a primary focus on a "value model." The value model is an explicit model of the most important stakeholder's preferences, and it is intended to capture them without regard to mutual consistency. That is, we want to be able to assess alternatives in the value system of each major stakeholder, realizing that the resulting preference orderings will not be the same. Any reconciliation necessary among them will be conducted later. Its concern is on the problem side of the problem–system tension. It is a synthesis activity in the sense that we are synthesizing problem descriptions, preferably several, with somewhat different scopes. In terms of Appendix A and Chapter 9, the associated heuristics are drawn mostly from modeling and prioritizing.

In solution structuring, we synthesize models of solutions, again multiple solutions that should differ in scope and scale. The heuristics that apply are drawn from those that cover modeling, aggregating, and partitioning.

Harmonization is a dominantly analytical activity in which we integrate problem and solution descriptions and assess value. Harmonization is a preparation for selection-abstraction. Selection is easy to understand; it is picking answers. An important distinction between the approach of systems architecting and most decision analysis texts is that we do not assume when we enter selection that there is a unitary, exclusive decision to make. At some point in the process, if the overall goal is to build a system, we must clearly make a decision about a preferred configuration. But we might travel down this process road many times before reaching such a unitary decision. Along the way, we may wish to hold onto multiple solution configurations, classes of solution configurations, and multiple problem descriptions. The best output of an architecture project may be a preferred class of solutions with downstream design processes to choose the best configuration from the preferred class. We make no decision before its time. As it was put in Rechtin (1991), *Hold onto the agony of decision as long as possible*. The notion of abstraction captures those cases where architecting has been completed even though no single configuration has been selected.

As an example of abstraction over selection, consider the case of a family of systems, say the collection of printers made by a single company. There are shared properties or components across the whole family (e.g., interfaces, software rendering engines, supply chains). These shared elements are the concern of the family-of-systems architect and are abstractions of the entire family. It is inaccurate to talk about selecting the whole family (though we might select the market-niche structure of the whole family), but it is accurate to consider selection of properties of the whole family abstract into a family-of-systems architecture. We refer to that form of selection as "abstraction."

Architectural projects ultimately produce architecture descriptions, a document. The term "document" should be interpreted broadly, as it may be a collection of linked models on an information system rather than a paper document. We illustrate this as a following step to Core Architecting in Figure 2.1. Architecture descriptions are developed at least partially in parallel with the architectural decision-making. But it is helpful to illustrate the separation of the two activities to emphasize that architecting is about decision-making, and architectures are about decisions. Architecture descriptions can only document those decisions. The quality (or lack thereof) of those decisions must stand on its own. An excellently drawn description will not make up for poor architecture decisions.

Finally, in practice, architects discover in the process where they need additional knowledge. Straightforward progress through an architecture study may be interrupted by the discovery that we do not know critical numbers related to the cost or performance of a key system element, or we do not understand the technicalities of a particular stakeholder problem, or we lack clear input on preferences from a stakeholder. In most cases, it is more effective to put such issues aside by making suitable assumptions, returning to the issues after completing an end-to-end pass-through architectural analysis, resolving those detailed issues in studies, and returning to another iterative cycle through the architecting process.

HEURISTICS ON HEURISTICS

A phenomenon observed as heuristics discovered by the USC graduate students is that the discoverers themselves began thinking heuristically. They found themselves creating heuristics directly from observation and discussion, and then trying them out on professional architects and engineers, some of whose experiences had suggested them (Most interviewees were surprised and pleased at the results). The resultant provisional heuristics were then submitted for review as parts of class assignments.

Kenneth L. Cureton, carrying the process one step further, generated a set of heuristics on how to generate and apply heuristics (Cureton 1991) from which the following were chosen.

Generating Useful Heuristics

- Humor [and careful choice of words] in a heuristic provides an emotional bite that enhances the mnemonic effect [Karklins].
- Use words that transmit the "thrill of insight" into the mind of the beholder.
- For maximum effect, try embedding both descriptive and prescriptive messages in a heuristic.
- Many heuristics can be applied to heuristics (e.g., *Simplify*! *Scope*!).
- Do not make a heuristic so elegant that it only has meaning to its creator and thus loses general usefulness.
- Rather than adding a conditional statement to a heuristic, consider creating a separate but associated heuristic that focuses on the insight of dealing with that conditional situation.

Applying Heuristics

- If it works, then it is useful.
- Knowing when and how to use a heuristic is as important as knowing what and why.
- Heuristics work best when applied early to reduce the solution space.
- Strive for balance—too much of a good thing or complete elimination of a bad thing may make things worse, not better!
- Practice, practice, practice!
- Heuristics are not reality, either!

A TAXONOMY OF HEURISTICS

The second step after finding or creating individual heuristics is to organize them for easy access so that the appropriate ones are at hand for the immediate task. The collection mentioned earlier in this chapter was accordingly refined and organized by architecting task.[3] In some ways, the resultant list—presented in Appendix A—was self-organizing. Heuristics tended to cluster around what became recognized as basic architecting tasks. For example, although certifying is shown last and is one of the last formal phases in a waterfall, it actually occurs at many milestones as "sanity checks" are made along the way and subsystems are assembled. The tasks, elaborated in Chapters 8 and 9, are as follows:

- Scoping and planning
- Modeling
- Prioritizing
- Aggregating
- Partitioning
- Integrating
- Certifying
- Assessing
- Evolving and rearchitecting

The list is further refined by distinguishing between two forms of heuristics. One form is descriptive; that is, it describes a situation but does not indicate directly what to do about it. Another is prescriptive; that is, it prescribes what might be done about the situation. An effort has been made in the appendix to group prescriptions under appropriate descriptions with some, but not complete, success. Even so, there are more than enough generally applicable heuristics for the reader to get started.

And then there are sets of heuristics that are domain-specific to aircraft, spacecraft, software, manufacturing, social systems, and so on. Some of these can be deduced or specialized from more general ones given here. Or, they can be induced or generalized from multiple examples in specialized subdomains. Still more fields are explored in Part III, adding further heuristics to the general list.

You are encouraged to discover still more, general and specialized, in much the same way the more general ones here were—by spotting them in technical journals, books, project reports, management treatises, and conversations.

Appendix A taxonomy is not the only possible organizing scheme, any more than all tool stores are organized in the same way. In Appendix A, one heuristic follows another, one-dimensionally, as in any list. But some are connected to others in different categories or could just as easily be placed there. Some are "close" to others, and some are further away. Ray Madachy, then a graduate student, using hypertext linking, converted the list into a two-dimensional, interconnected "map" in which the main nodes were architecting themes: Conception and design; the systems approach; quality and safety; integration, test, and certification; and disciplines (Madachy 1991a,b). To these were linked each of the 100 heuristics in the first systems architecting text (Rechtin 1991), which in turn were linked to each other. The ratio of heuristic-to-heuristic links to total links was about 0.2; that is, about 20% of the heuristics overlapped into other nodes.

The Madachy taxonomy, however, shared a limitation common to all hypertext methods—the lack of upward scalability into hundreds of objects—and consequently was not used for Appendix A. Nonetheless, it could be useful for organizing a modest-sized personal tool kit or for treating problems already posed in object-oriented form, for example, computer-aided design of spacecraft (Asato 1989).

Various other taxonomies are, of course, possible and may be more useful for particular toolkits. The literature has other collections of heuristics and approaches to validating and choosing them. Standard works, like the INCOSE Systems Engineering Body of Knowledge (SEBoK) (INCOSE 2024) and INCOSE Systems Engineering Handbook (INCOSE 2023), address the issue of heuristics and provide their own sets. There has been a heuristics project active within INCOSE for some time.

NEW DIRECTIONS

Heuristics are a popular topic in systems and software engineering, though they often go by another name or are formulated somewhat differently from here. A notable example is the concept known as a "pattern language." The idea of patterns and pattern languages comes from Christopher Alexander (Alexander 1964, 2018) and has been adapted to other disciplines by other writers. Most of the applications are to software engineering (Alexander 1999).

A pattern is a specific form of prescriptive heuristic. Several forms have been used in the literature, but all are similar. The basic form is a pattern name, a statement of a problem, and a recommended form of solution (to that problem). So, for example, a pattern in civil architecture (from Alexander) has the title "Masters and Apprentices," the problem statement describes the need for junior workers to learn while working from senior master workers, and the recommended solution consists of suitable arrangements of workspaces.

When a number of patterns in the same domain are collected together, they can form a pattern language. The idea of a pattern language is that it can be used as a tool for synthesizing complete solutions. The architect and client use the collected problem statements to choose a set that is well-matched to the client's concerns. The resulting collection of recommended solutions is a collection of fragments of a complete solution. It is the job of the architect to harmoniously combine the fragments into a whole.

In general, domain-specific, prescriptive heuristics are the easiest for apprentices to understand and use. So, patterns on coding in programming are relatively easy to teach and learn to use. This is borne out by the observed utility of software coding pattern books in university programming courses. Similarly, an easy entry to the use of heuristics is when they are attached as step-by-step guides in a structured development process. At the opposite end, descriptive heuristics on general systems architecting are the hardest to explain and use. They typically require the most experience and knowledge to apply successfully. The catalog of heuristics in Appendix A has heuristics across the spectrum.

CONCLUSION

Heuristics, as abstractions of experience, are trusted, nonanalytic guidelines for treating complex, inherently unbounded, ill-structured problems. They are used as aids to decision-making, value judgments, and assessments. They are found throughout systems architecting, from earliest conceptualization through diagnosis and operation. They provide bridges between client and builder, concept and implementation, synthesis and analysis, and system and subsystem. They provide the successive transitions from qualitative, provisional needs to descriptive and prescriptive guidelines, and thence to rational approaches and methods. Heuristics are of high relevance in engineering as well as architecting, and many other fields, but of course we survey only the architecting applications here.

This chapter has introduced the concept of heuristics as tools—how to find, create, organize, and use them for treating the qualitative problems of systems architecting. Appendix A provides a ready source of them organized by architecting task—in effect, a tool store of systems architecting heuristic tools.

NOTES

1 This process is one of inductive reasoning, "a process of truth estimation in the face of incomplete knowledge which blends information known from experience with plausible conjecture" (Klir 1985). More simply, it is an extension or generalization from specific examples. It contrasts with deductive reasoning, which derives solutions for specific cases from general principles.

2 Spinrad, Robert, Lecture at the University of Southern California, 1988.

3 The original 100 of Rechtin (1991) were organized by the phases of a waterfall. The list in Appendix A of this book recognizes that many heuristics apply to several phases, that the spiral model of system development would in any case call for a different categorization, and that many of the tasks described here occur over and over again during systems development.

REFERENCES

Alexander, C. (1964). *Notes on the Synthesis of Form*, Cambridge, MA: Harvard University Press.

Alexander, C. (1999). "The origins of pattern theory: The future of the theory, and the generation of a living world." *IEEE Software* **16**(5): 71–82.

Alexander, C. (2018). *A Pattern Language: Towns, Buildings, Construction*, Oxford: Oxford University Press.

Asato, M. (1988). *The Power of the Heuristic. AE 549 Reports*, Los Angeles, CA: University of Southern California.

Asato, M. (1989). *Spacecraft Design and Cost Model. AE 549*, Los Angeles, CA: University of Southern California.

Cureton, K. (1991). *Metaheuristics. AE 549 Reports*, Los Angeles, CA: University of Southern California.

INCOSE (2023). *INCOSE Systems Engineering Handbook*, New York: John Wiley & Sons.

INCOSE (2024). *Guide to the Systems Engineering Body of Knowledge*, San Diego, CA: International Council on Systems Engineering.

Kittay, E. F. (1990). *Metaphor: Its Cognitive Force and Linguistic Structure*, Oxford: Oxford University Press.

Klir, G. J. (1985). *Architecture of Systems Problem Solving*, New York: Plenum Press.

Madachy, R. (1991a). *Formulating Systems Architecting Heuristics for Hypertext. AE 549 Reports*, Los Angeles, CA: University of Southern California.

Madachy, R. (1991b). *Thread Map of Architecting Heuristics. AE 549 Reports*, Los Angeles, CA: University of Southern California.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Rechtin, E. (1991). *Systems Architecting: Creating and Building Complex Systems*, Englewood Cliffs, NJ: Prentice Hall.

Rowe, A. J. (1988). The meta logic of cognitively based heuristics. In: *Expert Systems in Business and Finance: Issues and Applications*. P. R. Watkins, and Eliot, L. B. (Eds), New York: John Wiley and Sons, (pp. 109–127).

Williams, P. (1992). *AE 549 Reports*, Los Angeles, CA: University of Southern California.

Part II

Introduction

New Domains, New Insights

Part II explores, from an architectural point of view, five domains beyond those of aerospace and electronics, the sources of most examples and writings to date. The chapters can be read for several purposes. For readers familiar with a domain, there are broadly applicable heuristics for more effective architecting of its products. For those unfamiliar with the domain, there are insights to be gained by understanding problems that differ in the degree but not in kind from their own. To coin a metaphor, if the domains can be seen as planets, then this part of the book corresponds to comparative planetology, the exploration of other worlds to benefit their own. The chapters can be read for still another purpose, as a template for exploring other, equally instructive, domains. An exercise for that purpose can be found at the end of Chapter 7, "Collaborative Systems."

Each of the chapters is preceded by a brief case study. Each of the case studies is chosen to be relevant to the chapter to which it is attached. Many students and readers have asked about case studies of real systems to assist in understanding the application of the materials. Unfortunately, really good engineering and architecting case studies are notoriously hard to obtain. The stories and details are rarely published. Books published on major systems are more likely to focus on the people involved than on the technical decision-making. Many of the most interesting stories are buried behind walls of proprietary information. By the time the full story can be published, it is often old. We, the authors, think the older stories carry timeless lessons, so we have included several here. Each includes some references back to the original literature, where it is readily available, so the interested reader can follow up with

DOI: 10.1201/9781003483595-5

further investigation of his or her own. In a few cases, we abstracted several cases into one where the original stories have not yet been published, and the combination makes the lessons clearer.

From an educational point of view, this part is a recognition that one of the best ways of learning is by example, even if the example is in a different field or domain. One of the best ways of understanding another discipline is to be given examples of problems it solves. And one of the best ways of learning architecting is to recognize that there are architects in every domain and at every level from which others can learn and with whom all can work. At the most fundamental level, all speak the same language and carry out the same process of systems architecting. Only the examples are different.

Chapter 3 explores systems for which form is predetermined by a builder's perceptions of need. Such systems differ from those that are driven by client purposes by finding their end purpose only if they succeed in the marketplace. The uncertainty of the end purpose has risks and consequences that it is the responsibility of architects to help reduce or exploit. Central to doing so are the protection of critical system parameters and the formation of innovative architecting teams. These systems can be either evolutionary or revolutionary. Not surprisingly, there are important differences in the architectural approach. The case study is an old one, but an excellent one, on the development of the DC-3 airplane.

Chapter 4 highlights the fact that manufacturing has its own waterfall, quasi-independent of the more widely discussed product waterfall, and that these two waterfalls must intersect properly at the time of production. A spiral-to-circle model is suggested to help understand the integration of hardware and software. Ultraquality and feedback are shown to be the keys to both lean manufacturing and flexible manufacturing, with the latter needing a new information flow architecture in addition. The case study is on the development of mass production, particularly its development at Ford and later Toyota.

Chapter 5 on sociotechnical systems introduces a number of new insights to those of the more technical domains. Economic questions and value judgments play a much stronger role here, even to the point of outright veto of otherwise worthwhile systems. A new tension comes to center stage, one central to social systems but too often downplayed in others until too late—the tension between facts and perceptions. It is so powerful in defining success that it can virtually mandate system design and performance, solely because of how that architecture is perceived. The case study is on architecting intelligent transportation systems.

Chapter 6 serves to introduce the domain of software as it increasingly becomes the center of almost all modern system designs. Consequently, whether stand-alone or as part of a larger system, software systems must accommodate to continually changing technologies and product usage. In very few other domains is annual, much less monthly, wholesale replacement of a deployed system economically feasible or even considered. In point of fact, it is considered normal in software systems, precisely because of software's unique ability to continuously and rapidly evolve in response to changes in technology and user demands. Software has another special property; it can be as hard or as soft as needed. It can be hard-wired if certification must be precise and unchanging. Or it can be as soft as a virtual environment molded at the will of a user. For these and other reasons, software practice is heavily dependent on heuristic guidelines and organized, layered modeling. It is a domain in which architecting development is very active, particularly in progressive modeling and rapid prototyping. The case study is on the transition from hierarchical to layered systems, a major point of contention in software systems. It is abstracted from several real cases familiar to the authors.

Chapter 7 introduces an old but newly significant class of systems, collaborative systems. Collaborative systems exist only because the participants actively and continuously work to keep it in existence. A collaborative system is a dynamic assemblage of independently owned and operated components, each one of which exists and fulfills its owner's purposes whether or not it is part of the assemblage. These systems have been around for centuries in programs of public works. But today, we find wholly new forms in communications (the Internet and World Wide Web), transportation (intelligent transportation systems), militaries (multinational reconnaissance-strike and defensive systems), and software (open-source software). The architecting paradigm begins to shift in collaborative systems because the architect no longer has a single client who can make and execute decisions. The architect must now deal with more complex relationships and must find architectures in less familiar structures, such as architecture through communication or command protocol specification. The case study is on the Global Positioning System (GPS), which did not start as a collaborative system, but which is rapidly evolving into one.

The nature of modern software and information-centric systems, and their central role in new complex systems, makes a natural lead into Part III, "Models and Modeling."

Case Study 2

The DC-3

While the DC-3 airplane was designed and built in the 1930s, it is not unknown for someone today to have flown on one. There were many flying into the 2000s. There will probably be some flying 100 years after their introduction, maybe even operating profitably in some remote area. The DC-3 is commonly cited as the most successful airplane ever built. What accounts for the extraordinary success of the DC-3 airplane? The history of the DC-3's development extensively illustrates many of the key lessons of systems architecting, especially the following:

1. The role of the very small architecting team in bringing vision and coherence to the system concept.
2. The cooperative nature of the effective architect–client relationship, even when the architect belongs to the builder organization.
3. The role of coupled technological and operational change in creating revolutionarily successful systems. New technology enabled a revolutionary change in commercial air operations with the DC-3, but only because that technology was used in a new concept of operations.
4. The role of evolutionary development in enabling revolutionary development.

The focus of this case study is on the decisions that defined the architecture of the DC-3, and how they compared to the decisions made by competitors at that time rather than a detailed recounting of all aspects of the history. The decisions, by Douglas and competitors, were in the context of the respective organizations and system sponsors and reflected different concerns. For the history itself, we draw on the Boeing 247 focused work (Van der Linden and Seely 2011) and a core DC-3 reference (Gradidge and Olson 2006). While the book "*The Boeing 247: The First Modern Airliner*" focuses on the Boeing 247, important parts of the airplane's story must be told in comparison to the Douglas series (DC-1, DC-2, and DC-3), which were under development at roughly the same time. Decisions that led to the DC-3 are meaningful when one sees how they compare to those made on the Boeing 247, and why the two teams reached different conclusions and different designs at the same time with access to essentially the same technology. More DC-3 information is at dc3history.org. One of the most valuable sources for the architectural history of the DC-3, and an exceptional source of architecting heuristics, is the paper "The Well Tempered Aircraft" by Arthur Raymond (1951).

DOI: 10.1201/9781003483595-6

THE HISTORY

In a room of the Smithsonian Air and Space Museum devoted to "America by Air" between World Wars I and II, three key airplanes can be seen together. They are the Ford Trimotor, the Boeing 247, and the DC-3. Of these, only the DC-3 can be seen outside of an air museum or historical air show. In 1930, the Ford Trimotor was state-of-the-art in passenger and cargo aircraft. It carried eight passengers and enabled passenger and cargo service across the USA. But, by modern standards, the airplane was barely usable. The large piston engine on the nose coupled noise and vibration (and sometimes exhaust) directly into the passenger and cargo areas. The framed fuselage put large spars directly through the passenger and cargo area, with obvious inconvenience for both types of service. Reliability and safety were far from modern standards. The knowledge of aerodynamics and structures, coupled with the available engine power-to-weight ratio, allowed no better. Regardless, it was such an improvement over its predecessor, and delivered such value, that 199 were built (see Figure CS2.1).

In the early 1930s, aeronautical technology was changing quickly. Engines were improving very rapidly in power and power-to-weight ratio, new structural concepts were being tested, and understanding of aerodynamics was improving very rapidly (from prototype airplanes, theoretical study, and the first generation of capable wind tunnels). These innovations enabled very significant improvements in capacity, range, speed, safety, and passenger comfort when used synergistically in new aircraft designs. Improved structural design methods, coupled with new materials and manufacturing methods, dropped the weight of an airplane of given volume. Better aerodynamics improved lift-to-drag ratio, a fundamental parameter in performance determination. Better engine power-to-weight ratio meant higher performance at a given weight, or an opportunity to cut weight while leaving other factors constant. The elements of a virtuous cycle of weight reduction and size and performance improvement were in place.

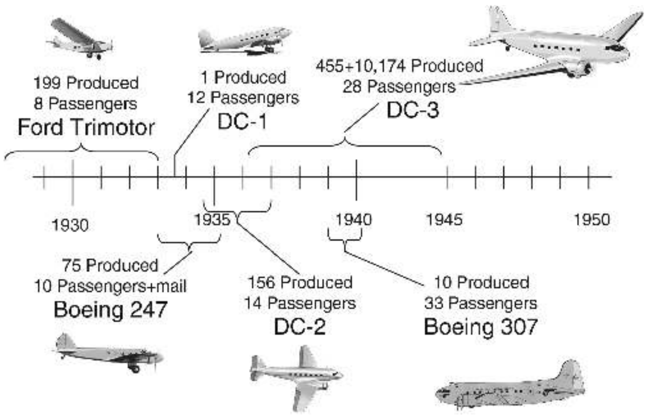


FIGURE CS2.1 Timeline of the DC-3 and related aircraft.

Two young companies riding the early boom in aeronautics, Boeing and Douglas Aircraft, were developing new airplane concepts exploiting these new technologies. One of the themes of this book is on how architecture should and does reflect sponsor needs and priorities. In a builder-architected system, those priorities include the concerns and interests of the builders and not just the client, operator, or user. The two companies faced very different business situations and clients.

In 1930, Boeing, along with what became United Airlines, Pratt & Whitney engines, and several other companies, were part of a vertically integrated company known as United Aircraft and Transport Corporation (UATC). At the time the most profitable airline business was carrying air mail under US Government contracts, in which UATC was dominant. As a result, when Boeing investigated how to exploit the emerging aeronautical technologies, they knew that government air mail contracts were the primary profit source. What we now think of as the regular business of airlines—carrying passengers and general freight—was financially ancillary to the airmail. Boeing's design studies for how to best exploit the new technology in engines, aerodynamics, and structures focused on an aircraft that was optimized for the routing structure imposed by the US Postal Service. Boeing also had to consider other concerns based on being part of UATC, in particular a preference to use Pratt & Whitney engines. The result was the Boeing 247. The airline arm of UATC quickly ordered 60, a large leap in production capacity for the Boeing of the time. Tellingly, UATC rejected an order for 247s from Trans World Airlines (TWA), a rival airline.

There is no doubt that the Boeing 247 was a revolutionary airplane, both technologically and in terms of user experience. It was far more comfortable than the Ford Trimotor, and much faster. But it was not revolutionary from a business–operational perspective. The 247 was intended to do business the way it was being done, just much better. The 247 had significant limitations in going beyond the air mail business. First, it was too small. The 10-passenger capacity was insufficient on economies of scale beyond the air mail business. Second, while the metal structure used the then-new stressed skin monocoque technique, it still had a structural span running crosswise through the passenger compartment, impeding passenger, and cargo loading and unloading. Both of these issues were the subject of extensive debate inside Boeing during development. They came about because of overall assumptions about how the technology would work and understanding of the demands from air mail contracts (which were known) and from passengers (that were more guesswork). Specifically, the arguments that won out were as follows:[1]

- There was little in the way of economies of scale going larger. Two airplanes of half the capacity would cost around half (or less) of a larger aircraft. Moreover, the main customer (the Post Office) preferred faster and more numerous planes.
- Passengers preferred a faster airplane and would give up on-board comfort to get one.
- Optimizing speed put the wing at mid-fuselage height and ran structural members through the cabin. However, this was acceptable since, as noted earlier, speed was considered primary and comfort secondary.

These arguments, seemly well-thought-out at the time, did not age well as others tried alternative approaches. And there was a strong push for alternatives. As noted above, TWA attempted to acquire Boeing 247s but was unable to. Instead, TWA approached Douglas aircraft, who was likewise considering how to incorporate the new technologies into new aircraft designs. Unlike Boeing, Douglas and their airline partners were thinking well beyond the immediate profit source of airmail. TWA specified a three-engine aircraft, in the belief that three engines were required to provide single-engine-out survivability over the Sierra Nevada or Rocky Mountains while also providing the capacity, speed, and range for profitable operation. This was a quandary for Douglas. While their sponsor had just expressed clear requirements, Douglas was well aware that three-engine aircraft (with one engine in the nose) had many serious drawbacks. Initially, Douglas decided to prototype an aircraft that was quite similar to the Boeing 247.

The DC-1 was produced contemporaneously with the Boeing 247 and was roughly the same size. Douglas and their customers realized the advantages of the new overall design, given the new technologies but believed the airplane was too small. The DC-1 was essentially a proof-of-concept airplane. Douglas and the airlines intended it to be a production representative airplane, but it served mainly to prove the concept and demonstrate the way forward. It also demonstrated the underlying requirement that led TWA to specify three engines, that the airplane be able to survive a single-engine-out condition anywhere in flight, was achievable in a two-engine airplane. The DC-1 had some design decisions significantly different from the 247, both technically and within the larger programmatic context. Douglas was not part of a vertically integrated company. While a vertically integrated enterprise may have scale advantages, those are likely to be outweighed by non-competitiveness in some components in a time of very rapidly changing technology. For example, Douglas was not tied to an in-house engine supplier. On the technical design front, Douglas was able to use the Jack Northrop wing design that was substantially lighter with better capacity than Boeing's, and did not put structural members in the cabin space, by placing the wing box under the fuselage. Douglas, and partners, correctly estimated that the speed advantages of the 247 design were marginal and would quickly disappear as technology advanced and that passengers would prefer comfort over marginal speed improvements. Both assumptions that the speed advantage was marginal and temporary and that passengers would prefer comfort turned out to be correct.

The DC-2 followed immediately on the DC-1 demonstration, was larger, and was commercially successful. It had a production run of 156 aircraft (see Figure CS2.1 for the times and figures), a considerable number for the time. The production run of the DC-2 was already larger than the Boeing 247, and nearly the size of the Ford Trimotor's, the previously most successful airplane. American Airlines, after some experience with the DC-2, approached Douglas about a further upsizing, with intent to use the airplane in cross-country sleeper service. Douglas and their team began design work immediately on the DC-3. It was much larger, with a passenger capacity double that of the DC-2. As it turned out, while the sleeper service was implemented (a mimic of established railroad sleeper car service), it quickly became apparent that the larger size and associated range and performance was highly successful in regular, daytime service.

This capacity and range change meant that regular passenger service could now be profitable independent of airmail transportation. Around the same time, the airmail business model was disrupted by the Airmail Act of 1934, which forced UATC to break up and ended the UATC dominance in airmail contracts. With airline operations freed from the structures of airmail routing and scheduling, the airlines were free to use the DC-3 to develop new markets. This was the operational revolution that was enabled by, then drove, the technological revolution.

The DC-3 production run was much larger than any previous airplane, reflecting its revolutionary success in the commercial airline business. Even though Douglas was confident of the excellence of the DC-3, the magnitude of the success was a surprise. The company chose an initial production to produce tooling with a design life of 50 units (which Raymond regarded as "rather daring"). That tooling lasted through hundreds of aircraft. With 455 of the initial commercial model produced, it was the foundation of the modern, then rapidly growing airline business.

Of course, the story does not end here. Boeing saw the success of the DC-3 and moved to counter with an even larger and higher performance aircraft. The breakup of UATC and the Great Depression greatly constrained Boeing's ability to respond. However, Boeing was well placed to continue to move up in aircraft size and performance by other work in their aircraft portfolio. At that time, Boeing was building the XB-15 bomber under a contract with the US Army Air Corps. They also undertook on their own to build a prototype advanced four-engine bomber known as the Model 299 (the predecessor of the B-17 Flying Fortress of World War II fame). Elements of the Model 299 were directly incorporated into a prototype of the Boeing 307. The 307 had a capacity of 33 passengers, a pressurized cabin, and much greater speed and range.

Here, history and chance intervenes in the story. With the success of the DC-3 airlines went to Douglas and committed to the DC-4, but that project was unsuccessful with the resulting aircraft being seen as unacceptable. Douglas had to entirely redesign the airplane with initial deliveries delayed into 1941. The model 299 and the 307 prototype were involved in a series of accidents. The original Model 299 crashed in 1935 during the bomber evaluations(Network 1963). The Air Corps regardless ordered limited numbers of a redesigned aircraft for additional development. The 307 was built from elements from the B-17 experimental program. The prototype 307 crashed in March 1939 during stability testing when it lost stability and broke up in flight (Network 1994). Among those killed in the crashes were Boeing's Chief Aerodynamicist and Chief Engineer.

The delays to both the DC-4 and Boeing 307 programs were significant. The 307 did not enter service until 1940 and the redesigned DC-4 until 1941. By then, US industry was rapidly converting to war production. After the attack on Pearl Harbor in 1941, essentially all airplane production was converted to war production, but in large measure, the conversion had already begun. The US Army Air Corps needed transports, bombers, fighters, and all types of aircraft. Boeing, with its advantages in large bombers, moved its production primarily to bombers. The DC-3 was an obvious choice as a transport. It was a proven, mature design with proven utility and reliability. Enormous contracts for producing military variants of the DC-3 came rapidly, and more than 10,000 were produced in various military configurations.

This huge production base became the foundation for the aircraft to fly productively for decades after production ended.

After World War II, the competition in commercial airplanes resumed, advancing from a new point in both technology and operations. The DC-3 existed in such large numbers that there was hardly room for a direct competitor. The technology for building and operating much larger aircraft had been extensively developed. The transports produced after World War II were larger still, mostly four-engine aircraft. And soon after, the transition to jet engines would revolutionize the architecture of commercial aircraft, and the airline industry, once again.

ARCHITECTURE INTERPRETATION

As interesting as the capsule history of the DC-3 may be, this history is not the primary focus here in this book. The reader may find many extensive histories of the DC-3 and its competitors. But we are interested here in understanding and interpreting its architecture—not just on its own—but in relationship to its competitors and within the context of its builders, sponsors, and users.

Three Story Variations

Three different but related interpretations can be considered in the DC-3 story. The first way of seeing the story is as one of architectural revolution fueled by technology. In this way, we see the DC-3 as a technology-enabled architectural jump over the Ford Trimotor. The moral of this story is that technological advance combined with architectural vision creates a revolutionary system. This story is, of course, true; but it is also incomplete. The DC-3 was a revolutionary advance over the Ford Trimotor, and it was a combination of technological advance and architectural vision. But it did not happen in one step, it did not happen in only one place, and it did not happen all at once. If the DC-3 was a technology-driven jump, then so was the Boeing 247. To understand the success of the DC-3 over the 247, we need to look beyond the first story of a technology-driven jump.

In the second story, we see the Boeing 247 and the DC-3 as a story in the hazards of optimality. The moral of the second story is that being optimal with respect to the problem as currently or originally understood is not always the best choice. The DC-3 achieved enormous success because it did not optimally serve existing markets; instead, it leapfrogged and enabled new markets. The revolution was not just in technology of airplanes, it was in the coupling of technological change with operational change. The DC-3 became a huge success when its owners changed their business model in response to its capabilities. In this story, we can see the Boeing 247 as a cautionary tale to not look too narrowly, especially in times of rapid change.

The third story expands the second by seeing what Boeing did after the appearance of the DC-3. When the DC-3 opened new markets, Boeing did not stand still. They had already invested in the 247, and it was being used on airmail routes, but they did not continue to build it in the face of the greater success of the DC-3. Instead, they followed where the DC-3 had revealed the market to be (larger, faster, higher-capacity aircraft) by building the 307. The 307 might have been a highly successful aircraft, except that World War II intervened and upset the competition with the forced conversion to war production.

The third story must color our perception of success and failure, and Boeing versus Douglas' decision-making. Boeing started the revolution with the 247. Boeing was eclipsed by the DC-3, but that must be viewed in the larger context of builder's strategic positions and capabilities. Boeing started with a stronger business position and a direct relationship with the leading customer, United Air Lines. Boeing also held a "real option"[2] on moving to even larger aircraft in a way that Douglas did not. Thus, Boeing could logically make a more conservative decision for the competitive and technological positioning of the 247 than made sense for Douglas. This leads naturally to the next question.

Was the Boeing 247 Successfully Architected?

It seems obvious that the DC-3 was very successfully architected. It is generally regarded as the most successful aircraft of all time, and beautifully combined technical and operational innovation. According to Raymond (1951), the combination was deliberate, if not entirely foreseen. The natural follow-on question is to ask how successfully was the Boeing 247 architected? Obviously, it was a much less successful aircraft. But it was the aircraft its sponsors requested. It did effectively exploit the new technology, and it did what was asked. The general question is, if a sponsor gets the system he or she asks for, and as a result loses in a competitive environment, did the architects perform either job effectively?

There is no universal answer to this question. The answer depends very much on how the development environment structures the relationship between the architect and sponsor. In the classical architecting paradigm, the architect must be careful not to substitute his or her own value judgments for those of the client. So, if the system reflects the client's value judgments, and the system is ultimately unsuccessful because those value judgments do not reflect reality, the architecting job has still been done well. But it is also traditionally well within the architect's responsibility to warn the client of the certain or likely consequences of proposed courses of action. If it is evident to the architect that the design process is leading to something that can be easily opposed by competitors, this must be made plain to the client. The client may choose to proceed anyway, but the consequences should be clear.

In some cases, the architect may have an ethical or even legal responsibility beyond that of the responsibility to the client. Public buildings must be built in accordance with public safety. A system architect working for a government has some responsibility beyond just the immediate acquisition program to the national interest. In our DC-3 story, the architect was part of the builder organization and so had a great stake in ultimate success or failure. A builder-architect cannot shrug off poor client decision-making as the builder-architect is also the client and rises or falls on the result. The architect in the builder-architected situation must respond to the concerns and priorities of the builder organization. The builder-architect has a level of ownership of the problem a third-party architect does not. The builder-architect also has possible level of conflict of interest, when the concerns of the builder and the client are inconsistent, that the third-party architect does not.

WHAT IS THE "ARCHITECTURE" OF THE DC-3?

Asking "What is the architecture of the DC-3" illustrates the contrast between architecture as physical design and architecture as concept development points of view. Both the Boeing 247 and DC-3 shared the same essential structural, technical features. Side by side they look very similar. Both were two-engine, hollow fuselage, and modern configuration transport aircraft. Both used very similar technology. In the sense of overall physical design, they are quite similar.

However, in performance attributes and in operational placement, they are quite different. The DC-3 is considerably larger and, more importantly, is enough larger for the performance margin to have great operational significance. The DC-3 performs missions that the Boeing 247 cannot and enables business models that the Boeing 247 cannot. In a larger context, the design of the DC-3 embodies a different business strategy than the Boeing 247. If we think of architecture as the technical embodiment of strategy, we see the distinct differences between the architectures of the two systems.

ART RAYMOND'S PRINCIPLES

One of the attractions of the DC-3 story is the excellent Art Raymond paper previously referenced. Raymond's paper provides a set of eight timeless principles for architecting that hold as well today as they did when first articulated:

1. **Proper Environment:** This includes the physical facilities in which designers work, but Raymond's focus was on the confidence and enthusiasm of the sponsors and adequate financing. In Raymond's words:

   > The thing above all else that makes a project go is the enthusiasm of its backers; not false enthusiasm put on for effect — sooner or later this is seen through — but rather the enthusiasm that comes from the conviction that the project is sound, worth-while, and due to succeed. (Raymond 1951)

2. **Good Initial Choice:** In Raymond's terms, a good initial choice is one that neatly combines value and feasibility. He particularly emphasizes the role of elegant compromise between conflicting factors and clearly identifying the need or mission for the aircraft. The biggest failures come not from systems that are technological failures, but from those that fail to meet any need well enough to generate demand.
3. **Excellence of Detail Design:** Although this book is focused on architecture as the initial concept, detailed design is likewise important. An excellent initial concept can be ruined by poor detailed design (although a poor initial concept is very unlikely to be saved by excellence in detailed design).
4. **Thorough Development:** Raymond's perspective on thorough development emphasizes design refinement after the first test flight. In Raymond's era, the refinement of flying qualities of airplanes was quite important and occurred mostly after the first flight. Calculations and wind tunnel tests were sufficient for basic performance, but refining handling qualities to a point of excellence required extensive flight testing.

5. **Follow-Through:** Follow-through refers to the system life cycle after delivery to the operator. In the case of a commercial aircraft, some of the important elements include operator and maintainer training, maintenance and service facilities, development of spare parts, design updates in response to service data, and technical manuals. The value of the system to its customers/operators is directly related to the quality of follow-through. From the perspective of systems architecting, the follow-through elements may be inside the boundaries of the initial concept development. The quality of the initial concept may be determined by its amenability to effective follow-through.
6. **Thorough Exploitation:** All successful aircraft are extensively modified during their operational lifetimes. The DC-3 was produced in an enormous number of variations, and even today there are firms that adapt modern avionics to the remaining DC-3 airframes. Successful systems are designed to accommodate a range of modifications. This is familiar in modern commercial aircraft where many interior configurations are available, usually several different choices of engine, freighter and passenger versions, and extended range or capacity versions.
7. **Correct Succession:** No matter how successful a system is, there comes a time when it is more effective to break away and re-architect. Conversely, breaking away when the time is not ripe incurs high cost to little effect. The essential judgment here is projection of technical and operational trends. There is an opportunity for succession when either (or better yet both) will move substantially over the time required to develop the successor system.
8. **Adaptiveness:** The DC-1, DC-2, and DC-3 sequence is the best illustration of adaptiveness. Adaptiveness really means responsiveness to the future environment as it unfolds, rather than as it was projected. Projections are the foundation of planning, and real strategy is the ability to adapt to the environment as it unfolds. In this story, we see several examples of adaptiveness in architecture. Douglas did not settle for the DC-1, even though it met the contractual specifications provided by TWA. Instead, they adapted to the operational environment as it developed, first with the improved DC-2 and then with the much-upsized DC-3. Likewise, Boeing illustrated effective adaptiveness in the sense of retaining (and then exercising) real options for larger aircraft. When their first attempt at a revolutionary aircraft was insufficient, they used large aircraft technology from their military aircraft to upsize their flagship commercial aircraft to the Boeing 307.

NOTES

1 See van der Linden, pages 32–40.
2 A "real option" is an investment allowing an organization an increased ability, or lower cost, to pursue future lines of action. A summary reference is Courtney, H. (2001). *20/20 Foresight: Crafting Strategy in an Uncertain World,* Harvard Business Press.

REFERENCES

Courtney,H. (2001).*20/20 Foresight: Crafting Strategy in an Uncertain World*, Cambridge, MA: Harvard Business Press.

dc3history. org. (2024).“The DC-3/Dakota Historical Society.” Retrieved 30-June, 2024, from-www.dc3history.org.

Gradidge, J. and D. D. Olson (2006). *The Douglas DC-1/CD-2/DC-3: The First Seventy Years*, Tonbridge, Kent:Air-Britain (Historians).

Network, A. S. (1963). “Boeing 299 Flying Fortress.” Retrieved 2024, from https://asn.flightsafety.org/wikibase/83555.

Network, A. S. (1994). “18 March 1939 Boeing S. 307 Stratoliner.” Retrieved 2024, from https://asn.flightsafety.org/asndb/342182.

Raymond, A. E. (1951).“The well–tempered aircraft.” *The Aeronautical Journal*55(490): 599–625.

Van der Linden, F. R. andV. J. Seely (2011). *The Boeing 247: The First Modern Airliner*, Washington, DC:University of Washington Press.

3 Builder-Architected Systems

No system can survive that doesn't serve a useful purpose.

Harry Hillaker[1]

INTRODUCTION: WHO ARCHITECTS VERSUS WHAT DRIVES THE ARCHITECTURE

The classical architecting paradigm of the third-party architect and separation of sponsor and builder is not the only way to create and build large complex systems, nor is it the only regime in which architects and architecting are important. Systems architected by the builder, there the architectural decisions are made in the builder organization, are common. While we will focus on that alternative, in this chapter, just looking at who does the architecting is a partial picture of the relevant concerns. We also need to consider the basis or the driver for undertaking the architecture effort. Two alternatives are "purpose-driven" and "technology-driven." We first encountered this in this book by identifying "purpose-driven" as the classical basis for architecting. In contrast, we can have "technology-driven" and perhaps other motivations.

In a purpose-driven system, the sponsor (provider of resources) and the users are both present and engaged when we architect. We can work with users to understand what they want, and we can work with the sponsor on what resources are available, and on what the sponsor is concerned with. Of course, it may well be that the sponsors and users are not aligned, and they may or may not be talking to each other and be willing to align, but the point is that both are available and engaged.

In a technology-driven system, the sponsor is present, but the users are not. The driver is the belief that some technology will enable something, and that something will be highly valued by a user community, but the user community is not engaged. One dimension of overall complexity will be how engageable those users are. A better case is they can be engaged. A more difficult case is they can be engaged but are expected to change their minds after any exposure to the system of interest (the classic ill-structured problem situation). Worse yet, there is no engagement possible since the existence of the users is hypothetical.

In the builder-architected case the architects are part of the builder organization, normally a company. Their client, the sponsor, is the company. The users are the customers of the system. The baseline case is where the customers are outside company, but we can have the case of internal customers when the system of interest is used inside. Manufacturing systems are usually of this type (since the company usually

DOI: 10.1201/9781003483595-7

builds their own factory) and internal information technology systems may be. A builder-architected system may be incremental (based on pre-existing systems) or something entirely new. Incremental development is more common.

Incremental Development for an Existing Customer

Most builder-initiated architectures are variations of existing ones; for example, consider jet aircraft, personal computers, smart automobiles, and follow-on versions of existing software applications. The original architectures, having proved by use to be sound, variations, and extensions should be of low risk. Extensive reuse of existing modules should be expected because design assumptions, system functions, and interfaces are largely unchanged. When systems of a type have been around for a number of years, especially from competitive suppliers, it is common for the architectures to settle to a relatively stable form.

Building on the DC-3 case study, consider the evolution of commercial aircraft. The general structure of the DC-3, a round main fuselage with a flat floor and open from front to back, is a consequence of monocoque construction and the demands of passenger and cargo operation. It has been stable from the DC-3 to airplanes developed in the 2000s. Likewise, the low wing, though it went from straight to swept as jet engines arrived, has been stable on large commercial aircraft for the same period. Propeller-driven aircraft generally have their engines embedded in the wing, and jet engines hang below the wing. Military transport jets are similar except that they are usually high-wing designs, which is superior for the operational profiles typical of military transports. These patterns have now been stable for decades. Every decade or so, there is interest in possible disruption to the pattern, with blended wing-body being the latest idea, but so far, none has had enough advantages to outweigh the established pattern. This is not to say that it will never happen, but architectures can remain stable for very long periods and displacing a very stable architecture is quite difficult.

The architect's responsibilities in the evolutionary case remain much the same as under the classical paradigm, but with an important addition: the identification of proprietary architectural features deemed critical to maintaining competitive advantage in the marketplace. Lacking this identification, the question "who owns what?" can become so contentious for both builder and customer that product introduction can be delayed for years. As was noted in the DC-3 case, where the in-house versus open supplier boundary lay was a key difference in the Boeing and Douglas aircraft surrounding the DC-3.

Far more important than these relatively low risks is the paradigm shift from function-to-form (purpose-driven) to one of form-to-function (form driven). Unlike the classical paradigm, in form-first architecting, one's customers judge the value of the product after rather than before the product has been developed and produced. In the classical paradigm, the customer is responsible for the value judgments, and so should expect to be satisfied with the resultant system. In a form-first, builder-architected system, the architect hopes the customer will find it satisfactory, but there are no guarantees. The judgment of success begins only after the system is built and delivered.

The resultant risk has spawned several risk-reduction strategies. The simplest is an early prototype demonstration to present to customers, with its associated risks of premature rejection. The more rapidly prototypes can be developed and delivered, the more rapidly feedback can be gained from customers. Another strategy is the open-source method for designing software, a process in which customers become developers, or at least active participants with developers. Anyone interested can participate, comment, submit ideas, develop software, and use the system, all at no cost to the participant. The project being tied together by the Internet (and some unique social conventions), everyone—and particularly the builder and potential clients—knows and can judge its utility. The risk of rejection is sharply reduced at the possible cost of control of design. The open-source community is a principal example of collaborative system assembly. We discuss that topic specifically in Chapter 7.

New Markets for Existing Products

The next level of architecting intensity is reached when the builder's motivation is to reach uncertain or "latent" markets in which the unknown customer must acquire the product before judging its value. Almost certainly, the product will have to be at least partially rearchitected in terms of cost, performance, availability, quantities produced, and so forth. To succeed in the new venture, architecting must be particularly alert, making suggestions or proposing options without seriously violating the constraints of an existing product line. Hewlett-Packard, in the 1980s (when they were primarily an engineering test equipment company), developed this architecting technique in a novel way. Within a given product line, say that of a "smart" analytic instrument, a small set of feasible "reference"[2] architectures are created, each of which is intended to appeal to a different kind of customer. In current terminology, we refer to "product-line-architectures" or the architecture of a "family-of-systems." The core idea is that the architecture of the product-line/family-of-systems remains stable but accommodates local changes to produce new products within the product-line. Latent markets discovered in the process can then be quickly exploited by expansion of the product line.

Several other chapters investigate additional variations on the family-of-systems idea. In Chapter 7, we discuss collaborative systems, a different arrangement of multiple systems into a greater whole. Chapter 6 and case study 5 deal with different conceptions of a family-of-systems, and the over-arching architecture, in this case in software and information systems. See Maier (2019) for a more comprehensive discussion of different variations including portfolios-of-systems as well as families and collaborative systems.

The original product line architecture can be maintained with few modifications or risks while multiple completed systems are offered to the market. Ideally, the architectural features of the product line are largely invariant, but the architectural features of individual products change rapidly. The product line sets out constraints and resources, and the individual products use them to produce valued features.

The architecture of a product-line or family-of-systems must be related to the architecture of its individual members. The architecture of the product line is dominantly, though not exclusively, the intersection of the architectures of the circumscribed products. The architecture of the product line is dominated by the common features. The two usual major reasons for doing product-line architectures are economies of scale and lowering the time and cost of developing new products within the product-line. Both objectives depend on being able to find things in common. Things in common can be produced in greater numbers (economies of scale). If there are many things in common, then the set of unique things that have to be developed to bring a new product to market is reduced and so the process should be faster and cheaper. Sometimes, though, we see work on product-line architectures trying to dictate things beyond the intersection of the individual architectures. In one sense only can the architecture of the product line be thought of as the union of the architectures of the products. This one exception to the "architecture-is-in-the-intersection" heuristic is the sense in which the product line defines the collection of niches into which each product will fit. The product line makes global decisions about where individual products can be developed, and where they cannot. In a product line of cars, or printers, or anything else we must decide where the clusters of related products will exist, and where they will *not* exist. The clusters should be placed where there is a nexus of related demand from users (a user or market driver) and where the builder has some proprietary advantage to offer.

New Products, New Markets

Of greatest risk are those form-first, technology-driven systems that create major qualitative changes in system-level behavior, changes in kind rather than degree. Systems of this type almost invariably require across-the-board new starts in design, development, and use. They most often arise when radically new technologies become available, such as jet engines, new materials, microprocessors, lasers, software architectures, and intelligent machines, and those technologies enable major changes to user operations. Although new technologies are infamous for creating unpleasant technological and even sociological surprises, by far, the greatest single risk in these systems is one of timing. Even if the form is feasible, introducing a new product either too early or too late can be punishing. As discussed in the DC-3 case study, Douglas Aircraft may not have beaten the Boeing 247 to market, but they were the first with the right set of features. In the next generation, Douglas Aircraft Company was too late into jet aircraft, losing out for years to The Boeing Company. Innumerable small companies have been too early, unable to sustain themselves while waiting for the technologies to evolve into engineered products. High-tech defense systems have suffered serious cost overruns and delays, most often due to a premature commitment to a critical new technology.

Technological Substitutions within Existing Systems

The second greatest risk is in not recognizing that before they are completed, technology-driven architectures will require much more than just replacing, one at a time, components of an older technology for those of a newer one. Painful experience shows that without widespread changes in the system and its management, technology-driven initiatives seldom meet expectations and too often cost more for less value. For example, direct replacements of factory workers with machines, of vacuum tubes with transistors, of large inventories with just-in-time deliveries, and of experienced analysts with computerized management information systems, all were disappointments when attempted by themselves in a system that was otherwise unchanged. Factory automation without architectural change has been a widely recognized failure. Failure rates of 50%–75% have been noted (Majchrzak 1988). Looking forward to the next case study, Lean Production is seen as a precondition to better use of automation, rather than an alternative (Womack et al. 2007). Technological substitution has been successful only when incorporated in concert with other matched and planned changes. It is not much of an exaggeration to say that the latter successes were well-architected, but the former failures were not. The DC-3 case study was an illustration. New technology in aircraft, without passing a threshold enabling operational change, was ineffective. And there had to be partners, airlines in the case of the DC-3, who were prepared to make those operational changes to exploit the new capability.

In automobiles, the most recent and continuing change is the insertion of ultraquality electronics and software between the driver and the mechanical subsystems of the car. This remarkably rapid evolution removes the driver almost completely from contact with, or direct physical control of, those subsystems. It considerably changes such overall system characteristics as fuel consumption, aerodynamic styling, driving performance, safety, and servicing and repair—as well as the design of such possibly unexpected elements as engines, transmissions, tires, dashboards, seats, passenger restraints, and freeway exits. From the 1990s to the 2000s, automotive control has moved decisively to computer interfaces, a trend evidently welcomed and used by the general public or it would not have been done. A telling indicator of the public's perception of automotive performance and safety was the virtually undisputed increase in national speed limits. Car crash death rates (per population, per motor vehicle, and per vehicle mile) have all dropped since the 1970s while speed limits have risen (Council 2022). Increasingly, the presence of Advanced Driver Assistance Systems (e.g., automatic braking) has been part of the improvement. Perhaps the most remarkable fact about this rapid evolution is that most customers were barely aware of it. This result came from a commitment to quality so high that a much more complex system could be offered that, contrary to the usual experience, worked far better than its simpler predecessor.

In aircraft, an equivalent, equally rapid, technology-driven evolution is "fly by wire," a change that, among other things, is forcing a social revolution in the role of the pilot and in methods of air traffic control. More is involved than the form-fit-function replacement of mechanical devices with a combination of electrical, hydraulic, and pneumatic units. Aerodynamically stable aircraft, which maintain steady flight with nearly all controls inoperative, are steadily being replaced with ones that are less stable, more maneuverable, and computer controlled in all but emergency conditions. The gain is a more efficient, potentially safer flight. But the transition has been as difficult as that between visual and instrument-controlled flight.

In inventory control, a remarkable innovation has been the very profitable combination in one system of point-of-sale terminals, of a shift of inventory to central warehouses and of just-in-time deliveries to the buyer. Note the word combination.

None of the components has been particularly successful by itself. The risk here is greater susceptibility to interruption of supply or transportation during crises.

In communications, satellites, packet switching, high-speed fiber-optic lines, e-mail, the World Wide Web, social media, and electronic commerce have combined for easier access to a global community, but with increasing concerns about privacy and security and the ease with which the technologies enables scale-up of various illegal activities and complex second and third order social impacts. The innovations now driving the communications revolution were not, individually, sufficient to create this revolution. It has been the interaction of the innovations, and the changes in business processes and personal habits connected to them, that have made the revolution.

In all of these examples, far more is affected than product internals. Affected also are such externals as manufacturing management, equity financing, government regulations, and the minimization of environmental impact, to name but a few. These externals alone could explain the growing interest by innovative builders in the tools and techniques of systems architecting. How else could well-balanced, well-integrated, financially successful, and socially acceptable total systems be created?

Consequences of Uncertainty of End Purpose

Uncertainty of end purpose, no matter what the reason, can have serious consequences. The most serious is the likelihood of serious error in decisions affecting system design, development, and production. Builder-architected systems are often solutions looking for a problem and hence are particularly vulnerable to the infamous "error of the third kind": working on the wrong problem.

Uncertainty in system purposes also weakens them as criteria for design management. Unless a well-understood basis for configuration control exists and can be enforced, system architectures can be forced off course by accommodations to crises of the moment. Some of the most expensive cases of record have been in attempts to computerize management information systems. Lacking clear statements of business purposes and market priorities, irreversible ad hoc decisions were made which so affected their performance, cost, and schedule that the systems were scrapped. Arguably, the best prevention against "system drift" is to decide on provisional or baseline purposes and stick to them. But what if those baseline purposes prove to be wrong in the marketplace?

ARCHITECTURE AND COMPETITION

In the classical architecting paradigm, there is little or no role for competition. The client knows what he or she wants, or learns through interaction with the architect. When a system is delivered that is consonant with the client's values, the client should be satisfied. If there were competitive offerings, then the client could have chosen one and avoided the expense of engaging with an architect. In many other cases, builder-architected systems prominent among them, success is judged more on competitive performance than on adherence to client values. Success in competition is certainly a significant surrogate for adherence to client values, but only partially.

To reconcile how architecting and architecture relate to competition, we must set the context of the organization's overall competitive strategy. Architecting cannot be talked about in the abstract; it has to be grounded in the strategies of the organization conducting it. In builder-architected systems, this means the competitive posture of the builder. Broadly speaking, we can identify three major competitive strategies with architectural consequences: disrupt and dominate, agile response, and attrition.

Disrupt and Dominate

This strategy is based on creating systems that disrupt existing operational patterns or markets and building barriers to prevent others from taking advantage of those disruptions. In the DC-3 case study, the DC-3 was a disruptive system in that it caused systematic change to how airlines did business. However, Douglas was unable to raise a strong barrier to prevent Boeing from entering the market space (although Douglas had a valuable lead of several years). The Apple iPod and iTunes music store combination is an example, or at least it was during the early 2000s, where patents, copyrights, secrecy of proprietary technologies, and exclusive contractual arrangements successfully formed barriers to competitive entry.

The architectural challenges in supporting this strategy are twofold. First, the quality of the architecting must be exceptional, as the architect must create beyond the boundaries of current systems. Great imagination is required, while simultaneously maintaining sufficient options (see the next section) to adapt to the inevitable failures of imagination. Second, the approach must allow protection from competitors who will employ an agile response strategy.

Agile Response

This strategy emphasizes the organization's capability to react more quickly and effectively than the competition. It is important to emphasize both speed and effectiveness because an ineffective response quickly delivered is still ineffective. A key distinction between the disrupt and dominate strategy and agile response is that agile response seeks to exploit the underlying flux in markets or military situations without disrupting their overall structure. An agile responder in a commercial environment produces new products within established markets faster and more effectively than the competition but does not try to create entirely new markets. The agile response strategy is especially effective in immature markets where changes in consumer preference and technology create many new opportunities. Put another way, it is less valuable to be first if whoever is first mostly helps customers discover what they actually want, and somebody else can then cover the revealed preference.

From an architectural perspective, the challenges for agile response are again twofold. First, to carry this strategy out effectively, the organization must be able to very rapidly conceive, develop, and deliver new systems. This means that architecting must be fast and must support a compressed development cycle. Second, at one higher level of abstraction, the architecture of the organization and its product lines must support agility. The organization and product lines must be structured to facilitate agility. Typically, the product-line architecture evolves much more slowly than the products and the product-line architecture sets out critical invariants, allowing rapid development and deployment. So use of project-line architectures for improving agility is effective primarily to the extent that the space for variation inside the product-line architecture is sufficient to cover the unknown demands.

Attrition

The classic example on the military side of the attrition strategy is to win by having more firepower, manpower, logistic power, and willingness to suffer than your opponent. A business equivalent strategy is to prevail through access to large amounts of low-cost capital, low-wage labor, and large distribution channels. When coupled with a strong organizational capability for learning and improvement, this is a powerful strategy, especially in mature markets where consumer preference changes slowly.

Architecting the attrition strategy is relatively slow and deliberate. The key architecture is the one embodied in the organization. Successful conduct of the attrition strategy is dependent on access to the requisite resources, cheaply and at a large scale. The strategy is likely to fail either when encountering a still larger and more fit competitor, or when the underlying environment (markets, operations, and technology) has an inherent rate of change high enough so that an agile response strategy becomes more effective, or when the change is sufficient to be open to disruption.

Reducing the Risks of Uncertainty of End Purpose

A powerful architecting guide to protect against the risk of uncertain purposes is to build in and maintain options. With options available, early decisions can be modified or changed later. A product-line architecture is a strategy for options, those options encapsulated in what the product-line enables. Other possibilities include the following: Build in options to stop at known points to guarantee at least partial satisfaction of user purposes without serious losses in time and money, for example, in databases for accounting and personnel administration. Create architectural options that permit later additions, a favorite strategy for automobiles and trucks. Provisions for doing so are often known as “hooks” in software to add applications and peripherals, “scars” in aircraft to add range and seats, “shunts” in electrical systems to isolate troubled sections, contingency plans in tours to accommodate cancelations, and forgiving exits from highways to minimize accidents.

In software, a general strategy is:

Use open architectures. You will need them once the market starts to respond.

As will be seen later, a further refinement of this domain-specific heuristic will be needed, but this simpler version makes the point for now.

Perhaps the biggest uncertainties are those barely visible because they are embedded in unexamined assumptions. The end of the Cold War was mostly a surprise, and its consequences ripple through to today. A massive infrastructure of National Security assumptions, built in huge institutions, reflected the belief that the Cold War would persist into the indefinite future. At the tactical level, many programs are built on poorly understood assumptions that sometimes are suddenly disrupted.

Risk Management by Intermediate Goals

Another strategy to reduce risk in the development of system-critical technologies is by scheduling a series of intermediate goals to be reached by precursor or partial configurations. At a more formal level, intermediate goals translate to program templates built on incrementalism (functional incremental spiral, risk-driven spiral, etc.). See Chapter 12 for additional details. Most programs have intermediate goals of some form as progress checkpoints. An incremental or risk-spiral program formalizes that into the structure of the program itself. As examples of building to intermediate goals, build simulators or prototypes to tie together and synchronize otherwise disparate research efforts. Build partial systems, demonstrators, or models to help assess the sensitivity of customer acceptance to the builder's or architect's value judgments, a widely used market research technique. As noted in the discussion of incremental and spiral programs, they also have traps. A paired trap is building the demonstrator that effectively resolves a high risk but doesn't look enough like the planned end-product to convince sponsors that progress has been made versus the flashy and convincing demonstration that dangerously elides the fact that it avoided dealing with the most serious technical risks. As will be seen in Chapter 7, if these goals result in stable intermediate forms, they can be powerful tools for integrating hardware and software.

Clearly, precursor systems have to be architected with the final product. If not, their failure in front of a prospective customer can play havoc with future acceptance and ruin any market research program. As one heuristic derived from military programs warns:

> The probability of an untimely failure increases with the weight of brass in the vicinity.

If precursors and demonstrators are to work well "in public," they better be well designed and well built.

Even if a demonstration of a precursor succeeds, it can generate excessive confidence, particularly if an untested requirement is critical. In one case, a U.S. Air Force (USAF) satellite control system successfully and very publicly demonstrated the ability to manage one satellite at a time; the critical task, however, was to control multiple, different satellites, a test it subsequently failed. Massive changes in the system as a whole were required. In another similar case, a small launch vehicle, arguably successful as a high-altitude demonstrator of single-stage-to-orbit, could not be scaled up to full size or full capability for embarrassingly basic mechanical and materials reasons.

These kinds of experiences led to the admonition: *Do the hard parts first.* Programmatically, this is almost always hard. Technically, it may not be hard at all, as when the "hard part" is a discrete piece of technology. It becomes both technical and programmatically difficult when the hard part is a unique function of the system as a whole. Such has been the case for a near-impenetrable missile defense system, a stealthy aircraft, a general aviation air traffic control system, a computer operating system, and a national tax reporting system. The only credible precursor, to demonstrate the hard parts, had to be almost as complete as the final product.

In risk management terms, if the hard parts are, perhaps necessarily, left to last, then the risk level remains high and uncertain to the very end. Leaving the highest risk to the end is the anti-thesis of the risk-driven spiral program template. The justification for the system therefore must be very high and the support for it very strong or its completion will be unlikely. For private businesses, this means high-risk venture capital. For governments, it means support by the political process, a factor in system acquisition for which few architects, engineers, and technical managers are prepared. Chapter 13 is a primer on the subject. When the authors have discussed this point with industry-experienced students, they frequently come up with what might be an anti-heuristic. Students state that, in their experience, pushing risk to late often happens because "doing the easy parts first" is how many programs show progress and build confidence. There is also an embrace of the sunk cost fallacy that once enough money has been invested, willingness to cancel goes down, even if the estimate of the cost-to-go hasn't changed that much because risks are still present.

THE "WHAT NEXT?" QUANDARY

One of the most serious long-term risks faced by a builder of a successful, technology-driven system is the lack of, or failure to win a competition for, a successor or follow-on to the original success.

The first situation is exemplified by a start-up company's lack of a successor to its first product. Lacking the resources in its early, profitless, years to support more than one research and development effort, it could only watch helplessly as competitors caught up and passed it by. Ironically, the more successful the initial product, the more competition it will attract from established and well-funded producers anxious to profit from a sure thing. Soon the company's first product will be a "commodity," something that many companies can produce at a rapidly decreasing cost and risk. Unable to repeat the first success, soon enough, the start-up enterprise fails or is bought up at fire-sale prices when the innovator can no longer meet payroll. This is common and sad but hard to avoid.

The second situation is the all-too-frequent inability of a well-established company that had been successfully supplying a market-valued system to win contracts for its follow-on. In this instance, the very strength of the successful system, a fine architecture matched with an efficient organization to build it, can be its weakness in a time of changing technologies and shifting market needs. The assumptions and constraints of the present architecture can become so ingrained in the thinking of participants that options simply do not surface.

In both situations, the problem is largely architectural, as is its alleviation.

For the innovative company, it is a matter of control of critical architectural features. For the successful first producer, it is a matter of knowing, well ahead of time, when purposes have changed enough that major rearchitecting may be required. Each situation will be considered in turn.

Controlling the Critical Features of the Architecture

The critical part of the answer to the start-up company's "what next" quandary is control of the architecture of its product through proprietary ownership of its basic features

(Alberthal et al. 1993, Morris and Ferguson 1993). This is the second half of a disrupt and dominate strategy. Examples of such features are computer operating systems, interface characteristics, communication protocols, microchip configurations, proprietary materials, patents, exclusive agreements with critical suppliers or distributors, and unique and expensive manufacturing capabilities. Good products, although certainly necessary, are not sufficient. They must also arrive on the market as a steadily improving product line, one that establishes, de facto, an architectural standard.

Surprisingly, one way to achieve that objective is to use the competition instead of fighting it. Because success invites competition, it may well be better for a start-up to make its competition dependent, through licensing, upon a company-proprietary architecture rather than to have it incentivized to seek architectural alternatives. Finding architectural alternatives takes time. But licensing encourages the competition to find new applications, add peripherals, and develop markets, further strengthening the architectural base, adding to the source company's profits and its own development base (Morris and Ferguson 1993). Heuristically: Successful architectures are proprietary, but open. "Open" here refers to being extensible by other than the originator, amenable to extension by other than the originator, though the proprietary nature implies an ability to control as well. Application frameworks on most major operating systems today allow those other than the operating system developer/owner to build applications on their own but do not cede control of the platform as a whole.

The computer industry has been a battleground of open source, open but proprietary, and closed architectures for decades. The greatest successes, if you measure by the market capitalization of the companies, have mostly been to very selective openness strategies. IBM made the PC hardware architecture very open, assuming they could dominate production, and were overwhelmed by the clone makers. They tried for years to reassert some level of proprietary control but could not succeed. Microsoft made sure that their operating systems would run on all of the clone hardware, and welcomed applications running on their software, but kept the operating system itself proprietary. Apple went the entirely proprietary route and fell behind, tried opening hardware in the 1990s (only to regret it), and later found extraordinary success with the largely closed architecture of the iPhone. At the same time, the open-source Internet infrastructure was sweeping away all proprietary network protocols, of which there used to be many, most now dead or restricted to small niches. Internet companies with huge market capitalizations (and obviously, there are many of them) did it based on application layer platforms running on the open infrastructure.

Decades earlier a different kind of architectural control was exemplified by the Bell telephone system with its technology generated by the Bell Laboratories, its equipment produced largely by Western Electric, and its architectural standards maintained by usage and regulation. Deregulation broke it up and the switch to cellular telephony finished the process. Others include Xerox in copiers, Kodak in cameras, and Hewlett-Packard in instruments. All these product-line companies began small, controlled the basic features, and prospered. But, as each of these also demonstrated, success is not forever. The ultimate failure of each of these was rooted in some combination of legacy or political change and a major technology shift leading to an operational shift that the incumbent could not, or would not, follow.

Thus, for the innovator, the essentials for continued success are not only a good product but also the generation, recognition, and control of its basic architectural features. Without these essentials, there may never be a successor product. With them, many product architectures, as architecturally controlled product lines, have lasted for years following the initial success. This adds even more meaning to the heuristic: *There's nothing like being the first success* (Rechtin 1991).

Abandonment of an Obsolete Architecture

A different risk-reduction strategy is needed for the company that has established and successfully controlled a product-line architecture and its market but is losing out to a successor architecture that is proving to be better in performance, cost, or schedule. There are many ways that this can happen. Perhaps the purposes that original architecture has satisfied can better be done in other ways. Typewriters were replaced by personal computers. Perhaps the conceptual assumptions of the original architecture no longer hold. Perhaps competitors found a way of bypassing the original architectural controls with a different architecture. Personal computers destroyed the market for Wang word processors and, eventually, for proprietary workstations. As a final example, cost risk considerations precluded building larger and larger spacecraft for the exploration of the solar system. Perhaps that example, too, will be superseded if the cost of large launch vehicles drops enough to change the risk posture in building very large spacecraft.

To avoid being superseded architecturally requires a strategy, worked out well ahead of time, to set to one side or cannibalize that first architecture, including the organization matched with it, and to take pre-emptive action to create a new one. One key move is the well-timed establishment of an innovative architecting team, unhindered by past success and capable of creating a successful replacement. Another key move is to make use of what they come up with, even when it interferes with past success. Just such a strategy was attempted by Xerox in a remake of the corporation as it saw its copier architecture start to fade and the rise of digital technologies. It declared redefinition as "the document company" and made extraordinarily innovative work (in new architectures) through the establishment of Xerox PARC (Palo Alto Research Center). While PARC was extraordinarily successful, the benefits were largely reaped by others rather than Xerox. While PARC was hardly a failure for Xerox, far greater profits were realized by others who exploited the inventions and architectures that came out of PARC. See Hiltzik (1999) for the now well-known story.

One can see PARC as a success, in that tremendous innovation was generated without destroying the old, successful architecture (Spinrad 1992). One can see it as a failure in that much of the value was realized by others when the legacy organization was capable of only changing so far. Xerox understood the necessity of making the architectural transition, and invested in it, for many years before being organizationally capable of actually making the transition. An important element of this was the problem of shifting operational concept, or failure to do so. For example, Xerox pioneered delivering an end-to-end system for desktop publishing. In the markets, Xerox was effectively serving at the time (most large organizations), they already had related capabilities, albeit far more manual. Manual or not, the capability was not essentially new, and so we had to make a different case to make headway. In contrast, in the small-business and individual market, there was very little such capability. As Apple Macintosh and other personal computer-based capabilities arrived a few years later they were entering that market where there was little legacy to compete against. The result, for them, was a dramatic success as they enabled capabilities in markets where there had been very little.

CREATING INNOVATIVE TEAMS

Clearly, the personalities of members of any team, particularly an innovative architecting team, must be compatible. A series of USC Research Reports by Jonathan Losk, Tom Pieronek, Kenneth Cureton, and Norman P. Geis, summarized in Geis (1993), based on the Myers-Briggs Type Indicator (MBTI) (Isabel Briggs Myers et al. 1998), strongly suggest that the preferred personality type for architecting team membership is NT. That is, members should tend toward systematic and strategic analysis in solving problems. As Cureton summarizes, "Systems architects are made and not born, but some people are more equal than others in terms of natural ability for the systems architecting process, and MBTI seems to be an effective measure of such natural ability. No single personality type appears to be the 'perfect' systems architect, but the INTP personality type often possesses many of the necessary skills."

Their work also shows the need for later including an ENTP (extroversion, intuition, thinking, and perceiving), a "field marshal" or deputy project manager, not only to add some practicality to the philosophical bent of the INTPs (introversion, intuition, thinking, perceiving) but to help the architecting team work smoothly with the teams responsible for building the system.

Creating innovative teams is not easy, even if the members work well together. The start-up company, having little choice, depends on good fortune in its recruiting of charter members. The established company, to put it bluntly, has to be willing to change how it is organized and staffed from the top down based almost solely on the conclusions of a presumably innovative team of "outsiders," albeit individuals chartered to be such. The charter is a critical element, not so much in defining new directions as in defining freedoms, rights of access, constraints, responsibilities, and prerogatives for the team. For example, can the team go outside the company for ideas, membership, and such options as corporate acquisition? To whom does the team respond and report—and to whom does it not? Obviously, the architecting team better be well designed and managed. Remember, if the team does not succeed in presenting a new and accepted architecture, the company may well fail.

One of the more arguable statements about architecting is the one by Frederick P. Brooks Jr. and Robert Spinrad that the best architectures are the product of a single mind. For modest-sized projects, that statement is reasonable enough. As projects get larger and larger, it remains true but changes form. The complexity and workload of creating large, multidisciplinary, technology-driven architectures would overwhelm any individual. The observation of a single mind is most easily accommodated by a simple but subtle change from "a single mind" to "a team of a single mind." Some would say "of a single vision" composed of ideas, purposes, concepts, presumptions, and priorities. It is also critical to understand the difference between composing multidisciplinary teams and how teams form decisions. The key to a coherent architecture is coherent decision making. Majority votes by large committees are practically the worst-case scenario for gaining coherence of decision making over a long series of related complex decisions.

One architect put the issue succinctly. When asked about the role of multidisciplinary teams, he said: "Multi-disciplinary teams covering all stakeholders and major subsystem areas are critical to effective space architecting, and I love using them. As long as I get to make all of the decisions." His point was simple—good architecting requires diversity of view but unity of decision.

Another way to think about what "single mind" can practically mean is through what teams really value from individuals. It is common on complex projects with large teams for there to be a dependence on a very small cadre who "understand the whole system." Talk to one of them and you realize they don't really understand the whole system, in the sense of all of the details all the way down. What they have is a form of "T-shaped" understanding. What sets them apart is understanding the whole, usually in terms of whole-system input-output threads or whole-system performance factors, coupled with in-depth understanding of how the end-to-end threads and performance factors come about. Usually that doesn't mean what happens to each bit on the chain, but it usually does mean understanding all the steps, at least logically. It is a true whole system perspective rather than subsystem or disciplinary perspective. Note that a systems engineer on a large program can be just as stovepiped in understanding as a mechanical or software engineer if all that the systems engineer understands is requirements, or reliability or some other system-level stovepipe.

In the simplest case, the single vision would be that of the chief architect and the team would work to it. For practical as well as team cohesiveness reasons, the single vision needs to be a shared one. In no system is that more important than in the entrepreneurially motivated one. There will always be tension between the more thoughtful architect and the more action-oriented entrepreneur. Fortunately, achieving balance and compromise of their natural inclinations works in the system's favor.

An important corollary of the shared vision is that the architecting team, and not just the chief architect, must be seen as creative, communicative, respected, and of a single mind about the system-to-be. Only then can the team be credible in fulfilling its responsibilities to the entrepreneur, the builder, the system, and its many stakeholders. Internal power struggles, basic disagreements on system purpose and values, and advocacies of special interests can only be damaging to that credibility.

As Ben Bauermeister, Harry Hillaker, Archie Mills, Bob Spinrad, and other friends have stressed in conversations with the authors, innovative teams need to be cultural in form, diverse in nature, and almost obsessive in dedication.

By cultural is meant a team characterized by informal creativity, easy interpersonal relationships, trust and respect, all characteristics necessary for team efficiency, exchange of ideas, and personal identification with a shared vision. To identify with a vision, they must deeply believe in it and in their chief. The members must acknowledge and follow the lead of their chief or the team disintegrates.

Diversity in specialization is to be expected; it is one of the reasons for forming a team. Equally important, a balanced diversity of style and programmatic experience is necessary to assure open-mindedness, to spark creative thinking in others, and to enliven personal interrelationships. It is necessary, too, to avoid the "groupthink" of nearly identical members with the same background, interests, personal style, and devotion to past architectures and programs. Indeed, team diversity is one of the better protections against the second-product risks mentioned earlier.

Consequently, an increasingly accepted guideline is that to be truly innovative and competitive in today's world: *The team that created and built a presently successful product is often the best one for its evolution—but seldom for creating its replacement.*

A major challenge for the architect, whether as an individual or as the leader of a small architectural team, is to maintain dedication and momentum not only within the team but also within the managerial structure essential for its support. The vision will need to be continually restated as new participants and stakeholders arrive on the scene—engineers, managers active and displaced, producers, users, and new clients. Even more difficult, it will have to be transformed as the system proceeds from a dream to a concrete entity, to a profit maker, and finally to a quality production. Cultural collegiality will have to give way to the primacy of the bottom line and, finally, to the necessarily bureaucratic discipline of production. Yet the integrity of the vision must never be lost, or the system will die.

The role of organizations in architectures, and the architecture of organizations, is taken up at much greater length by one of the present authors (Rechtin 2017).

ARCHITECTING "REVOLUTIONARY" SYSTEMS

A distinction to be made at this point is between architecting in precedented, or evolutionary, environments, and architecting unprecedented systems. Whether we call such systems "revolutionary," "disruptive," or "unprecedented" seems more a matter of fashion. What is important is that the system stands apart from all that came before it, and that is great change of businesses or militaries operate. One of the most notable features of Rechtin (1991) was an examination of the architectural history of clearly successful and unprecedented systems. A central observation is that all such systems have a clearly identifiable architect or small architect team. They were not conceived by the consensus of a committee. Their basic choices reflect a unified and coherent vision of one individual or a very small group. Further reflection, and study by students, has only reinforced this basic conclusion, while also filling in some of the more subtle details. The case study that opened this chapter is a fine example.

Unprecedented systems have been both purpose-driven and technology-driven. In the purpose-driven case, the architect has sometimes been part of the developer's organization and sometimes not. In the technology-driven case, the architect is almost always in the developer's organization. This should be expected as technology-driven systems typically come from intimate knowledge of emerging technology, and someone's vision of where it can be applied to advantage. This person is typically not a current user but is rather a technology developer. It is this case that is the concern of this section.

The architect has a lead technical role. But this role cannot be properly expressed in the absence of good project management. Thus, the pattern of a strong duo, project manager and system architect, is also characteristic of successful systems. In systems of significant complexity, it is very difficult to combine the two roles. A project manager is typically besieged by short-term problems. The median due date of things on the project manager's desk is probably yesterday. In this environment of immediate problems, it is unlikely that a person will be able to devote serious time to longer-term thinking and broad communication that are essential to good architecture.

The most important lesson in revolutionary systems, at least those not inextricably tied to a single mission, is that success is commonly not found where the original concept thought it would be. The Macintosh computer was a success because of desktop publishing, not what the market assumed in its original rollout (which was as a personal information appliance). Indeed, desktop publishing did not exist as a significant market when the Macintosh was introduced. As noted in the discussion on Xerox PARC, desktop publishing had been demonstrated several years before the introduction of the Macintosh computer, and a product-line was available that provided it. But, that product-line was very expensive and targeted at large organizations where the need was not apparent and had not sold well. This pattern of new systems becoming successful because of new applications has been common enough in the computer industry to have acquired a nickname, "the killer app(lication)." Taken narrowly, a "killer app" is an application so valuable that it drives the sales of a particular computer platform. Taken more broadly, a "killer app" is any new system usage so valuable that, by itself, it drives the dissemination of the system.

One approach to unprecedented systems is to seek the killer application that can drive the success of a system. A recent noncomputer example that illustrates the need, and the difficulty, is the search for a killer application for reusable space launch vehicles. Proponents believe that there is a stable economic equilibrium with launch costs an order of magnitude lower, and flight rates around an order of magnitude higher, than current. But, if flight rates increase and space payload costs remain the same, then total spending on space systems will have to be far higher (roughly an order of magnitude, counting only the payload costs). For there to be a justification for high flight rate launch, there has to be an application that will realistically exploit it. That is, some applications must attract sufficient new money to drive up payload mass.

Various proposals have been floated, including large constellations of communication satellites, space power generation, and space tourism. If the cost of payloads was reduced at the same time, their flight rate might increase without total spending going up so much. But the only clear way of doing that is to move to much larger-scale serial production of space hardware to take advantage of learning curve cost reductions. This clearly indicates a radical change to the architecture not only of launch, but to satellite design, satellite operations, and probably to space manufacturing companies as well. And all these changes need to take place synchronously for the happy consequence of lowered cost to result.

Now is not the first time the arguments on flight rate and economies of scale for space systems have been floated. They were an important part of the Space Shuttle discussion, and the numbers did not work out (Rechtin 1983). In the late 1990s, there was a burst of excitement about this approach, driven by multiple proposals for large constellations of communication satellites, first the Iridium system and then the proposals for Teledesic, Celestri, and others. But Iridium went bankrupt (being pulled from bankruptcy by a deal with the U.S. Department of Defense as an anchor customer), and the others were canceled. At the time of the writing (early 2020s), there is a new burst of excitement driven by the success of SpaceX in reusable boosters and the very large Starlink constellation going into operation. Relative to the point above, proliferated low earth orbit constellations for communications are driving serial production and lower payload costs per kilogram. The question is, is there a case beyond communications satellites? It still seems that something else with more mass, and associated revenue opportunities, than proliferated communication satellites will be necessary to drive high flight rate on large boosters. Other proliferated constellations are under development, in both commercial and government operational areas, but something much larger in mass and revenue still seems to be necessary.

Such synchronized changes have occurred in other industries. The semiconductor industry has experienced decades of 40% annual growth because such synchronized changes have become ingrained in the structure of the computer industry. As the production and design technology improve, the total production base (in transistor quantity and revenue) goes up. Lowered unit costs result in increased consumption of electronics even larger than the simple scale-up of each production generation. The resulting revenue increases are sufficient to keep the process going, and coordinated behavior in the production equipment supplier, design system supplier, and consumer electronic producers smooths the process sufficiently for it to run stably for decades.

In summary, the successful architect exploits what the market demonstrates as the killer application, assuming he or she can predetermine it. The successful innovator exploits the first-to-market position to take advantage of the market's demonstration of what it really wants faster than the second-to-market player does. The successful follower beats the first-to-market by being able to exploit the market's demonstration more quickly. Each is making a consistent choice of both strategy and architecture (in a technical sense). We explore this issue in depth in Chapter 12.

SYSTEMS ARCHITECTING AND BASIC RESEARCH

One other relationship should be established that between architects and those engaged in basic research and technology development. Each group can further the interests of the other. The architect can learn without conflict of interest. The researcher is more likely to become aware of potential sponsors and users.

New technologies enable new architectures, though not singly or by themselves. Consider solid-state electronics, fiber optics, software languages, and molecular resonance imaging for starters. Innovative architectures can provide the rationale for underwriting research, often at a very basic level. Yet, though both innovative architecting and basic research explore the unknown and unprecedented, there seems to be little early contact between their respective architects and researchers. The architectures of intelligent machines, the chaotic aerodynamics of active surfaces, the sociology of intelligent transportation systems, and the resolution of conflict in multimedia networks are examples of presumably common interests. Universities might well provide a natural meeting place for seminars, consulting, and the creation and exchange of tools and techniques.

New architectures, driven by perceived purposes, sponsor more basic research and technology development than is generally acknowledged. Indeed, support for targeted basic research undoubtedly exceeds that motivated by scientific inquiry. Examples abound in communications systems that sponsor coding theory, weapons systems that sponsor materials science and electromagnetics, aircraft that sponsor fluid mechanics, and space systems that sponsor the fields of knowledge acquisition and understanding.

It is therefore very much in the mutual interest of professionals in research and development (R&D) and systems architecting to know each other well. Architects gain new options. Researchers gain well-motivated support. Enough said.

HEURISTICS FOR ARCHITECTING TECHNOLOGY-DRIVEN SYSTEMS

General

- An insight is worth a thousand market surveys.
- Success is defined by the customer, not by the architect.
- In architecting a new program, all the serious mistakes are made in the first day.
- The most dangerous assumptions are the unstated ones.
- The choice between products may well depend upon which set of drawbacks the users can handle best.
- As time to delivery decreases, the threat to user utility increases.
- If you think your design is perfect, it is only because you have not shown it to someone else.
- If you do not understand the existing system, you cannot be sure you are building a better one.
- Do the hard parts first.
- Watch out for domain-specific systems. They may become traps instead of useful system niches, especially in an era of rapidly developing technology.
- The team that created and built a presently successful product is often the best one for its evolution—but seldom for creating its replacement. (It may be locked into unstated assumptions that no longer hold.)

Specialized

From Morris and Ferguson (1993):

- Good products are not enough. (Their features need to be owned.)
- Implementations matter. (They help establish architectural control.)
- Successful architectures are proprietary, but open. (Maintain control over the key standards, protocols, etc., that characterize them but make them available to others who can expand the market to everyone's gain.)

From Chapters 2 and 3

- Use open architectures. You will need them once the market starts to respond.

CONCLUSION

Technology-driven, builder-architected systems, with their greater uncertainty of customer acceptance, encounter greater architectural risks than those that are purpose-driven. Risks can be reduced by the careful inclusion of options, the structuring of their innovative teams, and the application of heuristics found useful elsewhere. At the same time, they have lessons to teach in the control of critical system features and the response to competition enabled by new technologies.

EXERCISES

1. The architect can have one of three relationships with the builder and client. The architect can be a third party, can be the builder, or can be the client. What are the advantages and disadvantages of each relationship? For what types of system is one of the three relationships necessary?
2. In a system familiar to you, discuss how the architecture can allow for options to respond to changes in client demands. Discuss the pros and cons of product versus product-line architecture as strategies in responding to the need for options. Find examples among systems familiar to you.
3. Architects must be employed by builders in commercially marketed systems because many customers are unwilling to sponsor long-term development; they purchase systems after evaluating the finished product according to their then-perceived needs. But placing the architect in the builder's organization will tend to dilute the independence needed by the architect. What organizational approaches can help to maintain independence while also meeting the needs of the builder organization?
4. The most difficult type of technology-driven system is one that does not address any existing market. Examine the history of both successful and failed systems of this type. What lessons can be extracted from them?

NOTES

1 Chief architect, General Dynamics YF-16, which became the F-16 Fighter. As stated in a University of Southern California (USC) Systems Architecting lecture, November 1989.

2 Not to be confused with "reference architectures" in the Standards sense, taken up in Chapter 11.

REFERENCES

Alberthal, L., et al. (1993). "Will architecture win the technology wars?" *Harvard Business Review* **71**(3): 160, 163, 166–170.

Council, N. S. (2022). "Motor Vehicle Safety." Retrieved 1-July, 2024, from injuryfacts.nsc.org/motor-vehicle/overview/introduction.

Geis, N. P. (1993). Profiles of Systems Architecting Teams. AE549 Research Reports. E. Rechtin. Los Angeles, CA.

Hiltzik, M. (1999). *Dealers of Lightning: Xerox PARC and the Dawning of the Computer Age*, New York: HarperCollins.

Isabel Briggs Myers, M. M., N. Quenk, and A. Hammer (1998). *MBTI Manual: A Guide to the Development and Use of the Myers-Briggs Type Indicator*, Dallas, TX: Consulting Psychologists Press.

Maier, M. W. (2019). "Architecting portfolios of systems." *Systems Engineering* **22**(4): 335–347.

Majchrzak, A. (1988). *The Human Side of Factory Automation: Managerial and Human Resource Strategies for Making Automation Succeed*, San Francisco, CA: Jossey-Bass/Wiley.

Morris, C. R. and C. H. Ferguson (1993). "How architecture wins technology wars." *Harvard Business Review* **71**(2): 86–96.

Rechtin, E. (1983). *A Short History of Shuttle Economics. NRC Reports*, Washington, DC: National Research Council.

Rechtin, E. (1991). *Systems Architecting: Creating and Building Complex Systems*, Englewood Cliffs, NJ: Prentice Hall.

Rechtin, E. (2017). *Systems Architecting of Organizations: Why Eagles Can't Swim*, Oxfordshire: Routledge.

Spinrad, R. J. (1992). *Lecture at the University of Southern California*, Los Angeles, CA: University of Southern California.

Womack, J. P., D. T. Jones, and D. Roos (2007) *The Machine that Changed the World: The Story of Lean Production—Toyota's Secret Weapon in the Global Car Wars that is Now Revolutionizing World Industry*, New York: Simon and Schuster.

Case Study 3

Mass and Lean Production

INTRODUCTION

Today, mass production is pervasive. Everything from cars to electronics is made in quantities of hundreds of thousands to millions. From the perspective of the 19th and early 20th centuries, products of extraordinary complexity are made in huge numbers. The story of mass production is significantly an architectural story. It is also a story of the interaction of architectures, in this case, the interaction and synergy between the architectures of system-products and systems that built those products. The revolution that took place in production was dependent on changes in how the produced systems were designed, and design changes had synergistic effects on production. The characteristics and structures of the surrounding human systems were also critical to the story, notions that we will take up in later chapters. In the later 20th century, the classical notion of mass production has been challenged by the methods of lean production. Lean is clearly an evolution of mass, in that both are methods of production at a similar scale of similar products. But lean production is a deep evolution, one that challenges essential principles and requires an architectural remaking.

This case study is a high-level survey that emphasizes the sweep of changes over time instead of details and the nature of architectural decision making in mass production. We start by reviewing the history of mass production, from an architecture perspective, focusing on the auto industry. We survey from the era of auto production as a cottage industry, through the development of mass production by the Ford Motor Corporation, to the era of competition from other U.S. manufacturers, and end with the development of the Toyota Production System (TPS). We strongly follow the history laid out in Sorensen and Williamson (2006), Womack et al. (2007), Ohno (2019), and Liker (2020), but with an interpretation in terms of the systems architecture concepts of this book.

AN ARCHITECTURAL HISTORY OF MASS PRODUCTION

The auto industry is hardly the only example of mass production, but it is usually considered as prototypical. The innovations in production at Ford, and later Toyota, substantially define the basic structures of modern mass production. The Ford system of production became the model for industry after industry, and the concepts filtered into society at large. The Toyota Production System is the prototype for Lean Production, now likewise a fundamental paradigm for organization in multiple industries, increasingly including service industries.

DOI: 10.1201/9781003483595-8

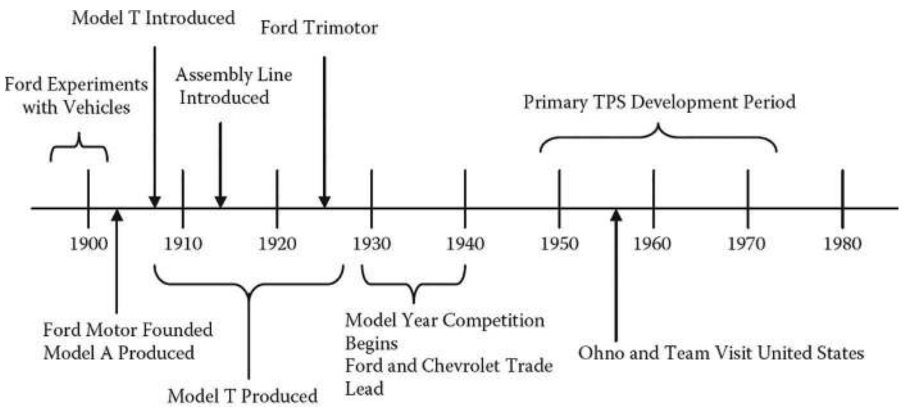

Model T Introduced
Ford Trimotor
Ford Experiments with Vehicles
Assembly Line Introduced
Primary TPS Development Period
1900
1910
1920
1930
1940
1950
1960
1970
1980
Ford Motor Founded
Model A Produced
Model T Produced
Model Year Competition Begins
Ford and Chevrolet Trade Lead
Ohno and Team Visit United States

FIGURE CS3.1 Key events in the architecture of automobile mass production.

In the sections following, we cover major blocks of time and consider how decisions about basic organizing structure of production were synergistic (or antagonistic) with how systems were designed. For convenience, refer to Figure CS3.1 for the sequencing and relationship of events.

Cottage Industry (1890s to 1900s)

As auto production began in the 1890s, it was a classic cottage industry. Small groups of workers assembled each vehicle in a shop. The process involved bringing in a stream of parts (or machining them locally) and assembling them as a small team in one place. Parts were not really interchangeable, each part typically required hand work to be fitted with others into assemblies. When the vehicle was complete, it was driven or otherwise moved away.

Automobiles built this way were very expensive. High prices and the small market went hand-in-hand and were strongly coupled with low production. Because the vehicles were expensive, they were a luxury item with a very narrow customer base. Because the market was small, economies of scale were limited and so prices were high.

Henry Ford understood the problem very well and was personally convinced that the way forward was in lower prices and larger production. He developed a conviction that high-quality automobiles could be, and should be, produced at cost low enough for average people to afford. The "car for the masses" would revolutionize society. He clashed repeatedly with his business partners over this, as they were convinced higher profits could be realized by concentrating on more expensive, high-margin vehicles. Over the short run, his partners were almost certainly right. Over the long run, the situation in automobiles was analogous in some ways to the situation in commercial aircraft just before the introduction of the DC-3 discussed in the case study before Chapter 3. The introduction of a new system would create a qualitative change in the structure of the market (and drive structural, architectural change in both production and systems).

Birth of Mass Production (1908–1913)

Ford's dream of a car for the masses was realized with the famous Model T. The Model T was introduced in 1908 and was eventually produced in numbers vastly greater than any car previously. For the first few years, it was produced at the Ford Piquette Avenue plant, Detroit, Michigan. In terms of this book, in the early days, Henry Ford was both sponsor and architect, although his role was clearly more sponsor as time went on (Sorensen and Williamson 2006). The architecture established at this point would become a decades-long invariant, although (Womack et al. 2007) point out how there was a substantial change from the first iteration to the later highly vertically integrated form. Ironically, the earliest forms bear significant relationships to the more flexible forms found in lean production. The architecting was done by a very small group, with leading credit probably best given to Charles Sorensen, at least according to Sorensen (Sorensen and Williamson 2006), with others having key roles. Sorensen had primary responsibility for the production system, with several others individually having leadership in other basic elements of the Ford production system architecture. According to Sorensen, the first experiments in the production line took place at the Piquette Avenue plant in mid-1908 on the Model N, an immediate predecessor to the Model T.

The Model N had been introduced as an incomplete prototype at the 1906 Detroit auto show. It was not disclosed that the show car was incomplete, and so the announced price of $500 was a sensation and generated terrific demand. The Model N demonstrated the latent demand for a solid, low-cost car. The Model T, with its superior engineering for production, was able to exploit that demand.

As Sorensen recounts, he and a small team spent Sundays during the summer of 1908 experimenting on the production floor of the Piquette Avenue plant. They laid out the parts required for a car from one end of the long narrow building floor to the other. They mounted a frame on skids and then dragged the skid down the floor, stopping along the way to add the parts that had been preplaced.

As an amusing aside, and as a wonderful indication of how obvious things go unnoticed when great innovations are made, Sorensen points out why the assembly line model was not actually used in production until 5 years later, in 1913. The main problem was that at the Piquette Avenue plant, the assembly floor was the third floor of the building, the top floor. In retrospect, this is laughable. Why put the place where you need to move all the production parts to and from three floors up off the ground? But in the early 1900s, this did not seem so obvious. When you make only a few cars, why put that messy operation on the ground floor, which has the nicer space for the staff (including sales)?

Once the Model T was introduced, and demand immediately exploded beyond the capacity of the Piquette Avenue plant, Ford built an all-new plant at Highland Park, Michigan, where the assembly line was brought to fruition in 1913. As we shall see in a later section of this case study, there is more to the structure of the Ford system than the assembly line, and those other structural elements play at least as important a role.

Competition from New Quarters (1920s to 1930s)

The Model T and its production system were based on a simple, virtuous cycle. Lowering costs allowed prices to be lowered, which increased sales and production, which enabled greater economies of scale, which lowered costs. Ford believed in the primacy of this virtuous loop, and other considerations were subordinated to maximizing production to gain economies of scale. Ford's pursuit of the Model T was driven by an innate belief in the value of a car for the masses. The vision was eventually overturned by an alternative vision spawned by market forces.

By the mid-1920s, Chevrolet was rapidly catching up to Ford in production numbers. They were catching up primarily by making better-looking, more-exciting cars, and marketing looks and excitement. Although the Model T was a very solid car, a new era had begun, based on market penetration of automobiles being large enough so that people began to see them as partially fashion-driven goods. When market penetration for automobiles became high, the purely utilitarian aspect of automobile ownership began to be replaced by automobiles as status symbols. When status played an important role, it quickly became the case that status was no longer conveyed simply by having a car, but by the car one had.

Model T production was shut down in 1927. Over the next decade, competition between Ford and its competitors (most famously General Motors, also Plymouth and Chrysler) moved to the model-year change system. Different models were produced for different market segments, and those models were regularly changed in external style and engineering features. The changes were synchronized with marketing campaigns to drive demand. Economies of scale in mass production were still of great importance, but the scale was not unlimited. The Model T had tested the outer envelope of focusing purely on cost reduction through scale and was displaced by a more complex mixture of engineering, production, and marketing.

The Toyota Production System (1940s to 1980s)

The development of the Toyota Production System (TPS) (Ohno 2019) can be said to have revolutionized manufacturing as did Ford's mass production system. Although the revolution was slower and less dramatic, it was in some ways more surprising as it occurred in an industry already apparently mature. By the 1950s, the automobile business appeared mature. Cars were much improved, but their architecture had changed little in decades, and the architecture of production likewise changed little. The revolution of the TPS has no dramatic moments like the assembly experiments at Piquette Avenue. The TPS revolution was a revolution by evolution, a case where incrementally change, accreted steadily enough and long enough, it takes on a qualitatively different flavor. Those incremental changes were driven, however, by more basic insights.

The TPS did not outwardly change the architecture of either cars or production. Both cars and factories built in accordance with TPS appear much the same as did cars and factories before. But the improvements in quality and cost brought Toyota from a nonentity in the business to a neck-and-neck contender for the largest auto manufacturer. This displacement of multiple, dominant, profitable firms is very unusual.

The architecture of the TPS is The Toyota Way (see below). Thus, the TPS is a sociotechnical system, and its architecture is likewise more social than technical. The most important elements are the shared principles and the means of their application.

METAPHOR OR VISION CHANGES

At each of the stages, the story is captured by a metaphor or basic vision. It is hard to know exactly how important the conceptual vision is, but the testimony of the people directly involved indicates that the coherent vision, the thing they could aim at, was an inspiration and guide, and they gave it great weight. Sorensen reports repeatedly that Henry Ford was devoted to his vision of cars for the masses, and his reluctance to recognize that it had run out of force caused great difficulty when it finally became obvious to everybody except Henry Ford that the time of the Model T was past. Ohno likewise speaks repeatedly of how various metaphors drove his thinking in how to re-organize production.

Craftsmen

Early automobiles were craftsmanly products, like bespoke suits. They were made by individual craftsmen and possessing one was a mark of status. Originally, there was no other choice, parts not being sufficiently interchangeable forced craft work at every stage of production. Being made by individual craftsmen, they carried the marks of those craftsmen (sometimes good, sometimes bad). Like nearly all craftsmanly products, these cars were very expensive.

The craftsmanly approach to cars is still not quite dead. A few cars, naturally very expensive and basically toys for adults, are built by individual teams of craftsman. The individual attention is a selling point, even if it objectively probably yields poorer quality than the best cars made in lean factories. As (Womack et al. 2007) point out, the pull of the craftsman ideal persisted for many decades and remained a barrier to full implementation of mass production, and later lean production, in some companies and market areas all the way to the end of the 20th century.

A Car for the Masses, or If We Build It, It Will Sell

Henry Ford's most famous quote is probably "The customer can have any color he wants, as long as it is black." Black was apparently chosen mostly because the high-quality black paint of the time was the fastest drying and thus allowed the production line to operate more efficiently. The paradigm for Ford operations from the introduction of the Model T to the mid-1920s was that the only real problem was making more Model Ts, cheaper. If they could be made, they could be sold or so the belief ran. This was the virtuous cycle of economies of scale and cost reductions. For roughly 15 years, this was an effective strategy and reflected the (temporary) correctness of Henry Ford's basic vision.

Cars as Fashion

By the mid-1920s, cars were no longer a rarity in the United States. There were enough reliable cars around that a used car market had begun. As Chevrolet and others introduced frequent style and model changes, they brought a fashion sensibility to automobiles. Henry Ford's simple vision of cheap transportation for the masses gave way to affordable status and transportation for the masses, and eventually a whole hierarchy of desire and status much like other mature product areas.

The Supermarket Metaphor and "Pull" Production

In Taiichi Ohno's book on the Toyota Production System, he makes a striking observation about his inspiration for the TPS. He says that when he toured the United States in 1956 to see the Ford and General Motors factories, he was more impressed by supermarkets. He adopted a supermarket metaphor for the organization of production. The idea was that the consumer (who in a production system is also a supplier to a later phase) can reach into the supermarket and get exactly what he or she needs, and the act of the consumer taking it "pulls" a replacement onto the shelf. The metaphor captures the idea of "pull" being the controlling factor in inventory and allowing inventory to match needs, becoming a natural way to control production flow. In contrast to Henry Ford's paradigm of pushing automobiles out, knowing they would be sold, the TPS model is to produce and deliver just what is sold and refill just what is taken. Ohno writes that the supermarket metaphor had been in use since the late 1940s, but his trip to the United States solidified his commitment to the metaphor.

The Toyota Way

Beyond the supermarket metaphor, Toyota promulgates a larger philosophy known as "The Toyota Way." The Toyota Way (Liker 2020) could be thought of as a metaphor or vision in the large, composed as it is of 14 principles that themselves are reasonably complex. The Toyota Way defines an overall approach to doing a production-oriented business in general and is not restricted to automobiles. It does not have a distinct end point (as Ford's vision did); rather, one of the principles is to embrace a sense of urgency for continuous improvement, regardless of current business conditions. The Toyota Way is, by design, a more embracing philosophy than a single vision. The 14 principles identified by Liker are essentially heuristics, as discussed in this book or are easily translated into such heuristics.

ELEMENTS OF THE ARCHITECTURE OF THE FORD PRODUCTION SYSTEM

The architecture of Ford mass production was not just the assembly line, or the River Rouge factory (Dearborn, Michigan), or the Model T. The architecture of the enterprise as a whole, the architecture that brought mass production its power, had four major components: Parts interchangeability, the production line, distributed production with synergistic system design, and management processes.

Parts Interchangeability

Womack points out that the true pre-condition for Ford's creation of mass production was the ability to build parts that were completely interchangeable (did not require manual rework as part of the assembly process). This required both technical and managerial advancements. Examples on the technical side include the development of machine tools capable of forming parts after hardening treatment rather than hardening (and thus distorting) parts already made. The management side involved things like standardization of measurement, specification, and inspection. With full parts interchangeability the assembly process could be simplified and standardized and broken down into steps small enough to allow for rolling assembly and the assembly line.

The Assembly Line

By far the most famous element of the mass production enterprise is the assembly line. As noted above, the experiments in fixing assembly stations and moving the vehicle down the factory floor began with the Model N in the Piquette Avenue plant. The physical constraints of the plant prevented full implementation until the Highland Park plant was built to produce the Model T.

The assembly line also led to a variety of other possibilities for efficiencies. Once the basic notion of configuring the flow to optimize material handling was present, the full power of engineering and statistics could be brought to bear to further improve the process. Moreover, assembly production should be (and eventually would be) synergistic with design. Automobiles eventually were designed to be easy to assemble within the Ford enterprise, and the enterprise adjusted itself to what it was possible to design.

Enterprise Distribution

The assembly line was just one of the major innovations that enabled mass production. As production volumes grew larger and larger the problem of factory scaling began to appear. There are upper limits to the practical size of a factory. Eventually, the major constraint is transportation. A factory in the Detroit area (or anywhere else) simply cannot bring arbitrarily large quantities of raw materials and parts and cannot move out arbitrarily large quantities of finished products. Eventually, transportation capacity runs out.

So, when it is necessary to build more factories in geographically distributed locations, how do we divide up the production tasks? The solution eventually arrived at in automobiles is to divide production along vehicle subsystem lines. So, engines are made in one location, chassis in another, bodies in still another, and all are brought together in assembly plants. The assembly plants can be located relatively close to major markets, and the others can be distributed based on what areas are favorable to the particular manufacturing task. The distribution of factories, especially final assembly, also came to match political constraints. Some work needed to be done in end-markets abroad, and systems of different tariffs on different levels of assembly drove aspects of enterprise distribution.

This division on vehicle subsystem lines is, or can be, synergistic with vehicle design. Design should be synergistic both with the detailed problem of assembly and the larger problem of how the production enterprise is distributed. For example, tight tolerance processes should be inside subsystems, and the interfaces between them should be less demanding. The subsystems should be designed in ways that facilitate testing and quality control at the point of production. Over time, the production processes, vehicle designs, and supplier networks coevolved.

Management Processes

The assembly line, distribution of plants, and vehicle subsystems are all obviously physical structures. But history also identifies certain management processes and the synergistic changes they drove as fundamental structural elements (that is, architectural elements) in the development of mass production.

Quality Assurance for Distributed Production

Consider how quality assurance and quality control change when production becomes distributed. If all production steps are under the same roof, when a problem appears, an engineer can simply walk from one part of the factory to another to understand the source of the problem. When the engine, frame, and transmission factories were in different parts of the United States, and the year was 1920, moving among the factories to straighten out problems was a serious burden.

Part of the success of mass production was the development of new quality assurance and control techniques to manage these problems. Similarly, new supplier management techniques were introduced. Many of the techniques like just-in-time production and negotiated learning curves that are considered very modern techniques were known to Ford and his architects, if not necessarily implemented in the same way or with the same priorities as would come to pass in lean production. In Ford's time, the sophistication level was much lower, and the technology did not allow optimization in the ways that it is possible today, but the concepts were already known.

Moving to the TPS era, as quality control improved, it eventually became possible to make architectural-level changes to the assembly process. For example, when very high-quality levels are attained, testing and inspection processes can be greatly reduced and simplified. When quality levels are very high stocks and work-in-progress can be reduced with little impact on production continuity. If the defect rate is low enough, it is no longer economic to conduct multistep inspection and testing processes. With an extremely low defect rate, testing and inspection can be pushed to the final, full system level.

As the concepts of lean production became widely known, and the advantages became obvious, the role of quality in enabling (and disabling) architectural shift likewise became apparent. The obvious temptation in implementing the lean concepts in TPS was to do so at the assembly plant. But doing so is critically dependent on the quality of parts. If the defect rate on parts is too high, and you stop the production line for root cause analysis every time a defect is discovered, the system will never run. If part quality is low then more stocks are required to avoid disruptions. The quality of the parts delivered to assembly is a pre-condition to leaning assembly. If the enterprise is attempting to implement lean concepts at assembly but cannot change the upstream supply chain, the effort is doomed. The change is architectural, something General Motors discovered when they attempted to extend the lessons learned from their joint assembly plant with Toyota to other assembly plants one at a time (Langfitt 2015).

The Toyota Production System can be seen to be driven by quality. First, it places the quality of the delivered product as the highest priority. It strives for quality through the continuous and incremental results of core heuristics, implemented as processes. Of particular relevance is the principle of root cause analysis of defects (Ohno 2019), the source for the "Five Whys" heuristic we cite and use in this book, albeit generalized to a broader set of circumstances, as is the goal with architectural heuristics. Continual use of root cause analysis leads to organizational learning, which itself has synergistic effects throughout.

Devotion to Component-Level Simplification

Ford and his architects were devoted to component-level simplification. They continually looked for ways to simplify the production of individual components and to simplify major subsystems. A major method was to cast larger and more complicated iron assemblies. This eventually resulted in the single-piece casting of the V8 engine block used in the most successful Ford immediately prior to World War II. That basic engine block casting design and technique was used for decades afterward.

The movement to larger and more complex castings is a fine example of the Simplify heuristic at work. A dictate to "simplify" sounds good, but how does one actually apply it? The application must be in the architect's current context. In the case of Ford and Sorenson, castings that were very complex to develop were ultimately "simple" because of the simplification they brought to the assembly process. Making the castings was only complex up to the point it was fully understood. Once it was understood, it could be carried out very consistently and allowed for great simplifications in downstream assembly.

Simplification is likewise part of the lean process, though seen and implemented in different ways. An aspect cited in Womack et al. (2007) makes for an interesting contrast to the practices in traditional mass production by Ford. Toyota involves their suppliers more deeply in the design process, and in a tiered relationship. The top-tier suppliers are invited to co-design subassemblies as a new car design evolves, frequently making for larger, more integrated subassemblies to be supplied. This increases the complexity of the "parts" delivered to the assembly plant (as they are now larger assemblies themselves) but means fewer such parts delivered to final assembly and off-loads much of the engineering design load to suppliers. This stands in contrast to Ford's emphasis on vertical integration of subassemblies and even low-level parts making into the huge factory complexes of the height of the mass production era.

Social Contract

On the labor relations front, Henry Ford is both famous and infamous. He is famous for introducing much higher wages, specifically targeting his wages to allow all of his workers to be able to realistically afford one of the cars they were building. This was consistent with Ford's overall vision of cars for the masses. After all, what masses could he be building cars for if not the masses that he himself employed? Henry Ford is also infamous for some of his other labor practices, such as his intrusions into the private lives of his workers. The architects of the TPS were well aware of both sides of Ford's labor relations and believed that the architecture of the production system must be reconciled with a stable social contract with the workers.

All systems of productivity improvement must reconcile the improvements that are in the interests of owners with the interests of the workers. If each improvement simply leads to higher worker production quotas and job losses, it is hardly likely that workers will be enthusiastic participants in the improvement process. In the rapid growth days of Ford, when wages were doubled over those otherwise prevailing, Ford workers had obvious reasons for believing their own interests were aligned with Ford's.

Toyota faced their own labor problems, but under worse circumstances in the early years. This led to their well-known lifetime employment policy something that became part of the social architecture of lean production. As a consequence of the policy, they were little constrained by other work rule limitations. Workers could be cross-trained and moved between jobs without the kind of work rule constraints typical under U.S. labor contracts. Toyota workers relied on the lifetime employment scheme and seniority-based pay rates and were willing to work otherwise in an interchangeable team member structure. Toyota did not have a problem sustaining the lifetime employment policy because of the very rapid growth rates from the 1960s onward, first in the Japanese market and then in export markets. Tying significant portions of compensation to annual profits also helped smooth out the financial demands of downturns without layoffs.

OTHER PERSPECTIVES

Manufacturing is an enormous, global endeavor. The literature is huge and we refer here only to some basic summaries. This case study's goal was to consider some of the most famous and familiar production models from the perspective of architecture, that is essential, defining forms and the purposes that drove them. They are also rich sources of heuristics that apply beyond their initial domain. A popular alternative perspective on manufacturing system architecture, extending to the overall enterprise, is the Theory of Constraints (TOC) (Rahman 1998, Goldratt 2012, 2017, Mabin and Balderstone 2020). TOC likewise finds its own metaphors, heuristics, and organizing principles.

CONCLUSION

Ford and Toyota are the two classic examples of mass production, with the first having created the architecture and the second having transformed it. Both have recognizable architectural histories and easily identified architects. Both created changes that have rippled into fields well beyond their own. Ford was able to pioneer mass production of systems as complex as the automobile. The architecture of the Ford production system was sociotechnical, but with a heavy emphasis on the technical. We can see directly the technical innovations that made it work and that defined its essential structures (the assembly line, distributed production, new management techniques).

The TPS architectural success was smaller in that it did not create a new industry, but was larger in that TPS succeeded against a backdrop of established and strong competitors. Likewise, TPS has reached its architectural influence well beyond the production of cars, a process that is still ongoing. The development of the TPS is also an example of where incremental change, sufficiently accumulated, can eventually become revolutionary. The architecture of the TPS is much more socio than technical. In its embodiment in the Toyota Way, it is described essentially as philosophy, albeit an operative philosophy, one directly usable in practical decision making.

REFERENCES

Goldratt, E. M. (2017). *Necessary but Not Sufficient: A Theory of Constraints Business Novel*, London: Routledge.

Goldratt, E. M. and C. Jeff (2012). *The Goal 30th Anniversary Edition: A Process of Ongoing Improvement*, Barrington, MA: North River Press.

Langfitt, F. (2015). *NUMMI. This American Life*, Washington, DC: National Public Radio.

Liker, J. K. (2020). *The Toyota Way: 14 Management Principles from the World's Greatest Manufacturer*, New York: McGraw Hill.

Mabin, V. J. and S. J. Balderstone (2020). *The World of the Theory of Constraints: A Review of the International Literature*, Boca Raton, FL: CRC Press.

Ohno, T. (2019). *Toyota Production System: Beyond Large-Scale Production*, University Park, IL**:** Productivity Press.

Rahman, S. U. (1998). "Theory of constraints: A review of the philosophy and its applications." *International Journal of Operations & Production Management* **18**(4): 336–355.

Sorensen, C. E. and S. T. Williamson (2006). *My Forty Years with Ford*, Detroit, MI: Wayne State University Press.

Womack, J. P., D. T. Jones, and D. Roos (2007). *The Machine that Changed the World: The Story of Lean Production—Toyota's Secret Weapon in the Global Car Wars that is Now Revolutionizing World Industry*, New York: Simon and Schuster.

4 Manufacturing Systems

INTRODUCTION: THE MANUFACTURING DOMAIN

Although manufacturing is often treated as if it were one step in the development of a product, it is also a major system in itself. As a complex system on its own, it has its own architecture; see Hays et al. (1988), particularly Chapter 7. It has a system function that its elements cannot perform by themselves—making other things with machines. And it has an acquisition waterfall for its construction quite comparable to those of its products. The architecture of the manufacturing system and the architecture of the system of interest must relate to each other. More broadly, both exist within the structure of the development program, which should be chosen consciously and deliberately to yield the desired properties for the client.

From an architectural point of view, manufacturing was quiet for a long time. Such changes, as were required, were largely a matter of continual, measurable, incremental improvement—a step at a time on a stable architectural base. Though companies came and went, it took decades to see a major change in its members. The percentage of sales devoted to research and advanced development for manufacturing, per se, was small. The need was to make the classical manufacturing architecture more effective—that is, to evolve and engineer it.

Beginning in the 1970s and accelerating through the 1990s and beyond, the world that manufacturing had supported for almost a century changed—and at a global scale. Driven by political change in China and other countries and by new technologies in global communications, transportation, sources, markets, and finance, global manufacturing became practical and then, shortly thereafter, dominant. It quickly became clear that qualitative changes were required in manufacturing architectures if global competition were to be met. In the order of conception, the architectural innovations were ultraquality, dynamic manufacturing (Hays et al. 1988), lean production (Womack et al. 2007, Ohno 2019), and "flexible manufacturing" (producing markedly different products on demand on the same manufacturing line). The results to date, demonstrated first by the Japanese and then spread globally, have greatly increased profits and market share and sharply decreased inventory, work-in-progress, and time to market. Each of these innovations will be presented in turn.

Even so, rapid change is still underway. As seen on the manufacturing floor, manufacturing research as such has yet to have a widespread effect. Real-time software is still a work-in-progress. Trend instrumentation, self-diagnosis, and self-correction, particularly for ultraquality systems, are far from commonplace. So far, the most sensitive tool for ultraquality is the failure of the product being manufactured.

DOI: 10.1201/9781003483595-9

MANUFACTURING IN CONTEXT

Before discussing architectural innovations in manufacturing, we need to place manufacturing in context. At some point, a system needs to be built, or it is of little interest. The building of the system is “manufacturing.” But there are several distinct scenarios.

Full Development Followed by Serial Production

This applies to and is common in situations where we build tens to millions of copies of a system after producing one or more complete prototypes. The prototypes, which may themselves be the end of a series of intermediate prototypes, are essentially identical to the system to be manufactured. The testing conducted on the prototypes is commonly referred to as “qualification” testing and is to show that the system to be built is fully suitable (in function, environmental suitability, and all other respects) for end use. It shows that the system to be manufactured meets the purposes of the client in operational use. Because the prototypes are not themselves to be delivered to customer use, they can be tested very strongly, indeed destructively, if desired and warranted.

There are several strategies by which we work up a series of prototypes to result in the representative manufactured system. The most common is usually referred to as breadboard-to-brassboard. In this strategy, each prototype contains the full functionality intended for the final system but is not packaged in an operationally representative way. The first development cycle, the breadboard, may exist just as open units in a lab, interconnected and discrete subsystems tested individually. A subsequent phase may be packaged into a surrogate platform that is not yet light or strong enough for final use. The development sequence culminates in the manufacturing of representative prototypes.

Incremental Development and Release

A contrasting strategy is to develop a series of prototypes where each is fully operationally suitable but contains less than the desired level of functionality. This is common in software-intensive systems. In software systems, the cost of manufacturing and delivery is quite low, nearly zero when software is electronically delivered. Thus, the cost impediment of frequent re-release does not exist as it does in systems where most of the value is in hardware.

An incremental development and release strategy facilitates an evolutionary approach to client desires. Instead of needing to get everything right at the beginning, the developer can experiment with suppositions as to what the client really wants. The client’s learning process using the system can be fed back into subsequent releases. A major issue in a frequent release strategy is that test and certification costs are re-incurred each time a release cycle is completed. If the release cycles are frequent (best for learning feedback), the cost of test and certification could rise quickly. The process can be cost-efficient only when the costs of test and certification can be driven down, usually by automation. In some sense, the process of testing and certification for release takes the place of serial production in the example of the serial production strategy above. Test and certification are the recurring cost of each release cycle, beyond the cost of developing each release.

Incremental development and release can be used outside of software-only systems. Where the incremental strategy is used outside of software systems, it is worth considering how cost and benefits interact. Consider space launch vehicles (rockets). When put into operation for paying customers, they are the classic example of ultraquality. Any measurable failure rate is unacceptable, or at least has been seen as unacceptable when the typical space mission is served by the building of a single spacecraft, or at most, a small number. A user wants confidence that the failure rate is very low because they have no replacement for a spacecraft that fails to achieve orbit. The best evidence of a low failure rate is a record of launches without failures. A logical way to play this out is the demand for very high assurance and to avoid iterations since any design or manufacturing iteration introduces uncertainty and might upset a high-reliability record. This of course also leads to high upfront costs and the process of generating assurance without flying is necessarily very expensive and has no real limits. You can always find something else on where there is some uncertainty to justify more analysis or more component testing.

In contrast, the launch vehicle provider SpaceX deliberately embraces frequent flights with an acceptance of less-than-perfect performance. This mimics the incremental release cycle with deliberate improvements between flights. The obvious weaknesses are two. First, each flight consumes a full launch vehicle, the expense of which has to be accepted. As they have achieved booster reuse, that element is no longer consumed and only incurs refurbishment costs. Second, any estimation by a user of the likely reliability level will have to take into account failed as well as successful flights. In the case of SpaceX, users have been willing to accept that in exchange for the much lower per-flight cost. That trade is more obviously advantageous when the user moves from singleton spacecraft for a mission to larger numbers of smaller satellites to achieve a mission. The trade is not obviously advantageous for missions where only a single large spacecraft is the logical choice.

There is a very important difference between classic incremental development and what we see in launch vehicles. In classic incremental development, the increments are in user-visible functionality. Much of what is happening from release to release is learning about the user preferences from actual use. The learning is about what is wanted, not (predominantly) about the quality of the system. In contrast, in the launch vehicle case, user preferences are fairly certain, and the level of functionality provided is straightforward (get to the desired orbit with acceptable orbital insertion accuracy). The uncertainties being resolved are with respect to design and manufacturing. In this way, the incremental process more closely resembles the risk spiral introduced by Boehm (1984).

At the time of this writing, the question of where stable equilibrium exists in flight rate, launch vehicle size, observed and estimated reliability, and operational costs are open with respect to SpaceX's longer-term direction, even if it appears to be settled for their workhorse launch vehicles. The Falcon 9 and Falcon Heavy series appear to have established a stable equilibrium point, as witnessed by their market share and the willingness of a wide range of mission sponsors to entrust their spacecraft to launch by the Falcon series. Whether or not the same scheme will scale to the much larger Starship launch vehicle and continue to the projected flight rate is open.

"Protoflight" Development and Manufacturing

In this strategy, which is common in one-of-a-kind items like spacecraft, the developmental unit is also the delivered manufactured unit. That is, rather than delivering a completed prototype to be manufactured, we deliver the completed prototype to be used (launched, in the case of a satellite). The primary advantage of the protoflight approach is cost. Obviously, when only a singular item needs to be delivered, the cost of manufacturing it is minimized by making only one.

The protoflight test quandary is a mirror image of the test quandary in the serial production case. In serial production, we can freely test the prototypes as thoroughly as we like, including destructively. But we must be concerned about whether or not the prototype units fully represent the manufactured units. Usually, if the production run is large enough, we will take units off the serial production line and test them as thoroughly as the prototypes were tested. In the protoflight case, we know that the prototype and the delivered system are identical (because they are the same unit), but we risk damaging the system during the test. Tests can change the state of the system, perhaps invisibly, and test processes are always vulnerable to accidents. We cannot test in certain ways because we cannot afford test-induced damage to the flight system. We must also continuously trade the risk of not revealing a defect because of lack of testing with the risk of creating defects through testing. The satellite business in particular is full of stories of protoflight systems that were damaged through accidents in testing (e.g., over-limit vibration testing and a weather satellite tipped off of its test stand).

In each of these cases, there is a relationship between the system architecture, the architecture of the program that builds the system, the test strategy, and the architecture of any systems used for testing. When we choose an overall program architecture, we induce constraints on how we can test the resulting system. The architecture of the system of interest will determine the sorts of test approaches that can be supported. That likewise affects what sorts of systems we can build for conducting tests. Each of these issues cannot be considered and resolved in isolation. In mature situations, there may be widely accepted solutions and established architecture breakdowns. In immature situations, there may be great leverage in innovative breakdowns.

Example: DARPA Grand Challenge—Recall the DARPA grand challenge for autonomous ground vehicles we introduced in the opening case study before Chapter 1. The competing teams all used the protoflight approach; they built, tested, and raced a single vehicle. Because the single vehicle had to be used for testing as well as racing, there were fundamental architectural choices that arbitrated between these needs. As examples, if more time was devoted to building a mechanically more complex vehicle, the amount of time available to use the vehicle in software testing would be reduced. Was superior mechanical performance worth less software testing time? Any test instrumentation needed to be built into the vehicle so it could be used in field experiments. But that same test instrumentation would need to be carried in the race or removed at the last minute (generating its own risks). Where should the trade-off in enhanced testing versus less system burden lie? A vehicle optimized for autonomous operation would not be drivable by a human, but a vehicle that can be alternately human or computer driven leads to much simpler field test operations. Is the loss of performance with retaining human drivability worth the lessened burden in field test operations? As a matter of historical record, different teams participating in the Grand Challenge events took distinctly different approaches along this spectrum, but the most successful teams took relatively similar approaches (simple, production-vehicle-based mechanical system available very early in the development cycle; extensive test instrumentation; and human drivability retained).

ARCHITECTURAL INNOVATIONS IN MANUFACTURING

Ultraquality Systems

At the risk of oversimplification, a common perception of quality is that quality costs money—that is, quality is a trade-off between cost and profit. Not coincidentally, there is an accounting category called "cost of quality." A telling illustration of this perception is the "DeSoto story." As the story goes, a young engineer at a DeSoto automobile manufacturing plant went to his boss with a bright idea on how to make the DeSoto a more reliable automobile. The boss's reply: "Forget it, kid. If it were more reliable it would last more years and we would sell fewer of them. It's called planned obsolescence." DeSoto no longer is in business, but the perception remains in the minds of many manufacturers of many products.

The difficulty with this perception is partly traceable to the two different aspects of quality. The first is quality associated with features like leather seats and air conditioning. Yes, those features cost money, but the buyer perceives them as value added and the seller almost always makes money on them. The other aspect of quality is the absence of defects. As it has now been shown, the absence of defects also makes money, and for both seller and buyer, through reductions in inventory, warranty costs, repairs, documentation, testing, and time to market. At least the absence of defects can do those things, provided that the level of product quality is high enough and the whole development and production process is architected for operation at that high level.

To understand why the absence of defects makes money, imagine a faultless process that produces a product with defects so rare that it is impractical to measure them; that is, none are anticipated within the lifetime of the product. Testing can be reduced to the minimum required to certify the system-level performance of the first unit. Delays and their costs can be limited to those encountered during development; if and when later defects occur, they can be promptly diagnosed and permanently eliminated. "Spares" inventory, detailed parts histories, and statistical quality control can be almost nonexistent. First-pass manufacturing yield can be almost 100% instead of today's highly disruptive 20%–70%. Service in the field is little more than replacing failed units for free. When failures are almost nonexistent many costly aspects of the design can be avoided (e.g., no maintenance access is needed if the only response to the rare failure is replacement of the whole system).

The only practical measurement of ultraquality would then be an end-system level test of the product. Attempting to measure defects at any subsystem level would be a waste of time and money—defects would have to be too rare to determine with high confidence. Redundancy and fault-tolerant designs would be unnecessary. Indeed, they would be impractical because, without an expectation of a specific failure (which then should be fixed), protection against rare and unspecified defects is not cost-effective. Thus, the level of quality achieved could affect the most appropriate architecture of the manufacturing system.

To some readers, this ultraquality level may appear to be hopelessly unrealistic. Suffice it to say that it has been approached for all practical purposes. For decades, no properly built unmanned satellite or spacecraft failed because of a defect known before launch (any defect would have been fixed beforehand). Microprocessors with millions of active elements, sold in the millions, now outlast the computers for which they were built. Like the satellites, they become technically and financially obsolete long before they fail. Television sets are produced with a production line yield of over 99%, far better than the 50% yield of a decade ago, with a major improvement in cost, productivity, and profit.

As a further example, the readers should note that consumer electronic products today are commonly unrepairable and, if defective within a warranty period, are simply replaced. This approach carries multiple benefits. If a system does not need to be repaired, the supplier need not maintain a repair and supply network and can sweep away all the costs associated with one. If repairs are not necessary, the unit can be designed without repair access or diagnostics, which commonly saves space and money and allows the use of manufacturing techniques (such as sealing) that themselves improve reliability.

Today's challenge, then, is to achieve and maintain such ultraquality levels even as systems become more complex. Techniques have been developed that certainly help, as outlined in Rechtin's (1991) Chapter 8. Two techniques that are commonly cited in manufacturing are:

Everyone in the production line is both a customer and a supplier, a customer for the preceding worker and a supplier for the next. Its effect is to place quality assurance where it is most needed, at the source. Every step of the chain is held to an equal level of responsibility.

Systematic root cause analysis and *The Five Why's* heuristic (Ohno 2019), a diagnostic procedure for finding the basic cause of a defect or discrepancy. Why did this occur? Then why did that, in turn, occur? Then, why that? and so on until the offending causes are discovered and eliminated. Properly understood, this is a learning process and the driver of systematic change. Doing root cause analysis is pointless unless one is committed to fixing it all the way down to the root so it does not re-occur.

To these techniques can be added a relatively new understanding: Some of the worst failures are system failures—that is, they come from the interaction of subsystem deficiencies which of themselves do not produce an end-system failure, but together can and do. Four catastrophic civil space system failures were of this kind: Apollo 1, Apollo 13, Challenger, and the Hubble Telescope. For Tom Clancy buffs, just such a failure almost caused World War III in his Debt of Honor. In all these cases, had any one of the deficiencies not occurred, the near-catastrophic end result could not have occurred. That is, though each deficiency was necessary, none were sufficient for end failure. As an admonition to future failure review boards, until a diagnosis is made that indicates that the set of presumed causes is both necessary and sufficient—and that no other such set exists—the discovery-and-fix process is incomplete and ultraquality is not assured.

Successful ultraquality has indeed been achieved, but there is a price that must be paid. Should ultraquality not be produced at any point in the whole production process, the process may collapse. Therefore, when something does go wrong, it must be fixed immediately; there are no cushions of inventory, built-in holds, full-time expertise, or planned workarounds. Because strikes and boycotts can have instantaneous effects, employee, customer, and management understanding and satisfaction are essential. Pride in work and dedication to a common cause can be of special advantage, as has been seen in the accomplishments of the zero-defect programs of World War II, the American Apollo lunar landing program, and the Japanese drive to world-class products.

In a sense, ultraquality-built systems are fine-tuned to near-perfection with all the risks thereof. Just how much of a cushion or insurance policy is needed for a particular system is an important value judgment that the architect must obtain from the client; the earlier, the better. That judgment has strong consequences for the architecture of the manufacturing system. Clearly, then, ultraquality architectures are very different from the statistical quality assurance architectures of only a few years ago. One can often note this from observation of the spaces. Are the spaces clean and free from clutter (like stocks of rejected parts and rework)? Are there wall charts that show quality status, and are they marked in "days since last defect" or the running average defect rate?

It also points out the likely folly of trying to implement the downstream consequences of ultraquality, like minimal buffers and end-inspection-only if the upstream quality prerequisites have not been achieved. Most important for what follows, it is unlikely that either lean production or flexible manufacturing can be made competitive at much less than ultraquality levels.

Dynamic Manufacturing Systems

The second architectural change in manufacturing systems is from comparatively static configurations to dynamic, virtually real-time, ones. Two basic architectural concepts have now become much more important. The first concerns intersecting waterfalls, and the second, feedback systems.

Intersecting Waterfalls

The development of a manufacturing system can be represented as a separate waterfall, distinct from, and intersecting with, that of the product it makes. Figure 4.1 depicts

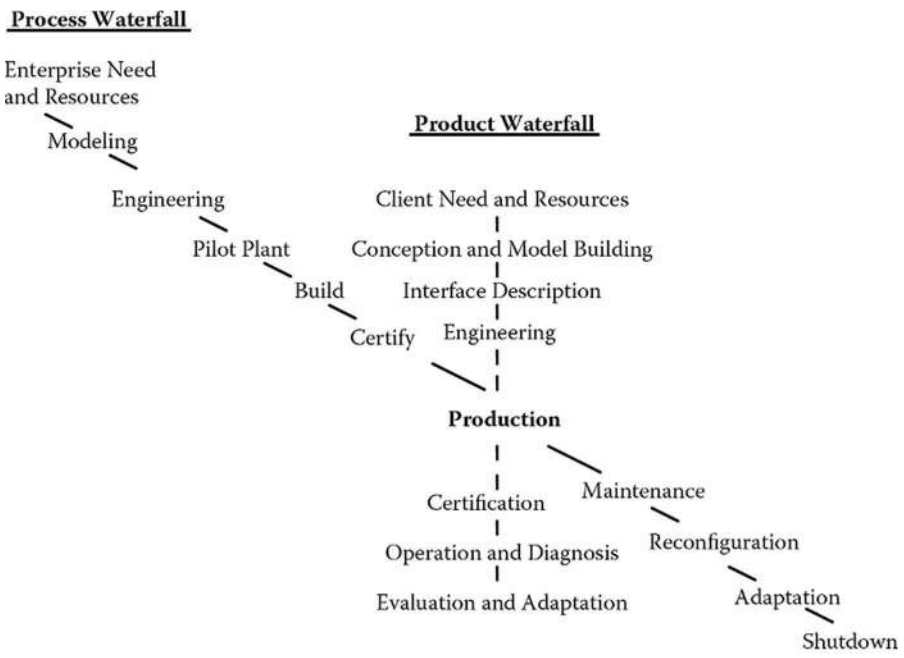


FIGURE 4.1 The intersecting process and product waterfalls.

the two waterfalls, the process (manufacturing) diagonally and the product vertically, intersecting at the time and point of production. The manufacturing one is typically longer in time, often decades, and contains more steps than the relatively shorter product sequence (months to years) and may end quite differently (the plant is shut down and demolished). Sketching the product development and the manufacturing process as two intersecting waterfalls helps emphasize the fact that manufacturing has its own steps, time scales, needs, and priorities distinct from those of the product waterfall. It also implies the problems its systems architect will face in maintaining system integrity, in committing well ahead to manufacture products not yet designed, and in adjusting to comparatively abrupt changes in product mix and type. A notably serious problem is managing the introduction of new technologies, safely and profitably, into an inherently high-inertia operation.

There are other differences. Manufacturing certification must begin well before product certification, or the former will preclude the latter; in any case, the two must interact. The product equivalent of plant demolition, not shown in the figure, is recycling, both now matters of national law in Europe. Like certification, demolition is important to plan early, given its collateral human costs in the manufacturing sector. The effects of environmental regulations, labor contracts, redistribution of usable resources, retraining, right-sizing of management, and continuing support of the customers are only a few of the manufacturing issues to be resolved—and well before the profits are exhausted.

Theoretically, if not explicitly, these intersecting waterfalls have existed since the beginning of mass production. But not until recently have they been perceived as having equal status, particularly in the United States. Belatedly, that perception is changing, driven in large part by the establishment of global manufacturing—clearly not the same system as a wholly owned shop in one's own backyard. The change is magnified by the widespread use of sophisticated software in manufacturing, which is a boon in managing inventory but a costly burden in reactive process control. Predictably, software for manufacturing process and control, more than any element of manufacturing, will determine the practicality of flexible manufacturing. As a point of proof (Hays et al. 1988) point out that a change in the architecture of [even] a mass production plant, particularly in the software for process control, can make dramatic changes in the capabilities of the plant without changing either the machinery or layout.

The development of manufacturing system software adds yet another production track. The natural development process for software generally follows a spiral, certainly not a conventional waterfall, cycling over and over through functions, form, code (building), and tests. The software spiral shown in Figure 4.2 is typical. It is partially shaded to indicate that one cycle has been completed with at least one more to go before the final test and use. One reason for such a departure from the conventional waterfall is that software, as such, requires almost no manufacturing, making the waterfall model of little use as a descriptor. The new challenge is to synchronize the stepped waterfalls and the repeating spiral processes of software-intensive systems. One of the most efficient techniques is through the use of stable intermediate forms (Brooks Jr 1995), combining software and hardware into partial but working precursors to the final system. Their important feature is that they are stable configurations; that is, they are reproducible, well-documented, progressively refined

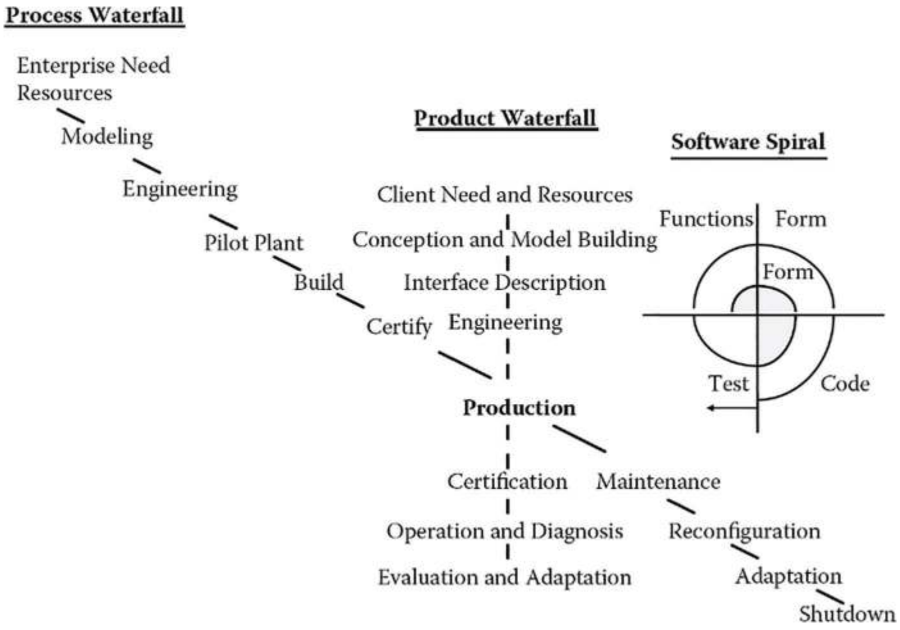


FIGURE 4.2 Product, process, and software system tracks.

baselines—in other words, they are identifiable architectural waypoints and must be treated as such. They can also act as operationally useful configurations, built-in "holds" allowing lagging elements to catch up, or parts of strategies for risk reduction as suggested in Chapter 3.

The Spiral-to-Circle Model

Visualizing the synchronization technique for the intersecting waterfalls and spirals of Figure 4.2 can be made simpler by modifying the spiral so that it remains from time to time in stable concentric circles on the four-quadrant process diagram. Figure 4.3 shows a typical development from a starting point in the function quadrant cycling through all quadrants three times—typical of the conceptualization phase—to the first intermediate form. There, the development may stay for a while, changing only slightly, until new functions call for the second form, say, an operational prototype

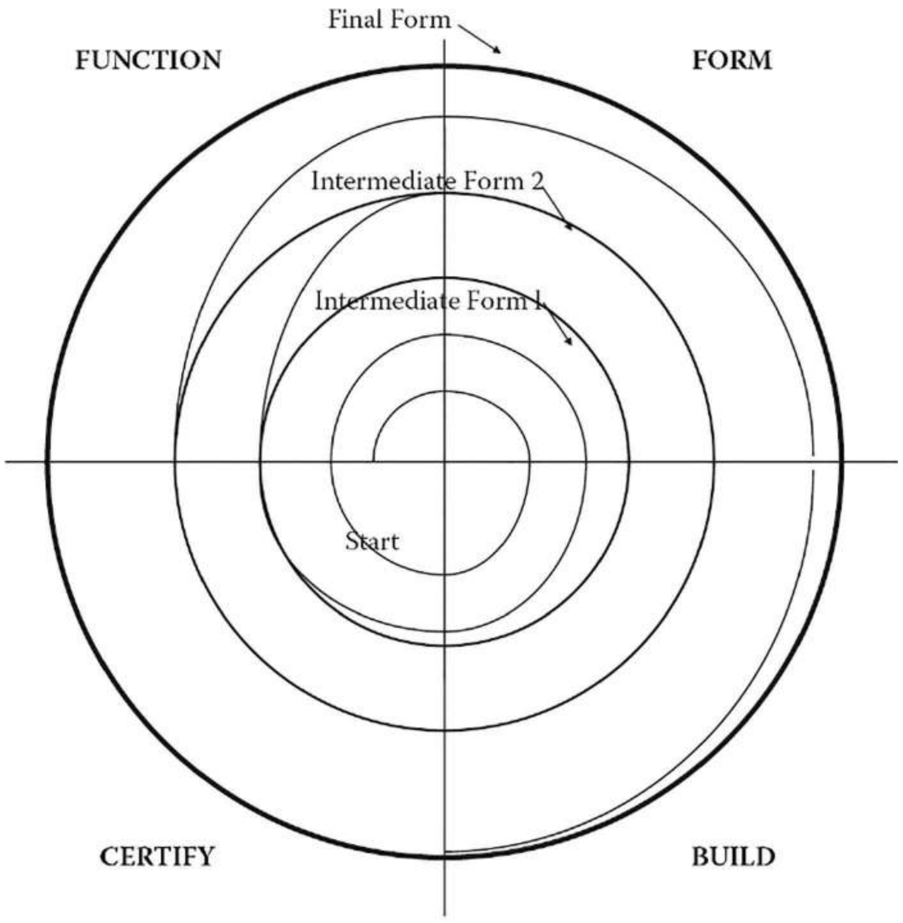


FIGURE 4.3 The spiral-to-circle model.

in Air Force procurement, that might be a "fly-before-buy" form. In space systems, it might be a budget-constrained "operational prototype" that is actually flown. In one program, it turned out to be the only system flown. But, to continue, the final form is gained, in this illustration, by way of a complete, four-quadrant cycle.

The spiral-to-circle model can show other histories, for example, a failure to spiral to the next form, with a retreat to the preceding one, possibly with less ambitious goals, or a transition to a still greater circle in continuing evolution, or an abandonment of these particular forms with a restart near the origin.

Synchronization can also be helped by recognizing that cycling also goes on in the multistep waterfall model, except that it is depicted as feedback from one phase to one or more preceding ones. It would be quite equivalent to software quadrant spiraling if all waterfall feedback returned to the beginning of the waterfall—that is, to the system's initial purposes and functions, and from there, down the waterfall again. If truly major changes are called for, the impact can be costly, of course, in the short run. The impact in the long run might be cost-effective, but few hardware managers are willing to invest.

The circle-to-spiral model for software-intensive systems in effect contains both the expanding-function concept of software and the stepwise character of the waterfall. It also helps understand what and when hardware and software functions are needed in order to satisfy requirements by the other parts of the system. For example, a stable intermediate software form of software should arrive when a stable, working form of hardware arrives that needs that software, and vice versa.

It is important to recognize that this model, with its cross-project synchronization requirement, is notably different from models of procurements in which hardware and software developments can be relatively independent of each other. In the spiral-to-circle model, the intermediate forms, both software and hardware, must be relatively unchanging and bug-free. A software routine that never quite settles down or that depends upon the user to find its flaws is a threat, not a help, in software-intensive systems procurement. A hardware element that is intended to be replaced with a "better and faster" one later is hardly better. Too many subsequent decisions may unknowingly rest on what may turn out to be anomalous or misunderstood behavior of such elements in system test.

To close this section, although this model may be relatively new, the process that it describes is not. Stable intermediate forms, blocks (I, II, and so forth), or "test articles" as they are called, are built into many system contracts and perform as intended. Yet there remains a serious gap between most hardware and software developers in their understanding of each other and their joint venture. As the expression goes, "These guys just don't talk to each other." The modified spiral model should help both partners bridge that gap, to accept the reasons both for cycling and for steps, and to recognize that neither acquisition process can succeed without the success of the other.

There should be no illusion that the new challenge will be easy to meet. Intermediate software forms will have to enable hardware phases at specified milestones—not just satisfy separate software engineering needs. The forms must be stable, capable of holding at that point indefinitely, and practical as a stopping point in the acquisition process if necessary. Intermediate hardware architectures must have sufficient flexibility to accommodate changes in the software—as well as in the hardware. And finally, the architects and managers will have a continuing challenge in resynchronizing the several processes so that they neither fall behind nor get too far ahead. Well-architected intermediate stable forms and milestones will be essential.

Concurrent Engineering

To return to Figure 4.2, this intersecting waterfall model also helps identify the source of some of the inherent problems in concurrent (simultaneous, parallel) engineering—in which product designers and manufacturing engineers work together to create a well-built product. Concurrent engineering in practice, however, has proven to be more than modifying designs for manufacturability. However defined, it is confronted with a fundamental problem, evident from Figure 4.2—namely, coordinating the two intersecting waterfalls and the spirals, each with different time scales, priorities, hardware, software, and profit-and-loss criteria. Because each track is relatively independent of the others, the incentives for each are to optimize locally, even if doing so results in an impact on another track or on the end product. After all, it is a human and organizational objective to solve one's own problems, to have authority reasonably commensurate with responsibilities, and to be successful in one's own right. Unfortunately, this objective forces even minor technical disagreements to higher, enterprise management where other considerations than just system performance come into play.

A Typical Example: A communications spacecraft design was proceeding concurrently in engineering and manufacturing until the question came up of the spacecraft antenna size. The communications engineering department believed that a 14-foot diameter was needed; the manufacturing department insisted that 10 feet was the practical limit. The difference in system performance was a factor of two in communications capability and revenue. The reason for the limit, it turned out, was that the manufacturing department had a first-rate subcontractor with all the equipment needed to build an excellent antenna, but no larger than 10 feet. To go larger would cause a measurable manufacturing cost overrun. The manufacturing manager was adamant about staying within his budget, having taken severe criticism for an overrun in the previous project. In any case, the added communications revenue gain was far larger than the cost of re-equipping the subcontractor. Lacking a systems architect, the departments had little choice but to escalate the argument to a higher level where the larger antenna was eventually chosen and the manufacturing budget increased slightly. The design proceeded normally until software engineers wanted to add more memory well after manufacturing had invested heavily in the original computer hardware design. The argument escalated, valuable time was lost, department prerogatives were again at stake, and so it went.

The example is not uncommon. A useful management improvement would have been to set up a trusted, architect-led team to keep balancing the system as a whole within broad top management guidelines of cost, performance, risk, and schedule.

If so established, the architectural team's membership should include a corporate-level (or "enterprise") architect, the product architect, the manufacturing architect, and a few specialists in system-critical elements, and no more. Such a structure does exist implicitly in some major companies, though seldom with the formal charter, role, and responsibilities of systems architecting.

Feedback Systems

Manufacturers have long used feedback to better respond to change. Feedback from the customer has been, and is, used directly to maintain manufacturing quality and indirectly to accommodate changes in design. Comparably important are paths from sales to manufacturing and from manufacturing to engineering.

The presence or absence of feedback paths, and their time constants, is something that can be deliberately controlled through the architecture of the program and organization that envelop a system of interest. Consider space exploration systems. An exploration organization can choose to pursue large, multimission systems that take a long time, or many more smaller, more rapidly turned over programs. Because the payload fraction of a spacecraft is generally higher as the spacecraft gets bigger, larger, multi mission spacecraft are generally more cost efficient. But, because they take much longer, the time constant on which the things learned on one mission can be fed back into the next is longer. The organization has fewer opportunities to incorporate their learning into future mission design. In a very mature mission area where needs change slowly, this might be a fair trade-off. In an immature mission area where each new payload reveals new questions and new preferences, a faster feedback loop yields dramatically different characteristics. Moreover, the pace of feedback affects the people in the organization. They, likewise, learn (and are held accountable) primarily when each full mission feedback loop closes. An organization with a fast feedback loop (but not too fast) is a rapidly learning organization. If the feedback is slow enough, coupled with staff turnover, there might not be any organizational learning at all.

What is new in manufacturing is that the pace has changed. Multiyear is now yearly, yearly is now monthly, monthly is now daily, and daily—especially for ultraquality systems—has become hourly, if not sooner. What was a temporary slowdown is now a serious delay. What used to affect only adjacent sectors can now affect the whole. What used to be the province of planners is now a matter of real-time operations.

Consequently, accustomed delays in making design changes, correcting supply problems, responding to customer complaints, introducing new products, reacting to competitors' actions, and the like were no longer acceptable. The partner to ultraquality in achieving global competitiveness was to counter the delays by anticipating them, in other words, using anticipatory feedback in as close to real-time as practical. The most dramatic industrial example to date has been in lean production (Womack et al. 2007), in which feedback to suppliers, coupled with ultraquality techniques, cut the costs of inventory in half and resulted in across-the-board competitive advantage in virtually all business parameters. More recently, criteria for certification, or those of its predecessor, quality assurance, are routinely fed back to conceptual design and engineering—one more recognition that quality must be designed in, not tested in.

A key factor in the design of any real-time feedback system is loop delay, the time it takes for a change to affect the system "loop" as a whole. In a feedback system, delay is countered by anticipation based on anticipated events (like a failure) or on a trend derived from the integration of past information. The design of the anticipation, or "correction," mechanism, usually the feedback paths, is crucial. The system as a whole can go sluggish on the one hand or oscillatory on the other. Symptoms are inventory chaos, unscheduled overtime, share price volatility, exasperated sales forces, frustrated suppliers, and, worst of all, departing long-time customers. Design questions are as follows: What is measured? How is it processed? Where is it sent? And, of course, to what purpose?

Properly designed feedback control systems determine transient and steady-state performance, reduce delays and resonances, alleviate nonlinearities in the production process, help control product quality, and minimize inventory. By way of explanation, in nonlinear systems, two otherwise independent input changes interact with each other to produce effects different from the sum of the effects of each separately. Understandably, the end results can be confusing, if not catastrophic. An example is a negotiated reduction in wages followed immediately by an increase in executive wages. The combination results in a strike; either alone would not.

A second key parameter, the resonance time constant, is a measure of the frequency at which the system or several of its elements try to oscillate or cycle. Resonances are created in almost every feedback system. The more feedback paths there are the more resonances. The business cycle, related to inventory cycling, is one such resonance. Resonances, internal and external, can interact to the point of violent, nonlinear oscillation and destruction, particularly if they have the same or related resonant frequencies. Consequently, a warning: Avoid creating the same resonance time constant in more than one location in a production system.

Delay and resonance times, separately and together, are subject to design. In manufacturing systems, the factors that determine these parameters include inventory size, inventory replacement capacity, information acquisition and processing times, fabrication times, supplier capacity, and management response times. All can have a strong individual and collective influence on such overall system time responses as time to market, material and information flow rates, inventory replacement rate, model change rate, and employee turnover rate. Few, if any, of these factors can be chosen or designed independently of the rest, especially in complex feedback systems.

Fortunately, there are powerful tools for feedback system design. They include linear transform theory, transient analysis, discrete event mathematics, fuzzy thinking, and some selected nonlinear and time-variant design methods. The properties of at least simple linear systems designed by these techniques can be simulated and adjusted easily. A certain intuition can be developed based upon long experience with them. For example,

- Behavior with feedback can be very different from behavior without it.
    - **Positive Example:** Provide inventory managers with timely sales information and drastically reduce inventory costs.
    - **Negative Example:** Ignore customer feedback and drown in unused inventory.

- Feedback works. However, the design of the feedback path is critical. Indeed, in the case of strong feedback, its design can be more important than that of the forward path.
  - **Positive Example:** Customer feedback needs to be supplemented by anticipatory projections of economic trends and of competitor's responses to one's own strategies and actions to avoid delays and surprises.
  - **Negative Examples:** If feedback signals are "out of step" or of the wrong polarity, the system will oscillate, if not go completely out of control. Response that is too little, too late is often worse than no response at all.
- Strong feedback can compensate for many known vagaries, even nonlinearities in the forward path, but it does so "at the margin."

  Example: Production lines can be very precisely controlled around their operating points; that is, once a desired operating point is reached, tight control will maintain it, but off that point or on the way to or from it (e.g., start up, synchronization, and shut down), off-optimum behavior is likely. Example: Just-in-time response works well for steady flow and constant volume. It can oscillate if flow is intermittent and volume is small.
- Feedback systems will inherently resist unplanned or unanticipated change, whether internal or external.

Satisfactory responses to anticipated changes, however, can usually be assured. In any case, the response will last at least one-time constant (cycle time) of the system. These properties provide stability against disruption. On the other hand, abrupt mandates, however necessary, will be resisted and the end results may be considerably different in magnitude and timing from what advocates of the change anticipated. Example: Social systems, incentive programs, and political systems notoriously "readjust" to their own advantage when change is mandated. The resultant system behavior is usually less than, later than, and modified from that anticipated.

- To make a major change in the performance of a presently stable system is likely to require a number of changes in the overall system design.

  Examples: The change from mass production to lean production to flexible production and the use of robots and high technology.

Not all systems are linear, however. As a warning against over-dependence on linear-derived intuition, typical characteristics of nonlinear systems are as follows:

- In general, no two systems of different nonlinearity behave in exactly the same way.
- Introducing changes into a nonlinear system will produce different (and probably unexpected) results if they are introduced separately than if they are introduced together.
- Introducing even very small changes in input magnitude can produce very different consequences even though all components and processes are deterministic.

Example: Chaotic behavior (noiselike but with underlying structure) close to system limits is such a phenomenon. Example: When the phone system is saturated with calls and goes chaotic, the planned strategy is to cut off all calls to a particular sector (e.g., California after an earthquake) or revert back to the simplest mode possible (first come, first serve). Sophisticated routing is simply abandoned — it is part of the problem. Example: When a computer abruptly becomes erratic as it runs out of memory, the simplest and usually successful technique is to turn it off and start it up again (reboot), hoping that not too much material has been lost.

- Noise and extraneous signals irreversibly intermix with and alter normal, intended ones, generally with deleterious results.

  Example: Modification of feedback and control signals is equivalent to modifying system behavior — that is, changing its transient and steady-state behavior. Nonlinear systems are therefore particularly vulnerable to purposeful opposition (jamming, disinformation, overloading).
- Creating nonlinear systems is of higher risk than creating well-understood, linear ones.

The risk is less that the nonlinear systems will fail under carefully framed conditions than that they will behave strangely otherwise. Example: In the ultraquality spacecraft business, there is a heuristic: If you cannot analyze it, do not build it—an admonition against unnecessarily nonlinear feedback systems.

The two most common approaches to nonlinearity are, first, when nonlinearities are both unavoidable and undesirable, minimizing their effect on end-system behavior through feedback and tight control of operating parameters over a limited operating region. Second, when nonlinearity can improve performance, as in discrete and fuzzy control systems, be sure to model and simulate performance outside the normal operating range to avoid "nonintuitive" behavior.

The architectural and analytic difficulties faced by modern manufacturing feedback systems are that they are neither simple nor necessarily linear. They are unavoidably complex, probably contain some nonlinearities (limiters and fixed time delays), are multiply interconnected, and are subject to sometimes drastic external disturbances, not the least of which are sudden design changes and shifts in public perception. Their architectures must therefore be robust and yet flexible. Though inherently complex, they must be simple enough to understand and modify at the system level without too many unexpected consequences. In short, they are likely to be prime candidates for the heuristic and modeling techniques of systems architecting.

Feedback can be thought of at levels beyond the individual system and the manufacturing enterprise in normal operation. At the strategic level, we configure our enterprises with some level of feedback based on achieving, or failing to achieve success. Some measures of success are tied to enterprise-level feedback behavior.

Consider the strategic problem of an enterprise with a scientific research purpose that conducts missions of varying duration, from a few years to a decade. Suppose that the organization can make tradeoffs between the duration and complexity of their missions (as many organizations can). They can choose shorter, simpler missions or longer but more complex (able to do more investigation) missions. As noted above, it impacts the learning process for the organization. The choice is the choice of feedback loop time constant, at both a scientific and enterprise-programmatic level, from mission to mission. The scientific discoveries on one mission will affect the scientific questions posed on the next mission. When unexpected things are found on one mission, like clear evidence of water on Mars, it deeply affects the enterprise's preference for what to look for on subsequent missions. Likewise, the success or failure of various systems on a mission will affect their potential use on subsequent missions.

The duration of a mission is, in effect, the feedback time constant. The shorter that time constant, the more rapidly scientific discoveries and mission results are fed back into future missions. If the major goal of the enterprise is to produce unexpected scientific discoveries, then a shorter time constant may be an effective trade-off for less single mission cost-effectiveness. Reprising the maligned slogan "Faster, Better, Cheaper," it could be that faster is better at the enterprise level, even if it is not better at a single mission level. But note, such a conclusion is dependent on the overall objectives of the enterprise being subject to change from feedback. If the overall enterprise objectives are stable, and more like stewardship, a shorter time constant would not be a good trade-off. A single enterprise might well have objectives that map to different program loop time constants.

Consider the challenge of satellite-based earth environmental observation (for which we have a large constellation of satellites from multiple enterprises). There are several distinct missions involved each with different "time constant" needs.

- Scientific research seeks new results. One of the surest paths to new results is to make new observations (new phenomenology, new levels of resolution). Results are only new once, so this is well met by constant turnover and innovation in observation technologies.
- Weather forecasting relies on having records of observations long enough to build predictive systems and on being able to validate the forecasts as the predictive systems update. This requires stable observation records long enough to cover cycles in the phenomena of interest, coupled with improvements at a rate compatible with the ability to incorporate those improvements into better predictive models and validate those predictive models. Improvements to weather forecasting come only from the coupled improvements to observations and predictive models. If changes to one have a time constant much longer than the other, then improvement will likely be limited by the slower loop.
- Climate modeling is like weather forecasting, except the time constants are much longer. Now, the basic observation interval is at least a year, and decades of precisely comparable records are needed for some trending to be recognized. This mission area will benefit from stability and lack of change in observations in contrast to steady improvement.

Similar effects can be imagined at the management level of the enterprise. If the mission time constant is short enough, it will last no more than one person's normal assignment period. Program managers, architects, systems engineers, principal investigators, and others will serve for the full duration of a mission. Instead of end-to-end mission success or failure being fed back over several different leaders (as commonly happens with decade-long programs), accountability for success or failure is attached directly to those leaders. The effects on personnel policies and organizational learning should be obvious.

LEAN PRODUCTION

One of the most influential books on manufacturing of the last part of the twentieth century was the 1990 bestseller, *The Machine That Changed the World: The Story of Lean Production*, since updated (Womack et al. 2007). Although the focus of this extensive survey and study was on automobile production in Japan, the United States, and Europe, its conclusions are applicable to manufacturing systems in general, particularly the concepts of quality and feedback. Subsequent to the publication of "Machine" there was a serious effort on the part of United States and European auto companies to adapt (Ingrassia and White 1994). But a declaration of a "Comeback" was too soon, as is well known (Ingrassia 2011, Langfitt 2015). The story of lean production systems is by no means neat and orderly. Although the principles can be traced back to 1960 and even before, effective implementation took decades. Lean production certainly did not emerge full-blown. Ideas and developments came from many sources, some prematurely. Credits were sometimes misattributed. Many contributors were very old by the time their ideas were widely understood and applied. Quality was sometimes seen as an end result instead of as a prerequisite for any change. Improvement efforts went on for years with only limited apparent end results. Then, seemingly, within a few years, everything worked. But when others attempted to adopt the process, they often failed. Why?

One way to answer such questions is to diagram the lean production process from an architectural perspective. Figure 4.4 is an architect's sketch of the lean production waterfall derived from the texts of the just-mentioned books, highlighting (boldfacing) its nonclassical features and strengthening its classical ones. The most apparent feature is the number and strength of its feedback paths. Two are especially characteristic of lean production: the supplier waterfall loop and the customer-sales-delivery loop. Next evident is the quality policies box, crucial not only to high quality but to the profitable and proper operation of later steps, just-in-time inventory, rework, and implicit warranties. Quality policies are active elements in the sequence of steps, are a step through which all subsequent orders and specifications must pass, and are as affected by its feedback input as any other step; that is, policies must change with time, circumstance, technology, and product change and process imperatives.

Research and development (R&D) is not "in the loop" but instead is treated as one supplier of possibilities, among many, including one's competitors' R&D. As described in the 1990 study, R&D is not a driver, though it would not be surprising if its future role were different. Strong customer feedback, in contrast, is very much within the loop, making the loop responsive to customer needs at several key points. Manufacturing feedback to suppliers is also a short path, in contrast with the stand-off posture of much U.S. procurement.

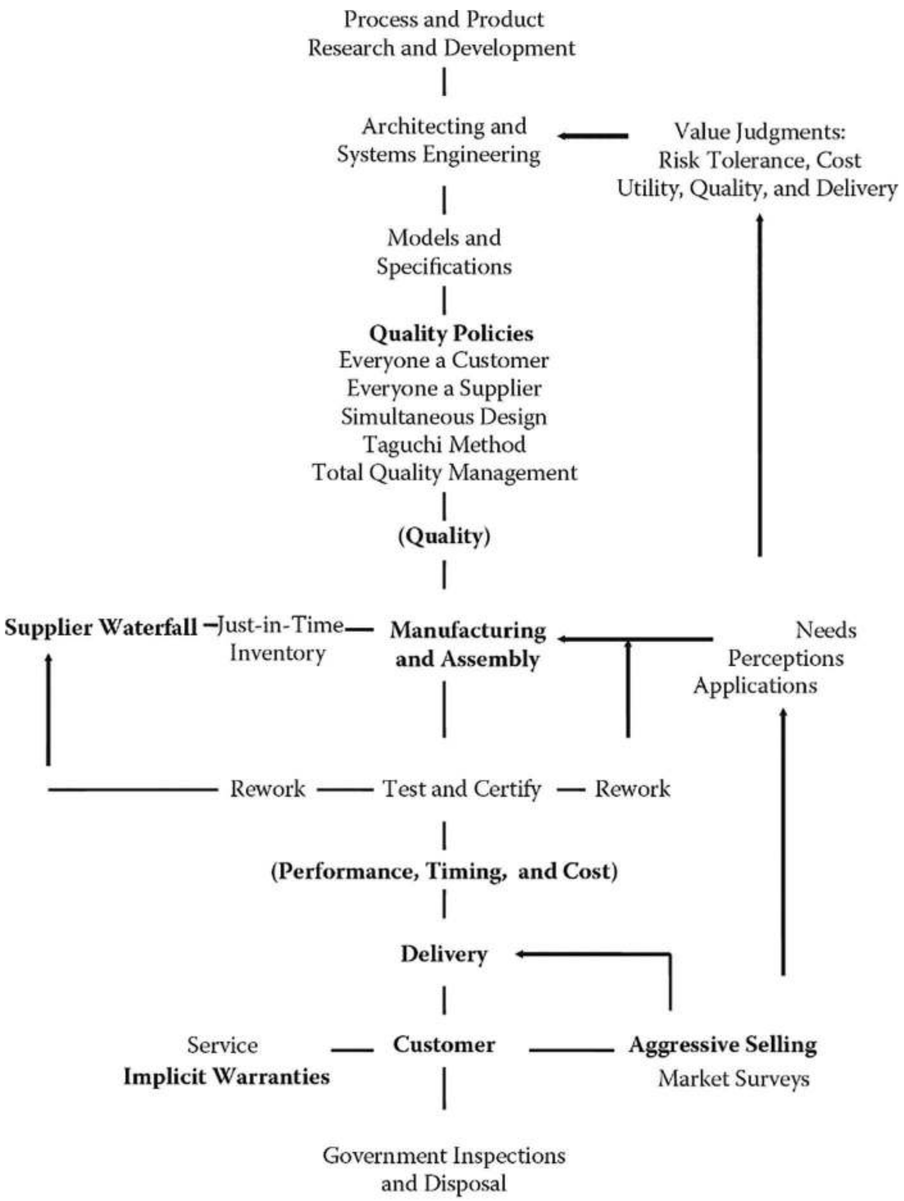


FIGURE 4.4 An architect's sketch of lean production.

The success of lean production has induced mass producers to copy many of its features, not always successfully. Several reasons for lack of success are apparent from the figure. If the policy box does not implement ultraquality, little can be expected from changes further downstream regardless of how much they ought to be able to contribute. Just-in-time (JIT) inventory is an example. Low-quality supply mandates a cushion of inventory roughly proportional to the defect rate; shifting that inventory from the manufacturer back to the supplier without a simultaneous quality upgrade simply increases transportation and financing costs. To decrease inventory, decrease the defect rate, then apply the coordination principles of JIT, not before.

Another reason for limited success in converting piecemeal to lean production is that any well-operated feedback system, including those used in classical mass production, will resist changes in the forward (waterfall) path. The "loop" will attempt to self-correct. And it will take at least one loop time constant before all the effects can be seen or even be known. To illustrate, if supply inventory is reduced, what is the effect on sales and service inventory? If customer feedback to the manufacturing line is aggressively sought, as it is in Japan, what is the effect on time to market for new product designs?

A serious question raised in both books is how to convert mass production systems into lean production systems. It is not, as the name "lean" might imply a mass production system with the "fat" of inventory, middle management, screening, and documentation taken out. It is to recognize lean production as a different architecture based on different priorities and interrelationships. How then to begin the conversion? What is both necessary and sufficient? What can be retained?

The place to begin conversion, given the nature of feedback systems, is in the quality policies step. In lean production, quality is not a production result determined postproduction and posttest; it is a prerequisite policy imperative. Indeed, Japanese innovators experienced years of frustration when total quality management (TQM), JIT, and the Taguchi methods at first seemed to do very little. The level of quality essential for these methods to work had not yet been reached. When it was, the whole system virtually snapped into place with results that became famous. Even more important for other companies, unless their quality levels are high enough, even though all the foregoing methods are in place, the famous results will not—and cannot—happen.

Conversely, at an only slightly lower level of quality, lean systems sporadically face at least temporary collapse, as discussed in Womack et al. (2007). As a speculation, there appears to be a direct correlation between how close to the cliff of collapse the system operates and the competitive advantage it enjoys. Backing off from the cliff would seem to decrease its competitive edge, yet getting too close risks imminent collapse—line shutdown, transportation jam-up, short-fuse customer anger, and collateral damage to suppliers and customers for whom the product is an element of a still larger system production.

To summarize, lean production is an ultraquality, dynamic feedback system inherently susceptible to any reduction in quality. It depends upon well-designed, multiple feedback. Given ultraquality standards, lean production arguably is less complex, simpler, and more efficient than mass production. And, by its very nature, it is extraordinarily, fiercely, competitive.

FLEXIBLE MANUFACTURING

Flexible manufacturing is the capability of sequentially making more than one product on the same production line. In its most common present-day form, it customizes products for individual customers, more or less on demand, by assembling different modules (options) on a common base (platform). Individually tailored automobiles, for example, have been coming down production lines for years. But with one out of three or even one out of ten units having to be sent back or taken out of a production stream, flexible manufacturing in the past has been costly in inventory, complex in operation, and high-priced per option compared to all-of-a-kind production.

What changed are customer demands and expectations, especially in consumer products. Largely because of technological innovation, more capability for less cost now controls purchase rate rather than wearout and increasing defect rate—an interesting epilogue for the DeSoto story! Cyclic demand is very bad for lean production, hypothetically making long lived vehicles dangerous for the maker, as in the DeSoto story. As (Womack et al. 2007) notes, the extremely demanding mandatory Japanese auto inspection process makes it undesirable to keep a vehicle longer than 6 years, probably significantly smoothing domestic demand for Japanese auto makers. One consequence of the change is more models per year with fewer units per model, the higher costs per unit being offset by use of techniques such as preplanned product improvement, standardization of interfaces and protocols, and lean production methods.

A natural architecture for the flexible manufacturing of complex products would be an extension of lean production with its imperatives—additional feedback paths and ultraquality-produced simplicity—and an imperative all its own, human-like information command and control.

At its core, flexible manufacturing involves the real-time interaction of a production waterfall with multiple product waterfalls. Lacking an architectural change from lean production, however, the resultant multiple information flows could overwhelm conventional control systems. The problem is less that of gathering data than of condensing it. That suggests that flexible manufacturing will need judgmental, multiple-path control analogous to that of an air traffic controller in the new "free flight" regime. Whether the resultant architecture will be fuzzy, associative, neural, heuristic, or largely human, is arguable.

To simplify the flexible manufacturing problem to something more manageable, most companies today would limit the flexibility to a product line that is forward and backward compatible, uses similar modules (with many modules identical), keeps to the same manufacturing standards, and is planned to be in business long enough to write off the cost of the facility out of product-line profits. In brief, production would be limited to products having a single basic architecture, for example, producing either Macintosh computers, Hitachi TV sets, or Motorola cellular telephones, but not all three on demand on the same production line.

Even that configuration is complex architecturally. To illustrate: A central issue in product-line design is where in the family of products, from low-end to high-end, to optimize. Too high in the series, and the low end is needlessly costly. Too low, and the high end adds too little value. A related issue arises in the companion manufacturing system. Too much capability, and its overhead is too high; too little, and it sacrifices profit to specialty suppliers.

Another extension from lean production to flexible manufacturing is much closer coordination between the design and development of the product line and the design and development of its manufacturing system. Failure to achieve this coordination, as illustrated by the problems of introducing robots into manufacturing, can result in warehouses of unopened crates of robots and in-work products that cannot be completed as designed. Clearly, the product and its manufacturing system must match. At the elementary level, this means that the system must be composed of subsystems that distribute cleanly over the manufacturing enterprise. More specifically, their time constants, transient responses, and product-to-machine interfaces must match, recognizing that any manufacturing constraint means a product constraint and vice versa. At a more sophisticated level, the elements of the process, like quality measurement and control, must be matched across the product-system and manufacturing system boundaries.

As suggested earlier, the key technological innovation is likely to be the architecture of the joint information system. In that connection, one of the greatest determinates of the information system's size, speed, and resolution is the quality of the end product and the yield of its manufacturing process—that is, their defect rates. The higher these defect rates, the greater the size, cost, complexity, and precision of the information system that will be needed to find and eliminate them quickly.

Another strong determinant of information system capacity is piece part count, another factor dependent on the match of product and manufacturing techniques. Mechanical engineers have known this for years: whenever possible, replace a complicated assembly of parts with a single, specialized piece. Nowhere is the advantage of piece part reduction as evident as in the continuing substitution of more and more high-capacity microprocessors for their lower-capacity predecessors. Remarkably, this substitution, for approximately the same cost, also decreases the defect rate per computational operation. It appears to be an inevitable consequence of the different parts of Moore's law. As technology allows more and more transistors per unit area, the cost of the fabrication plant likewise rises. The rise in cost of the fabrication plant drives the market toward increasing standardization of parts (to spread large capital costs over many units). Increasing standardization means greater regularization, and higher quality to achieve economic throughput in an expensive fabrication plant. The end-user value added can then come only from software (a topic we take up later).

And, especially for product lines, the fewer different parts from model to model, the better, as long as that commonality does not decrease system capability unduly. Once again, there is a close correlation between reduced defect rate, reduced information processing, reduced inventory, and reduced complexity—all by design.

Looking further into the future, an extension of the lean production architecture is not the only possibility for flexible manufacturing. It is possible that flexible manufacturing could take a quite different architectural turn based on a different set of priorities. Is ultraquality production really necessary for simple, low-cost, limited-warranty products made largely from commercial, off-the-shelf (COTS) units (for example, microprocessors and flat screens)? Or is the manufacturing equivalent of parallel processors (pipelines) the answer? Should some flexible manufacturing hark back to the principles of special, handcrafted products or one-of-a-kind planetary spacecraft? The answers should be known in less than a decade, considering the profit to be made in finding them.

HEURISTICS FOR ARCHITECTING MANUFACTURING SYSTEMS

- The product and its manufacturing system must match. (In many ways.)
- Keep it simple. (Ultraquality helps.)
- Partition for near-autonomy. (A trade-off with feedback.)
- In partitioning a manufacturing system, choose the elements so that they minimize the complexity of material and information flow (Savagian, 1990, USC).
- Watch out for critical misfits. (Between intersecting waterfalls.)
- In making a change in the manufacturing process, how you make it is often more important than the change itself. (Management policy.)
- When implementing a change, keep some elements constant to provide an anchor point for people to cling to. (Schmidt, 1993, USC) (A trade-off when a new architecture is needed.)
- Install a machine that even an idiot can use and pretty soon everyone working for you is an idiot. (Olivieri, 1991, USC) (An unexpected consequence of mass production Taylorism—see next heuristic.)
- Everyone a customer, everyone a supplier.
- To reduce unwanted nonlinear behavior, linearize!
- If you cannot analyze it, do not build it.
- Avoid creating the same resonance time constant in more than one location in a [production] system.
- The five why's. (A technique for finding basic causes, and one used by every inquisitive child to learn about the world at large.)

CONCLUSION

Modern manufacturing can be portrayed as an ultraquality, dynamic feedback system intersecting with that of the product waterfall. The manufacturing systems architect's added tasks, beyond those of all systems architects, include (1) maintaining connections to the product waterfall and the software spiral necessary for coordinated developments, (2) assuring quality levels high enough to avoid manufacturing system collapse or oscillation, (3) determining and helping control the system parameters for stable and timely performance, and (4) looking farther into the future than do most product-line architects.

EXERCISES

1. Manufacturing systems are complex systems that need to be architected. If the manufacturing line is software intensive, and repeated software upgrades are planned, how can certification of software changes be managed?
2. The feedback lags or resonances of a manufacturing system of a commercial product interact with the dynamics of market demand. Give examples of problems arising from this interaction and possible methods for alleviating them.
3. Examine the following hypothesis: Increasing quality levels in manufacturing enable architectural changes in the manufacturing system that greatly

increase productivity but may make the system increasingly sensitive to external disruption. For a research exercise, use case studies or a simplified quantitative model.

4. Does manufacturing systems architecture differ in mass production systems (thousands of units) and very low-quantity production systems (fewer than ten produced systems)? If so, how and why?
5. Current flexible manufacturing systems usually build very small lot sizes from a single product line in response to real-time customer demand; for example, an automobile production line builds each car to order. Consider two alternative architectures for organizing such a system, one centralized and one decentralized. The first would use close centralized control, centralized production scheduling, and resource planning. The other would use fully distributed control based on disseminating customer/supplier relationships to each work cell; that is, each job and each work cell interact individually through an auction for services. What would be the advantages and disadvantages of both approaches? How would the architecture of the supporting information systems (extending to sales and customer support) have to differ in the two cases?

REFERENCES

Boehm, B. W. (1984). "Software engineering economics." *IEEE Transactions on Software Engineering* **4**(1): 4–21.

Brooks Jr, F. P. (1995). *The Mythical Man-Month (Anniversary ed.)*, Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Hays, R., S. Wheelwright, and K. Clark (1988). *Dynamic Manufacturing: Creating the Learning Organization*, New York: Free Press.

Ingrassia, P. (2011). *Crash Course: The American Automobile Industry's Road to Bankruptcy and Bailout-and Beyond*, Westminster, MD**:** Random House Trade Paperbacks.

Ingrassia, P. and J. B. White (1994). *Comeback: The Fall & Rise of the American Automobile Industry*, New York: Simon and Schuster.

Langfitt, F. (2015). *NUMMI. This American Life*, Washington, DC: National Public Radio.

Ohno, T. (2019). *Toyota Production System: Beyond Large-Scale Production*, New York: Productivity Press.

Olivieri, (1991) USC. In Rechtin, E., (ed.). *Collection of Student Heuristics in Systems Architecting, 1988–1993*, Los Angeles, CA: University of Southern California (unpublished but available to students and researchers on request).

Rechtin, E. (1991). *Systems Architecting: Creating and Building Complex Systems*, Englewood Cliffs, NJ: Prentice Hall.

Savagian, (1990) USC. In Rechtin, E., (ed.). *Collection of Student Heuristics in Systems Architecting, 1988–1993*, Los Angeles, CA: University of Southern California (unpublished but available to students and researchers on request).

Schmidt, (1993) USC. In Rechtin, E., (ed.). Collection of Student Heuristics in Systems Architecting, 1988–1993, Los Angeles, CA: University of Southern California (unpublished but available to students and researchers on request).

Womack, J. P., D. T. Jones, and D. Roos (2007). *The Machine that Changed the World: The Story of Lean Production—Toyota's Secret Weapon in the Global Car Wars that is Now Revolutionizing World Industry*, New York: Simon and Schuster.

Case Study 4

Intelligent Transportation Systems

INTRODUCTION

Intelligent transport systems (ITSs) (Maier 1997) are members of a class of systems in which humans and their behavior are inextricably part of the system. We refer to these as "sociotechnical systems." They are also systems whose architectures are distributed, in both a logical and physical sense, and are equally distributed in their development, procurement, and management. The key characteristics of such systems, which will jointly help define the concept of a collaborative system in Chapter 7, include (1) the lack of a single client with ownership and developmental responsibility for the system, (2) considerable uncertainty in system purposes and a recognition that purposes will evolve in unknown directions over its lifetime, and (3) the need for extensive voluntary cooperation in their deployment and use. This last point, creation through voluntary cooperation and interaction, will be the central insight of Chapter 7. In Chapter 5, we deal more generally with the concept of sociotechnical systems, where humans and their behaviors are inside rather than outside the system boundary.

ITS concepts have been around for several decades but started getting serious attention in the 1990s. ITS in general refers to transportation-related guidance, control, and information systems. These systems use computer and information technology to address transportation functions at the level of individual vehicles, roadways, and large transportation networks. The motivator for developing such systems is the belief that they will improve transport network flow, improve safety, reduce environmental impact, and be large commercial market opportunities. Many believe that the transport improvements gained through the application of information technology promise to be cheaper and less environmentally damaging than further expansion of the physical transport infrastructure. Over the long term, ITS could evolve into automated highways where vehicles are automatically controlled for even larger gains in system performance.

ITS is an area that has evolved extensively over the lifetime of this book. At the time of the first edition, planning was defined by concepts documented in reports like (IVHS 1992). When the references used here were written, intelligent transportation services were mostly speculative; only a very limited set was available. At the time of the writing of this fourth edition, a variety of intelligent transport services are commonly available. In-vehicle navigation assistance is common, either from built-in systems or the integration of smartphones into the vehicle. The systems provide route planning and real-time route guidance. Phone apps provide multi-modal guidance, integrating knowledge of public transport systems and private vehicles. Information

DOI: 10.1201/9781003483595-10

technology was integral to the rise in ride-hailing services. Position monitoring systems are in fairly wide use in commercial vehicles and public transportation fleets. Real-time traffic conditions are available through websites (Government and private) and via many apps. Several different online services provide free maps of virtually every city, and nonurban areas in industrialized countries, with route planning services. Many metro areas use intelligent traffic control methods in managing their stoplights, road admittance systems, and demand lanes.

In most areas, what does not exist today is the interconnection of these various services and the centralized exploitation of both information and controls. This is striking because many of the early concepts and proponents emphasized the role and value of centralized control. Some experiments toward centralized systems have been made in a few cities, and there is continuing interest in further integration, although less than in the first run of enthusiasm in the 1990s. The actual experience points out that the split between public and private control and responsibility is an architectural choice. When certain services are to be provided voluntarily by market means, that is a choice on the overall structure of the system. When certain services are reserved to government control, likewise that is a choice on the overall structure of the system. To understand the architecture of social systems, one element is the division between public and private means. This highlights the sociotechnical nature. Humans are inside the system, and human constructs are very much part of the architecture.

The purpose of this case study is to illustrate the use of architecting concepts and heuristics in sociotechnical systems, not to write a history of intelligent transportation systems. The case study should be read in that spirit. The comparison between what was suggested in the 1990s and how architecture was reasoned then can be illuminated by what has played out, but the key is to follow the reasoning in the context of its time.

ITS CONCEPTS

Possible ITSs have been extensively described in the literature (IVHS 1992). The most common decomposition of ITS services is into five core service areas plus automated highways, which has been considered somewhat farther out. The automated highway notion has been significantly superseded by attention to autonomous driving. The five core services as have been usually defined (and mapped to typical provisioning methods) are:

Advanced Traveler Information Services (ATIS)

ATIS is the provision of accurate, real-time information on transportation options and conditions to a traveler anytime, anywhere. For example,

1. Computer-assisted route *planning* to a street location anywhere in the country. This service could be coupled with traffic prediction and multimode transport information. Extensive capabilities in this category are available from various smartphone apps, web services, and dedicated in-vehicle navigation systems.

2. Computer-assisted route *guidance* to a street location anywhere in the country, again possibly coupled with real-time traffic information and predictions. While logically distinct, #1 and #2 are usually transparently joined in all popular applications.
3. Access to full public transportation schedules in a distant city before leaving for that city. Again, this is widely available today in smartphone apps and websites.
4. Broadcast of real-time and predictive traffic conditions on the major roads of an urban area. Today, this is available with moderate fidelity through apps, websites, and radio (for the few who still listen).
5. Emergency situation and location reporting, manual and automated. Various private services, some tied to particular auto manufacturers, now provide this. Depending on the service, you could have this with virtually global coverage via very small satellite transceivers.

Advanced Traffic Management Systems (ATMS)

The intent of ATMS is to improve the carrying capacity and flow of the road network by integrating traffic sensors, remotely operated traffic signals, real-time monitoring and prediction, and dissemination of route information. The service components of ATMS include sensing traffic conditions in real time over wide areas, real-time prediction of traffic conditions, and remotely controlling traffic signals and signage from central control centers to optimize road network conditions. A long-term concept in ATMS is coupling traffic management with route selection in individual vehicles.

ATMS exist today, although their penetration has been less than many of the advocates hoped. Coupling of traffic management with individual route selection is almost nonexistent, wide area prediction is limited, but wide area real-time monitoring is common. Your in-vehicle navigator will make route recommendations that try to compensate for traffic and road conditions. The navigation applications know about road closures and slowdowns and re-routes in response. If the routing uses any cooperative mechanisms with local traffic managers, it is not evident, but like many such applications, what goes into the private, proprietary algorithms is opaque. Some jurisdictions make use of considerable centralized control, including additional mechanisms not previously listed, such as demand pricing.

Advanced Vehicle Control Systems (AVCS)

AVCS covers driver assistance systems within a single vehicle. This is an area of continuing advancement with some steady roll-out in production vehicles. Some examples include:

1. Partially automated braking systems. Today, antilock brake automation is common, with additional levels of automation rare.
2. Obstacle warning and avoidance. Backup sensors and cameras are common, as are blind spot warning systems and other types of proximity warnings. Some stability enhancement in emergency avoidance maneuvers has been done.

3. Automated driver assistance in distance following or lateral lane keeping.
4. Vision enhancement in reduced visibility conditions. Again, a few systems are available.
5. Various levels of true autonomous driving are in experimental stages as of the writing of this fourth edition. The development of these capabilities to the point of on-the-street regular use has been protracted and significantly controversial. Much of the controversy speaks directly to issues in how the machines interact with people and how people interact with the machines.

Commercial Vehicle Operations (CVO)

CVO dealt with the automation of regulatory functions and record keeping, especially in interstate travel. The goal was reduction of time and expense due to regulatory requirements in road transport. More broadly, this covers a wide basket of services relevant to business use of transportation. Although the roll-out to public infrastructure has been limited, there has been extensive usage in private fleets. Some examples of proposed CVO capabilities include:

1. Weigh-in-motion for trucks.
2. Electronic license/tag/permit checking and record keeping.
3. Hazardous cargo monitoring (coupled with navigation and position reporting).
4. Position monitoring and reporting for fleet management.

Position monitoring and reporting have come into very wide use in commercial operations, and not just for vehicles. Tagging of transportation containers and individual packages is common, as is data recording on more sensitive units (e.g., monitoring of the state of a refrigerated container). There are many specialized suppliers of hardware and software for niche application areas.

Advanced Public Transport (APT)

The goal of APT is performance improvements in public transport through the application of intelligent vehicle and highway system (IVHS) technology. Some examples include:

1. Real-time monitoring of bus, subway, or train position coupled with waiting area displays and vehicle dispatch. These systems have proved popular and reasonably effective where deployed.
2. Electronic fare-paying systems to improve stopping times and allow time-sensitive pricing. Many systems have moved to smart cards and related electronic payment systems.

ITS ARCHITECTURE AND SOCIOTECHNICAL ISSUES

An ITS is unquestionably a sociotechnical system in the sense that humans and their behavior are irreducible components. People decide whether or not to use route planning and guidance systems. When given route advice, they choose to use it or ignore it. People choose to purchase (or not) various components of an ITS. At the political level, people make joint decisions through their government on whether to make infrastructural investments. So, any discussion of the architecture of an ITS must include people, and the architecting of an ITS must incorporate the sociotechnical nature of the system. The heuristics of sociotechnical systems are the primary focus of Chapter 5, with Chapter 7 taking up the somewhat narrower, more specialized case of collaborative systems. To illustrate, consider how the issues have been resolved, in practice, for the ITSs that now exist and are emerging.

Who is the Client for an Architect?

Borrowing a phrase (Schuman 1994), and ITS will be a system no one owns. At the broad level, this is clearly true. Nobody owns or controls the whole of ITS systems and capabilities that operate in the United States, or Japan, or any other major country. ITS planning, at least in the United States, has been at an advisory level. Purchase, deployment, and use have been the result of distributed decisions by governments and consumers. Studies from the 1990s saw that as likely (Federal Highway Administration 1991). In consequence, the integrity of any ITS architecture must be maintained through some similarly distributed means. When a single client (agency, company, or individual) commissions a system, the integrity of that system can be maintained by an architect hired by the client. When no client with that power exists, no architect with the power can exist either. This complicates the architecting problem. Lacking the power to directly establish and maintain the architecture, the architect must find indirect means to do so.

In systems architecting, it is common for the actual users to be different from the system sponsor. When this occurs, the architect must be conscious of the possibility that the preferences and needs of the ultimate users are different from those of the sponsors, or as perceived by the sponsors. The system might seem perfectly satisfactory to the sponsors, and yet be unacceptable to the users. If the system is rejected by the users, the sponsor is unlikely to perceive the system as successful.

This situation is even more extreme in the case of an ITS. To date, there has been little centralized architecting of ITSs, at least of those elements deployed and widely used in the United States. The U.S. Department of Transportation (DOT) sponsored rather extensive ITS architecture studies in the 1990s. But the U.S. DOT has only limited authority to direct or mandate transportation developments in the United States. Execution is up to states, metropolitan traffic authorities, cities, and individuals. By analogy, an architect for an ITS is more in the position of an urban planner than a civil architect. The urban planner has a great deal of influence but relatively little power. For an urban planner to be effective requires considerable political skill, and sponsors who understand the limits of their own paper. Through city governments and zoning boards, urban planners can possess a "no" authority, that is, the ability to say "no" to nonconformant plans. However, their ability to say "yes" is much less and happens only if the government employing the urban planner is willing to commit funds.

Actual execution has been dominated by private companies pursuing targeted capabilities they think are economically valuable. Google and Apple (and some others) found value in building massive map databases and routing advice and building that into their phone app environments. Companies like Garmin, which specialized in small GPS terminals, found that providing services running on or via those terminals was a better business. Iridium, who built a global satellite network to support telephony, found that facilitating a market for object tagging and tracking was much more economically viable.

Public or Private?

In the first wave of excitement over ITSs, many of the concepts seemed to assume a dominantly public infrastructure. But, in practice today, most of the interesting traveler information services are private. In-vehicle GPS navigators were privately produced and purchased and are now largely replaced by smartphone integration. The online map services are private and mostly ride on the ecosystems of phone apps. Because the architecture of an ITS is not currently centrally directed by government (at least in the United States), it is probably not a surprise that the private side has been the side that has most extensively developed.

Even if more centralized architecting had been done, the result might well have been the same, although there might have been niche areas where centralized decision-making could take hold. As an elaboration, consider the problem of social collaboration and fully coupled control. Fully coupled control is the idea that route recommendation and guidance to travelers can be used in conjunction with traffic control mechanisms and that *the guidance can itself be used as a traffic control mechanism*. In order for a fully coupled control scheme to be effectively used, the driver compliance rate must be high. Why will drivers willingly comply with centralized route guidance that is knowingly being computed with the benefit of the whole in mind? Will people just game the system?

There is a useful heuristic from (Rechtin 1991):

> In introducing technological and social change, how you do it is often more important than what you do. If social cooperation is required, the way in which a system is implemented and introduced must be an integral part of its architecture.

Using this heuristic requires identifying those architectural characteristics that lead to cooperative acceptance and use. In the ITS case, what system characteristics are most likely to foster public cooperation and acceptance? The answer will not be identical for all countries and cultures. We suggest (Maier 1997) for the United States that ITS general acceptance will be greater for those services that are privately and voluntarily contracted for.

The deployment mode for an architectural element should be, in order of preference:

1. Private development and purchase.
2. Private development and purchase subject to governmental guidelines or standards.
3. Private development and purchase subject to government mandating.
4. Government-financed development and private purchase.
5. Government mandating of deployment with direct finance.

A consequence of using these criteria is that ITS services should be partitioned to support and encourage private development of particular packages. Such packaging requires groupings that provide income streams a private firm can capture and defensible markets. This criterion suggests that the technical architecture should support such decentralization in development and deployment even where centralization would be more "efficient," say on a life-cycle cost basis.

Although the issue of public–private partitioning is not so prominent in early writings on ITSs, as noted, it has played an important role in actual development. The greatest ferment in ITSs has been in private systems, and private systems have avoided problems of perception of invasion of privacy and monitoring. Of course, the reality is that private monitoring is also monitoring, and people know that very well, but they find it acceptable. This only reemphasizes that perceptions may matter more than facts.

FACTS AND PERCEPTIONS

Continuing on that same line of facts versus perceptions, consider how ITS-like sociotechnical systems are judged as successes or failures. How would we know if ITSs were successful or not? A traditional systems engineering approach would immediately appeal to measures of effectiveness, probably measures like average speed on the roads, road throughput, life-cycle cost, incident rate, and various other measures that are easy to imagine and cite. But do the actual stakeholders of ITSs (government authorities and the traveling public) perceive those as measures of success? Another heuristic: *Success is in the eye of the beholder*, suggests not. In an earlier edition of this book, we framed the following thought experiment. Fifteen years after the original framing of the thought experiment is about when the fourth edition of this book was released and so makes it timely to consider how the thought experiment has played out in reality:

Scenario 1: It is 15 years later than the present. Intelligent transportation systems are widely deployed in most major urban areas and are very heavily used. Most urban areas have five or more competing, private traffic information service providers. There is extremely active, competitive development of supporting communication, display, and algorithmic systems. Market penetration of rich services is above 85%. But traffic in major urban areas is much worse than current. Most measures of effectiveness (e.g., average travel time and average speed) have decayed.

Scenario 2: Again, it is 15 years beyond the present. ITSs are likewise widely deployed and widely used. Now the various measures are substantially improved. But deployment and compliance have come only from vigorous mandates and enforcement. In many jurisdictions, the mandates have been tossed out by popular vote. Effectiveness is demonstrably highest in those jurisdictions with the strictest enforcement and the least responsiveness to popular demand.

Although the reader may consider the scenarios fanciful and unrealistic, that is not the point. Perhaps they seem unrealistic in the light of what has happened, and that there never seemed to be much serious consideration of a Scenario 2 like case. The point is to ask, seriously, whether a system should be judged by whether it gives the users what they want, regardless of whether the architect thinks what they want makes any sense. The classical paradigm says what the sponsors want is what matters, not what the architect thinks makes sense. In a sociotechnical and collaborative system, where voluntary interaction is essential to system operation, what the users think they want is more important than what the sponsor wants. What we have now is a situation more like Scenario 1, because the dominant deployments are private.

Although the exact scenarios are not reality, the underlying observation about success not necessarily coming from the original objectives has been proved in practice. In several cases where ITSs have been found successful in use, a major source of satisfaction has been through the impact on travel time variance and predictability, and not on average speed. That is, the most valued impact has been how it enables users to accurately predict how long a trip will take and make travel times more consistent, rather than making the average time shorter. This has been particularly true in CVO, where understanding and control of uncertainty deeply impacts many use-cases.

Architecture as Shared Invariants

One way of envisioning the architecture of something as complex as an ITS is by looking at the shared invariants. For an ITS, this means looking for the things shared across many or all users and that do not change much with time. In the sense of shared invariants, some of the things that could be an ITS architecture include the following:

- Shared positioning services. This would most obviously be GPS but extends to alternative, non-GPS mechanisms like those tied to cellular or beacon networks.
- Map data, specifically the network of roads and their positions relative to GPS locations.
- How digital traffic messages are encoded. How do you digitally indicate that the flow at a given point on a given road has some value, in a system-independent way.
- Mobile communication networks over which traffic information flows (networks that may not belong to ITS).

Much less of this kind of invariant definition has been done than could have been done. In practice, it has been hard to get centralized attention to elements that are supportive behind the scenes but are not delivering services that benefit directly.

Dominance of Economics

Finally, a theme of sociotechnical systems that is strongly evident in ITS is the role of, or the dominance of, economics. Today, what we have in an ITS has largely been driven by what makes a profitable business, in many cases rather independently of doing anything about travel. The online map services, for example, are widely used and popular. In the course of a few short years, people have gone from mostly using paper maps to referring to maps on their phones to just following phone directions (and perhaps not being able to read maps). Many websites are now modified to simply link to one of the main map services whenever a location must be provided.

What drives the map services? Because they are free and rather modest appendages to larger Internet firms, the answer is either advertising or data sales from aggregating user interactions. A popular web service attracts users, a large user base brings advertising dollars. Map services are a particularly fine advertising target because the nature of the search strongly suggests what the user is looking for, hence assisting in advertising targeting. User queries, sufficiently aggregated, are a very valuable information source that can be monetized. A user can opt out of advertising and data collection by using an offline version instead, but usage rates have proven that most users are not reluctant to abandon a small slice of privacy about their location searches in order to have continuously updated, online map information. It works because the economics works.

The broader lesson to consider in sociotechnical systems is that business is usually a part of them. “Stable forms” include the notion of economically stable, not just technically stable. In Chapter 13 in Part IV, we will discuss politicotechnical systems. In a politicotechnical system, stability is likewise critical, but there it comes mostly from the nature of the constituency instead of the business model.

REFERENCES

Federal Highway Administration (1991). *Intelligent Vehicle Highway Systems: A Public Private Partnership*, Washington, DC: Federal Highway Administration.

IVHS (1992). *Strategic Plan for Intelligent Vehicle-Highway Systems in the United States*, Washington, DC: IVHS American.

Maier, M. W. (1997). “On architecting and intelligent transport systems.” *IEEE Transactions on Aerospace and Electronic Systems* **33**(2): 610–625.

Rechtin, E. (1991). *Systems Architecting: Creating and Building Complex Systems*, Englewood Cliffs, NJ: Prentice Hall.

Schuman, R. (1994). Developing an architecture that no one owns: The US approach to system architecture. *Proceedings of the First World Congress on Applications of Transport Telematics and Intelligent Vehicle-Highway Systems*, Paris, France.

5 Social Systems

INTRODUCTION: DEFINING SOCIOTECHNICAL SYSTEMS

Social: Concerning groups of people or the general public.
Technical: Based on physical sciences and their application
Sociotechnical Systems: Technical works involving significant social participation, interests, and concerns.

Sociotechnical systems are technical works involving the participation of groups of people in ways that significantly affect the architecture and design of those works. Historically, the most conspicuous has been the large civil works—monuments, cathedrals, urban developments, dams, and roadways among them. Lessons learned from their construction provide the basis for much of civil (and systems) architecting and its theory (Lang 1987).

More recently, others, although physically quite different, have become much more sociotechnical in nature than might have been contemplated at their inception—ballistic missile defense, air travel, information networks, welfare, and health delivery. Few can even be conceived, much less built, without major social participation, whether planned or not. Experiences with them have generated a number of strategies and heuristics of importance to architects in general. Several are presented here. Among them are three heuristics: the four who's, economic value, and the tension between perceptions and facts. In the interests of informed use, as with all heuristics, it is important to understand the context within which they evolved, the sociotechnical domain. Then, at the end of the chapter, there are a number of general heuristics of applicability to sociotechnical systems in particular.

PUBLIC PARTICIPATION

At the highest level of social participation, members of the public directly use—and may own a part of—the system's facilities. At an intermediate level, they are provided a personal service, usually by a public or private utility. Most importantly, individuals and not the utilities—the architect's clients—are the end users. Examples are highways, communication and information circuits, aviation traffic control, and public power. Public cooperation and personal responsibility are required for effective operation. That is, users are expected to follow established rules with minimal policing or control. Drivers and pilots follow the rules of the road. Communicators respect the twin rights of free access and privacy.

At the highest level of participation, participating individuals choose and collectively own a major fraction of the system structure—cars, trucks, aircraft, computers, telephones, electrical and plumbing hardware, and so on. In a sense, the public "rents" the rest of the facilities through access charges, fees for use, and taxes. Reaction to a facility failure or a price change tends to be local in scope, quick, and focused. The public's voice is heard through specialized groups such as automobile clubs for highways, retired persons associations for health delivery, professional societies for power and communications services, and similar groups. Market forces are used to considerable advantage through adverse publicity in the media, boycotts, and resistance to stock and bond issues on the one hand, and through enthusiastic acceptance on the other. Recent examples of major architectural changes strongly supported by the public are superhighways, satellite communications, entertainment cable networks, jet aircraft transportation, health maintenance organizations, and a slow shift from polluting fossil fuels to alternative sources of energy. Since the publication of the first edition of this book, there has been the massive rise of social media. Social media is clearly a public utility in the sense described above, though the regulatory model has not matched that of other public utilities. Compared to the time it took for other public utilities to form and stabilize we are still in an early period, so perhaps the final form of how social media is treated is yet to be determined.

Systems of this sort are best described as collaborative systems, systems that operate only through the partially voluntary initiative of their components. This collaboration is an important subject in its own right, because the Internet, the World Wide Web, and open-source software are collaborative assemblages. We address this topic in detail in Chapter 7.

At the other extreme of social participation are social systems used solely by the government, either directly or through sponsorship, for broad social purposes delegated to it by the public; for example, National Aeronautics and Space Administration (NASA) and U.S. Department of Defense (DoD) systems for exploration and defense, Social Security, and Medicare management systems for public health and welfare, research capabilities for national well-being, and police systems for public safety. The public pays for these services and systems only indirectly, through general taxation. The architect's client and end user are the government. The public's connection with the design, construction, and operation of these systems is sharply limited. Its value judgments are made almost solely through the political process, which is the subject of Chapter 13. They might best be characterized as "politicotechnical."

The phrase "system-of-systems" is now commonly used in the systems engineering literature, although not with a consistent definition. Because the term system-of-systems is ambiguous on its face (is any system whose subsystems are complex enough to be regarded as systems a system-of-systems?), we prefer the terms we use here. In many writings, sociotechnical systems and systems-of-systems are conflated. In others, collaborative systems and systems-of-systems are conflated. Families of systems, product-lines, and portfolios-of-systems are also groupings of systems, each distinct in its own way and each is associated with a distinct set of best practices. For the purposes of this chapter, we will establish the foregoing definition for sociotechnical systems and will explore the notion of collaborative systems at some length.

Narrowly Sociotechnical: Humans as Essential Elements

The primary focus of this chapter is on systems where the actions of the public interacting with the system are central, where the human actions greatly influence how the system operates, how well it performs, and whether it can be successful. We should also note that human behavior can become an integral part of system operation and limits, even at a small scale, when humans and their choices are an irreducible part of the system. In complex enterprises like manufacturing, weather forecasting, or medical services, humans make individual decisions about what system elements to use (or ignore) and how to use them. There are organizational norms, but those norms may allow for great flexibility on the part of the humans involved to choose what tools to use and how to prioritize their use. Those choices may have strong effects on overall effectiveness but not be closely controlled, either by choice or because social structures may make close control impossible.

As an example, the present author interviewed weather forecasters on their use of different data sources as part of the effort to identify future observation capabilities. A striking finding was how much things varied among different forecasters even within the same mission area. Forecasters have access to many observation sources, both in their direct form (the observation itself) and the results of large numerical models that have assimilated the same observational data. Some forecasters treat the numerical model results as the foundation and look to the source data primarily to identify or explore anomalies. Other forecasters build their own estimates of atmospheric conditions from observations and then use the numerical model results to cross-check. As the timescale of the forecasts shifts, the weight of practice also shifts. For longer-term forecasts (multiple days out), the impact of global observations is decisive and almost no forecasters rely on manual estimates. The numerical model is king in this domain. At the opposite end, for very rapidly developing situations, like tornados, real-time assessment of observations is king. Numerical models are lagging indicators. But there is a broad middle ground where neither has proven to be inarguably decisive. Most likely the boundary will move over time and the situations where algorithms will be clearly superior to human assessment will grow, but how quickly that turns into established practice will be determined as much to social factors as by technical progress.

THE FOUNDATIONS OF SOCIOTECHNICAL SYSTEMS ARCHITECTING

The foundations of sociotechnical systems architecting are much the same as for all systems: a systems approach, purpose orientation, modeling, certification, and insight. Social system quality, however, is less a foundation than a case-by-case trade-off; that is, the quality desired depends on the system to be provided. In nuclear power generation, modern manufacturing, and manned space flight, ultra-quality is imperative. But in public health, pollution control, and safety, the level of acceptable quality is only one of many economic, social, political, and technical factors to be accommodated.

But if sociotechnical systems architecting loses one foundation, ultra-quality, it gains another—a direct and immediate response to the public's needs and perceptions. Responding to public perceptions is particularly difficult, even for an experienced architect. The public's interests are unavoidably diverse and often incompatible. The groups with the strongest interests change with time, sometimes reversing themselves based on a single incident. Three Mile Island was such an incident for nuclear power utilities. Pictures of the Earth from a lunar-bound Apollo spacecraft suddenly focused public attention and support for global environmental management. An election of a senator from Pennsylvania avalanched into widespread public concern over health insurance and Medicare systems.

The Separation of Client and User

In most sociotechnical systems, the client, the buyer of the architect's services, is not the user. This fact can present a serious ethical, as well as technical, problem to the architect: how should conflicts between the preferences, if not the imperatives, of the utility agency and those of the public (as perceived by the architect) be treated when preferences strongly affect system design?

It is not a new dilemma. State governments have partly resolved the problem by licensing architects and setting standards for systems that affect the health and safety of the public. Buildings, bridges, and public power systems come to mind as systems that affect public safety. Information systems are already on the horizon. The issuing or denial of licenses is one way of making sure that public interest comes first in the architect's mind. The setting of standards provides the architect some counterarguments against overreaching demands by the client. But these policies do not treat such conflicts as that of the degree of traffic control desired by a manager of an intelligent transportation system (ITS) as compared with that of the individual user/driver, or the degree of governmental regulation of the Internet to assure a balance of access, privacy, security, profit making, and system efficiency.

One of the ways of alleviating some of these tensions is through economics. Economics has important insights, as economics is fundamentally the study of social constructs. In addition, markets have evolved a variety of mechanisms, such as market segmentation, that effectively deal with essential problems that arise from the nature of sociotechnical, social, and collaborative systems architecting.

Socioeconomic Insights

Social economists bring two special insights to sociotechnical systems. The first insight, which might be called the four who's, asks four questions that need to be answered as a self-consistent set if the system is to succeed economically—*Who benefits? Who pays? Who provides? And, as appropriate, Who loses?*

Example: The answers to these questions of the Bell Telephone System were: (1) the beneficiaries were the callers and those who received the calls; (2) the callers paid based on usage because they initiated the calls and could be billed for them; (3) the provider was a monopoly whose services and charges were regulated by public agencies for public purposes; and (4) the losers were those who wished to use the telephone facilities for services not offered or to sell equipment not authorized for connection to those facilities. The telephone monopoly was incentivized to carry out widely available basic research because it generated more and better service at less cost, a result the regulatory agencies desired. International and national security agreements were facilitated by having a single point of contact, the Bell System, for all such matters. Standards were maintained and the financial strategy was long term, typically 30 years. The system was dismantled when the losers invoked antitrust laws, creating a new set of losers, complex billing, standards problems, and a loss of research. Arguably, it enabled the Internet sooner than otherwise. Subsequently, separate satellite and cable services were established, further dismantling what had been a single service system. The dismantlement also may have assisted in allowing the rapid rollout of wireless cellular telephone systems. In other countries, some of the most successful wireless cellular rollouts have occurred where wireless companies were the only allowed alternative to a government-sponsored telephone monopoly.

Example: The answers to the four who's for the privatized Landsat system, a satellite-based optical-infrared surveillance service, were as follows: (1) the beneficiaries were those individuals and organizations who could intermittently use high-altitude photographs of the Earth; (2) because the value to the user of an individual photograph was unrelated to its cost (just as is the case with weather satellite TV), the individual users could not be billed effectively; (3) the provider was a private, profit-making organization that understandably demanded a cost-plus-fixed-fee contract from the government as a surrogate customer; and (4) when the government balked at this result of privatization, the Landsat system faced collapse. Research had been crippled, a French government-sponsored service (SPOT) had acquired appreciable market share, and legitimate customers faced loss of service. Privatization was reversed and the government again became the provider.

Example: Serious debates over the nature of their public health systems are underway in many countries, triggered in large part by the technological advances of the last few decades. These advances have made it possible for humanity to live longer and in better health, but the investments in those gains are considerable. The answers to the four who's are at the crux of the debate. Who benefits—everyone equally at all levels of health? Who pays—regardless of personal health or based on need and ability to pay? Who provides—and determines the cost to the user? Who loses—anyone out of work or above some risk level, and who determines who loses?

Regardless of how the reader feels about any of these systems, there is no argument that the answers are matters of great social interest and concern. At some point, if there are to be public services at all, the questions must be answered and decisions made. But who makes them and on what basis? Who and where is "the architect" in each case? How and where is the architecture created? How is the public interest expressed and furthered?

Looking forward, the Four Who's heuristic, while especially important in socio-technical systems, is very broadly applicable in all systems architecting efforts as a guide in stakeholder identification. This is somewhat more tactical than the application here, but one marker of a good heuristic is the flexibility with which it can be applied. In Chapters 8–10, where we build a general process to carry out architecting studies, the Four Who's is foundational in stakeholder identification.

With the rapidly growing potential for autonomy in systems, perhaps the "fifth who" will soon take its place: Who takes responsibility? We can increasingly build systems that act autonomously in situations where historically humans have made decisions and chosen how to act. For example, autonomous vehicles and medical diagnostic systems. Historical practice has been that the system operator is primarily held responsible for the actions of the systems which the operator controls, modified somewhat by rules for liability in design and production. Autonomous systems challenge these historical arrangements, and what new arrangements are agreed will likely have a very strong effect on whether such systems will be deployed and if they will be successful. When an autonomous vehicle crashes is the owner responsible or the producer? Are those in the vehicle treated as operators or passive passengers? Can you sue a system developer for medical malpractice? In the near-term, these issues are likely to be finessed by interposing a human in the decision chain. But over the longer term, probably there will be no choice but to come to grips with issue more directly.

The issue will likely be most salient if or when autonomous systems evolve to the point they are undisputably superior in performance to human operators, but still imperfect. It is easy to imagine that autonomous driving and autonomous medical diagnosis might evolve to the point that it is more reliable than the average human, or even highly able humans, and yet still imperfect. The performance envelope might be complex, as it is for human operators, and differ substantially from that of human operators. If the economic advantages of autonomous operation are large enough, there will be pressure to find socially acceptable approaches to allocating responsibility. Past history suggests this may end up in a patchwork, perhaps treated distinctly in different societies.

The second economic insight is comparably powerful: In any resource-limited situation, the true value of a given service or product is determined by what a buyer is willing to give up to obtain it. Notice that the subject here is value, not price or cost.

Example: The public telephone network provides a good example of the difference between cost and value. The cost of a telephone call can be accurately calculated as a function of time, distance, routing (satellite, cable, cellular, landline, and so forth), location (urban or rural), bandwidth, facility depreciation, and so on. But the value depends on content, urgency, priority, personal circumstance, and message type, among other things. As an exercise, try varying these parameters and then estimating what a caller might be willing to pay (give up in basic needs or luxuries). What is then a fair allocation of costs among all users? Should a sick, remote, poor caller

> have to pay the full cost of remote TV health service, for example? Should a business that can pass costs on to its customers receive volume discounts for long-distance calling via satellite? Should home TV be pay-per-view for everyone? Who should decide on the answers?

These two socioeconomic heuristics, used together, can alleviate the inherent tensions among the stakeholders by providing the basis for compromise and consensus among them. The complainants are likely to be those whose payments are perceived as disproportionate to the benefits they receive. The advocates, to secure approval of the system as a whole, must give up or pay for something of sufficient value to the complainants that they agree to compromise. Both need to be made to walk in the other's shoes for a while. And therein can be the basis for an economically viable solution.

The Interaction between the Public and Private Sectors

A third factor in sociotechnical systems architecting is the strong interplay between the public and private sectors, particularly in advanced democracies where the two sectors are comparable in size, capability, and influence—but differ markedly in how the general public expresses its preferences.

By the middle of the 1990s, the historic boundaries between public and private sectors in communications, health delivery, welfare services, electric power distribution, and environmental control were in a state of flux. This chapter is not the place to debate the pros and cons. Suffice it to say, the imperatives, interests, and answers to the economists' questions are sharply different in the two sectors, see the discussion in Rechtin (1991, pp. 270–274) for additional insights. The architect is well advised to understand the imperatives of both sectors prior to suggesting architectures that must accommodate them. For example, the private sector must make a profit to survive; the public sector does not and treats profits as necessary evils. The public sector must follow the rules; the private sector sees rules and regulations as constraints and deterrents to efficiency. Generally speaking, the private sector does best in providing well-specified things at specified times. The public sector does best at providing services within the resources provided.

Because of these differences, one of the better tools for relieving the natural tension between the sectors is to change the boundaries between them in such negotiable areas as taxation, regulation, services provided, subsidies, billing, and employment. Because perceived values in each of these areas are different in the two sectors and under different circumstances, possibilities can exist where each sector perceives a net gain. The architect's role is to help discover the possibilities, achieve balance through compromise on preferences, and assure a good fit across boundaries.

Architecting a system, such as a public health system, that involves both the public and private sectors can be extraordinarily difficult, particularly if agreement does not exist on a mutually trusted architect, on the answers to the economist's questions, or on the social value of the system relative to that of other socially desirable projects. The problem is only exacerbated by the fundamental difficulties of diverse preferences. Public-sector projects look for a core of common agreement and an absence of "losers" sufficient to generate a powerful negative constituency.

Private ventures look for segments of users (more is often better) with the resources to fund their own needs.

Facts versus Perceptions: An Added Tension

Of all the distinguishing characteristics of social systems, the one that most sharply contrasts them with other systems is the tension between facts and perceptions about system behavior. To illustrate the impact on design, consider the following: Architects are well familiar with the trade-offs between performance, schedule, cost, and risk. These competing factors might be thought of as pulling architecting four different directions as sketched in Figures 5.1 and 5.2, which can be thought of as the next echelon or ring—the different sources or components of performance, schedule, cost, and risk. Notice that performance has an aesthetic component, as well as technical and sociopolitical sources. Automobiles are a clear example. Automobile styling often is more important than aerodynamics or environmental concerns in their architectural design. Costs also have several components, of which the increased costs to people of cost reduction in money and time being especially apparent during times of technological transition, and so on.

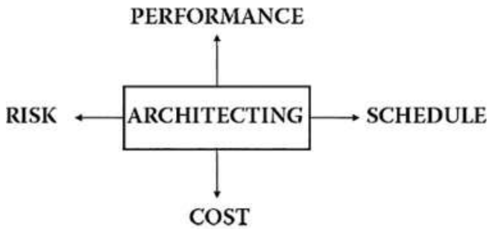


FIGURE 5.1 Four basic tensions in architecting.

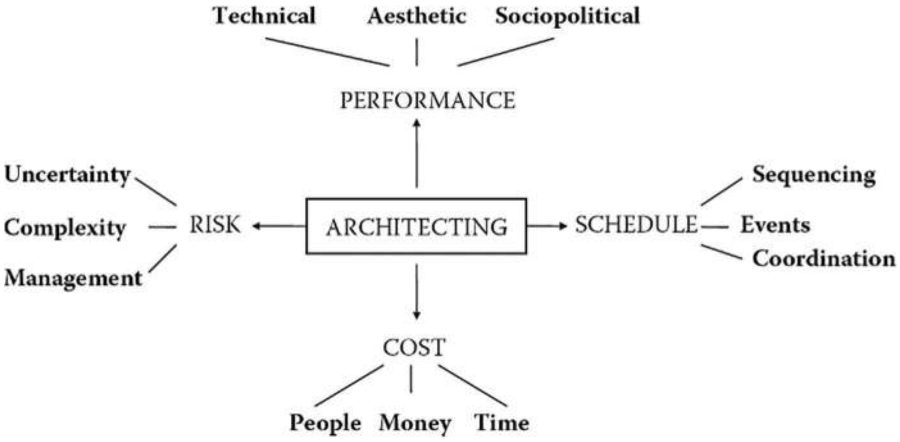


FIGURE 5.2 Underlying sources of the four tensions.

To these well-known tensions must be added another, one that social systems exemplify but which exist to some degree in all complex systems—namely, the tension between perceptions and facts, as shown in Figure 5.3. Its sources are shown in Figure 5.4. This added tension may be dismaying to technically trained architects, but it is all too real to those who deal with public issues. Social systems have generated a painful design heuristic: It is not the facts; it is the perceptions that count. Some real-world examples include the following.

- It makes little difference what facts nuclear engineers present about the safety of nuclear power plants, their neighbors' perception is that someday their local plant will blow up. Remember Three Mile Island and Chernobyl? A. M. Weinberg of Oak Ridge Associated Universities suggested perhaps the only antidote: "The engineering task is to design reactors whose safety is so transparent that the skeptical elite is convinced, and through them the general public" (Weinberg 1990).
- Airline travel has been made so safe that the most dangerous part of travel can be driving to and from the airport. Yet, every airliner crash is headline

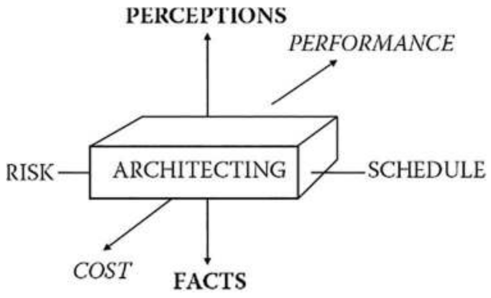


FIGURE 5.3 Adding facts versus perceptions.

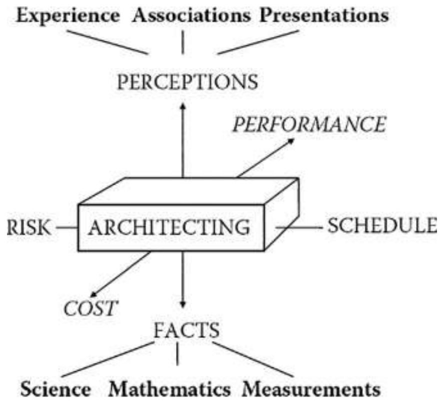

Experience Associations Presentations
PERCEPTIONS
PERFORMANCE
RISK
ARCHITECTING
SCHEDULE
COST
FACTS
Science Mathematics Measurements

FIGURE 5.4 Sources of facts and perceptions.

news. A serious design concern, therefore, is how many passengers an airliner should carry—200? 400? 800?—because, even though the average accident rate per departure would probably remain the same, more passengers would die at once in the larger planes and a public perception might develop that larger airliners are less safe, facts notwithstanding.

- One of the reasons that health insurance is so expensive is that health care is perceived by employees as nearly "free" because almost all its costs are paid for either by the employee's company or the government. The facts are that the costs are either passed on to the consumer, subtracted from wages and salaries, taken as a business deduction against taxes, or paid for by the general taxpayer, or all of the above. As any economist will explain, free goods are overconsumed.
- One of the most profound and unanticipated results of the Apollo flights to the Moon was a picture of the Earth from afar, a beautiful blue, white, brown, and green globe in the blackness of space. We certainly had understood that the Earth was round, but that distant perspective changed our perception of the vulnerability of our home forever, and with it, our actions to preserve and sustain it. Just how valuable was Apollo, then and in our future? Is there an equivalent value today?

Like it or not, the architect must understand that perceptions can be just as real as facts, just as important in defining the system architecture, and just as critical in determining success. As one heuristic states, the phrase, "I hate it," is a direction (Gradous 1993). There have even been times when, in retrospect, perceptions were "truer" than facts that changed with the observer, circumstance, technology, and better understanding. Some of the most ironic statements begin with, "It can't be done, because the facts are that…"

Alleviating the tension between facts and perceptions can be highly individualistic. Some individuals can be convinced—in either direction—by education, some by prototyping or anecdotes, some by A. M. Greenberg's antidote given earlier, some by better packaging or presentation, and some only by the realities of politics. Some individuals will never be convinced, but they might be accepting. In the end, it is a matter of achieving a balance of perceived values. The architect's task is to search out that area of common agreement that can result in a desirable, feasible system.

Looking more broadly, this is just a strengthened version of the basic admonition that an architect must know his or her client and what communicates to that client. It does no good to communicate precise and accurate representations that the client does not understand. Some clients are convinced only by prototypes. Some are convinced by analyses. In any case, the architect must understand what the audience in the domain of interest understands and will accept.

HEURISTICS FOR SOCIAL SYSTEMS

- Success is in the eyes of the beholder (not the architect).
- Do not assume that the original statement of the problem is necessarily the best, or even the right one. (Most customers would agree.)

- In conceptualizing a social system, be sure there are mutually consistent answers to the Four Who's: Who benefits? Who pays? Who supplies (provides)? And, as appropriate, Who loses?
- In any resource-limited situation, the true value of a given service or product is determined by what one is willing to give up to obtain it.
- The choice between the architectures may well depend upon which set of drawbacks the stakeholders can handle best. (Not on which advantages are the most appealing.)
- Particularly for social systems, it is not the facts, it is the perceptions that count. (Try making a survey of public opinion.)
- The phrase, "I hate it." is a direction. (Or were you not listening?)
- In social systems, how you do something may be more important than what you do. (A sometimes bitter lesson for technologists to learn.)
- When implementing a change, keep some elements constant as an anchor point for people to cling to. (At least until there are some new anchors.)
- It is easier to change the technical elements of a social system than the human ones. (Enough said.)

CONCLUSION

Social systems, in general, place social concerns ahead of technical ones. They exemplify the tension between perception and fact. More than most systems, they require consistent responses to questions of who benefits? who pays? who supplies? (provides, builds, and so forth), and, sociologically at least, who loses?

Perhaps more than other complex systems, the design and development of social ones should be amenable to insights and heuristics. Social factors, after all, are notoriously difficult to measure, much less predict. But, like heuristics, they come from experience, from failures and successes, and from lessons learned.

EXERCISES

1. Public utilities are examples of sociotechnical systems. How are the heuristics discussed in this chapter reflected in the regulation, design, and operation of a local utility system?
2. Apply the four who's to a sociotechnical system familiar to you. Examples: the Internet, air travel, communication satellites, or a social service.
3. Many efforts are underway to build and deploy intelligent transport systems using modern information technologies to improve existing networks and services. Investigate some of the current proposals and apply the four who's to the proposal.
4. Pollution and pollution control are examples of a whole class of sociotechnical systems where disjunctions in the four who's are common. Discuss how governmental regulatory efforts, both through mandated standards and pollution license auctions, attempt to reconcile the four who's. To what extent have they been successful? How did you judge success?

5. Among the most fundamental problems in architecting a system with many stakeholders is conflicts in purposes and interests. What architectural options might be used to reconcile them?
6. Give an example of the application of the heuristic, "When introducing technological change, how you do it is often more important than what you do."

REFERENCES

Gradous, L. (1993). *AE549 Student Report*, Los Angeles, CA: University of Southern California.

Lang, J. (1987). *Creating Architectural Theory*, New York: van Nostrand Rheinhold.

Rechtin, E. (1991). *Systems Architecting: Creating and Building Complex Systems*, Englewood Cliffs, NJ: Prentice Hall.

Weinberg, A. (1990). "Engineering in an age of anxiety: The search for inherent safety. Perspectives of nuclear power and carbon dioxide abatement." *Issues in Science and Technology* **6**: 2.

Case Study 5

MedInfo and Layered Systems

The discussion in Chapter 6 on software and information technology-centric systems will make extensive use of the concept of layered systems and multi-view representations. Layering stands in contrast to the classic notion of hierarchical decomposition, where any system is seen as composed of subsystems, which are in turn composed of smaller subsystems, and so on, until we reach a level of sufficient simplicity. Software is naturally constructed as a layered system rather than in the classic hierarchy. The software does not have the classic whole-part hierarchy of hardware. Or, rather, to the extent that it does there is a multiple hierarchy. As with the other chapters in Part II, we introduce the core concepts with examples taken from life before proceeding with the chapter. The case study in this section differs from some of the others in that it is not drawn from a single, named system. For this case study, it has been more convenient to combine and abstract several interrelated situations the authors have encountered over time. The individual stories either illustrate only a limited range of issues or are not available to publish with full acknowledgment. Nevertheless, a reader with experience should have little trouble drawing parallels in his or her own personal experiences. The core of the situation is the desire to transform a system from being organized (and constructed) dominantly around a hierarchical form to a layered form. In one common terminology, this is "de-stovepiping."

Key points to consider in this case study include the following.

- The contrasting logic of layered versus hierarchical construction and the presence of multiple, equally valid hierarchies. In each, ask what constitutes components, what constitutes connectors between components, and how each relates to others.
- The technical structure is (or should be) a reflection of business strategy. Choosing a layered architecture is foremost a business, or operational, strategic choice.
- The implementation, along with the means of implementation, matters a great deal in whether or not the business strategy is realized. Simply converting a hierarchical architecture to a layered architecture does not embody a coherent business strategy. Implementation of the strategy requires details of the implementation (a repeat of the heuristic of variable technical depth, but in a different guise).

DOI: 10.1201/9781003483595-12

- Going from hierarchical to layered is underdetermined. While the layered form is recognizable as-is, its implementations vary and differ in ways that require a more complex understanding of the multi-view concept. In particular, in software, we need to distinguish layering and hierarchy in logical, run-time, and development perspectives.

Many of these points are echoed and expanded in later chapters. For example, the relationship between business strategy and architecture is studied in depth in Chapter 12. The multi-view concept is extensively explored in Chapters 9 and 10 and we discuss the software-standard multi-view formulation known as “4+1” in Chapter 11. We introduce many of the key points here, which we will return to at greater length in later chapters.

The case study concerns the product-line architecture, or family-of-systems architecture if you prefer, of a fictitious company, MedInfo. In real life, the situation described has been encountered by the authors in both commercial business and Government agency contexts. In some contexts, it is referred to as the “de-stovepiping problem,” for reasons that will become apparent. The generality of the situation described to both business and Government groups should be apparent, but we make a few specific comments on different applications at the end.

BUSINESS BACKGROUND

Our fictitious case study company, MedInfo, makes a wide variety of medical imaging systems, including conventional x-rays, computed tomography (CT), magnetic resonance imaging (MRI), and others. The systems are sold to hospitals and clinics, both in the United States and internationally. Wherever one of the MedInfo systems is deployed, it will be integrated into the user’s technical infrastructure, at least as far as possible. At the starting point of this story, the systems are structured as “stovepipes”; that is, each system is designed, manufactured, sold, and operated as its own, stand-alone, system, which is illustrated in Figure CS5.1.

As described, the collection of MedInfo systems is a portfolio of systems (Maier 2019). The collection is a portfolio because they share common management (the company) and objectives (serving the market and making money). Each component system has its own objectives related to the specific mission/market area addressed. The collective can be said to also have objectives, relating to overall market coverage, risk profile, and competitive positioning. To be a family-of-systems in addition to a portfolio, there would have to be a common design or manufacturing linking the component systems. For example, shared computing hardware or shared software. To be a collaborative system or system-of-systems, they would need to interact with each other to produce some larger, collective functionality.

The progression of the MedInfo business has been dominated by steady upgrades to individual systems and the occasional introduction of new imaging systems. The upgrade path is what one would expect: additional user features, lowered cost, greater throughput, enhanced sensitivity, coverage area, and so forth. The management

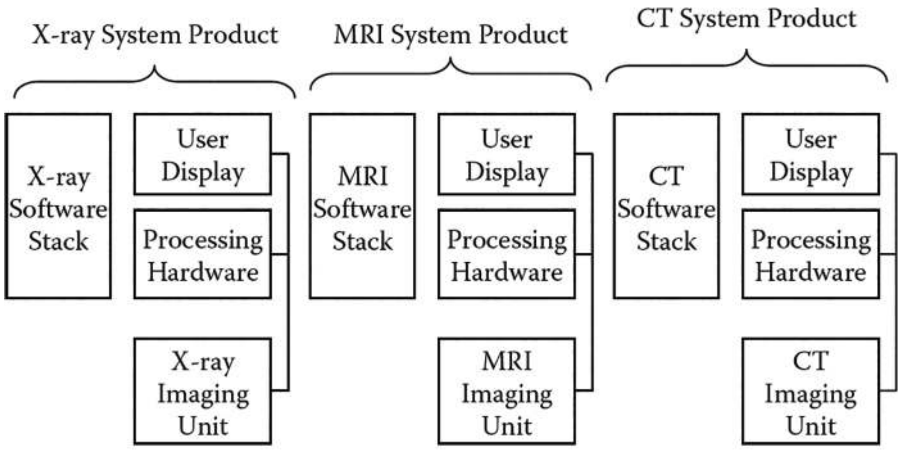


FIGURE CS5.1 MedInfo's initial situation. Multiple products are structured as stand-alone systems. Their nature as a product line is restricted to marketing and branding.

model for the portfolio is likewise simple as one would expect. Each of the products has an individual product manager. That manager is responsible end-to-end for that product. The manager leads design efforts, runs development and production, and is responsible for field performance, maintenance, and support. Although each product manager has many subordinates responsible for different aspects, all responsibility for that product ultimately lies with the product manager.

Each system has associated with it a supply chain of subcontractors and other suppliers. The subsystems or components supplied are each defined through specifications and interface control documents, written as needed based on the patterns of interconnection within each system.

MOTIVATION FOR CHANGE

If MedInfo has a solid record of success with things as they are, what motivation is there for change, especially relatively radical architectural change? The motivation for change is clearly not incremental improvement. Incremental, steady improvement is clearly possible and is already being realized with the current architecture. However, business strategy issues are pushing MedInfo toward restructuring their family of products. MedInfo management has identified four reasons for strategic change to how their products are developed, manufactured, and deployed (and hence how they need to be architected):

1. Software cost reduction
2. User demand for interconnection and integration
3. Rate of product turnover
4. Lateral and vertical product space expansion

Software Cost Reduction

MedInfo management has noticed that the fraction of total product development cost expended on software has risen steadily. A decade ago, it was 30% or less, but now it exceeds 70%. What used to be hardware-dominated products are now software-dominated products. The shift comes from multiple causes. The first is the continued commodification of hardware. Custom processors have disappeared, and larger and more complex hardware units are available through subcontracting. Second, and related, is that competitive differentiation is increasingly software-based. When competitors have access to the same hardware components, it is possible to competitively differentiate only through software.

User demands are also increasingly focused on software capabilities, such as processing algorithms, display forms, user customization, and the ability to support process automation. A major source of user demand, and a source of the movement toward higher-value fractions in software, is the need for interconnection and integration.

User Demand for Interconnection and Integration

Users are increasingly dissatisfied with stand-alone systems. A radiologist might need access to five different imaging technologies during the day and have to report from any or all of them within a hospital network. Few radiologists (much less other types of doctors) are happy with five computers on their desks and with manual file transfer among systems. Images need to be shared among a more diverse user base. The neurosurgeon wants to see the detailed MRI of the patient's spine, not just the radiologist's report, as does the specialized physical therapist. Users are increasingly demanding both interconnection and integration.

A very simple form of integration is collapsing the number of displays and computers on the desk needed to access multiple imaging systems. A more significant form of interconnection and integration is the ability to move data from different imaging systems onto a common reporting platform. A complex form of integration is being able to combine, overlay, and otherwise jointly process images from different systems. The most complex form of integration is where integration leads itself to new products and new concepts of operation by customers. An example here would be integrating medical imaging data into multidisciplinary diagnostic decision support systems, perhaps offering that diagnostic decision support system as its own product not integrated with imagers.

Rate of Product Turnover

In MedInfo's world, as in many other product spaces, there is increasing pressure to turn over products more quickly. User expectations for product cycles are getting shorter. As MedInfo's competition works to lower development cycle time, MedInfo is forced to match, at least. Management does not want to be average in the industry, but they want to lead it. As a practical matter, the most effective approach to increasing product turnover is to reduce the development size of each product, something at odds with the increasing demands for functionality, interconnection, and integration. The demands on capability drive up the size of each product's software base, and the need to increase product turnover would be best served by driving down the size of each product's software base.

Lateral and Vertical Product Space Expansion

Finally, the pressure to grow makes it important to continuously challenge the horizontal and vertical boundaries of the product space. If MedInfo machines are going to be integrated into larger medical information systems, then failing to move one's own boundary outward to encompass some of that larger information space leaves one open to being laterally consumed. If the processing and user interface side of a MedInfo imager is subsumed into a shared information system, the information system supplier will want to capture that part of the value stream and push MedInfo back to being a narrower hardware supplier. The threat of being subsumed is equally an opportunity to subsume. MedInfo thinks itself as having the greatest expertise in imaging hardware, but if they build an integrated diagnostic platform, part of the point would be to consume imagery from other providers. Integrated system markets can easily become "winner take all" markets, meaning one better try to be the winner.

THE LAYERED ALTERNATIVE

As MedInfo systems become software-dominated (in development cost) and increasingly interconnected, it becomes obvious that different products share a great deal of software. Networking, data storage and transformation, significant processing, and much user interface code are either the same or could easily be the same. Achieving integration is largely a matter of achieving protocol sharing, which is often most easily accomplished by code sharing. Systems that build with layers that carefully isolate the parts with the greatest potential to change from each other, through relatively invariant layers, generally have the highest rates of effective change. However, hierarchical system decomposition with end-to-end product managers with complete vertical responsibility does not encourage the discovery and management of that shared code.

An alternative architecture for the MedInfo product line that emphasizes horizontal layering is illustrated in Figure CS5.2. As we discuss later, this simplified picture ignores key details. It emphasizes a transition to a shared software infrastructure but does not specify if that shared infrastructure is in the form of source code, object libraries, or run-time services. Each is a distinct choice leading to different sets of advantages and disadvantages. Ignoring that complexity for a moment, in this variant, the actual deployed products may or may not look different than before. If a customer desires a stand-alone system, he can receive a stand-alone system. On the other hand, if a customer desires that the system display on a shared display system, then that can be accommodated. In either case, the same subsystems are present as before, but now those subsystems are drawn from a shared base. There is extensive, designed-in code sharing among the different elements of the family, with the shared elements forming layers.

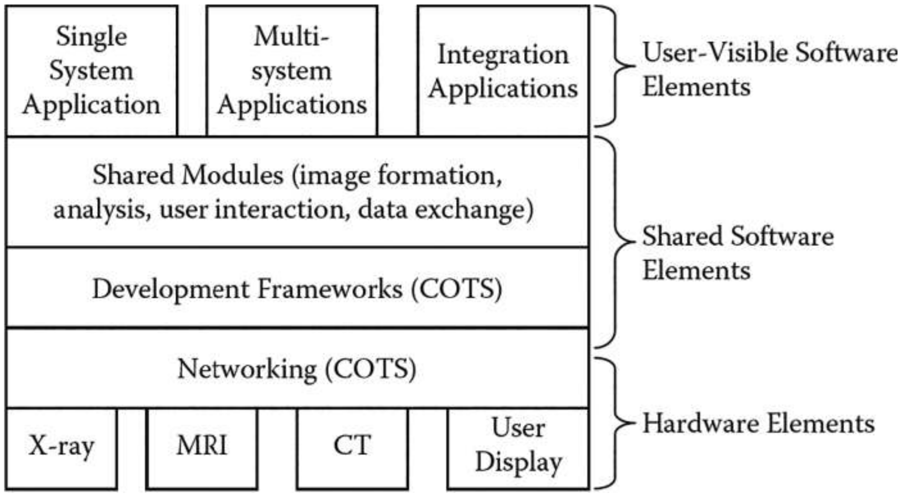

Single System Application
Multi-system Applications
Integration Applications
User-Visible Software Elements
Shared Modules (image formation, analysis, user interaction, data exchange)
Development Frameworks (COTS)
Shared Software Elements
Networking (COTS)
X-ray
MRI
CT
User Display
Hardware Elements

FIGURE CS5.2 Transformed structure of MedInfo product line. The product line allows the "mixing and matching" of hardware elements and assembles end-to-end applications from a large base of shared modules and commercial products.

In a classic hierarchy, a lower-level element is a "part of" a higher-level element. This is the relationship among the elements of a chair. The legs, seat, and back are all parts of the chair. A set of chair legs belongs to exactly one instance of a chair (although the design and manufacturing of those legs may be shared across many identical chairs). In a layered system, a lower-layer element provides services to a higher-layer element. The lower element does not belong to the upper, the lower elements is used by the upper element. An individual lower element may be used by multiple higher elements simultaneously, unlike the classic whole-part relationship, where being a part of one whole precludes being a part of a different whole. A layered architecture has a component relationship of the "uses" form instead of the "part-of" form.

The layered model is borrowed originally from communication networks, where it originated as the well-known seven-layer model. The seven-layer model of the ISO Open System Interconnect (OSI) standard is now of largely historic interest, having been replaced by the 5+ layered model of the Internet. In both, the lower four layers (physical, data link, network, and transport) and the top layer (application) are largely the same. What is different is what is in between. The original OSI model defined two specific in-between layers, the session and the presentation. In practice, these are not used, at least not used in the functional forms originally envisioned by the OSI model. Many Internet applications are simply written directly onto the transport layer, with no intermediate layers. In modern development libraries, and in this case study, the middle area is occupied by various forms of middleware (e.g., message servers, .NET, Common Object Request Broker Architecture [CORBA®]). The contents of these middle layers are different from what was envisioned in the OSI model and are not standardized. Certain capabilities appear repeatedly but are not standards.

Before we take up the complexities of moving to the layered architecture from the stovepiped form, we have to recognize that we have not really defined what it is. Layering must be implemented and there are distinctly different methods of implementation, and each will carry its own set of strengths and weaknesses. Moreover, if we want to assess the extent to which an architecture shift will provide benefits on the four strategic objectives, we have to know more information. To maximize the contrast among alternatives, consider three relatively pure approaches to achieving the layered picture of CS4.2, with the ability to deliver stand-alone products as outlined above.

1. **Logical Layering:** In this approach, the layers are defined by protocols, but each development team independently implements those protocols. Each implementation should be of the same protocol set but will be its own implementation. Functionally, each implementation is the same, but they do not share code.
2. **Development Sharing (Code Sharing):** This approach means that development teams draw from a shared code base. Common functions are implemented once in the code base and then copied over and re-used by each development team. There are at least two sub-forms here: object code libraries and source code sharing.
   a. In the first form, the development teams draw from a shared library of compiled software. This is the typical programming library, just at a larger scale. The shared libraries may be from commercial suppliers or developed and shared in-house. Since they are object code libraries, the developers who use them do not change them.
   b. In the second form, the developers draw from a source code library, the pieces of which they want are then compiled into their application. This retains the "stovepiped" run-time structure of the non-layered-architecture, but with potentially extensive code sharing. Because the sharing is of source code, the developers can modify it if needed. This brings up the obvious issue of whether and how modifications are fed back into the shared library, which is an important implementation topic if this approach is chosen.
3. **Run-Time Sharing:** Here the commonality is expressed by having the shared elements implemented as running programs that are accessed by other components. The equivalent of a stovepipe application is now a much smaller application that pulls together all of the required functionality by accessing concurrently running service providers. Those service providers can field service requests from other applications simultaneously.

As a practical matter, to do well against objective #1, software development cost reduction, MedInfo needs to drive down the volume of code developed. Productivity improvements are also relevant, but the biggest impact is from developing less code. Sharing code is an obvious path to achieving that, assuming that real sharing can be achieved. Code sharing has been very effective but has also been an area where the promise outstrips the reality. For interconnection and integration, code sharing guarantees consistency of interfaces but does not guarantee actual interconnection. A run-time approach may provide better guarantees of interconnection, but even then details really matter.

Various real companies have made MedInfo's hypothesized transition from a hierarchical to a layered architecture. One can easily find reports on how the transition went that emphasize the following.

1. The success of the transition was critical to realizing the business's strategic objectives (those discussed above).
2. The transition was intensely traumatic for staff and management, leading to extensive attrition and financial difficulties while trying to carry it off.

THE PAIN OF THE TRANSITION

The first source of pain is how end-to-end management responsibility changes. In a stovepiped world, the product manager has everything necessary within his or her scope of responsibility, and the scope of responsibility is clear. When there are problems, there is no doubt about where to go to demand a fix, and one point of decision on how to make the fix. Once the fix is made, the scope of its impact is on the product for which the manager is responsible. Of course, the product manager may or may not have enough resources of the right capability to be highly competitive, but there is no doubt where decision-making responsibility sits. In the layered construct, the situation becomes more complex.

In the layered construct, the end-user product is now assembled out of components shared across the product family, whether the sharing is logical, in code, or at run-time. It may be delivered, in part, on platforms out of the product manager's responsibility. For example, in the layered construct, it may be that an x-ray imager is delivered by providing the imaging hardware and software, but the software and user interface reside entirely on computers shared with other imaging systems, which may not be made by MedInfo (in a more highly integrated case). A large part of the imaging and user interface software is shared with other products in the MedInfo family. Being able to do this is a major part of the stated MedInfo business strategy.

When things go wrong, either during development or in deployment, who is responsible for the fix? The product manager no longer has vertical control over the elements that combine to produce a valuable product. If a change is required in the shared code base, that change could conceivably impact all of the other products that use the shared elements. Various companies and government development groups have reported that handling this diffusion of end-to-end responsibility was the most difficult aspect of the change to a layered architecture. It is not practical to make the chief executive officer (CEO) the first convergence point for technical issues across the different products. There must be a point to resolve the issues at a lower level, but conceivably there is no point of common financial responsibility at that lower level.

The obvious answer is team formation and team responsibility. It is not hard to institute cross-product or cross-functional engineering teams, but it is likewise not hard to make such teams toothless when all financial accountability resides elsewhere. It is one thing to build an organization on the basis of cross-functional teaming, and quite another to break up and re-arrange one whose teams are purely product-oriented, and have the new structure work effectively and quickly. Just because an answer is obvious does not make it easy.

Related to management responsibility is how MedInfo must do quality management. In the stovepiped construct, quality can be managed product by product. The quality requirements can come directly from the expectations of the market for each product. But how do we do quality management for shared infrastructure components? Granted, coming up with relevant quality measures is no problem for the experienced, but where do the thresholds lie, given that quite different thresholds might apply to different products within the family? If trade-offs for shared components resolve quite differently in different product applications, which trade-off should be selected? And how do we enforce standards when those standards do not directly relate to delivered customer quality but do have immediate financial consequences (probably bad consequences for somebody)?

Again, various companies have similarly reported on the difficulty of these issues and on how they were successfully dealt with. One heuristic that stands out is:

> Subsystem quality requirements that are sufficient when the component is an element of a stovepiped system are unlikely to be sufficient when the component is shared across many systems.
>
> Or, the quality requirements on the components of a shared layer are likely to be much more demanding than when those components are not shared.

This is a familiar concept from hardware-centric systems. If a part used in one model of a car has a systematic defect, the impact of the resulting recall will be modest. If the part is shared across multiple product lines, the resulting recall may bankrupt the manufacturer. More difficult quality requirements may require new quality assessment tools. Some highlights have included the following.

- The transition to a layered, family-of-systems architecture drove the development and adoption of a massively parallel and automated software regression testing system. All unit-level regression tests needed to be run automatically over a weekend (and they were run every weekend).
- All heavily shared libraries (in a case where they used the shared code in a shared development environment approach) were required to be written with what are known as assertion statements (Rosenblum 1995) on all function entrances and exits. All designs with assertions had to be formally reviewed before production. All software had to be tested with the assertions compiled. Any calling function that causes an assertion to fail is assumed to be at fault and must correct itself.

A related problem in end-to-end management is how subcontracting or outsourcing is organized. In the hierarchical construct, subcontracting tends to follow physical lines. A subcontractor delivers a box or a board, and the specification is written at the box or board level. In a layered system, one can likewise imagine doing the subcontracting of a whole layer, or components within a layer. But new difficulties are introduced, such as the following.

- The specification for a layer typically looks nothing like the specification for a box. Is the expertise available in-house to write specifications and manage subcontracts when the interfaces change in dramatic ways?
  - This came up for one of the authors. After an extended discussion and study on a layered de-stovepiping of a system, the system manager's response was "I get the point, and it would be great, but I don't have anybody who understands how to write requirements and specifications for this. Where would I get them, given they'd also need to know the application domain really well?"
- Testing and integration of layered elements presupposes access to the shared programming libraries. How will shared programming environments be managed with a subcontractor? If the prime contractor has selected some overall software framework to facilitate integration, will all subcontractors buy licenses to the same framework? Is that financially feasible? How will the configurations of the separately purchased frameworks be managed to ensure compatibility?
- What happens when a subcontractor supplying a component that cuts across the whole family-of-systems goes out of business, or decides to drop support, or simply releases a poor-quality version?

All of these, leaving aside the detailed technical issues, fall generally under the heading of management culture and skills. It is not as if there are no solutions to these issues, many companies and government departments have encountered them and solved them. The impact on management culture and practices is most likely when companies frequently report high attrition as a cost of transition. In one case known to the author, the chief operating officer (COO) of a major company responded when asked how his company had successfully managed a stovepipe to layered transition that others had failed at, "We became successful when management attrition reached 50%." Unfortunately, this does not appear to be uncommon.

RESULTS

Solving the problems imposed by changing architectures is possible, but typically quite painful. Is the solution and the pain of arriving at the solution worth the strategic gains? In our composite example, MedInfo answers "yes," but with a certain degree of qualification. The transition from a stovepiped to a layered system addresses the business objectives, at least it can when the devilish details are worked out.

- An effective layered architecture can drop the total line of code count across a family-of-systems. If the total size is dropped, cost and development time advantages can be expected to follow. However, even where there is a software size savings, the savings can be lost if the newer development environment has a higher overhead, is much more expensive, or if access constraints make development more difficult.
- A layered architecture can allow much more complete integration among elements of a product line when all elements of the line have made the

transition. The endpoint might be very integrated, but it might be a very long march to get to the point at which significant benefits are realized. Management needs to know where the cut-over point is to make a rational decision.

- If the layered architecture effectively isolates areas of change from each other, it can allow for much faster product evolution. The key is a good choice of invariants. The invariants must flow from a wise identification of things that change, and where an invariant structure can isolate change. The Transmission Control Protocol/Internet Protocol (TCP/IP) that form the basis of the Internet are an outstanding example.
- The transition is almost invariably very painful. The pain is related much more to the difficulties of the human enterprise than to inherent difficulties in the technologies involved. The new architecture is not more complex than the old one; it is simply different, and many success factors relevant to the old one must be replaced before the new one can be equally as successful.

FURTHER POINTS

The 4+1 model for software architecture description (Kruchten 1995), covered in more detail in Chapter 11, standardizes the expression of the architecture of a software system to the Logical, Development, Run-Time, Execution, and Use-case views. The first three correspond to the three sharing strategies outlined for MedInfo. There is a strong parallelism in the strategies for establishing a family-of-systems approach on the architecture description views when using 4+1. There is also correspondence between the quality factors typically covered by each view and where the sharing strategy is established. The issues with how management maps to the component systems is something that need to be added to 4+1, depending on the development context.

Essentially the same scenario as for MedInfo also applies to many government systems. De-stovepiping has often been a government priority for the same reasons that the hypothetical MedInfo company saw it as a priority. Often interconnection and integration opportunities are leading needs or increasing the rate of turnover or cost reduction. The issues arise both in administrative systems (like payroll, maintenance, and personnel management) as well as in operational systems like the software that supports complex kill chains.

REFERENCES

Kruchten, P. B. (1995). "The 4+ 1 view model of architecture." *IEEE Software* **12**(6): 42–50.
Maier, M. W. (2019). "Architecting portfolios of systems." *Systems Engineering* **22**(4): 335–347.
Rosenblum, D. S. (1995). "A practical approach to programming with assertions." *IEEE Transactions on Software Engineering* **21**(1): 19–31.

6 Software and Information Technology Architecture

Today I am more convinced than ever. Conceptual integrity is central to product quality. Having a system architect is the most important step toward conceptual integrity.

Frederick P. Brooks, Jr.
The Mythical Man-Month after Twenty Years

INTRODUCTION: THE STATUS OF SOFTWARE ARCHITECTING

Software is rapidly becoming the centerpiece of complex system design, in the sense that an increasing fraction of system performance and complexity is captured in software, and that software considerations drive overall system development. Software is increasingly the portion of the system that enables the unique behavioral characteristics of the system. Competitive developers of end-user system products find themselves increasingly primarily focused on developing software, even though the system combines both hardware and software. The reasons stem from software's ability to create intelligent behavior and the technical-economic trends in commoditized hardware. This capability of software is matched against increasing maturity in many other fields containing complex systems. For example, the physical architectures of aircraft have been slowly varying since 1970, and the physical architectures of spacecraft have been slowly varying since at least 1990. In contrast, software architectural innovation is continuing and rapid.

Although detailed quantitative data are hard to come by, anecdotal stories tell a consistent story. A wide variety of companies in different industries (e.g., telecommunications, consumer electronics, industrial controls) have reported a dramatic shift in the relative engineering efforts devoted to hardware and software.[1] Where, up until the 1980s, the ratio was typically 70% hardware and 30% software, it is now typically reversed, with 30% hardware and 70% software (or even more heavily biased to software), and continuing to move to software dominance. This should not be surprising, given how the semiconductor industry has changed. Where product developers used to build from relatively simple parts (groups of logic gates), they now use highly integrated microprocessors with most peripheral devices on the chip. The economies of scale in semiconductor design and production have pushed the industry toward integrated solutions, where the product developer primarily differentiates through software. Moreover, microcontrollers have become so inexpensive and have such low power consumption that they can be placed in nearly any product, even throwaway products. The microprocessor-based products acquire their functionality by the software that executes on them. The product developer is transformed from

DOI: 10.1201/9781003483595-13

a hardware designer to a hardware integrator and software developer. As software development libraries become larger, more capable, and accepted, many of the software developers will be converted to software integrators.

Continuing the trend, many companies do not worry about hardware at all. They develop software and deploy it on cloud computing environments procured and managed by others. This is true even for a major segment of military applications. The commoditization of hardware continues and the value-added is concentrated in software, at least as for the actual implementation of the value-added element.

The largest market for software today is usually termed "information technology," which is a term encompassing the larger domain of computers and communications applied to business and public enterprises. We consider both here, as software architecture as a field is becoming a distinct specialty. What has usually been called software architecture, at least in the research community, is usually focused on developing original software rather than building information-processing systems through the integration of large software and hardware components. But this too is changing as cloud computing and related environments supply large componentized development and deployment environments. Information technology practice is less concerned with developing complete original applications and more and more concerned with building systems through integration. What is usually called enterprise architecture, to the extent that it deals with the architecture of software, is normally dealing with integrating large, preexisting software applications.

The focus of this chapter is less on the architecting of software (though that is discussed here and in Part III) than it is on the impact of software on system architecting. The software possesses two key attributes that affect architecting. First, well-architected software can be very rapidly evolved. The evolution of deployed software is much more rapid than the evolution of deployed hardware, because an installed base of software can be regularly replaced at a moderate cost. The cost of "manufacturing" software is essentially zero (although the cost of certifying it for use may be high), and so unlike in hardware systems, regular total replacement is efficient. As a result, annual replacement is becoming slow, quarterly replacement is common, and more frequent limited updates are normal. Annual or more frequent field software upgrades are normal for operating systems, databases, end-user business systems, large-scale engineering tools, and communication and manufacturing systems. This puts a demanding premium on software architectures, because they must be explicitly designed to accommodate future changes and to allow repeated certification with those changes.

Second, software is an exceptionally flexible medium. Software can easily be built that embodies logically complex concepts such as layered languages, rule-driven execution, data relationships, and many others, all the way up to modern machine learning systems. Software can accommodate learning relationships, either in development or dynamically while operating. This flexibility of expression makes the software an ideal medium with which to implement system "intelligence." In both the national security and commercial worlds, intelligent systems are far more valuable to the user and far more profitable for the supplier than their simpler predecessors.

A combination of technical and economic trends favors building systems from standardized computer hardware and system-unique software, especially when computing must be an important element of the system. Building digital hardware at very high integration levels yields enormous benefits in cost per gate but requires comparably large capital investments in design and fabrication systems. These costs are fixed, giving a strong competitive advantage to high production volumes. Achieving high production volumes requires that the parts be general purpose and so usable across a wide range of applications. For a system to reap the benefits of very high integration levels, its developers must either use the standard parts (available to all other developers as well) or be able to justify the very large, fixed expense of custom development. If standard hardware parts are selected, the remaining means to provide unique system functionality is software.

Logically, the same situation applies to writing software. Software production costs are completely dominated by design and testing. Actual production is nearly irrelevant. So, there is likewise an incentive to make use of large programming libraries or components and amortize the development costs over many products. This has been a trend for decades and continues. Even though much of the software engineering community is frustrated with the pace of software reuse, there are many successful examples. The first obvious one is operating systems. Very few groups who use operating systems write one from scratch anymore. Either they use an off-the-shelf product from one of the remaining vendors, or they use an open-source distribution and customize it for their application. Databases, scripting languages, and Web applications are all examples of successful reuse of large software infrastructures.

The rapid proliferation of open-source software is likewise a successful example of wide-scale software reuse. When the source code is completely open and available for modification and redistribution, many groups have built vigorous communities of developers and users. The availability of the source code, along with the licensing terms for redistribution, appear to be key to making this form of reuse work, as is the quality of the design.

Large development libraries, cloud computing environments, and now even machine learning are examples of forms of software reuse. The case of cloud computing and development libraries is obvious. Machine learning effectively fits in as well, since large language models require large, centralized training efforts and can then be used as the base for more targeted learning applications.

A consequence of software's growing complexity and central role is the recognition of the importance of software architecture and its role in system design. An appreciation of sound architecture and skilled architects is broadly accepted. The soundness of the software architecture will strongly influence the quality of the delivered system and the ability of the developers to further evolve the system. When a system is expected to undergo extensive evolution after deployment, it is usually more important that the system be easily evolvable than that it be exactly correct at first deployment.

Software architecture is frequently discussed, from both academic and industrial perspectives (Garlan and Perry 1995). Within the software architecture community, there is limited consensus on the borders of what constitutes "architecture." Many groups focus on architecture as a high-level physical structure, primarily of source code. A distillation of commonly used ideas is that the architecture is the overall structure of a software system in terms of components and interfaces. This definition would include the major software components, their interfaces with each other and the outside world, and the logic of their execution (single-threaded, interrupted, multithreaded, combination). This is often added to principles that define the system's design and evolution, an interesting combination of heuristics with the structure to define architecture. A software architectural "style" is seen as a generic framework of components or interfaces that defines a class of software structures. The view taken in this book, and in some of the most used literature (Kruchten 1995), is more expansive than just high-level physical structure and includes other high-level views of the system: behavior, constraints, and execution allocation as well.

Systems-of-systems architecture incorporates key ideas from software architecture. Object-orientation, spiral process models, and rapid prototyping are all familiar from software architecture. The carryover should not be a surprise; the systems for which architecting is particularly important are behaviorally complex, data-intensive, and software rich. This is widely occurring in military systems, such as ballistic missile defense, autonomous systems, and kill chains. Examples of software-centered systems of similar scope exist in the civilian world, such as social media, the Internet, global cellular telephony, health care, manned space flight, and flexible manufacturing operations.

The consequences to software design of this accelerating trend to smarter systems are now becoming apparent. For the same reason that guidance and control specialists became the core of systems leadership in the past, software specialists will become the core in the future. In the systems world, software will change from having a support role (usually after the hardware design is fixed) to becoming the centerpiece of complex systems design and operation. As more of the behavioral complexity of systems is embodied in software, it will become the driver of system configuration. Hardware will be selected for its ability to support software instead of the reverse. This is now common in business information systems and other applications where compatibility with a software legacy is important.

If software is becoming the centerpiece of system development, it is particularly important to reconcile the demands of system and software development. Even if 90% of the system-specific engineering effort is put into software, the integrated system is still the end product. It is the system, not the software inside, that the client wishes to acquire. The two worlds share many common roots, but their differing demands have led them in distinctly different directions. Part of the role of systems architecting is to bring them together in an integrated way.

SOFTWARE AS A SYSTEM COMPONENT

How does the architecture and architecting of software interact with that of the system as a whole? Software has unique properties that influence the overall system structure.

1. Software provides a palette of abstractions for creating system behavior. Software is extensible through layered programming to provide abstracted user interfaces and development environments. Through the layering of software, it is possible to directly implement concepts such as relational

data, natural language interaction, logic programming, and learning that are far removed from their computational implementation. Software does not have a natural hierarchical structure, at least not one that mirrors the system-subsystem-component hierarchy of hardware.
2. It is economically and technically feasible to use evolutionary delivery for software. If architected to allow it, the software component of a deployed system can be completely replaced on a regular schedule.
3. Software cannot operate independently. Software must always be resident on some hardware system and, hence, must be integrated with some hardware system. That it has to run on something does not mean there has to be a fixed mapping between software elements and computers. The execution allocation may be flexible and very dynamic, assuming the right architecture is in place. The interaction between, and integration with, this underlying hardware system becomes a key element in software-centered system design.

For the moment there are no emerging technologies that are likely to take software's place in implementing behaviorally complex systems. Perhaps some form of biological or nano-agent technology will eventually acquire similar capabilities. In these technologies, behavior is expressed through the emergent properties of interacting organisms. However, the design of such a system can be viewed as a form of logic programming in which the "program" is the set of component construction and interface rules. Then the system, the behavior that emerges from component interaction, is the expression of an implicit program, a highly abstracted form of software.

System architecting adapts to software issues through its models and processes. To take advantage of the rich functionality, there must be models that capture the layered and abstracted nature of complex software. If evolutionary delivery is to be successful, and even just to facilitate successful hardware/software integration, the architecture must reconcile the continuously changing software with the much less frequently changing hardware.

SOFTWARE FOR MODERN SYSTEMS

Software plays disparate roles in modern systems. Mass market application software, one-of-a-kind business systems, real-time analysis and control software, and human interactive assistants are all software-centered systems, but each is distinct from the other. The software attributes of rich functionality and amenability to evolution match the characteristics of modern systems. These characteristics include the following.

1. Storage of large volumes of information, and semiautonomous and intelligent interpretation of that information,.
2. Provision of responsive human interfaces that mask the underlying machines and present their operation in metaphor.
3. Semiautonomous adaptation to the behavior of the environment and individual users.

4. Real-time control of hardware at rates beyond human capability with complex functionality.
5. Constructed from mass-produced computing components and unique system software, with the capability to be customized to individual customers.
6. Coevolution of systems with customers, as experience with system technology changes perceptions of what is possible.

The marriage of high-level and domain-specific languages with general-purpose computers allows behaviorally complex, evolutionary systems to be developed at a reasonable cost. Although the engineering costs of a large software system are considerable, they are much less than the costs of developing a pure hardware system of comparable behavioral complexity. Such a pure hardware system could not be evolved without incurring large manufacturing costs on each evolutionary cycle. Hardware-centered systems do evolve, but at a slower pace. They tend to be produced in similar groups for several years, and then make a major jump to new architectures and capabilities. The time of the jump is associated with the availability of new capabilities and the programmatic capability of replacing an existing infrastructure.

The layering of software as a mechanism for developing greater behavioral complexity is exemplified in the continuous emergence of new software languages and in Internet and Web applications being built on top of distributed infrastructures. The trend in programming languages is to move closer and closer to application domains. The progression of language is from machine-level (machine and assembly languages) to general-purpose computing (FORTRAN, Pascal, C, C++, Ada) to domain-specific (MATLAB®, SQL, PERL, and other scripting languages). The movement to machine learning frameworks, increasingly built on large language models, can be seen to be a continuation of the same pattern. At each level, the models are closer to the application, and the language components provide more specific abstractions. By using higher and higher-level languages, developers are effectively reusing the coding efforts that went into the language's development. Moreover, the new languages provide new computational abstractions or models not immediately apparent in the architecture of the hardware on which the software executes. Consider a logic programming language like PROLOG. A program in PROLOG is more in the nature of hypothesis and theorem-proof than arithmetic and logical calculation. But it executes on a general-purpose computer as invisibly as does a C or even FORTRAN program. Prompt programming for large language models (like GPT) is an even further expression of the idea, now moved into natural language rather than formal language.

SYSTEMS, SOFTWARE, AND PROCESS MODELS

An architectural challenge is to reconcile the integration needs of software and hardware to produce an integrated system. This is both a problem of representation or modeling and of process. Modeling aspects are taken up subsequently in this chapter, and in Part III. On the process side, hardware is best developed with as little iteration in production as possible, but software can (and often should) evolve through much iterations. Hardware should follow a well-planned design and production cycle to minimize cost, with large-scale production deferred to as close to final delivery as possible (consistent with adequate time for quality assurance). However, software cannot be reliably developed without access to the targeted hardware platform for much of its development cycle. Production takes place nearly continuously, with release cycles now often happening daily in many advanced development organizations.

Software distribution costs are comparatively so low that repeated complete replacement of the installed base is considered normal practice. When software firms ship their yearly (or more frequent) upgrades, they ship a complete product. Firms commonly "ship" patches and limited updates on the Internet, eliminating even the cost of media distribution. The cycle of planned replacement is so ingrained that some products (e.g., software development tools) are distributed as an online subscription.

In contrast, the costs of hardware are often dominated by the physical production of the hardware. If the system is mass-produced, this will clearly be the case. Even when production volumes are very low, as in unique customized systems, the production cost is often comparable to or higher than the development cost. As a result, it is uneconomic, and hence impractical, to extensively replace a deployed hardware system with a relatively minor modification. Any minor replacement must compete against a full replacement, a replacement with an entirely new system designed to fulfill new or modified purposes.

One important exception to the rule of low deployment costs for software is where the certification costs of new releases are high. For example, one does not casually replace the flight control software of a 777 aircraft any more than one casually replaces an engine. Extensive test and certification procedures are required before a new software release can be used. Certification costs are analogous to manufacturing costs in that they are a cost required to distribute each release but do not contribute to product development.

Waterfalls for Software?

For hardware systems, the process model of choice is a waterfall (in one of its pure or more refined incarnations). The waterfall model defines development stages and tries to keep iterations local—that is, between adjacent tasks such as requirements and design. Upon reaching production, there is no assumption of iteration, except the large-scale iteration of system assessment and eventual retirement or replacement. This model fits well within the traditional architecting paradigm as described in Chapter 1.

Software can, and sometimes does, use a waterfall model of development. The literature on software development originally embraced the sequential paradigm of requirements, design, coding, testing, and delivery. But dissatisfaction with the waterfall model for software led to the spiral model and variants including agile models today. Essentially, all successful software systems are iteratively delivered. Application software iterations are expected as a matter of course. Weapon systems and manufacturing software are also regularly updated with refined functionality, new capabilities, and fixes to problems. One reason for software iterations is to fix problems discovered in the field. A waterfall model tries to eliminate such problems by doing a very high-quality job of requirements development. Indeed, the success of a waterfall development is strongly dependent on the quality of the requirements. But in some systems, the evolvability of software can be exploited to reach the market faster and avoid costly, and possibly fruitless, requirements searches. The logic of focusing on requirements development to prevent defects that have to be discovered in the field and then fixed is based on the typical accelerating cost of defect repair the farther one is into the development process. But this neglects the cost of discovering the defects. You cannot fix what you cannot find.

Example: Data communication systems have an effective requirement of interoperating with whatever happens to be present in the installed base. Deployed systems from a global range of companies may not fully comply with published standards, even if the standards are complete and precise (which they often are not). Hence, determining the "real" requirements to interoperate is quite difficult. The most economical way to do so may be to deploy to the field and compile real experience, the alternative being test labs of unlimited size holding everything one might ever encounter in the field. But focusing on field discovery, in turn, requires that the systems support the ability to diagnose the cause of interoperation problems and be economically modifiable once deployed to exploit the knowledge gained.

In contrast, a casual attitude toward evolution in systems with safety or mission-critical requirements can be tragic.

Example: The Therac 25 was a software-controlled radiation treatment machine in which software and system failures resulted in six deaths (Leveson and Turner 1993, Leveson 2017). It was an evolutionary development from a predecessor machine. The evidence suggests that the safety requirements were well understood but that the system and software architectures both failed to maintain the properties. The system architecture was flawed in that all hardware safety interlocks (which had been present in the predecessor model) were removed, leaving software checks as the sole safety safeguard. The software architecture was flawed because it did not guarantee the integrity of treatment commands entered and checked by the system operator.

One of the most extensively described software development problems is customized business systems. These are corporate systems for accounting, management, and enterprise-specific operations. They are of considerable economic importance, built in fairly large numbers (though no two are exactly alike), and are developed in an environment relatively free of government restrictions. Popular and widely published development methods have strongly emphasized detailed requirements development followed by semiautomated conversion of the requirements to program code—an application-specific waterfall.

Even though this waterfall is better than ad hoc development, results have been disappointing. In spite of years of experience in developing such business systems, large development projects regularly fail. As Tom DeMarco noted years ago

(DeMarco and Lister 2013), "somewhere, today, an accounts payable system development is failing" in spite of the thousands of such systems that have been developed in the past. Relatively routine software projects still fail at embarrassing rates. Part of the reason is the relatively poor state of software engineering compared to other fields. Another part is the moving goalposts, the expected level of complexity keeps rising even as the tools for construction get better. An important reason is the lack of an architectural perspective and the benefits it brings (DeMarco 1995).

The architect's perspective, as discussed in this book, is to explicitly consider implementation, requirements, and long-term client needs in parallel. A requirements-centered approach assumes that a complete capture of documentable requirements can be transformed into a satisfactory design. However, existing requirements practices often fail to capture performance requirements and ill-structured requirements like modifiability, flexibility, and availability. Even where these nonbehavioral requirements are captured, they cannot be transformed into an implementation in any semiautomated way. And it is the nature of serious technological change that the impact will be unpredictable. As technology changes and experience is gained, what is demanded from systems changes as well.

The original spiral model (the risk spiral) did not embrace evolution because its focus was risk resolution. Its spirals were strictly risk-based and designed to lead to a fixed system delivery. The risks were assumed to be primarily internal, in the nature of technology or implementation. Rapid prototyping envisions evolution, but only on a limited scale. The kind of spiraling used in functional incrementalism and agile fully embraces evolution and discovery of requirements by putting things in front of users. Software processes, as implemented, spiral through the waterfall phases but do so in a sequential approach to moving release levels. This modified model was introduced in Chapter 4, in the context of integrating software with manufacturing systems, and it will be further explored below.

One point of reconciliation is that we could see the primary risks in a system as an expression of user needs. Specifically, no matter how we ask the users they won't realize (or discover) their needs until provided with a system to use. Equivalently, we expect user operations to change given the system of interest, and the change process cannot start until they are provided with some functionally relevant system. In this case, the functional increments could be seen as spirals in a risk-spiral model, but now focused on gathering information from users.

Spirals for Hardware?

Using a spiral model for hardware acquisition is equivalent to repeated prototyping. A one-of-a-kind, hardware-intensive system cannot be prototyped in the usual sense. A complete "prototype" is, in fact, a complete system. If it performs inadequately, it is a waste of the complete manufacturing cost of the final system. Each one, from the first article, needs to be produced as though it were the only one. As was discussed previously, under the "protoflight" development strategy, the prototype is the final system. A true prototype for such a one-of-a-kind system must be a limited version or component intended to answer specific developmental questions. We would not "prototype" an aircraft carrier, but we might well prototype individual pieces and build subscale models for testing. The development process for one-of-a-kind systems needs to place a strong emphasis on requirements development and attention to detailed purpose throughout the design cycle. Mass-produced systems have greater latitude in prototyping because of the prototype-to-production-cost ratio, but still have less compared to software. However, the initial "prototype" units still need to be produced. If they are to be close to the final articles, they need to be produced on a similar manufacturing line. But setting up a complete manufacturing line when the system is only in the prototype stage is very expensive. Setting up the manufacturing facilities may be more expensive than developing the system. As a hardware-intensive system, the manufacturing line cannot be easily modified, and leaving it idle while modifying the product to be produced represents a large cost.

There are hardware-heavy systems that are pursuing what appear to be more incremental development cycles, but closer examination generally confirms the points made above. SpaceX famously launches, in development not just in operations, at a very fast pace compared to other developers. There are significant changes to hardware between flights, reflecting what has been learned in previous flights. The first flights of the full Starship vehicle with booster came after flights of Starship alone ("hopping" attempt flights) and flights by subscale vehicles. There are a few things to consider about this strategy. First, it is dependent on being able to manufacture the test vehicles quickly (quickly by the standards of space launch vehicles) and at a cost where using up a vehicle in a test flight is affordable. The economics has to be such that the cost of the information gained from a test flight, where the vehicle is lost, is less than it would have cost to gain the information by other means (e.g., analysis, ground test). A significant amount of the iteration between flights will be in software. If the hardware changed dramatically then it would disrupt the manufacturing preparations (like tooling and material supply) that are necessary to do continuous and rapid production. While the hardware may evolve from launch to launch, it does so within stable forms. This notion of architecture as invariants has been a theme throughout the book.

Integration: Spirals and Circles

What process model matches the nature of evolutionary, mixed technology, behaviorally complex systems? As was suggested earlier, a spiral and circle framework seems to capture the issues. The system should possess stable configurations (represented as circles) and development should iteratively approach those circles. The stable configurations can be software release levels, architectural frames, or hardware configurations. Again, this is the idea of architecture as invariants. The presence of invariants around which other elements change and evolve is a key to being able to evolve.

This process model matches the accepted software practice of moving through defined release levels, with each release produced in cycles of requirements-design-code-test. Each release level is a stable form that is used while the next release is developed. Three types of evolution can be identified. A software product, like an operating system or shrink-wrapped application, has major increments in behavior indicated by changes in the release number, and more minor increments by changes in the number after the "point." Hence, a release 7.2 product would be major version seven, second update. The major releases can be envisioned as circles, with the minor releases cycling into them. On the third level are those changes that result in new systems or re-architected old systems. These are conceptually similar to the major releases but represent even bigger changes. The process with software annotations is illustrated in Figure 6.1. By using a side view, one can envision the major

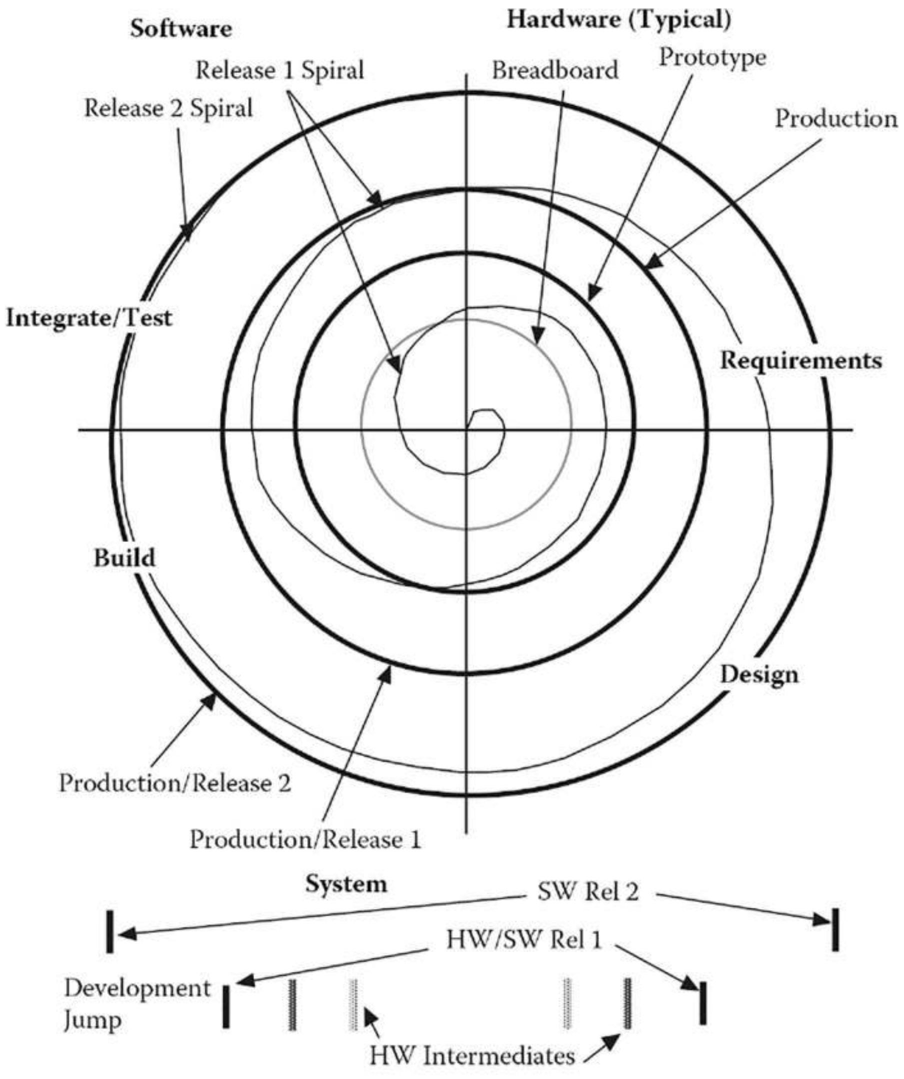


FIGURE 6.1 A typical arrangement of spirals and circles in a project requiring hardware and software integration. This illustrates the stable intermediate configurations of hardware (typically breadboard and prototype) integrating with a software spiral. Software is developed on stable intermediate systems.

releases as vertical jumps. The evolutionary spiral process moves out to the stable major configurations and then jumps up to the next major change.

In practice, evolution on one release level may proceed concurrently with the development of a major change.

Example: The Internet and World Wide Web provide numerous examples of stable intermediate forms promoting evolution. The architecture of the Internet, in the sense of an organizing or unifying structure, is clearly the Internet Protocol (IP), the basic packet switching definition. IP defines how packets are structured and addressed, and how the routing network interacts with the packets. It determines the kinds of services that can be offered on the Internet, and in so doing constrains application construction. As the Internet has undergone unprecedented growth in users, applications, and physical infrastructure, IP has remained stable. The dominant version of IP is changing form v4 to v6, but the process is slow. A truly complete transition is still a way off, given how the nature of IPv4 underpins so much of how the Internet operates. The World Wide Web has similarly undergone tremendous growth and evolution on top of a simple set of elements, the Hypertext Transfer Protocol (HTTP) and the Hypertext Markup Language (HTML). Both the Internet and the World Wide Web are classic examples of systems with nonphysical architecture, a topic that becomes central in the discussion of collaborative systems in Chapter 7.

Hardware–software integration adds to the picture. The hardware configurations must also be stable forms but should appear at different points than the software intermediates on the development timeline. Some stable hardware should be available during software development to facilitate that development. A typical development cycle for an integrated hardware–software system illustrates parallel progressions in hardware and software with each reaching different intermediate stable forms. The hardware progression might be breadboard, production prototype, production, and then (possibly) field upgrade. The software moves through a development spiral aiming at a release 1.0 for the production hardware. The number of software iterations may be many more than for the hardware. In the late development stages, new software versions may be built weekly (Maguire 1994). Before that, there will normally be partial releases that run on the intermediate hardware forms (the breadboards and the production prototypes). Hardware–software codesign research is working toward environments in which developing hardware can be represented faithfully enough so that physical prototypes are unnecessary for early integration. Such tools may become available, but iteration through intermediate hardware development levels is still the norm in practice.

A related problem in designing a process for integration is the proper use of the heuristic: Do the hard part first. Because software is evolved or iterated, this heuristic implies that the early iterations should address the most difficult challenges. Unfortunately, honoring the heuristic is often difficult. In practice, the first iterations are often good-looking interface demonstrations or constructs of limited functionality. If interface construction is difficult or user acceptance of the interface is risky or difficult, this may be a good choice. But if operation to time constraints under loaded conditions is the key problem, some other early development strategy should be pursued. In that case, the heuristic suggests gathering realistic experimental data on loading and timing conditions for the key processes of the system. That data can then be used to set realistic requirements for components of the system in its production configuration.

Example: Call distribution systems manage large numbers of phone personnel and incoming lines as in technical support or phone sales operations. By tying the system into sales databases, it is possible to develop sophisticated support systems that ensure that full customer information is available in real time to the phone personnel. To be effective, the integration of data sources and information handling must be customized to each installation and evolve as the understanding of what information is needed and available develops. But, because the system is real time and critical to customer contact, it must provide its principal functionality reliably and immediately upon installation.

Incremental development can lead to dysfunctional relationships to the Do The Hard Part First heuristic. There can be a temptation to push the hard parts to later iteration cycles. At first, this might be reasonable as it is clear that something is sufficiently difficult that it cannot be resolved in a single iteration cycle. But when that slip compounds, and other decisions are made and implemented around not fixing the hard problem, the difficulty of fixing it may compound. If the group starts and abandons different approaches to fixing the hard problems, things only get worse. See Goldstein (2005) for an example of where this took place.

Thus, an architectural response to the problems of hardware–software integration is to architect both the process and the product. The process is manipulated to allow different segments of development to match themselves to the demands of the implementation technology. The product is designed with interfaces that allow the separation of development efforts where the efforts need to proceed on very different paths. How software architecture becomes an element of system architecture, and more details on how this is to be accomplished, are the subjects to come.

THE PROBLEM OF HIERARCHY

A central tenet of classic systems engineering is that all systems can be viewed in hierarchies. A system is composed of subsystems that are composed of small units. A system is also embedded in higher-level systems in which it acts as a component. One person's system is another person's component. A basic strategy is to decompose any system into subsystems, decompose the requirements until they can be allocated to subsystems, carefully specify and control the interfaces among the subsystems, and repeat the process on every subsystem until you reach components you can buy or are the products of disciplinary engineering. Decomposition in design is followed by integration in reverse. First, the lowest-level components are integrated into the next-level subsystems, those subsystems are integrated into larger subsystems, and so on, until the entire system is assembled.

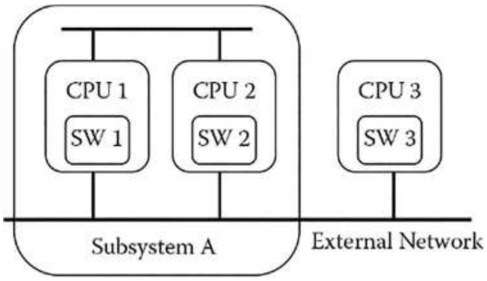


FIGURE 6.2 System/hardware hierarchy view of a system.

Because this logic of decomposition and integration is so central to classical systems engineering, it is difficult for many systems engineers to understand why it often does not match software development very well. To be sure, some software systems are very effectively developed this way. The same logic of decomposition and integration matches applications built in procedural languages where the development effort writes all of the application's code. In these software systems, the code begins with a top-level routine, which calls first-level routines, which call second-level routines, and so forth, to primitive routines that do not call others. In a strictly procedural language, the lower-level routines are contained within or encapsulated in the higher-level routines that use them. If the developer organization writes all the code, or uses only relatively low-level programming libraries, the decomposition chain terminates in components much like the hardware decomposition chain terminates. Like in the classical systems engineering paradigm, we can integrate and test the software system in much the same way, testing and integrating from the bottom up until we reach the topmost module.

As long as the world looks like this, on both the hardware and software sides, we can think of system decompositions as looking like Figure 6.2. This figure illustrates the world, and the position of software, as classical systems engineers would portray it. Software units are contained within the processor units that execute them. Software is properly viewed as a subsystem of the processor unit.

However, if we instead went to the software engineering laboratory of an organization building a modern distributed system and asked the software engineers to describe the system hierarchy, we might get a very different story. Much modern software is written using object-oriented abstractions, built-in layers, and makes extensive use of very large software infrastructure objects (like operating systems, databases, and middleware) that do not look very much like simple components or the calls to a programming library. The transition is illustrated in Figure 6.3 and was discussed in the "Case Study 5" (prior to this chapter). When expanded to the level of interacting bodies of code, the world looks as illustrated in Figure 6.4. Each of these issues (object orientation and layering) creates a software environment that does not look like a hierarchical decomposition of encapsulated parts, and to the extent that a hierarchy exists, it is often quite different from the systems/hardware hierarchy. We consider each of these issues in turn.

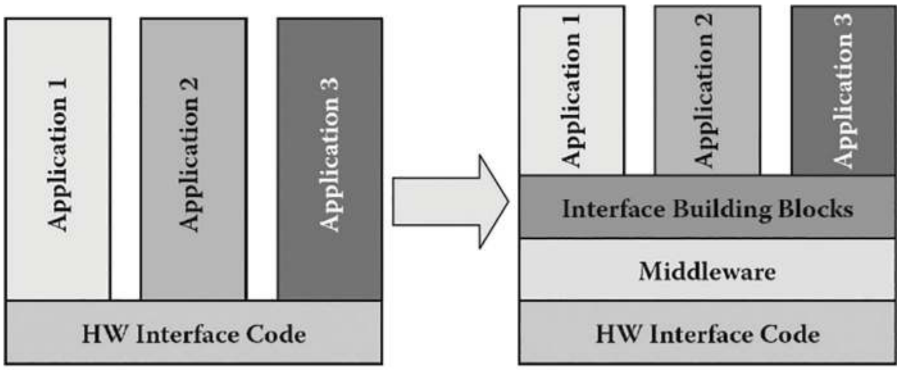


FIGURE 6.3 Thick applications have been replaced by much thinner implementations that rely on thicker shared infrastructure layers for years, and the process continues. This transition is of high value but introduces problems in quality control and development methods often unfamiliar to groups accustomed to building thick applications.

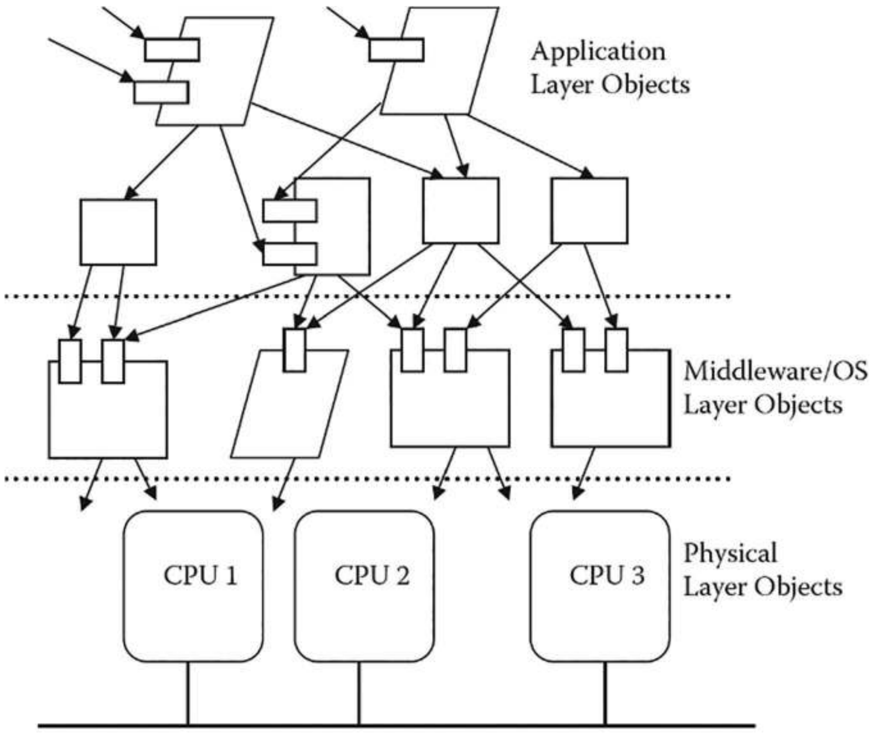


FIGURE 6.4 Layered software hierarchy view of a system.

Object Orientation

The software community engages in many debates about exactly what "object-oriented" should mean, but only the fundamental concepts are important for systems architecting. An object is a collection of functions (often called methods) and data. Some of the functions are public; that is, they are exposed to other software objects and can be called by them. Depending on the specific language and runtime environment, calling a function may be a literal function call, or it may simply mean sending a message to the target object, which interprets it and takes action. Objects can be "active"; that is, they can run concurrently with other objects. In some software environments, concurrent objects can freely migrate from computer to computer over an intervening network. Often the software developer does not know, and does not want to know, on which machine a particular object is running and does not want to directly control its migration. Concurrent execution of the objects is passed to a distributed operating system, which may control object execution through separately defined policies.

In object-oriented systems, the number of objects existing when the software executes can be indeterminate. An object has a defining "template" (although the word "template" means something slightly different in many object-oriented languages) is known as a "class." A class is analogous to a type in procedural programming. So, just as one can declare many variables of type "float," so one can declare many objects corresponding to a given class. In most object-oriented languages, the creation of objects from classes happens at runtime, when the software is executing. If objects are not created until runtime, the number of them can be controlled by external events.

As systems modeling languages, like SysML, have become standard in the community, the object-oriented perspective has become much more common. In SysML, the concept of blocks, commonly used to model hardware whole-part hierarchies follows the established object-oriented logic of packaging data and functions together. Similarly, the SysML versions respect other object-oriented concepts such as inheritance. The concepts frequently scale well to hardware environments, albeit with some adjustments.

This is a very powerful method of composing a software system, and as the growing popularity of SysML demonstrates, also in systems engineering. Each object is really a computational machine. It has its own data (potentially a very large amount) and as much of its own program code as the class author decides. This sort of dynamic object-oriented software can essentially manufacture logical machines, in arbitrary numbers, and set them to work on a network, in response to events that happen during program execution. To compare this to classical notions of decomposition, it is as though one could create subsystems on the fly during system operation.

Layered Design

The objects are typically composed in a layered design as is further illustrated in Figure 6.4. Layers are a form of hierarchy, with a critical difference. In a layered system, the lower-level elements (those making up a lower layer) are not contained in the upper-layer elements. The elements of a layer interact to produce a set of services, which are made available to the next higher layer (in a strictly layered system). Objects in the next higher layer can use the services offered by the next lower layer but cannot otherwise access the lower-layer objects. Within a layer, the objects normally treat each other as peers; that is, no object is contained within another object. However, object orientation has the notion of encapsulation. An object has internals, and the internals (functions and data) belong to that object alone, although they can be duplicated in other objects with the same class.

A modern distributed application may be built as a set of interacting, concurrent objects. The interaction concurrent objects exist at runtime but do not (necessarily) have a one-to-one relationship to elements in the development environment nor a static relationship to hardware execution elements. The objects interact with a lower layer, often called "middleware services." The middleware services are provided by externally supplied software units. Some of the services are part of commercial operating systems; others are individual commercial products. Those middleware components ride on lower layers of network software, supplied as part of the operating system services. In a strong distributed environment, the application programmers, who are writing the objects at the top level, do not know what the network configuration is on which their objects ride. Of course, if there are complex performance requirements, it may be necessary to know and control the network configuration and to program with awareness of its structure. But in many applications, no such knowledge is needed, and the knowledge of the application programmers about what code is actually running ceases when the thread of execution leaves the application and enters the middleware and operating systems.

The hierarchy problem is that at this point the software hierarchy and the hardware hierarchy have become disconnected. To the software architect, the natural structure of the system is layers of concurrent objects, again as illustrated in Figures 6.3 and 6.4. This means the systems and software architects may clash in their partitioning of the system, and inappropriate constraints may be placed on one or the other. Before investigating the issue of reconciliation, we must complete the discussion with the nature of software components.

This distinction between the logical expression (the original concept of the objects), the operation of concurrent elements in execution (the runtime), their potentially dynamic mapping to hardware execution elements, and the non-one-to-one relationship to code in development environments is the foundation of multiview representation. Specifically, it explains the origin of the 4+1 concept (Kruchten 1995) of software architecture representation. The "four" are the logical, runtime, execution, and development. The "+1" is the set of use cases or mission threads that connect or run-through the four.

Large, Autonomous Components

When taking a decompositional approach to design, the designer decomposes until he or she reaches components that can be bought or easily built. In both hardware and software, some of the components are very large. In software, in particular, the design decomposition often results in very large software units, such as operating systems and databases. Both of these are now often millions of lines of programming language code and possess rich functionality. More significantly, they act semiautonomously when used in a system. An operating system is not a collection of functions to be passively called by an application. To be sure, that is one of the services offered by modern operating systems. But modern operating systems manage program memory, schedule program units on processors, and synchronize concurrent objects across multiple processors. An advanced operating system may present unified services that span many individual computers, possibly widely geographically spread.

These large and autonomous components change architecting because the architect is strongly incentivized to adapt to the components rather than trying to adapt the components themselves. In principle, of course, the architect and client need not adapt. They can choose to sponsor a from-scratch development instead. They can try to figure out how to sponsor modifications to a complete operating system. But the cost of attempting to replicate the enormous software infrastructure that applications now commonly reuse is prohibitive. So, for example, the market dominance and complexity of very large databases force us to use commercial products in these applications. The commercial products support particular kinds of data models and do not support others. The architecture must take account of the kinds of data models supported, even when those are not a natural choice for the problem.

Cloud computing environments include many large autonomous components. These are not just operating systems and databases, but much larger collections of components useful as infrastructure in large applications. There is a strong incentive to make use of these components, to architect within their framework, to gain the advantages of using the associated infrastructures. In some ways, this is no different than thinking about working with other large components but emphasizes the increasingly domain-specific nature of architecting practices.

RECONCILING THE HIERARCHIES

Our challenge is to reconcile the systems and software worlds. Because software is becoming the dominant element, in terms of its cost pacing what can be developed, one might argue for simply adopting software's models and abandoning the classic systems view. This is inappropriate for several reasons. First, the migration of software to object-oriented, layered structures is only partial. Legacy software goes away very slowly, sometimes terrifyingly slowly. The infrastructure for supporting distributed, concurrent, object-oriented applications is strong in some environments (cloud computing) and immature in others (real-time systems) and may never fully transition. The divide between application areas that emphasize scalability, speed of development, and geographic reach and those that emphasize reliability, real-time and predictable response, and safety, is likely to continue.

Second, both approaches are fundamentally valid. Figures 6.2 and 6.4 are correct views of the system, they just represent different aspects. No single view can claim primacy. As we move into complex, information-centric systems, we will have to accept the existence of many views, each representing different concerns, and each targeted at a different stakeholder audience. The architect, and eventually systems engineers, will have to be sure that the multiple views are consistent and complete with respect to the stakeholder's concerns.

Third, every partitioning has its advantages and drawbacks. Building a system in which each computational unit has its own software confined within it has distinct advantages. In that case, each unit will normally have much greater autonomy (because it has its own software and does not depend on others). That means each unit can be much more easily outsourced or independently developed. Also, the system does not become dependent on the presence of some piece of software infrastructure. Software infrastructure elements (operating systems and middleware) have a poor record for on-schedule delivery and feature completeness. Anybody depending on an advanced feature of an operating system to be delivered more than a year out runs a high risk of being left with nothing when the scheduled delivery date comes by and the operating system vendor has decided to delay the feature to a future version or has simply pushed the delivery cycle out another year.

Nevertheless, the newer approaches have tremendous advantages in many situations. Consider the situation when the units in Figure 6.2 share a great deal of functionality. If separate development teams are assigned to each, the functionality is likely to be independently developed as many times as there are units. Redundant development is likely to be the least of the problems; however, because those independent units probably interact with each other, the test burden has the potential to rise as the square of the number of units. Appropriate code sharing—that is, the use of layered architectures for software—can alleviate both problems.

THE ROLE OF ARCHITECTURE IN SOFTWARE-CENTERED SYSTEMS

In software, as in systems, the architect's basic role is the reconciliation of a physical form with the client's needs for function, cost, certification, and technical feasibility. The mindset is the same as described for system architecting in general, though the areas of concentration are different. System architecting heuristics are generally good software heuristics, though they may be refined and specialized. Several examples are given in Chapter 9. In addition, there are heuristics that apply particularly to software. Some of these are mentioned at the end of this chapter.

The architect develops the architecture. Following Brooks' term (Brooks Jr 1995), the architect is the user's advocate. As envisioned in this book, the architect's responsibility goes beyond the conceptual integrity of the systems as seen by the user, to the conceptual integrity of the system as seen by the builder and other stakeholders. The architect is responsible for both what-the-system-does and well as how-the-system-does-it. But that responsibility extends, on both counts, only as far as is needed to develop a satisfactory and feasible system concept. After all, the sum of both is nearly the whole system, and the architect's role must be limited if an individual or small team is to carry it out. The latter role, of defining the overall implementation structure of the system, is closer to some of the notions of software architecture in recent literature.

The architect's realm is where views and models combine. Where models that integrate disparate views are lacking, the architect can supply the insight. When disparate requirements must interact if satisfaction is to be achieved, the architect's insight can ensure that the right characteristics are considered foremost and that an architecture that can reconcile disparate requirements is developed. The perspective required is predominantly a system perspective. It is the perspective of looking at the software and its underlying hardware platforms as an integrated whole that delivers value to the client. Its performance as a whole, behavioral and otherwise, is what gives it its value.

Architecting for evolution is also an example of the greatest leverage at the interfaces heuristic. Make a system evolvable by paying attention to the interfaces. In software, interfaces are very diverse. With a hardware emphasis, it is common to think of communication interfaces at the bit, byte, or message level. However, in software communication, interfaces can be much richer and capture extensively structured data, flow of control, and application-specific notions. Interfaces can look very close to how stakeholders see the logical exchange of information meaningful in the operations domain. Current work in distributed computing is a good example. The trend in middleware is to find abstractions well above the network socket level that allow flexible composition. Network-portable languages like Java allow each machine to express a common interface for mobile code (the Java virtual machine). The ambition of service-oriented architectures is to provide a rich set of intermediate abstractions to allow end-user development to be further abstracted away from the low-level programming details.

Programming Languages, Models, and Expression

Models are languages. A programming language is a model of a computing machine. Like all languages, they have the power to influence, guide, and restrict our thoughts. Programmers with experience in multiple languages understand that some problems will decompose easily in one language, but only with difficulty in another, an example of fitting the architecture of the solution to that of a prescriptive solution heuristic. The development of programming languages has been the story of moving successively higher in abstraction from computing hardware.

The layering of languages is essential to complex software development because a high-level language is a form of software reuse. Assembly languages masked machine instructions; procedural languages modeled computer instructions in a more language-like prose. Modern languages containing object and strong structuring concepts continue the pattern by providing a richer palette of representation tools for implementing computing constructs. Each statement in FORTRAN, Pascal, or C reuses the compiler writer's machine-level implementation of that construct. The same logic scales to application-specific languages like mathematical languages or databases. A statement in a mathematical language like MATLAB® or Mathematica may invoke a complex algorithm requiring long-term development and deep expertise. A database query language encapsulates complex data storage and indexing code. The current enthusiasm for service-oriented architectures is (or should be) the same phenomenon. By assembling abstractions closer to what end users are interested in, while maintaining a low enough level of abstraction to be reusable, we greatly enhance development productivity.

One way of understanding this move up the ladder of abstraction is a famous software productivity heuristic on programmer productivity. A purely programming-oriented statement of the heuristic is

> Programmers deliver the same number of lines of code per day regardless of the language they are writing in.

Hence, to achieve high software productivity, programmers must work in languages that require few lines of code. See works like (Jones 1985, 2008) for data that contributes to this heuristic, if not being the singular source of it. This heuristic can be used to examine various issues in language and software reuse. The nature of a programming language, and the available tools and libraries, will determine the amount of code needed for a particular application. Obviously, writing machine code from scratch will require the most code. Moving to higher-level languages like C++ or Ada will reduce the amount of original code needed, unless the application is fundamentally one that interacts with the computing hardware at a very low level. Still, less original code will be required if the language directly embodies application domain concepts, or, equivalently, application-specific code libraries are available.

Application-specific languages imitate domain languages already in use and make it suitable for computing. One of the first and most popular is spreadsheets. The spreadsheet combines a visual abstraction and a computational language suited to a range of modeling tasks in business offices, engineering, and science. An extremely important category is database query languages. Today, it would be quite unusual to undertake an application requiring sophisticated database functionality and not using an existing database product and its associated query language. Another category includes mathematical languages. These languages, such as Mathematica, MacSyma, and MATLAB®, use well-understood mathematical syntax and then process those languages into computer-processable form. They allow the mathematically literate user to describe solutions in a language much closer to the problem than a general-purpose programming language.

Large language models, trained on Internet-scale collections of text, carry this to a further extreme with their ability to respond to natural language prompts. They likewise carry to an extreme the full range of architectural and engineering concerns. One might ask how to do quality assurance on software capable of "hallucinations" (del Campo and Leach 2022, Ahmad et al. 2023, Zhang et al. 2023). Answers may well be found, but probably not within the technique set of traditional quality assurance.

The value of application-specific approaches is seen in projects that build application specifics on top of an established higher-level programming language. For example, in the application area of bioinformatics or biological computation, there are multiple projects that overlay biology-specific elements onto some other popular language. There is a Biopython, BioPerl, BioRuby, and a BioJava (Cock et al. 2009, Ryu 2009).

Application-specific programming languages are likely to play an increasingly important role in all systems built in reasonably large numbers. The only impediment to the use of these abstractions in all systems is the investment required to develop the language and its associated application generator and tools. One-of-a-kind systems will not usually be able to carry the burden of developing a new language along with a new system unless they fit into a class of systems for which a "meta-language" exists. Some work along these lines has been done, for example, in command and control systems (Tracz 1995, Bergner et al. 2005). As mentioned before, service-oriented architectures are a currently fashionable take on the same theme.

Architectures, "Unifying" Models, and Visions

Architectures in software can be definitions in terms of tasks and modules, language or model constructs, or, at the highest abstraction level, metaphors. Because software is the most flexible and ethereal of media, its architecture, in the sense of a defining structure, can be equally flexible and ethereal.

The most famous example is the original use of the desktop metaphor by Macintosh, a true architecture. To a considerable degree, when the overall human interface guidelines are added, this metaphor defines the nature of the system. It defines the types of information that will be handled and it defines much of the logic or processing. The guidelines force operation to be human-centered; that is, the system continuously parses user actions in terms of the effects on objects in the environment. As a result, Macintosh and Microsoft Windows programs are dominated by a main event loop. The foremost structure the programmer must define is the event loop, a loop in which system-defined events are sequentially stripped from a queue, mapped to objects in the environment, and their consequences evaluated and executed.

The power of the metaphor as architecture is twofold. First, the metaphor suggests much that will follow. If the metaphor is a desktop, its components should operate similarly to their familiar physical counterparts. This results in fast and retentive learning "by association" with the underlying metaphor. Second, it provides an easily communicable model for the system that all can use to evaluate system integrity. System integrity is maintained when the implementation of the metaphor is clear.

DIRECTIONS IN SOFTWARE ARCHITECTING

Software architecture and architecting have received considerable attention recently. There have been several special issues of IEEE Software magazine devoted to software architecture. Starting with Shaw and Garlan's book (Shaw and Garlan 1996), a whole series of books and other publications appeared. Much of the work in software architecture focuses on architectural structures and their analysis. Just as the term "architectural style" has a definite meaning in civil architecture, the usage is attached to style in software work. In the terminology of this book, work on software architecture styles is attempting to find and classify the high-level forms of software and their application to particular problems.

Focusing on architecture is a natural progression of software and programming research that has steadily ascended the ladder of abstraction. Work on structured programming led to structured design and to the multitasking and object-oriented models later generalized to multidisciplinary and multiview approaches described in Chapter 10. The next stage of the progression is to further classify the large-scale structures that appear as software systems become progressively larger and more complex.

The most developed work in software architecture primarily addresses the product of architecting (the structure or architecture) rather than the process of generating it. The published studies cover topics such as classifying architectures, mapping architectural styles to particularly appropriate applications, and using software frameworks to assemble multiple related software systems. Work on software architecture patterns is effectively work on process, in that it provides directive guidance in forming a software architecture. This book presents some common threads of the architectural process that underlie the generation of architectures in many domains. Once a particular domain is entered, such as software, the architect should make full use of the understood styles, frameworks, or patterns in that domain.

The flavor of current work in software architecture is best captured by reviewing some of its key ideas. These include the classification of architectural styles, patterns and pattern languages in software, and software frameworks.

Architectural Styles

At the most general level, a style is defined by its components, connectors, and constraints. The components are the things from which the software system is composed. The connectors are the interfaces by which the components interact. A style sets the types of components and connectors that will make up the system. The constraints are the requirements that define system behavior. In the current usage, the architecture is the definition in terms of form, which does not explicitly incorporate the constraints. To understand the constraints, one must look to additional views.

As a simple example, consider the structured design models described previously. A pure structured style would have only one component type, the routine, and only one connector type, invocation with explicit data passing. A software system composed using only these components and connectors could be said to be in the structured style. But the notion of style can be extended to include considerations of its application and deviations.

David Garlan and Mary Shaw give this discussion of what constitutes an architectural style (Garlan and Shaw 1993) p. 6:

> An architectural style, then defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style. Additionally, a style might define topological constraints on architectural descriptions (e.g. no cycles). Other constraints—say, having to do with execution semantics—might also be part of the style definition.
>
> Given this framework, we can understand what a style is by answering the following questions: What is the structural pattern—the components, connectors, and topologies? What is the underlying computational model? What are the essential invariants of the style—its "load bearing walls"? What are some common examples of its use? What are the advantages and disadvantages of using that style? What are some of the common specializations?

Garlan and Shaw have gone on to propose several root styles (Garlan and Shaw 1993, Garlan 1995). As an example, their first style is called "pipe and filter." The pipe and filter style contains one type of component, the filter, and one type of connector, the pipe. Each component inputs and outputs streams of data. All filters can potentially operate incrementally and concurrently. The streams flow through the pipes. Likewise, all stream flows are potentially concurrent. Because each component acts

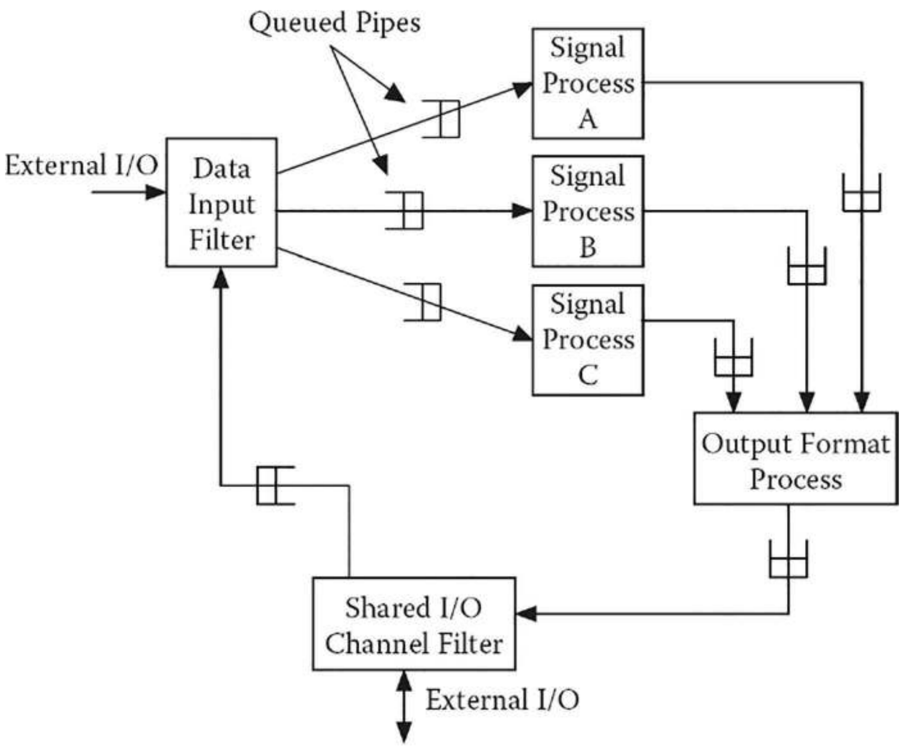


FIGURE 6.5 A pipe and filter system. Data flow through the system in pipes, which may actually have several types depending on their semantics for queuing, data push or pull, and so forth. Data are processed in filters that read and write pipes.

to produce one or more streams from one or more streams, it can be thought of as an abstract sort of filter. A pipe and filter system is schematically illustrated in Figure 6.5.

UNIX shell programs and some signal-processing systems are common pipe and filter systems. The UNIX shell provides direct pipe and filter abstractions with the filter's concurrent UNIX processes and the pipe's interprocess communication streams. The pipe and filter abstraction is a natural representation for block-structured signal-processing systems, in which concurrent entities perform real-time processing on incoming sampled data streams.

Some other styles proposed include object-oriented, event-based, layered, and blackboard. An object-oriented architecture is built from components that encapsulate both data and function and which exchange messages. An event-based architecture has as its fundamental structure a loop that receives events (from external interfaces or generated internally), interprets the events in the context of the system state, and takes actions based on the combination of event and state. Layered architectures emphasize horizontal partitioning of the system with explicit message passing and function calling between layers. Each layer is responsible for providing a well-defined interface to the layer above. A blackboard architecture is built from a set of concurrent components that interact by reading and writing asynchronously to a common area.

Each style carries its advantages and weaknesses. Each of these styles is a description of an implementation from an implementer's point of view, and specifically from the software implementer's point of view. They are not descriptions from the user's point of view, or even from the point of view of a hardware implementer on the system. A coherent style, at least of the type currently described, gives a conceptual integrity that assists the builder but may be no help to the user. Having a coherent implementation style may help in construction, but it is not likely to yield dramatic improvements in productivity or quality, because it does not promise to dramatically cut the size of what must be implemented.

This is reflective of a significant fraction of the software architecture literature. The primary focus is on the structure of the software, not on the structure of the problem that the software is to solve. The architecture description languages being studied are primarily higher-level or more abstract descriptions of programming language constructs. User concerns enter the current discussion typically through analysis. So, for example, an architecture description language developer may be concerned with how to analyze the security properties of a system description written in the language. This approach might be termed "structuralist." It places the structure of the software first in modeling and attempts to derive all other views from it. There is an intellectual attraction to this approach because the structural model becomes the root. If the notation for structure can be made consistent, then the other views derived from it should retain that consistency. There is no problem of testing consistency across many views written in different modeling languages. The weakness of the approach is that it forces the stakeholders other than the software developers to use an unfamiliar language and trust unfamiliar analyses. In the security example, instead of using standard methods from the security community, those concerned with security must trust the security analysis performed on the architectural language. This approach may grow to be accepted by broad communities of stakeholders, but it is likely to be a difficult sell.

In contrast to the perspective that places structure first in architecture, this book has repeatedly emphasized that only the client's purpose should be first. The architect should not be removed from the purpose or requirements; the architect should be immersed in them. This is a distinction between architecting as described here and as is often taught in software engineering. We do not assume that requirements precede architecture. The development of requirements is part of architecting, not its preconditions.

The ideal style is one that unifies both the user's and builder's views. The mathematical languages mentioned earlier are examples. They structure the system from both a user's and an implementer's point of view. Of course, the internals of the implementation of such a complex software system will contain many layers of abstraction. Almost certainly, new styles and abstractions specific to the demands of implementation in real computers will have to arise internally. When ideal styles are not available, it is still reasonable to seek models or architectural views that unify some set of considerations larger than just the software implementer. For the implementation of complex systems, it would be a useful topic of research to find models or styles that encompass a joint hardware–software view.

Architecture through Composition

Patterns, styles, and layered abstraction are inherent parts of software practice. Except for the rare machine-level program, all software is built from layered abstractions. High-level programming languages impose an intellectual model on the computational machine. The nature of that model inevitably influences what kinds of programs (systems) are built on the machine.

The modern trend is to build systems from components at higher and higher levels of abstraction. It is necessary because no other means are available to build very large and complex systems within acceptable time and effort limits. Each high-level library of components imposes its own style and lends itself to certain patterns. The patterns that match the available libraries are encouraged, and it may be very difficult to implement architectures that are not allowed in the libraries.

Example: Graphical Macintosh and Windows programs are almost always centrally organized around an event loop and handlers, a type of event-driven style. This structure is efficient because the operating systems provide a built-in event loop to capture user actions such as mouse clicks and key presses. However, because neither had multithreading abstractions until the from-the-ground-up operating system revamps of the 1990s, a concurrent, interacting object architecture was difficult to construct. Many applications would have benefited from a concurrent interaction object architecture, but these architectures were very difficult to implement within the constraints of existing libraries. As both systems evolved, direct support for multithreaded, concurrent processes has worked its way into all aspects of both systems, user interfaces included.

The logical extension is to higher and higher-level languages and from libraries to application-specific languages that directly match the nature of the problem they were meant to solve. Modern mathematical software packages are, in effect, very high-level software languages designed to mimic the problem they are meant to solve. The object of the packages is to do technical mathematics. So rather than provide a language into which the scientist or engineer must translate mathematics, the package does the mathematics. This is similar to computer-aided design packages, and indeed to most of the shrink-wrap software industry. These packages surround the computer with a layered abstraction that closely matches the way users are already accustomed to working.

Actually, the relationship between application-specific programming language, software package, and user is more symbiotic. Programmers adapt their programs to the abstractions familiar to the users. However, users eventually adapt their abstractions to what is available and relatively easy to implement. The best example is the spreadsheet. The spreadsheet as an abstraction partially existed in paper form as the general ledger. The computer-based abstraction has proven so logical that users have adapted their thinking processes to match the structure of spreadsheets. It should probably be assumed that this type of interactive relationship will accelerate when the first generation of children to grow up with computers reaches adulthood.

HEURISTICS AND GUIDELINES IN SOFTWARE

The software literature is a rich source for heuristics. Most of those heuristics are specific to the software domain and are often specific to restricted classes of a software-intensive system. The published sets of software heuristics are quite large. The newer edition of Brook's *The Mythical Man-Month: Essays in Software Engineering* (Brooks Jr 1995) includes a new chapter, "The Propositions of the Mythical Man-Month: True or False?" which lists the heuristics proposed in the original work. The new chapters reinforce some of the central heuristics and reject a few others as incorrect.

The heuristics given in Man-Month are broad ranging, covering management, design, organization, testing, and other topics. Several other sources give specific design heuristics. The best sources are detailed design methodologies that combine models and heuristics into a complete approach to developing software in a particular category or style. We discuss some of the software-specific aspects of modeling methods in Chapters 10 and 11. Even more specific guidelines are available for the actual writing of code. A book by McConnell (2004) contains guidelines for all phases and a detailed bibliography.

From this large source set, however, there are a few heuristics that particularly stand out as broadly applicable and as basic drivers for software architecting:

- Choose components so that each can be implemented independently of the internal implementation of all others.
- Programmer productivity in lines of code per day is largely independent of language. For high productivity, use languages as close to the application domain as possible.
- The number of defects remaining undiscovered after a test is proportional to the number of defects found in the test. The constant of proportionality depends on the thoroughness of the test but is rarely less than 0.5.
- Very low rates of delivered defects can be achieved only by very low rates of defect insertion throughout software development, and by layered defect discovery—reviews, unit test, system test.
- Software should be grown or evolved, not built.
- The cost of removing a defect from a software system rises exponentially with the number of development phases since the defect was inserted.
- The cost of discovering a defect does not rise. It may be cheaper to discover a requirement defect in customer testing than in any other way, hence the importance of prototyping.
- Personnel skill dominates all other factors in productivity and quality.
- Do not fix bugs later; fix them now.

As has been discussed, the evolvability of software is one of its most unique attributes. A related heuristic is that a system will develop and evolve much more rapidly if there are stable intermediate forms than if there are not. In an environment where wholesale replacement is the norm, what constitutes a stable form? The previous discussion has already talked about releases as stable forms and intermediate hardware configurations. From a different perspective, the stable intermediate forms are the unchanging components of the system architecture. These elements that do not change provide the framework within which the system can evolve. If they are well chosen—that is, if they are conducive to evolution—they will be stable and facilitate further development. A sure sign the architecture has been badly chosen is the need to change it on every major release. The architectural elements involved could be the use of specific data or control structures, internal programming interfaces, or hardware–software interface definitions. Some examples illustrate the impact of architecture on evolution.

Example: The point-to-point protocol (PPP) is a publicly defined protocol for computer networking over serial connections (such as modems). Its goal is to facilitate broad multivendor interoperability and to require as little manual configuration as possible. The heart of the protocol is the need to negotiate the operating parameters of a changing array of layered protocols (e.g., physical link parameters, authentication, IP control, AppleTalk control, compression, and many others). The list of protocols is continuously growing in response to user needs and vendor business perceptions. PPP implements negotiation through a basic state machine that is reused in all protocols, coupled with a framework for structuring packets. In a good implementation, a single implementation of the state machine can be "cloned" to handle each protocol, requiring only a modest amount of work to add each new protocol. Moreover, the common format of negotiations facilitates troubleshooting during tests and operations. During the protocol development, the state machine and packet structure have been mapped to a wide variety of physical links and a continuously growing list of network and communication support protocols.

Example: In the original Apple Macintosh operating system, the architects decided not use to the feature of their hardware to separate "supervisor" and "user" programs. They also decided to implement a variety of application programming interfaces through access to global variables. These choices were beneficial to the early versions because they improved performance. But these same choices (because of backward compatibility demands) greatly complicated efforts to implement advanced operating system features such as protected memory and preemptive multitasking. In the end, the dramatic evolution of the operating system required wholesale replacement, with limited backward compatibility. Another architectural choice was to define the hardware–software interface through the Macintosh Toolbox and the published Apple programming guidelines. The combination proved to be both flexible and stable. It allowed a long series of dramatic hardware improvements and even a transfer to a new hardware architecture, with few gaps in backward compatibility (at least for those developers who obeyed the guidelines). Even as the old operating system was entirely replaced, the old programming interface survived through minimal modifications allowing recompilation of programs.

Example: The Internet Protocol combined with the related Transmission Control Protocol (TCP/IP) has become the software backbone of the global Internet. Its partitioning of data handling, routing decisions, and flow control has proven to be robust and amenable to evolutionary development. The combination has been able to operate across extremely heterogeneous networks with equipment built by countless vendors. Although there are identifiable architects of the protocol suite, control of protocol development is quite distributed with little central authority. In contrast, the proprietary networking protocols developed and controlled by major vendors performed relatively poorly at scaling to diverse networks. One limitation in the current IP protocol suite that has become clear is the inadequacy of its 32-bit address space. However, the suite was designed from the beginning with the capability to mix protocol versions on a network. As a result, the deployed protocol version has been upgraded several times (and will be again to IPv6).

EXERCISES

1. Consult one or more of the references for software heuristics. Extract several heuristics and use them to evaluate a software-intensive system.
2. Requirements defects that are delivered to customers are the most costly to *remove* because of the likelihood they will require extensive rework. However, *discovering* such defects any time before customer delivery is likewise very costly because only the customers' reaction may make the nature of the defect apparent. One approach to this problem is prototyping to get early feedback. How can software be designed to allow early prototyping and feedback on the information gained without incurring the large costs associated with extensive rework?
3. Pick three software-intensive systems of widely varying scope, for example, a pen computer-based data-entry system for warehouses, an internetwork communication server, and the flight control software for a manned space vehicle. What are the key determinants of success and failure for each system? As an architect, how would these determinants change your approach to concept formulation and certification?
4. Examine some notably successful or unsuccessful software-intensive systems. To what extent was success or failure due to architectural (conceptual integrity, feasibility of concept, certification) issues and to what extent was it due to other software process issues?
5. Are their styles analogous to those proposed for software that jointly represent hardware and software?

NOTE

1 The numbers are anecdotal but reflect private communications to one of the present authors from a wide variety of sources.

REFERENCES

Ahmad, M. A., L. Yaramis, and T. D. Roy. (2023). "Creating trustworthy LLMS: Dealing with hallucinations in healthcare AI." arXiv preprint arXiv:2311.01463.

Bergner, K., et al. (2005). Dosam–domain-specific software architecture comparison model. *International Conference on the Quality of Software Architectures*, Berlin: Springer.

Brooks Jr, F. P. (1995). *The Mythical Man-Month (Anniversary ed.)*, Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Cock, P. J., et al. (2009). "Biopython: Freely available Python tools for computational molecular biology and bioinformatics." *Bioinformatics* **25**(11): 1422.

del Campo, M. and N. Leach (2022). *Machine Hallucinations: Architecture and Artificial Intelligence*, New York: John Wiley & Sons.

DeMarco, T. (1995). On systems architecture. *Proceedings of the 1995 Monterey Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Specification-Based Software Architectures*, Monterey, CA.

DeMarco, T. and T. Lister (2013). *Peopleware: Productive Projects and Teams*, Boston, MA: Addison-Wesley.

Garlan, D. (1995). What is style. *Proceedings of Dagshtul Workshop on Software Architecture*, Saarbruecken, Germany.

Garlan, D. and D. E. Perry (1995). "Introduction to the special issue on software architecture." *IEEE Transactions on Software Engineering* **21**(4): 269–274.

Garlan, D. and M. Shaw (1993). *In Introduction to Software Architecture*, Pittsburgh, PA: Carnegie Mellon University.

Goldstein, H. (2005). "Who killed the virtual case file?[case management software]." *IEEE Spectrum* **42**(9): 24–35.

Jones, C. (1985). *Programming Productivity*, New York: McGraw-Hill, Inc.

Jones, C. (2008). *Applied Software Measurement: Global Analysis of Productivity and Quality*, New York: McGraw-Hill Education Group.

Kruchten, P. B. (1995). "The 4+1 view model of architecture." *IEEE Software* **12**(6): 42–50.

Leveson, N. G. (2017). "The Therac-25: 30 years later." *Computer* **50**(11): 8–11.

Leveson, N. G. and C. S. Turner (1993). "An investigation of the Therac-25 accidents." *Computer* **26**(7): 18–41.

Maguire, S. (1994). *Debugging the Development Process*, Redmond, WA**:** Microsoft Press.

McConnell, S. (2004). *Code Complete*, London: Pearson Education.

Ryu, T.-W. (2009). "Benchmarking of BioPerl, Perl, BioJava, Java, BioPython, and Python for primitive bioinformatics tasks and choosing a suitable language." *International Journal of Contents* **5**(2): 6–15.

Shaw, M. and D. Garlan (1996). *Software Architecture: Perspectives on an Emerging Discipline*, Englewood Cliffs, NJ: Prentice-Hall, Inc.

Tracz, W. (1995). "DSSA (domain-specific software architecture) pedagogical example." *ACM SIGSOFT Software Engineering Notes* **20**(3): 49–62.

Zhang, Y., et al. (2023). "Siren's song in the AI ocean: a survey on hallucination in large language models." arXiv preprint arXiv:2309.01219.

Case Study 6

The Global Positioning System

The Global Positioning System (GPS) is one of the great success stories of the late 20th century. Most readers of this book will have had some experience with GPS, either in a car or with a handheld unit, though its usage is now so ubiquitous that many users are not even aware that they are using it. GPS is a large system that has undergone extensive evolution. Early in its history, as well as in the history of its predecessor programs, it was a centrally controlled system. As it has evolved and become extremely successful, control has gradually migrated away from the GPS program office. The larger enterprise of satellite-based position, navigation, and timing in which GPS is a component is no longer controlled wholly by a single program office; indeed, it is no longer controlled by the U.S. government. Non-U.S. systems like the European Galileo system, the Chinese Beidou, and the refreshed Russian GLONASS have turned the Global Navigation Satellite System (GNSS) into a full-fledged collaborative system, the subject of the next chapter.

GPS is a fitting case study to conclude Part II, looking either forward or backward. Looking forward, to Chapter 7, GPS is partially a collaborative system and partially not. As with most partial cases, the fact that it is not firmly in or out provides a better discussion of just what the border is. Looking back at the earlier chapters, we can see many of the other points we sought to emphasize. GPS was largely (though not exclusively) a technology-driven system where the sponsors were well in front of user demand. GPS illustrates the principles of architecture through invariants, the importance of a small architecture team, and the ripe timing with respect to both technology and user and institutional needs. Most lessons one wants to learn in architecture have occurred during the lifetime of the system.

THE HISTORY

The moment (actually the weekend) when the GPS architecture was determined can be easily identified — but that moment was the result of a long series of earlier steps. The history of satellite navigation leads up to GPS and then extends beyond it. That GPS came about when it did and in the form it did is, at least, somewhat surprising. It need not have come about, and certainly, there was no inevitability in when it came about. The fact that it did when and how it did, and how successful it has become, illustrates major points in the systems architecting of both conventional and collaborative systems.

DOI: 10.1201/9781003483595-14

Some Notes on Sources

A system as famous as the GPS has attracted considerable attention. There are both scholarly and popular accounts of many elements of its development. During the period between the origination of the overall concept of satellite navigation (in the late 1950s) and when the GPS was architected, there was extensive study. Much of that study was non-public but has since been revealed, but much of it was public and published in technical forums. In the years since the GPS reached operational capability, there have been many retrospective articles.

The author is indebted to his aerospace colleague Glenn Buchan who wrote an exceptionally fine case study from which we have drawn a great deal. The Air Force Institute of Technology has also published an extensive case study of the Global Positioning System, though it focuses more on the program events after the initial concept was formed than before (O'Brien and Griffin 2007). Two histories by the primary leader of the program, Dr. Bradford Parkinson, are Parkinson and Gilbert (1983) and Parkinson et al. (1995). Interestingly, for the purposes of this discussion, there has been considerable debate over who "invented" versus who architected the GPS. See Easton (1985, 2006) and the additional responses by Parkinson and Gilbert to the earlier article in the Proceedings of the IEEE.

GPS is publicly well known enough that there is now even a film, named after the holiday weekend when the core team met in the Pentagon to hash out what became the successful concept (Sylvester 2018). Like most films, it takes a point of view, one that might not be shared by all of the participants.

The Origins of GPS: The Foundational Programs

Position determination and navigation are fundamental to military operations, with air, sea, or land. For all of military history, extensive technological effort has been expended to improve navigational capabilities. At the beginning of the space era (late 1950s), each of the U.S. military services viewed the navigation problem quite differently.

Inertial Navigation and Its Limits

The U.S. Air Force and Navy were heavily investing in inertial navigation systems. Inertial navigation was particularly well suited to nuclear missiles. It was sufficiently accurate to guide a missile (flight times of 30 minutes or less) with a nuclear warhead (assuming the launch position is accurately known) and was immune to external interference. For the Air Force, this was a sufficient solution, as launch points were fixed and their locations known a priori precisely. For the Navy, the initial positioning problem was significant. Inertial navigation systems inherently drift on time scales of days. Thus, an inertial navigation system in a ship or submarine, although very accurate over a day or so, must be "recentered" regularly or its accuracy will continuously degrade. Because ships, and especially ballistic missile submarines, are expected to operate at sea for months at a time, the problem of correcting the drifting inertial units was central.

A naval ship requires global position determination to moderate high accuracy (tens to hundreds of meters) modest update rates (once to a few times a day). Ships know their altitude, so only two-dimensional positioning is required. In the 1960s, these capabilities were provided mainly by land-based radio navigation systems (e.g., LORAN [long-range navigation]), but these systems worked much less well than desired. Strategic aircraft would use global three-dimensional position determination but could operate only with the capabilities available at the time.

Weapon Delivery

In the 1950s, air-delivered weapons were still notoriously inaccurate, though guided weapons were a topic of vigorous research and important capabilities were coming into service. A hardened, fixed target, like a bridge, might require tens of sorties to destroy. A moving target, like a tank, could only be hit by a pilot flying low and in short-range direct sight. At the time, the primary solution of interest was sensor-guided weapons. Some of these were already worked well, like infrared-guided air-to-air missiles. Extensive work at the time was devoted to developing command, infrared, and radar-guided weapons, in all modes including surface-to-air, air-to-air, and air-to-surface.

The accuracy requirements for weapon delivery are very challenging. Accuracy must be a few meters at the most, three-dimensional position is required, and the platforms move very rapidly. The belief was that only sensor guidance was suitable for the task.

The Transit Program

Shortly after the launch of Sputnik in 1957, Frank McClure of Johns Hopkins Applied Physics Laboratory (APL) determined that measurements of Doppler shifts on the Sputnik signal could be used to infer lines of position at the receiver. Given multiple satellites in orbit and precision orbit determination, it would be possible to construct a navigation system. This concept was supported by the Advanced Research Projects Agency (ARPA) and led to a dedicated satellite launch in 1960. By 1965, 23 satellites had been launched and the Transit system had been declared operational (Kershner and Newton 1962, Danchik 1998).

Transit squarely addressed the Navy navigation problem. Transit provided a two-dimensional position update a few times a day with moderate accuracy, and it did it globally, in all weather, without any land-based infrastructure. From an architectural perspective, Transit was purpose-driven, had a clear architect and architecture, and the alignment between the user stakeholders and developers was close. As a result, it was a stable, evolving, and successful system. The witness to the strength of the linkage between the user base and the developers is that Transit satellites were being launched until 1988, and the system operated until 1996, long after GPS became operational.

Transit was important in the history of GPS in several respects.

1. Transit provided a useful service for roughly two decades before GPS became operational. Transit demonstrated the feasibility and utility of satellite navigation.

2. Building and operating Transit forced the resolution of important technical issues. In particular, it led to great improvements in orbit determination, gravity models, and in computing and predicting signal delays due to propagation through the atmosphere.
3. Transit set a precedent for commercial use. Transit was made available for commercial use in the late 1960s, and the number of commercial-use receivers came to far outnumber the number of military-use receivers (a harbinger of what was to come in GPS).

TIMATION

A related problem to navigation is the problem of time transfer or clock synchronization. This is the problem of accurately synchronizing clocks in distant locations, or of transferring a precise time measurement in one location to a distant location. This was a natural Navy concern as it was the foundation of the revolution in navigation in the 18th century when chronometers allowed the determination of longitude.

At the Naval Research Laboratory (NRL) in the 1960s, Roger Easton made two key realizations. He first realized that satellite clocks with appropriate signal transmission could be used to allow precision time transfer. He then realized that simultaneous time transfer from multiple synchronized clocks was equivalent to position determination (either two or three dimensions, depending on the number of satellites). These realizations were translated into a concept known as TIMATION (Easton 1972).

TIMATION was envisioned as a navigation system that would begin as an improved version of Transit and evolve into a full-blown system not unlike what GPS is today (Easton 1969). The early versions would use small constellations of low Earth-orbiting satellites (like Transit). This is relatively simple to build but provides only intermittent access (a few times a day, like Transit). Over time, the constellation would grow in size and move to higher orbits until global continuous coverage was achieved.

Project 621B

At the same time, the U.S. Air Force was studying satellite navigation through the Space and Missile Center (SMC) and its research center, The Aerospace Corporation. The project was known as 621B. The Aerospace Corporation president, Ivan Getting, had conceived of important elements of the satellite navigation concept in the 1950s and strongly advocated for it as the Aerospace Corporation president. The Air Force concentrated on longer-term, more ambitious goals consonant with Air Force mission needs. The Air Force concept was based on three-dimensional, high-precision, global position determination, and it was also concerned with electronic warfare and other military factors. 621B used the notion of simultaneous measurement of three delays to three known satellite locations, like Roger Easton's concept but with key differences. First, 621B performed measurement (or "pseudoranging") with the then-new scheme of digital pseudorandom-coded signals. Project 621B also makes a comprehensive study of alternatives for how to distribute the ranging calculations (looking at both passive and active signaling versions) and for how to get sufficient clock accuracy. The tradespace included high-accuracy atomic clocks on satellites, on user equipment, and distributed on the ground with various over-the-air synchronization schemes. The alternative set was covered in an Aerospace Corporation report (classified at the time of writing in the late 1960s) known as the "Nakamura Report" after the lead author. This report identified the approach (atomic clocks on orbit, with all position computation done in user terminal) eventually used in the GPS as the technically preferred, but that its technical risk was very high and other alternatives had better risk profiles.

The difference in signal approach was technically very aggressive for the time but led to key advantages. At the time, processing digital signals several megahertz (MHz) wide was very challenging (though it became trivial as the microchip revolution proceeded in the 1980s). However, the digitally coded signal had significant jam and interference resistance and largely solved the frequency coordination problem. With pseudorandom-coded signals, all transmitters in the system could operate at the same frequency and rely on the code processing to separate them.

THE ORIGIN OF GPS

By the early 1970s, Transit was a stable program with a satisfied user base, TIMATION and Project 621B were proceeding with their own ideas, and the larger vision of satellite navigation was not proceeding. The larger vision of a single U.S. constellation providing global navigation did exist. All three players presented global concepts at a conference in 1969 (Easton 1969, Kershner 1969, Woodford 1969). The differing stakeholder groups were engaged in bureaucratic warfare and made little headway beyond the scope of their own programs. But, in a remarkably short interval, an architecturally sweet compromise would be found among them all and converted into a successful development program.

Parkinson and Currie

The person most often associated with the success of the GPS is Bradford Parkinson, though see the references cited previously for additional history and controversy. Parkinson arrived in Los Angeles, California, in late 1972 to run the 621B program and quickly became a believer in the merits of global satellite navigation. Through happy, though accidental, circumstances, he was able to spend unusual amounts of time with the Director of Defence Research and Engineering (DDR&E) Malcolm Currie. Parkinson convinced Currie of the merits of satellite navigation, and Currie was convinced that Parkinson was the right person to lead the effort. Parkinson was tasked to form a joint proposal to be presented to the Defense Systems Acquisition Review Council (DSARC). The proposal he presented was essentially the 621B concept, and the DSARC immediately rejected it.

The Fateful Weekend

Over Labor Day weekend 1973, Parkinson assembled a small team of experts from each of the programs and areas and closeted them away from interference. Over the long weekend, they reached a consensus on a revised concept. The meeting where

TABLE CS6.1
Features of the Revised 1973 NAVSTAR (to become GPS) Concept

Concept of operations	Measure pseudoranges to four (or more) satellites and process to compute both the master time and three-dimensional position. All position computations occur in the receivers that operate entirely passively.
Constellation	21–24 satellites in inclined half-geosynchronous orbits.
Source of time	Atomic clocks on satellites updated from the ground. Receiver time computed from multiple satellites simultaneously with position.
Signal	Pseudorandom code at L-band (1,200 MHz). Two codes: one narrow and unencrypted, one wide and encrypted.

this happened has become known as "The Lonely Halls Meeting." The meeting has even been the subject of a documentary movie (Sylvester 2018). The revised concept combined features from the predecessor programs. The fundamental features of the revised concept are shown in Table CS6.1.

For those knowledgeable about GPS, the features should look familiar. They are virtually identical to those of today's operational GPS system. The revised concept was again presented to the DSARC after Currie had assured Parkinson that a true joint program would have strong support from his level and was approved.

As a result of DSARC approval, a joint program office was formed (the NAVSTAR program office), with Parkinson as the head. They were able to begin development very rapidly by incorporating major elements of the preexisting programs, particularly TIMATION. Critical space elements of the NAVSTAR-GPS concept were taken directly from TIMATION, and so the in-development TIMATION hardware was very useful for prototype demonstrations.

THE LONG ROAD TO REVOLUTION

Of course, the formation of the GPS concept is far from the end of the story. Although the essential architecture, in terms of its basic structure in signals, processing distribution, and constellation, was set in 1973 and has remained largely invariant, there was a very long road to successful development and an operational revolution. During the long road to operations, a key transition was taking place, the transition from a centrally controlled or "monolithic" system to a collaborative system in which there was no central authority over the whole. We discuss these two key points in the following sections.

The Timeline to Operation

The key events in the timeline to operations (and the present day) are shown in Figure CS6.1.

There were more than 10 years of serious activity, including the deployment of an operational system, before the architecture of GPS was set. After the architecture was set, development continued for roughly 20 years before GPS became fully

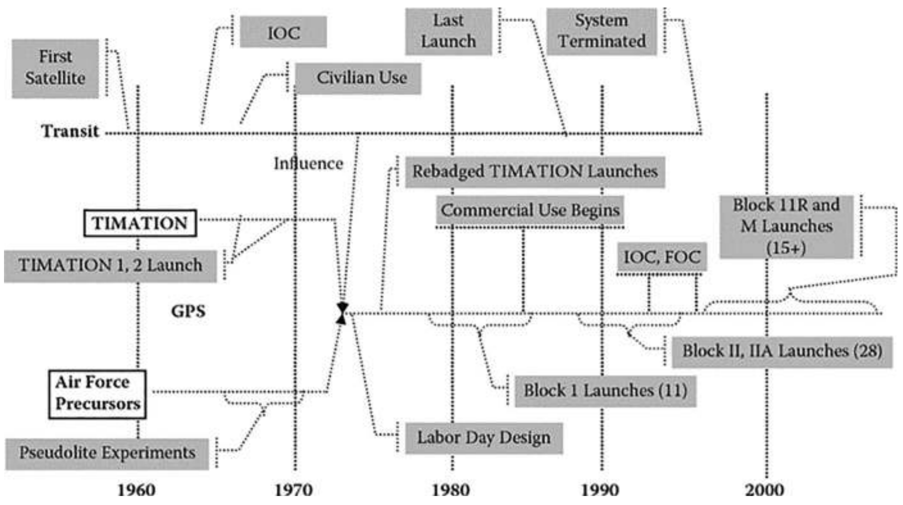


FIGURE CS6.1 Timeline for development of the Global Positioning System (GPS).

operational. Although it took 20 years to full operational capability, GPS delivered real utility much earlier. Not only did it deliver real utility, it had already broken out of the confines of the Joint Program Office and had a large commercial and multiagency component. By the early 1990s, there was even an international component, albeit an unintended one, in the Soviet Union's GLONASS system.

Commercial Markets and the Gulf War

The Transit system set a precedent for allowing civilian use of a military satellite navigation system. From very early on, the technology was available to commercial manufacturers and the navigation receivers could be openly purchased. By the end of the 1960s, the number of civilian Transit receivers (generally on civilian ships) greatly exceeded the number of military receivers. This precedent was repeated with GPS in the 1980s. The GPS signal architecture has the C/A code (on one frequency) sent unencrypted while the P-code signal, sent on both frequencies, is encrypted. The technical information needed to build a receiver was not originally publicly available, but there was nothing in the designs that would prevent its commercial licensing. These choices were deliberate and made with consideration of the Transit precedent.

In 1984, in the wake of the Soviet Union shoot-down of the off-course KAL 007 commercial airliner, it became U.S. policy to allow the free use of the C/A (coarse/acquisition) coded signal. The information necessary to build a C/A code receiver was made freely available to private industry. Because the C/A code has a narrower bandwidth than the military signal and is only transmitted on one frequency, the accuracy achieved is considerably less than with the military signal. In the 1980s and 1990s, accuracy was further degraded through the deliberate introduction of clock noise (known as "selective availability"). These degradations were intended to leave the accuracy sufficient for civilian navigational purposes while precluding use for specific military purposes like weapons control.

The microelectronics evolution of the 1980s enabled commercial development of GPS chipsets. Those chipsets in turn led to low-cost commercial receivers, which in turn benefited the military program. As the commercial market expanded, receivers dropped very quickly in size and cost. Commercial firms also began developing innovative software applications and even augmenting transmitter infrastructure systems.

The first Gulf War in 1991 gave a major impetus to GPS development. The use of receivers by ground troops in the war and GPS support for widely televised precision air strikes led to considerable publicity for GPS. The satellite constellation was sufficiently mature to provide substantial capability, although initial operational capability (IOC) had not been declared. The very public demonstration of the effectiveness of GPS in supporting guided weapons led to further interest in new guidance types. The continuing receiver cost reduction, and a politically driven requirement for weapons for stealth aircraft, led to the Joint Direct Attack Munition (JDAM) concept, a highly successful approach to precision weapons where a low-cost GPS receiver and guidance unit are mated to legacy "dumb" bombs.

Revolution in the Second Generation

The GPS revolution did not come with its deployment in its intended mission and in its original context. The original slogan of the program office was "Drop five bombs in the same hole," coupled with a goal to build cheap receivers. Although that capability has long ago been achieved, it itself has not been as valuable as newly conceived applications. It was really in the second generation, the generation after GPS reached full operational capability, that the revolution began, with applications and markets well outside those in the original architectural concept.

Ubiquitous GPS

Between the achievement of GPS's full operational capability and the present day, the number of GPS applications and receivers has exploded. GPS went from a specialized navigation device to something that could be included almost as an afterthought in other devices (e.g., cell phones for E911 service). Some application areas transitioned to deep dependence on GPS. Among them are surveying and time synchronization in power transmission and telecommunications networks. The ubiquity of GPS was made possible by receiver costs being driven down by Moore's law advances in digital electronics (which depended on GPS having a digital signal) and the development of new applications.

GPS Policy Frequently Lagged GPS Application

As GPS spread widely in usage and applications, there were ongoing conflicts over the appropriate policy response. An example of the policy lag relative to the technology was the period in the 1990s when the U.S. Federal Aviation Administration (FAA) and Coast Guard were deploying GPS enhancement transmitters at the same time the U.S. Air Force was maintaining the accuracy degrading selective availability features. One group within the U.S. Government was investing to enhance GPS civilian accuracy while another group was actively degrading it, sometimes in the same geographic areas. Eventually, after the year 2000, it became United States policy to abjure the use of selective availability. Even later, the GPS satellite acquisition program began flying satellites with dual and eventually triple-frequency civilian code transmitters, necessary for higher accuracy without terrestrial augmentation.

The policy lag had several sources. One was simply that the innovation-driven expansion of applications, enabled by cheap receivers, occurred much more quickly than policy could adapt. Probably more important is that there was no point below the White House where policy could be effectively coordinated. The control of the design and acquisition was in the U.S. Air Force GPS program office. But that office was an Air Force office within the Department of Defence, not a group with multiagency responsibility. As time went on, various other offices acquired important responsibilities, such as the FAA mentioned above, but those offices had no point of common authority convergence below the White House. In significant measure, control of GPS was beyond the U.S. Government by the early 2000s. The receiver market was dominated by commercial firms, and there were increasing innovations coming from non-governmental sources. Beyond that, international elements were appearing. The first non-U.S. satellite navigation system was the Russian GLONASS. This was followed by the European Galileo, the Chinese Beidou, and the Japanese regional augmentation system. GPS had morphed from a large and complicated, but reasonably conventional, system to a collaborative system, one not under the centralized control of any single entity.

GPS-Guided Weapons

A massive increase in the number of military receivers came with the development of GPS-guided weapons. As receiver costs dropped, largely because of the availability of commercial GPS chips, the cost of a receiver became less than that of even a very simple weapon. At this point, it became feasible to attach a GPS-based guidance system to a huge number of previously unguided weapons. The canonical example is a GPS-based guidance unit attached to 500–2,000 lb "dumb" bombs, known as the JDAM.

Even though the JDAM is a fine example of lateral exploitation (to use Art Raymond's term from the DC-3 story), the concepts of operation associated with the JDAM are more revolutionary. With a JDAM, especially the lighter-weight 500 lb JDAM, a large high-altitude aircraft, like a B-52 or B-1 bomber, could become a close support aircraft. This concept of operation was invented on-the-fly during the Afghan war in 2001–2002. Large aircraft could loiter persistently in the battle area waiting for calls from ground troops. To make the concept work, ground troops needed to be able to precisely measure the GPS coordinates of targets (easily achieved with laser rangefinders coupled to GPS receivers) and communicate directly to the aircraft overhead. When the whole concept was in place, it could be rapidly improved by realizing that smaller guided bombs worked as well, or better, than large guided bombs in close support, given accurate guidance. With smaller bombs, the large aircraft could carry many more and were not limited by persistence over the target, as older dedicated close support aircraft had been. The synergistic effects were large in combining the technology of GPS with changed concepts of operation and repurposed platforms.

ARCHITECTURE INTERPRETATION

GPS provides us with important lessons applicable to other systems that relate back to the topics of Part I and Part II. The lessons are: Right idea, right time, right people; Be technically aggressive, but not suicidal; Consensus without compromise; Architecture through invariants; and Revolution through coupled change.

Right Idea, Right Time, Right People

GPS would almost certainly not have happened when it did and how it did without particular people being in the right place at the right time. It was not obvious that the U.S. Navy and Air Force could reach a consensus on a global navigation concept, sell that concept through the acquisition bureaucracy, and then maintain it for more than the decade it took to become firmly established. Without Parkinson in the key position at that time, it is unlikely that the Air Force program would have discovered and adopted key Navy ideas and expanded its scope enough to become an established program. No stakeholder group in the Air Force needed global satellite navigation badly enough to allow the program to survive.

On the Navy side, they had a stable program plan. The TIMATION concept was intended to eventually lead to a global, high-precision system. Had the Navy been left alone, would something like GPS eventually have emerged? Obviously, we cannot ever know for sure, the experiment cannot be carried out. A global, three-dimensional system was in the minds of Navy developers, as demonstrated by Easton (1969). But two factors speak against the Navy's concept ever growing into the GPS system as it exists today. First, there was no Navy stakeholder with a combination of need and resources to grow the system to the level of capability now provided by GPS. Navy needs were well met by more incremental improvements that were more aligned with the limited resources of Navy space programs. Second, the most important Air Force contribution was the signal, the digital pseudorandom coded ranging signal used in the current GPS. This was a technically aggressive choice in 1973 and was unnecessary for the Navy mission (indeed it had drawbacks for the Navy mission). However, the pseudorandom noise (PRN) signal used in GPS provides it with significant jam resistance and considerably eases the problem of frequency management in crowded areas (such as urban and suburban areas of industrialized countries). The signal, and its placement in L-band, allows high-precision location (from tens-of-meters to meter-level accuracy) to be achieved without severe frequency management problems. This has been an important factor in the long-term success of GPS but was of little relevance to the Navy mission as understood in the 1970s.

Be Technically Aggressive, But Not Suicidal

Parkinson and his team made technically aggressive choices, with wisdom that is obvious in retrospect but was not so obvious in prospect. The most important over the long term was to base GPS ranging on the digital PRN signal. In the 1970s, processing a digital signal with a modulation rate from 1 to 10 MHz was very difficult, requiring many boards of custom hardware. With decades of advance in Moore's law, processing the same signals today is a trivial hardware exercise that easily fits into communications chipsets. Even though choosing an all-digital approach was aggressive in the 1970s, it was central to the achievement of cheap receivers in the 1990s. The price/performance curve for digital electronics has moved orders of magnitude in the intervening decade, but the same curve for analog hardware has moved much less. By the 1990s, commercial firms were able to enter the GPS market with receivers in form factors and prices acceptable to a wide consumer base only because most of the processing required was digital.

The choice of half-geosynchronous orbits was also aggressive, but not excessively so. The half-geosynchronous orbit allows for global simultaneous visibility to four satellites with a constellation of 25 or so satellites. The exact number depends on the specification for occasional brief outages. Higher orbits reduce the number of satellites required modestly but considerably increase the satellite weight (because of the higher power required). Lower orbits either incur a large radiation exposure penalty (in the Van Allen belts) or cause the number of satellites to increase enormously (potentially to hundreds), although lower orbits result in smaller and simpler satellites. Building satellites that survive in the half-geosynchronous orbit is more challenging than in low Earth orbit (because of higher radiation levels), but not excessively so.

Finally, the selected architecture of GPS placed precision clocks on the satellites, and not on the receivers. This meant that receivers needed only digital processing, and all sophisticated computation was done on the ground, not in the receivers. Over the long term, this was very beneficial. It meant that improved processing techniques could be deployed with each new generation of receiver, and receiver generation times have been far shorter than spacecraft generation times. However, it also required that atomic clocks operate precisely and reliably on satellites for up to a decade. Again, the previous work by the Navy had proven the possibility and had explored various design options for precision clocks in orbit, including both crystal and atomic clocks. Although the technology had to be matured by the GPS program, and they had to pursue a parallel sourcing strategy since there was no fully capable supplier at the time, the essential trades had already been made, and the data required for those trades had been acquired in well-designed experiments.

In all three cases, Parkinson's team made technically aggressive decisions but did not incur excessive risk. Although processing megabit/second PRN signals was challenging in the 1970s, it had already been demonstrated. The Project 621B experiments, along with other projects, had accomplished the task several times. Likewise, the space environment at half-geosynchronous was known, and the techniques for surviving in it were known and tested (albeit uncommon and expensive). High-precision clocks had been, and were being, flown in orbit, although the required long-term accuracy and reliability had not been demonstrated. In all three cases, the trade-offs could be made with concrete knowledge of the issues. That could not have been the case a few years earlier.

Consensus without Compromise

Even though the architecture of GPS reflected a fusion of ideas from many sources, it was not a watered-down compromise. The fusion genuinely took the best aspects of the approaches of several stakeholders and abandoned inferior aspects. There is little evidence of political compromise, the sort that might have insisted that “Air Force equities require that we do this and Navy equities require that we do that.” Instead, the elements selected from the different component programs were those that reflected consensus best choices or cut through consensus to adopt a clear strategic position.

Using the simultaneous position and time determination method from four pseudoranges can be seen as a clear consensus choice. Essentially, all parties would now agree, and agreed even then, that it represented the best choice overall. The impact on overall system simplicity is clear. All position determination is done through the signals broadcast by the satellites, no auxiliary terrestrial signal is required, the receiver is nearly all digital, and all serious computation is done in the receiver.

In the case of the choice of the all-digital signal, the consensus is not clear, but the choice reflects a very clear strategic choice. The all-digital, L-band signal was easy to frequency manage, allowed all satellites to share the same frequency, and led to cheap receivers. On the downside, the L-band signal penetrates poorly in buildings and even foliage, requires more difficult antennas, and the all-digital signal was challenging to process in the 1970s. The choice made a clear strategic decision; GPS would ride the improvements in digital electronics. It would exploit technology developments that were well underway but still far from ready. Parkinson’s team could have made a variety of other choices that would have compromised among the players and been easier in the near-term but would have missed the long-term opportunities.

Architecture as Invariants

GPS is an example of architecture as invariants. Between the origin of the joint program and 2007, the signals were unchanging, and the orbits underwent minimal change. Of these two, the signals were much more significant as an invariant. The constellation had already been morphed by the inclusion of terrestrial transmitters for local accuracy improvement. Only in the 2010s and 2020s has any change begun to appear in the signals. Currently launched satellites add new military signals, known as the M-code, to augment the legacy military codes. A copy of the unencrypted civilian signal has been added at a second frequency. The second signal improves accuracy by allowing direct measurement of ionospheric delay. This capability has been available for military users since the inception of GPS but has been unavailable to civilian users.

Architecture through invariants is particularly effective when evolution is important to long-term success. In the case of GPS, the invariant signals have allowed the decoupled evolution of the constellation and receivers. The receivers have undergone extensive evolution programmatically and technically without that change having to be coupled to change in the satellites. At the current time, receivers are developed dominantly by commercial firms with no formal relationship to the GPS program office.

Revolution through Coupled Change

The greatest impact of GPS came only through coupled change to affiliated systems and concepts of operation. The original slogan was "five bombs in the same hole and cheap receivers." The latter was achieved and then achieved beyond the original expectations. The former was never as important as was originally thought. Instead, the proliferation of cheap receivers enabled a whole range of new applications. Some of the innovations include the following.

- Extremely compact and low-cost receivers could be distributed to individual soldiers. Small units can accurately and continuously determine their position, and reporting of those positions enables new networked operational concepts.
- As receivers became cheap enough, they could be placed on weapons. Instead of guiding the weapon delivery platform to weapon release, GPS now guides the weapon from release to impact.
- The existence of precision-guided weapons at costs little larger than their unguided predecessors has resulted in a radical shift in the dominant operational concept for aerial weapons delivery (at least for the United States), from less than 10% of all aerial weapons being guided to roughly 90% in a period of 15 years.
- As guided weapons became the norm, the operational concept for platforms has shifted. In the Afghan war that began in 2001, the B-1 bomber was used for close air support, by loitering for long periods at high altitudes and dropping GPS-guided bombs on targets located by ground troops. Close air support was performed by small aircraft whose pilots delivered weapons visually and is now more effectively performed by an airplane designed originally for delivering strategic nuclear weapons.
- Ubiquitous position determination has been instrumental in enabling autonomous systems of every sort, from quadcopter drones to land vehicles to ships. The ease of constructing and operating autonomous vehicles has enabled multiple new concepts of operation in civilian and military areas in a process that is just getting started.
- The practice of surveying has been pervasively impacted by GPS. Surveyors, because they do not need position determination at high update rates, have been able to exploit a wide range of unanticipated processing techniques based on collecting long segments of GPS signals from a fixed location.
- The ability of GPS to provide very high accuracy, globally referenced time has led to its embedding into electric power and telecommunications control systems.
- GPS is now typically included in cell phones at a marginal cost to support electronic 911 service. The ability to track large numbers of cell phone users will lead to a wide range of new applications.

Revolution through coupled change is exemplified in GPS but is hardly unique to GPS. The most dramatic impacts of new technologies typically come from uses beyond the originally envisioned application. Those dramatic applications typically involve rethinking the problem being solved and the concept of operation involved. A simple application of new technology to old concepts of operation is almost never as valuable as what can be realized from creating new concepts of operation.

CONCLUSION

GPS is an exceptional example of architecture in a revolutionary system. Its original development is a classic example of architecting by a very small team with a tightly defined mission with the challenges of new technology. As it has developed, it has illustrated evolution toward a collaborative system and revolution through changes to concepts of operation. GPS is not quite a collaborative system. It is still run by a single, joint program office. However, many of the factors that drive GPS development are out of the control of that program office. Commercial receiver builders design in whatever features they wish. Several agencies continue to develop and deploy augmentation transmitters. And, most strikingly, international players are deploying competing programs that may contain compatible or interfering signals.

The greatest impact of GPS has been in areas outside the original conception of its use, and that success is a testament to the quality of the architecture. The core architecture of GPS (the signal and the position determination method) has been robust to extensive lateral exploitation. The willingness of the program office and sponsors to cede control of some segments and applications, allowing a collaborative system to form, has been central to long-term success. The multitude of applications is a witness to the basic insight that ubiquitous, global satellite navigation would be tremendously valuable.

REFERENCES

Danchik, R. J. (1998). "An overview of transit development." *Johns Hopkins APL Technical Digest* **19**(1): 19.

Easton, R. (1985). "Comments on NAVSTAR: Global positioning system - ten years later." *IEEE Proceedings* **73**(1): 168.

Easton, R. (2006). "Who invented the global positioning system?" The Space Review.

Easton, R. L. (1969). *Mid-Altitude Navigation Satellites. EASCON 69*, Washington, DC: IEEE.

Easton, R. L. (1972). "The role of time/frequency in Navy navigation satellites." *Proceedings of the IEEE* **60**(5): 557–563.

Kershner, R. and R. Newton (1962). "The transit system." *The Journal of Navigation* **15**(2): 129–144.

Kershner, R. B. (1969). *Low Altitude Navigation Satellite System. EASCON 69*, Washington, DC: IEEE.

O'Brien, P. and J. Griffin (2007). *Global Positioning System Systems Engineering Case Study*, Port San Antonio, TX**:** Air Force Center for Systems Engineering.

Parkinson, B. W. and S. W. Gilbert (1983). "NAVSTAR: Global positioning system—ten years later." *Proceedings of the IEEE* **71**(10): 1177–1186.

Parkinson, B. W., et al. (1995). "A history of satellite navigation." *Navigation* **42**(1): 109–164.

Sylvester, T. (2018). The Lonely Halls Meeting. USA, Pool Room Studios**:** 1 hour 29 minutes.

Woodford, J. B., Melton, W. C. and Dutcher, R. L. (1969). *Satellite Systems for Navigation Using 24 Hour Orbits. EASCON 69*, Washington, DC: IEEE.

7 Collaborative Systems

INTRODUCTION: COLLABORATION AS A CATEGORY

Most of the systems discussed so far have been the products of deliberate and centrally controlled development efforts. There was an identifiable client or customer (singular or plural), clearly identifiable builders, and users. Client, in the traditional sense, means the person or organization who sponsors the architect, and who has the resources and authority to construct the system of interest. The role of the architect existed, even if it was hard to trace to a particular individual or organization. The system was the result of deliberate value judgment by the client and existed under the control of the client. However, many systems are not under central control, either in their conception, their development, or their operation. The Internet is a canonical example, but many others exist, including electrical power systems, multinational defense systems, joint military operations, and intelligent transportation systems. These systems are all collaborative, in the sense that they are assembled and operate through the voluntary choices of the participants, not through the dictates of an individual client. These systems are built and operated only through a collaborative process.

Some systems are born as collaborative systems and others evolve that way. The Internet was not originally a collaborative system but has long ago passed out of centralized control. Global Positioning System (GPS) was not originally a collaborative system but is already at least partially one. GPS is part of a much larger Global Navigation Satellite System (GNSS) that is clearly a collaborative system. Other systems, such as those architected to be multicompany or multigovernmental collaborations are, or should be, considered as collaborative systems from the beginning.

A problem in this area is the lack of standard terminology for categories of system. Any system is an assemblage of elements that together possess capabilities not possessed by an element. This is just saying that a system possesses emergent properties, indeed that possessing emergent properties is the defining characteristic of a system. A microwave oven, a laptop computer, and the Internet are all systems; but each can have radically different problems in design and development.

So what do we call an assemblage of components each individually complex enough to be considered a system that, which acting together, produce still more emergent behavior? We could call them a system-of-systems, but that fails to distinguish that not all assemblages are the same. There are many different part-whole relationships, even when the components are themselves systems in their own right. Back in Chapter 3, we discussed product-lines where the component systems don't interact with each other but are from a family with common design and (often) shared parts. A venture capital or R&D manager might have a portfolio of systems that relate to common funding and management but not by design or interaction. Some very complex systems, such as a major space telescope, have many components

DOI: 10.1201/9781003483595-15

that are systems in their own right, but the integrated whole operates only as an integrated whole. Contrast that with intelligent transport (case study 4), where the parts interact voluntarily and may form and re-form various federations.

From an architecting and engineering perspective, what matters is which assemblages with different characteristics require different practices in architecting and engineering. If can we architect or design them in the same way, then differences are of little consequence. Publications by the authors (Maier 1998, 2019) have identified the classes of Collaborative System, Family-of-Systems, and Portfolio-of-Systems. The definition of Collaborative System published in Maier (1996) and Maier (1998) has become the most widely accepted definition of a system-of-systems (DoD 2008), albeit with some significant variations suggested later (Dahmann and Baldwin 2008) and various fine points of distinction.

The concept of a Collaborative System (or system-of-systems) discussed in this chapter covers systems distinguished by the voluntary nature of the systems' assembly and operation. Examples of systems in this category include most intelligent transport systems (IVHS 1992), military C4I and Integrated Battlespace (Butler et al. 1996), and partially autonomous flexible manufacturing systems (Hays 1988). Most readers will likely prefer the term equating of collaborative systems and systems-of-systems, as that has become accepted usage. One can refer to the authors publication where this originated for the full argument (Maier 1998).

What exactly is a collaborative system? In this chapter, a system is a "collaborative system" when its components

1. Are complex enough to be regarded as systems in their own right and interact to provide functions not provided by any of the components alone; that is, the components in combination make up a system.
2. The component systems fulfill valid purposes in their own right and continue to operate to fulfill those purposes if disassembled from the overall system.
3. The component systems are managed (at least in part) for their own purposes rather than the purposes of the whole. The component systems are separately acquired and integrated but maintain a continuing operational existence independent of the collaborative system.

A separate issue is how the components come to be combined together. Our interest here is in systems deliberately constructed. Some people are interested in nondeliberate combinations that form recognizable systems. Some refer to these as "organic" systems, and there are a variety of interesting examples in human society. DeMarco presented an example known as the "Bombay Box-Wallah" system (DeMarco 1995) (which even served as a character of a sort in a major movie (Batra 2013)), and others have described the operation of an ungoverned, yet organized urban environment (Girard and LAMBOT 1993).

We know the classification as a collaborative system (or not) is important because misclassification leads to serious problems. Especially important is a failure to architect for robust collaboration when direct control is impossible or inadvisable.

This can arise when the developers believe they have greater control over the evolution of a collaborative system than they actually do. In believing this, they may fail to ensure that critical properties or elements will be incorporated by failing to provide a mechanism matched to the problem.

As with other domains, collaborative systems have their own heuristics, and familiar heuristics may have new application. To find them for collaborative systems, we look first at important examples and then generalize to find the heuristics. A key point is the heightened importance of interfaces, and the need to see interfaces at many layers. The explosion of Internet technology has greatly facilitated collaborative system construction, but we find that the "bricks-and-mortar" of Internet-based collaborative systems are not at all physical. The building blocks are communication protocols, often at higher layers in the communications stack that are familiar from past systems.

COLLABORATIVE SYSTEM EXAMPLES

Systems built and operated voluntarily are not unusual, even if they seem very different from classical systems engineering practice. Most of the readers of this book will be living in capitalist democracies where social order through distributed decisions is the philosophical core of government and society. Nations differ in the degree to which they choose to centralize versus decentralize decision making, but the fundamental principle of organization is voluntary collaboration. This book is concerned with technological systems, albeit sometimes systems with heavy social or political overtones. So, we take as our examples the systems whose building blocks are primarily technical. The initial examples are the Internet, intelligent transportation systems (for road traffic), and joint air defense systems.

The Internet

When we say "the Internet," we are not referring to the collection of applications that have become so popular (e-mail, World Wide Web, chats, and social media). We are referring to the underlying communications infrastructure on which the distributed applications run. A picture of the Internet that tried to show all physical communications links active at one time would be a sea of lines with little or no apparent order. But, properly viewed, the Internet has a clear structure. The structure is a set of protocols called TCP/IP (Transmission Control Protocol/Internet Protocol). Their relationship to other protocols commonly encountered in the Internet is shown in Figure 7.1, modeled after (Peterson and Davie 2007). The TCP/IP suite includes the IP, TCP, and User Datagram Protocol (UDP) protocols in Figure 7.1. Note in Figure 7.1 that all the applications shown ultimately depend on IP. Applications can use only communications services supported by IP. IP, in turn, runs on many link and physical layer protocols. IP is "link friendly" in that it can be made to work on nearly any communications channel. This has made it easy to distribute widely but prevents much exploitation of the unique features of any particular communication channel.

The TCP/IP family protocols are based on distributed operation and management. All data are encapsulated in packets, which are independently forwarded through the

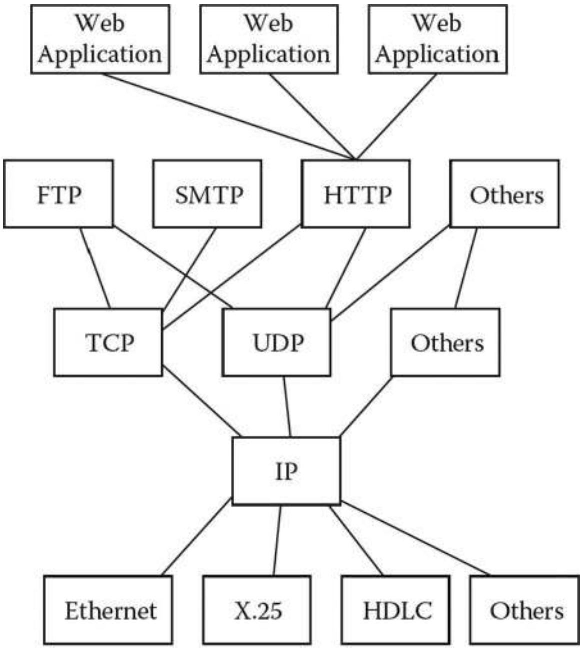


FIGURE 7.1 Protocol dependencies in the Internet.

Internet. Routing decisions are made locally at each routing node. Each routing node develops its own estimate of the connection state of the system through the exchange of routing messages (also encapsulated as IP packets). The distributed estimates of connection state are not, and need not, be entirely consistent or complete. Packet forwarding works in the presence of some errors in the routing tables (although introduction of bad information can also lead to collapse).

The distributed nature of routing information, and the memoryless forwarding, allows the Internet to operate without central control or direction in the classic sense. Of course, control exists, but it is a collaborative, decentralized mechanism based on agreements-in-practice between the most important players. A decentralized development community matches the decentralized nature of control and decentralized architecture itself. There is no central body with *coercive* power to issue or enforce standards. There is a central body that issues standards, the Internet Engineering Task Force (IETF), but its practices are unlike nearly any other standards body. The IETF approach to standards is, fundamentally, to issue only those which have already been developed and deployed. The IETF acts more in a role of recognizing and promulgating standards than in creating them. Its apparently open structure (almost anybody can go to the IETF and try and form a working group to build standards in a given area) actually has considerable structure, albeit structure defined by customary practices rather than mandates.

Broadly (not looking at all details), Figure 7.1 was as valid in 1990 and 2000 as it is in 2025. In detail, there are differences. Web-based applications were a limited, new thing in 1990. The lower-level physical communications protocols in 1990 were dominated more by ethernet than the vastly broader use of wireless by 2000 and especially by 2025. Various other protocols have come and gone, or been refined, in the same time period. In that time period, effectively many generations on the Internet time scale, the TCP/IP portion of the diagram has been pretty stable. Only toward the 2025 end of the time span has there been a major evolution at that layer with a slow transition from IPv4 to IPv6. This stability has been essential in how the Internet has evolved. It is a leading example of the principle of "Architecture is in the Invariants" discussed in this book.

The organization accepts nearly any working group that has the backing of a significant subset of participants. The working group can issue "Internet-drafts" with minimal overhead. For a draft to advance to the Internet equivalent of a published standard, it must be implemented and deployed by two or more independent organizations. All Internet standards are available for free, and strong efforts are made to keep them unencumbered by intellectual property. Proprietary elements are usually accepted only as optional extensions to an open standard. But under the surface, the practices of the organization create different forms of virtual mandates. Anybody can try to form a working group, but working code and a willingness to open source it speaks far louder than procedures. Powerful organizations can find their efforts in the IETF stymied by smaller players, if those players are faster and more willing to distribute working implementations, and forge alliances with others who will demonstrate interoperability in working systems.

Distributed operation, distributed development, and distributed management are linked. The Internet can be developed in a collaborative way largely because its operation is collaborative. Because the Internet uses best-effort forwarding and distributed routing, it can easily offer new services, as long as those new services depend only on best-effort operation, without changing the underlying protocols. In contrast, services requiring hard network-level guarantees cannot be offered. New services can be implemented and deployed by groups that have no involvement in developing or operating the underlying protocols, but only so long as those new services do not require any new underlying services. So, for example, groups were able to develop and deploy IP-Phone (a voice over the Internet application) without any cooperation from TCP/IP developers or even Internet service providers. However, the IP-Phone application cannot offer any quality-of-service guarantees because the protocols it is built on do not offer simultaneous delay and error rate bounding.

In contrast, networks using more centralized control can offer richer building block network services, including quality-of-service guarantees. However, they are much less able to allow distributed operation. Also, the collaborative environments that have produced telecommunications standards have been much slower moving than the Internet standards bodies. They have not adopted some of the practices of the Internet bodies that have enabled them to move quickly and rapidly capture market share. Of course, some of those practices would threaten the basic structure of the existing standards organizations.

In principle, a decentralized system like the Internet should be less vulnerable to destructive collective phenomena and be able to locally adapt around problems. In practice, both the Internet with its distributed control model and the telephone system with its greater centralization have proven vulnerable to collective phenomena.

Distributed control protocols like TCP/IP are prone to collective phenomena in both transmission and routing; see Bertsekas and Gallager (2021) particularly Chapter 6. Careful design and selection of parameters have been necessary to avoid network collapse phenomena. One reason is that the Internet uses a "good intentions" model for distributed control, which is vulnerable to nodes that misbehave either accidentally or deliberately. There are algorithms known that are robust against bad intentions faults, but they have not been broadly incorporated into network designs. The decentralized nature of the system has made it especially difficult to defend against coordinated distributed attacks (e.g., distributed denial of service attacks). Centralized protocols often deal more easily with these attacks because they have strong knowledge of where connections originate and can initiate aggressive load-shedding policies under stress.

Wide-area telephone blackouts have attracted media attention and shown that the more centralized model is also vulnerable. Complex collective phenomena occur in large networked systems with or without centralized control. The argument about decentralized versus centralized fault tolerance has a long history in the electric power industry and, even today, has not reached full resolution.

Intelligent Transportation Systems

The goal of most initiatives in intelligent transportation is to improve road traffic conditions through the application of information technology. See IVHS (1992), Maier (1997), and Barbaresso et al. (2014) for additional broad discussion. We already discussed several aspects of ITSs in "Case Study 4," preceding Chapter 5. We pick out one issue to illustrate how a collaborative system may operate and the architectural challenges in making it happen.

One intelligent transportation concept is called "fully coupled routing and control." In this concept, a large fraction of vehicles are equipped with devices that determine their position and periodically report it to a traffic monitoring center. The device also allows the driver to enter his or her destination when beginning a trip. The traffic center uses the traffic conditions report to maintain a detailed estimate of conditions over a large metropolitan area. When the center gets a destination message, it responds with a recommended route to that destination, given the vehicle's current position. The route could be updated during travel if warranted. The concept is referred to as fully coupled because the route recommendations can be coupled with traditional traffic controls (e.g., traffic lights, on-ramp lights, and reversible lanes).

Obviously, the concept brings up a wide array of sociotechnical issues. Many people may object to the lack of privacy inherent in their vehicle periodically reporting its position. Many people may object to entering their destination and having it reported to a traffic control center. Although there are many such issues, we narrow down once again to just one that best illustrates collaborative system principles. The concept works only if:

1. A large fraction of vehicles have, and use, the position reporting device.
2. A large fraction of drivers enter their (actual) destination when beginning a trip.
3. A large fraction of drivers follow the route recommendations they are given.

When the concept was first floated (1980s) these conditions were hypothetical, at best. Today, they describe how a lot of people normally drive, but the mechanisms are "central" only in that the number of application providers is low (mostly Google, Apple, Waze, and a few others in the United States, a similar small set globally). Vehicles on the roads are mostly privately owned and operated for the benefit of their owners. With respect to the collaborative system conditions, the concept meets it if using the routing system is voluntary, as it is today. The vehicles continue to work whether or not they report their position and destination. And vehicles are still operated for their owner's benefit, not for the benefit of some "collective" of road users. So, if we are architecting a collaborative traffic control system, we have to explicitly consider how the three conditions above needed to gain the emergent capabilities are ensured.

One way to ensure them is to not make the system collaborative. Under some social conditions, we can ensure conditions one to three by making them legally mandatory and providing enforcement. It is a matter of judgment whether or not such a mandatory regime could be imposed.

If one judges that a mandatory regime is impossible, then the system must be collaborative. Given that it is collaborative, there are many architectural choices that can enhance the cooperation of the participants. For example, we can break apart the functions of traffic prediction, routing advice, and traditional controls and allocate some to private markets. Imagine an urban area with several "Traffic Information Provider" services. These services are private and subscription based, receive the position and destination messages, and disseminate the routing advice. Each driver voluntarily chooses a service, or none at all. If the service provides accurate predictions and efficient routes, it should thrive. If it cannot provide good service, it will lose subscribers and die.

Such a distributed, market-based system may not be able to implement all of the traffic management policies that a centralized system could. However, it can facilitate social cooperation in ways the centralized system cannot. A distributed, market-based system also introduces technical complexities into the architecture that a centralized system does not. In a private system, it must be possible for service providers to disseminate their information securely to paying subscribers. In a public, centralized system, information on conditions can be transmitted openly. In the 1990s, this would have been (indeed was) a discussion about possibilities. At the writing of this 4th edition, it describes how the situation as it evolved.

Joint Air Defense Systems

A military system may seem like an odd choice for collaborative systems. After all, military systems work by command, not voluntary collaboration. Leaving aside the social issue that militaries must always induce loyalty, which is a social process, the degree to which there is a unified command on military systems or operations is variable. A system acquired and operated as a single service can count on central direction. A system belonging to a single nation but spanning multiple services can theoretically count on central direction, but in practice, it is likely to be largely collaborative. A system that comes together only in the context of multiservice, multinational, coalition military operations cannot count on central control and is always a collaborative system.

All joint military systems and operations have a collaborative element, but here we consider just air defense. An air defense system must fuse a complex array of sensors (ground radars, airborne radars, beacon systems, human observers, and other intelligence systems) into a logical picture of the air space, and then allocate weapon systems to engage selected targets. If the system includes elements from several services or nations, conflicts will arise. Nations, and services, may want to preferentially protect their own assets. Their command channels and procedures may affect greater self-protection, even when ostensibly operating solely for the goals of the collective.

Taking a group of air defense systems from different nations and different services and creating an effective integrated system from them is the challenge. The obvious path might be to try and convert the collection into something resembling a single-service air defense system. This would entail unifying the command, control, and communications infrastructure. It would mean removing the element of independent management that characterizes collaborative systems. If this could be done, it is reasonable to expect that the resulting integrating system would be closer to a kind of point optimum. But the difficulties of making the unification are likely to be insurmountable.

If, instead, we accept the independence, then we can try and forge an effective collaborative system. The technical underpinnings are clearly important. If the parts are going to collaborate to create integrated capabilities greater than the sum of the parts, they are going to have to communicate. So, even if command channels are not fully unified, communications must be highly interoperable. In this example, as in other sociotechnical examples, the social side should not be ignored. It is possible that the most important unifying elements in this example will be social. These might include shared training or educational background, shared responsibility, or shared social or cultural background.

Air defense systems have another important aspect, they are subject to direct attack. Put directly, they are subject to unexpected and violent disassembly in normal use. Their resilience to direct attack is an essential property. An air defense system that achieves great efficiency until it is attacked is useless. There is a good argument that configuring air defense systems as collaborative systems is an essential resilience strategy. Being a collaborative system means the components have independent capabilities and independent management. They can carry out air defense missions whether connected to the whole or not (and so can continue to operate even if collective links are lost). The component commanders are empowered to carry out a partial air defense mission independent of the whole, and so the initiative to operate whether fully connected or not is part of the inherent design.

ANALOGIES FOR ARCHITECTING COLLABORATIVE SYSTEMS

One analogy that may apply is the urban planner, just as we used the civil architect as an analogy for the systems architect. The urban planner, like the architect, develops overall structures. The architect structures a building for effective use by the client; the urban planner structures effective communities. The client of an urban planner is usually a community government, or one of its agencies. The urban planner's client and the architect's client differ in important respects. The architect's client is making a value judgment for himself or herself, and presumably has the resources to put into action whatever plan is agreed to with the architect. When the architect's plan is received, the client will hire a builder. In contrast, the urban planner's client does not actually build the city. The plan is to constrain and guide many other developers and architects who will come later, and hopefully guide their efforts into a whole greater than if there had been no overall plan. The urban planner and client are making value judgments for other people, the people who will one day inhabit the community being planned. The urban planner's client usually lacks the resources to build the plan on their own but can certainly stop something from being built if it is not in the plan (e.g., via zoning). To be successful, the urban planner and client have to look outward and sell their vision. They cannot bring it about without the other's aid, and they normally lack the resources and authority to do it themselves.

Urban planning also resembles architecting in the spiral or evolutionary development process more than in the waterfall. An urban plan must be continuously adapted as actual conditions change. Road capacity that was adequate at one time may be inadequate at another. The mix of businesses that the community can support may change radically. As actual events unfold, the plan must adapt and be resold to those who participate in it, or it will be irrelevant.

Another analogy for collaborative systems is in business relationships. A corporation with semi-independent division is a collaborative system if the divisions have separate business lines, individual profit and loss responsibilities, and also collaborate to make a greater whole. Now consider the problem of a postmerger company. Before the merger, the components (the companies who are merging) were probably centrally run. After the merger, the components may retain significant independence but be part of a greater whole. Now if they are to jointly create something greater, they must go through a collaborative system instead of their traditional arrangement. If the executives do not recognize this and adapt, it is likely to fail. A franchise that grants its franchisees significant independence is also like a collaborative system. It is made up of independently owned and operated elements, which combine to be something greater than they would achieve individually.

COLLABORATIVE SYSTEM HEURISTICS

As with builder-architecting, manufacturing, sociotechnical, and software-intensive systems, collaborative systems have their own heuristics. The heuristics discussed here have all been given previously, either in this book or its predecessor. But saying that they have been given previously does not mean that they have been explored for their unique applications in collaborative systems. For most people, heuristics do not stand alone as some sort of distilled wisdom. They function mainly as guides or "outline headings" to relevant experience. What is different here is their application—or the experience with specific respect to collaborative systems that generated the heuristic. Looking at how heuristics are applied to different domains gives a greater appreciation for their use and applicability in all domains.

Stable Intermediate Forms

The heuristic on stable intermediate forms is given originally as:

Complex systems will develop and evolve within an overall architecture much more rapidly if there are stable intermediate forms than if there are not.

The original source of this heuristic is the notion of self-support during construction. It is good practice in constructing a building or bridge to have a structure that is self-supporting during construction rather than requiring extensive scaffolding or other weight-bearing elements that are later removed. The idea generalizes to other systems where it is important to design them to be self-supporting before they reach the final configuration. In the broader context, "self-supporting" can be interpreted in many ways beyond physical self-support. For example, we can think of economic and political notions of "self-support."

Stability in the more general context means that intermediate forms should be technically, economically, and politically self-supporting. Technical stability means that the system has all of the elements required to operate on its own to fulfill useful purposes. Economic stability means that the system generates and captures revenue streams adequate to maintain its operation. Moreover, it should be in the economic interests of each participant to continue to operate rather than disengage. Political stability can be stated as the system has a politically decisive constituency supporting its continued operation, a subject we return to in Chapter 13. In collaborative systems, it cannot be assumed that all participants will continue to collaborate. The system will evolve based on continuous self-assessments of the desirability for collaboration by the participants:

- As noted above, integrated air defense systems are subject to unexpected and violent "reconfiguration" in typical use. As a result, they are designed with numerous fall-back modes, down to the anti-aircraft gunner working on his own with a pair of binoculars. Air defense systems built from weapon systems with no organic sensing and targeting capability have frequently failed in combat when the network within which they operate has come under attack.
- The Internet allows components nodes to attach and detach at will. Routing protocols adapt their paths as links appear and disappear. The protocol encapsulation mechanisms of IP allow an undetermined number of application layer protocols to simultaneously coexist.

Policy Triage

This heuristic gives guidance in selecting components and in setting priorities and allocating resources in development. It is given originally as:

The triage**:** Let the dying die. Ignore those who will recover on their own. And treat only those who would die without help.

Triage can apply to any system, but it especially applies to collaborative systems. Part of the scope of a collaborative system is deciding what *not* to control. Attempting to overcontrol will fail for lack of authority. Undercontrol will eliminate the system nature of the integrated whole. A good choice enhances the desired collaboration.

- The Motion Picture Experts Group (MPEG), when forming their original standard from video compression, chose to standardize only the information needed to decompress a digital video stream (Sweet 1997, Chiariglione 1998). The standard defines the format of the data stream and the operations required to reconstruct the stream of moving picture frames. However, the compression process is deliberately left undefined. By standardizing decompression, the usefulness of the standard for interoperability was assured. By not standardizing compression, the standard leaves open a broad area for the firms collaborating on the standard to continue to compete. Of course, to be decompression standard compliant the trade space of compression is hugely reduced, only certain approaches are possible. But the remaining trade space is still quite large and contains great latitude for creativity, intellectual property, and specialization to media types. Interoperability increases the size of the market, a benefit to the whole collaborative group, while retaining a space for competition eliminates a reason to not collaborate with the group. Broad collaboration was essential both to ensure a large market and to ensure that the requisite intellectual property would be offered for license by the participants.

Leverage at the Interfaces

Two heuristics, here combined, discuss the power of the interfaces:

> The greatest leverage in system architecting is at the interfaces. The greatest dangers are also at the interfaces.

When the components of a system are highly independent, operationally, and managerially, the architecture of the system is the interface. The architect is trying to create emergent capability. The emergent capability is the whole point of the system. But the architect may only be able to influence the interfaces among the nearly independent parts. The components are outside the scope and control of an architect of the whole.

- The Internet oversight bodies concern themselves almost exclusively with interface standards. Neither physical interconnections nor applications above the network protocol layers are standardized. Both areas are the subject of standards but not the standards process of the IETF.

One consequence is attention to different elements than in a conventional system development. For example, in a collaborative system, issues like life-cycle costs are of low importance. The components are developed collaboratively by the participants, who make choices to do so independently of any central oversight body. The design team as a whole, cannot choose to minimize life-cycle cost, nor should they, because the decisions that determine costs are outside their scope. The central design team can choose interface standards and can use them to maximize the opportunities for participants to find individually beneficial investment strategies.

Ensuring Cooperation

> If a system requires voluntary collaboration, the mechanism and incentives for that collaboration must be designed in.

In a collaborative system, the components actively choose to participate or not. Like a free market, the resulting system is the Web of individual decisions by the participants. Thus, an economists' argument that the costs and benefits of collaboration should be superior to the costs and benefits of independence for each participant individually should apply. As an example, the Internet maintains this condition because the cost of collaboration is relatively low (using compliant equipment and following addressing rules), and the benefits are high (access to the backbone networks). Similarly, in MPEG video standards, compliance costs can be made low if intellectual property is pooled, and the benefits are high if the targeted market is larger than the participants could achieve with proprietary products. Without the ability to retain a competitive space in the market (through differentiation on compression in the case of MPEG), the balance might have been different. Alternatively, the cost of noncompliance can be made high, though this method is less used.

An alternative means of ensuring collaboration is to produce a situation in which each participant's well-being is partially dependent on the well-being of the other participants. This joint utility approach is known, theoretically, to produce consistent behavior in groups. A number of social mechanisms can be thought of as using this principle. For example, strong social indoctrination in military training ties the individual to the group and serves as a coordinating operational mechanism in integrated air defense.

Another way of looking at this heuristic is through the metaphor of the franchise. The heuristic could be rewritten for collaborative systems as follows:

> Consider a collaborative system a franchise. Always ask why the franchisees choose to join, and then choose to remain as members.

VARIATIONS ON THE COLLABORATIVE THEME

The two criteria provide a sharp definition of a collaborative system, but they still leave open many variations. Some collaborative systems are really centrally controlled, but the central authority has decided to devolve authority in the service of system goals. In some collaborative systems a central authority exists, but power is expressed only through collective action. The participants have to mutually decide and act to take the system in a new direction. And, finally, some collaborative systems lack any central authority. They are entirely emergent phenomena.

We call a collaborative system where central authority exists and can act as a closed collaborative system. Closed collaborative systems are those in which the integrated system is built and managed to fulfill specific purposes. It is centrally managed during long-term operations to continue to fulfill those purposes and any new purposes the system owners may wish to address. The component systems maintain an ability to operate independently, but their normal operational mode is subordinated to the centrally managed purpose. For example, most single-service air defense networks are centrally managed to defend a region against enemy systems, although the component systems retain the ability to operate independently and do so when needed under the stress of combat.

Open collaborative systems are distinct from the closed variety in that the central management organization does not have coercive power to run the system. The component systems must, more or less, voluntarily collaborate to fulfill the agreed-upon central purposes. The Internet is an open collaborative system. The IETF works out standards but has no power to enforce them. IETF standards work because the participants choose to implement them without proprietary variations, at least for the most part.

As the Internet becomes more important in daily life, in effect, as it becomes a new utility like electricity or the telephone, it is natural to wonder whether the current arrangement can last. Services on which public safety and welfare depend are regulated. Public safety and welfare, at least in industrial countries, are already dependent on Internet operation. So, will the Internet and its open processes eventually come under regulation? To some extent, in some countries, it already has. In other ways, the movement is toward further decentralization in international bodies. Clearly, the international governing bodies have less control today over the purposes for which the Internet is used than did U.S. authorities when it was being rapidly developed in the 1990s.

Virtual collaborative systems lack both a central management authority and centrally agreed-upon purposes. Large-scale behavior emerges, and may be desirable, but the overall system must rely upon relatively invisible mechanisms to maintain it.

A virtual system may be deliberate or accidental. Some examples are the current form of the World Wide Web and national economies. Both “systems” are distributed physically and managerially. The World Wide Web is even more distributed than the Internet in that no agency ever exerted direct central control, except at the earliest stages. Control has been exerted only through the publication of standards for resource naming, navigation, and document structure. Although, essentially just by social agreement, major decisions about Web architecture are filtered through very few people. Websites choose to obey the standards or not at their own discretion. The system is controlled by the forces that make cooperation and compliance with the core standards desirable. The standards do not evolve in a controlled way; rather, they emerge from the market success of various innovators. Moreover, the purposes the system fulfills are dynamic and change at the whim of the users.

National economies can be thought of as virtual systems. There are conscious attempts to architect these systems, through politics, but the long-term nature is determined by highly distributed, partially invisible mechanisms. The purposes expressed by the system emerge only through the collective actions of the system’s participants.

In addition to the terms above, two other categories are commonly cited in the literature, particularly U.S. Department of Defense: Acknowledged and Directed. These are defined DoD (2008, pp. 5) as:

> **Acknowledged:** Acknowledged SoS have recognized objectives, a designated manager, and resources for the SoS; however, the constituent systems retain their independent ownership, objectives, funding, and development and sustainment approaches. Changes in the systems are based on collaboration between the SoS and the system.

Directed: Directed SoS are those in which the integrated system-of-systems is built and managed to fulfill specific purposes. It is centrally managed during long-term operation to continue to fulfill those purposes as well as any new ones the system owners might wish to address. The component systems maintain an ability to operate independently, but their normal operational mode is subordinated to the central managed purpose.

MISCLASSIFICATION

Two general types of misclassification are possible. One is to incorrectly regard a collaborative system as a conventional system, or the reverse. Another is to misclassify a collaborative system as closed, open, or virtual.

In the first case, system versus collaborative system, consider open-source software. Open-source software is often thought of as synonymous with Linux (or, perhaps more properly, GNU/Linux), a particular open-source operating system. Actually, there is a great deal of open source, "free" software that is not related to Linux in any way. The success of the Linux model has spawned an open-source model of development that is now widely used for other software projects and some nonsoftware projects. Software is usually considered open source if the source code is freely available to a large audience, who can use it, modify it, and further redistribute it under the same open conditions by which they obtained it. Because Linux has been spectacularly successful, many others have tried to emulate the open-source model. The open-source model is built on a few basic principles (Raymond 1999), perhaps heuristics. These include, from Eric Raymond:

1. Designs, and initial implementations, should be carried out by gifted individuals or very small teams.
2. Software products should be released to the maximum possible audience, as quickly as possible.
3. Users should be encouraged to become testers, and even codevelopers, by providing them source code.
4. Code review and debugging can be arbitrarily parallelized, at least if you distribute source code to your reviewers and testers.
5. Incremental delivery of small increments, with a very large user/tester population, leads to very rapid development of high-quality software.

Of course, a side effect of the open-source model is losing the ability to make any significant amount of money distributing software you have written. The open-source movement advocates counter that effective business models may still be built on service and customization, but some participants in the process are accustomed to the profit margins normally had from manufacturing software. A number of companies and groups outside of the Linux community have tried to exploit the success of the Linux model for other classes of products, with mixed results. But as of the time of this writing, there are some success stories.

Some of this can be understood by realizing that open-source software development is a collaborative system. Companies or groups that have open-sourced their software without success typically run into one of two problems that limit collaboration. First, many of the corporate open-source efforts are not fully open. For example, both Apple and Sun Microsystems open-sourced large pieces of strategic software, primarily in the 1990s. But both released software under licenses that significantly restrict usage compared to the licenses in the Linux community. They (Apple and Sun) argued that their license structure was necessary for their corporate survival and can lead to a more practical market for all involved. Their approach was more of a cross between traditional proprietary development and true open-source development. However, successful open-source development is a social phenomenon, and even the perception that it is less attractive or unfair may be sufficient to destroy the desired collaboration. In both cases, they later had to alter their strategy: in Sun's case, more toward full openness, and in Apple's case, backing away from it.

An interesting counter to the hypothesis that the quality of open-source software is due to the breadth of its review (corresponding to Eric Raymond's last point) may simply be wrong. The real reason for the quality may be that Darwinian natural selection is eliminating poor-quality packages—the disappointed companies among them. In a corporation, a manager can usually justify putting maintenance money into a piece of software the company is selling even when the piece is known to be of very low quality. It will usually seem easier, and cheaper, to pay for "one more fix" than to start over and rewrite the bad software from scratch—this time correctly. But in the open-source community, there are no managers who can insist that a programmer maintain a particular piece of code. If the code is badly structured, hard to read, prone to failure, or otherwise unattractive, it will not attract the volunteer labor needed to keep in the major distributions, and it will effectively disappear. If nobody works on the code, it does not get distributed and natural selection has culled it.

For the second case of misclassification, classification within the types of collaborative systems, consider a multiservice integrated battle management system. Military C4I systems are normally thought of as closed collaborative systems. As the levels of integration cross higher and higher administrative boundaries, the ability to centrally control the acquisition and operation of the system lessen. In a multiservice battle management system, there is likely to be much weaker central control across service boundaries than within those boundaries. A mechanism that ensures components will collaborate within a single service's system-of-systems, say a set of command operational procedures, may be insufficient across services.

In general, if a collaborative system is misclassified as closed, the builders and operators will have less control over purpose and operation than they may believe. They may use inappropriate mechanisms for insuring collaboration and may assume cooperative operations across administrative boundaries that will not reliably occur in practice. The designer of a closed collaborative system can require that an element behave in a fashion that is not to its own advantage (at least to an extent). In a closed collaborative system, the central directive mechanisms exist, but in an open collaborative system, the central mechanisms do not have directive authority. In an open collaborative system, it is unlikely that a component can be induced to behave to its own detriment. In an open collaborative system, the central authority lacks real authority and can proceed only through the assembly of voluntary coalitions.

A virtual collaborative system misclassified as open may show very unexpected emergent behaviors. In a virtual collaborative system, neither the purpose nor structure is under direct control, even of a collaborative body. Hence, new purposes and corresponding behaviors may arise at any time. The large-scale distributed applications on the Internet, for example USENET and the World Wide Web, exhibit this. Both were originally intended for exchange of research information in a collaborative environment but are now used for diverse purposes, some undesired and even illegal.

STANDARDS AND COLLABORATIVE SYSTEMS

The development of multicompany standards is a laboratory for collaborative systems. A standard is a framework for establishing some collaborative systems. The standard (e.g., a communication protocol or programming language standard) creates the environment within which independent implementations can coexist and compete.

Example: Telephone standards allow equipment produced by many companies in many countries to operate together in the global telephone network. A call placed in country and traverse switches from different manufacturers and media in different countries with nearly the same capabilities as if the call were within a single country on one company's equipment.

Example: Application programming interface (API) standards allow different implementations of both software infrastructure and applications to coexist. So, operating systems from different vendors can support the same API and allow compliant applications to run on any systems from any of the vendors.

Historically, there has been a well-established process for setting standards. There are recognized national and international bodies with the responsibility to set standards, such as the International Standards Organization (ISO), the American National Standards Institute (ANSI), and so forth. These bodies have a detailed process that has to be followed. The process defines how standards efforts are approved, how working groups operate, how voting is carried out, and how standards are approved. Most of these processes are rigorously democratic (if significantly bureaucratic). The intention is that a standard should reflect the honest consensus of the concerned community and is thus likely to be adopted.

Since 1985, this established process has been run over, at least within the computer field, by Internet, Web, and open-source processes. The IETF, which never votes on a standard in anything like the same sense as ISO or ANSI, developed the TCIP/IP standards which have completely eclipsed the laboriously constructed Open Systems Interconnect (OSI) networking standard. Moreover, the IETF model has spread to a wide variety of networking standards. As another example, in operating systems, the most important standards are either proprietary (from Microsoft, Apple, and others) or defined by open-source groups (Linux and BSD). Again, the traditional standards bodies and their approaches have played only a little role.

Because the landscape is still evolving, it may be premature to conclude what the new rules are. It may be that we are in a time of transition, and that after the computing market settles down we will return to more traditional methods. It may be that when the computer and network infrastructure is recognized as a central part of the public infrastructure (like electricity and telephones), it will be subjected to similar regulation and will respond with similar bureaucratic actions. Or it may be that traditional standards bodies will recognize the principles that have made the Internet efforts so successful and will adapt. Some fusion may prove to be the most valuable yet. In that spirit, we consider what heuristics may be extracted from the Internet experience. These heuristics are important not only to standards efforts, but to collaborative systems as a whole because standards are a special case of collaborative system.

Economists call something a "network good" if it increases in value the more widely it is consumed, see Varian (2014, Chapter 36). So, for example, telephones are network goods. A telephone that does not connect to anybody is not valuable. Two cellular telephone networks that cannot interoperate are much less valuable than if they can interoperate. The central observation is that:

Standards are network goods and must be treated as such.

Standards are network goods because they are useful only to the extent that other people use them. One company's networking standard is of little interest unless other companies support it (unless, perhaps, that company is a monopoly). What this tells standards groups is that achieving large market penetration is critically important. Various practices flow from this realization. The IETF, in contrast to most standards groups, gives its standards away for free. A price of zero encourages wide dissemination. Also, the IETF typically gives away reference implementations with its standards. That is, a proposal rarely becomes a standard unless it has been accompanied by the release of free source code that implements the standard. The free source code may not be the most efficient, may not be fully featured, probably does not have all the extras in interface that a commercial product should have, but it is free and it does provide a reference case against which everybody else can work. The IETF culture is that proponents of an approach are rarely given much credibility unless they are distributing implementations.

The traditional standards organizations protest that they cannot give standards away because the revenue from standard sales is necessary to support their development efforts. But, the IETF has little trouble supporting its efforts. Its working conferences are filled to overflowing and new proposals and working groups are appearing constantly. Standards bodies do not need to make a profit, indeed should not. If they can support effective standards development they are successful, though removing the income of standards sales might require substantial organizational change.

Returning to collaborative systems in general, the example of standards shows the importance of focusing on real collaboration, not the image of it. Commitment to real participation in collaboration is not indicated by voting; it is indicated by taking action that costs something. Free distribution of standards and reference implementations lowers entrance costs. The existence of reference implementations provides clear conformance criteria that can be explicitly tested.

CONCLUSION

Collaborative systems are those that exist only through the positive choices of component operators and managers. These systems have long existed as part of the civil infrastructure of industrial societies but have come into greater prominence as high-technology communication systems have adopted similar models, as centralized systems have been decentralized through deregulation or divestiture, and as formerly independent systems have been loosely integrated into larger wholes. What sets these systems apart is their need for voluntary actions on the part of the participants to create and maintain the whole. This requires that the architect revisit known heuristics for greater emphasis and additional elaboration. Among the heuristics that are particularly important are:

1. **Stable Intermediate Forms:** A collaborative system designer must pay closer attention to the intermediate steps in a planned evolution. The collaborative system will take on intermediate forms dynamically and without direction, as part of its nature.
2. **Policy Triage:** The collaborative system designer will not have coercive control over the system's configuration and evolution. This makes choosing the points at which to influence the design more important.
3. **Leverage at the Interfaces:** A collaborative system is defined by its emergent capabilities, but its architects have influence on its interfaces. The interfaces, whether thought of as the actual physical interconnections or as higher-level service abstractions, are the primary points at which the architect can exert control.
4. **Ensuring Cooperation:** A collaborative system exists because the partially independent elements decide to collaborate. The designer must consider why they will choose to collaborate and foster those reasons in the design.
5. **A collaboration is a network good**; the more of it there is, the better. Minimize entrance costs and provide clear conformance criteria.

EXERCISES

1. The Internet, multimedia video standards (MPEG), and the GSM digital cellular telephone standard are all collaborative systems. All of them also have identifiable architects, a small group of individuals who carried great responsibility for the basic technical structures. Investigate the history of one of these cases and consider how the practices of the collaborative system architect differ from architects of conventional systems.
2. In a collaborative system, the components can all operate on their own whether or not they participate in the overall system. Does this represent

a cost penalty to the overall system? Does it matter? Discuss from the perspective of some of the examples.

3. Collaborative systems in computing and communication usually evolve much more rapidly than those controlled by traditional regulatory bodies, and often more rapidly than those controlled by single companies. Is this necessary? Could regulatory bodies and companies adopt different practices that would make their systems as evolvable as collaborative (e.g., Internet or Linux) while retaining the advantages of the traditional patterns of control?

EXERCISES TO CLOSE PART II

Explore another domain much as builder-architected, sociotechnical, manufacturing, software, and collaborative systems are explored in this part. What are the domain's special characteristics? What more broadly applicable lessons can be learned from it? What general heuristics apply to it? Some suggested, heuristic-domains to explore include the following:

1. Telecommunications in its several forms: point-to-point telephone network systems, broadcast systems (terrestrial and space), and packet-switched data (the Internet).
2. Electric power, which is widely distributed with collaborative control, is subject to complex loading phenomena (with a social component) and is regulated. (Hill, David J., Special Issue on Nonlinear Phenomena in Power Systems: Theory and Practical Implications, *Proceedings of the IEEE*, Vol. 83, Number 11, November, 1995.)
3. Transportation, in both its current form and in the form of proposed intelligent transportation systems.
4. Financial systems, including global trading mechanisms and the operation of regulated economics as a system.
5. Space systems, with their special characteristics of remote operation, high initial capital investment, vulnerability to interference and attack, and their effects on the design and operation of existing earth-borne system performing similar functions.
6. Existing and emerging media systems, including the collection of competing television systems of Internet distribution, private broadcast, public broadcast, cable, satellite, and video recording.

REFERENCES

Barbaresso, J., et al. (2014). *USDOT's Intelligent Transportation Systems (ITS) ITS strategic plan, 2015–2019, United States*. New York: Department of Transportation.

Batra, R. (2013). *The Lunchbox*. India: UTV Motion Pictures (India), Sony Pictures Classics (North America): 105 minutes.

Bertsekas, D. and R. Gallager (2021). *Data Networks*, Nashua, NH: Athena Scientific.

Butler, S., et al. (1996). “Architectural design of a common operating environment.” *IEEE Software* **13**(6): 57–65.

Chiariglione, L. (1998). “Impact of MPEG standards on multimedia industry.” *Proceedings of the IEEE* **86**(6): 1222–1227.

Dahmann, J. S. and K. J. Baldwin (2008). Understanding the current state of US defense systems of systems and the implications for systems engineering. *2008 2nd Annual IEEE Systems Conference,* IEEE, New York.

DeMarco, T. (1995). On systems architecture. *Proceedings of the 1995 Monterey Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Specification-Based Software Architectures*, New York.

DoD. (2008). *Systems Engineering Guide for Systems of Systems*, Washington, DC: Department of Defense.

Girard, G. and J. Lambot (1993). *City of Darkness-Life in Kowloon Walled City*, San Francisco, CA: Watermark Press.

Hays, R., S. Wheelwright, and K. Clark (1988). *Dynamic Manufacturing: Creating the Learning Organization*, New York: Free Press.

IVHS (1992). *Strategic Plan for Intelligent Vehicle-Highway Systems in the United States*, Washington, DC: IVHS American.

Maier, M. W. (1996). *Architecting Principles for Systems-of-Systems. INCOSE International Symposium*, Boston, MA: Wiley Online Library.

Maier, M. W. (1997). “On architecting and intelligent transport systems.” *IEEE Transactions on Aerospace and Electronic Systems* **33**(2): 610–625.

Maier, M. W. (1998). “Architecting principles for systems-of-systems.” *Systems Engineering: The Journal of the International Council on Systems Engineering* **1**(4): 267–284.

Maier, M. W. (2019). “Architecting portfolios of systems.” *Systems Engineering* **22**(4): 335–347.

Peterson, L. L. and B. S. Davie (2007). *Computer Networks: A Systems Approach*, Cambridge, MA: Morgan Kaufmann.

Raymond, E. (1999). “The cathedral and the bazaar.” *Knowledge, Technology & Policy* **12**(3): 23–49.

Sweet, W. (1997). “Chiariglione and the birth of MPEG.” *IEEE Spectrum* **34**(9): 70–77.

Varian, H. R. (2014). *Intermediate Microeconomics with Calculus: A Modern Approach*, New York: WW Norton & Company.

Part III

Introduction

Processes and Models

INTRODUCTION TO PART III

What is the product of an architect? Although it is tempting to regard the building or system as the architect's product, the relationship is necessarily indirect. The system is built by the developer, not the architect. The architect acts to translate between the problem domain concepts of the client and the solution domain concepts of the builder. We've emphasized aspects of this relationship in the preceding chapters. Even in Chapter 3 on builder-architected systems, we made the point that some degree of separation between the architect and those doing the development is useful. At a minimum, the concerns of architect and developer are different. The architect is necessarily customer and sponsor focused, and the developer engineering focused.

A key point of emphasis has been on architectural decisions and how those decisions look in different domains (builder-architected, information systems, etc.). The product of the architect captures those decisions, the decisions that will ultimately shape the value, cost, and risk of the system to eventually be built. The product is thus a fractional design. It is not the complete design, the complete design is produced only when there is a commitment to follow the development process very deeply. One useful concept is that architecture is often a class of design solutions, not a single design solution. The class is defined by the core decisions it shares, but which are distinct from those made in other classes (other architectures).

The architectural part of the design emerges from the problem domain-solution domain translation. Great architects go beyond the role of intermediary to make a visionary combination of technology and purpose that exceeds the expectation of

DOI: 10.1201/9781003483595-16

builder or client. But the system cannot be conceived or built as envisioned unless the architect has a mechanism to communicate the decisions that define that vision and track construction against it. In any system of appreciable complexity, there will be numerous alternatives and tradeoffs, and an outstanding design cannot emerge without a means to create and navigate those alternatives and tradeoffs. The concrete, deliverable products of the architect are models of the system. Those models are developed through architectural processes.

Individual models alone are point-in-time representations of a system. Architects need to see and treat each as a member of one of several progressions. The architect's first models define the system concept. As the concept is found satisfactory and feasible, the models progress to the detailed, technology-specific models of design engineers. The architect's original models come into play again when the system must be certified.

A CIVIL ARCHITECTURE ANALOGY

Once again, civil architecture provides a familiar example of modeling and progression. An architect is retained to ensure that the building is pleasing to the client in all aspects (aesthetically, functionally, and financially). One product of the civil architect is intangible; it is the conceptual vision that the physical building embodies, and that satisfies the client. But the intangible product is not worth much without a series of increasingly detailed tangible products, all models of some aspect of the building. We will follow the same logic in the systems world, albeit with somewhat different choices of models. Table PIII.1 lists some of the models in the civil analogy and their purposes.

The progression of models during the civil design life cycle can be visualized as a steady reduction of abstraction. Early models may be quite abstract. They may convey only the basic floor plan, associated order-of-magnitude budgets, and renderings encompassing only major aesthetic elements. Early models may cover many disparate designs representing optional building structures and styles. As decisions are made, the range of options narrows and the models become more specific. Eventually, the models evolve into construction drawings and itemized budgets and pass into the hands of the builders. As the builders work, models are used to control

TABLE PIII.1
Models and Purposes in Civil Architecture

Model	Purpose
Physical scale model	Convey look and site placement of building to architect, client, and builder
Floor plans	Work with client to ensure building can perform basic functions desired
External renderings	Convey look of building to architect, client, and builder
Budgets, schedules	Ensure building meets client's financial performance objectives, manage builder relationship
Construction blueprints	Communicate design requirements to builder, provide construction acceptance criteria

the construction process and to ensure the integrity of the architectural concept. Even when the building is finished, some of the models will be retained to assist in future project developments and to act as an as-built record for building alterations.

Note that the civil architect models listed in Table PIII.1 lack some familiar to systems work, like explicit functional models, only because they are usually (but not always) unnecessary in civil architecture. In a typical home, one does not need an explicit functional model of a kitchen to understand and assess the functional suitability of that kitchen design. But if the building-of-interest were a 1,000 room hotel and event center, the situation would be very different. Dinner for five and dinner for 5,000 are obviously not the same thing, even if implementation building blocks are superficially similar.

Making architectural decisions and building the models are obviously intertwined but still distinct activities. One could build a fine set of models that embodied terrible decisions, and excellent decisions could be embodied in an incompetently built set of models. The first case will undoubtedly lead to disappointment (or disaster), and the second case very likely will. The only saving grace in the second case is that later implementers might recognize the situation and work to correct it. The focus of this book is on decisions over descriptions, but in this part, we address the issues of modeling and description directly.

Models are generated from some process. We can sit down and creatively start sketching, but for complex systems that will quickly reach a limit. We need certain pieces of the design, we need the pieces to fit together. It is not enough just to identify the models, and their associated decisions; we need to understand how to develop those models and how the models emerge from effective processes. This is what we are trying to accomplish in Part III.

GUIDE TO PART III

Part III is organized around the abstract architecting process model first introduced in Chapter 2, see Figure 2.1. The chapters of Part III extend and make concrete the processes of that model, emphasizing how we use models to both capture and guide the decision-making required at each phase. Part III develops the concrete elements of architectural practice, defines the progress of those elements models of systems, and embeds them in larger development processes. Throughout we ground the discussion in current Model-Based Systems Engineering (MBSE)/Digital Engineering (DE) methods and practices, adapted for the needs of architecting.

This is a significant change from previous versions of this book. Over the lifetime of the book MBSE has extensively developed. Some elements, like multiple views and integrated models, pre-date even the first edition of the book, although they were immature and not standardized at that time. Since that time, there has been extensive standardization and some consensus of notations. Where in the earlier editions, we just surveyed integrated modeling methods, as none were really consensus choices, we can now define some specific processes on specific model notations. While nothing is really an industry standard, ISO/IEC 42010 (ISO 2022) concepts and SysML (Friedenthal et al. 2014) notations are widely enough taught and used to be a common basis for discussion.

We start the chapter with another, brief, case study. Here we describe a hypothetical application of drone technology to support mountain search and rescue (SAR) operations. This application is not really hypothetical; SAR teams now make frequent use of drones in a variety of missions. Our case study is hypothetical in that we structure it in an organized way to provide a domain example for the process, methods, and modeling discussions. Part III uses other case studies introduced earlier, in particular, the DARPA Grand Challenge introduced at the beginning of the book and material drawn from published architecture studies in which the authors have been involved in.

Chapter 8 provides a terminology foundation, with examples, for what is covered in detail in the subsequent chapters. It defines key terminology in architecture description, like view and viewpoint, and defines the most common views used in architecting systems. Because architecture is multidimensional and multidisciplinary, architecture usually requires many partially independent views. The chapter reviews six basic views and major categories of models for each view. It also introduces viewpoint as an organizing abstraction for writing architecture description standards. Because a coherent and integrated product is the ultimate goal, the models chosen must also be designed to integrate with each other. That is, they must define and resolve their interdependencies and form a complete definition of the system to be constructed.

Chapter 9 elaborates on the APM-ASAM architecting process through heuristic guidelines and a modeling progression. This fits with the general concept of design progression—the idea that models, heuristics, evaluation criteria, and many other aspects of the system evolve on parallel tracks from the abstract to the specific and concrete. Put another way, design begins conceptually, moves to solution-neutral functional and physical specifics, then a specific expression as trans-disciplinary elements, and finally as specific disciplinary designs. Progression ties architecting into the more traditional engineering design disciplines. Even when a system is dominated by the concerns of one engineering domain, it will contain a core of problems not amenable to rational, mechanistic solution that are closely associated with reconciling customer or client need and with technical capability. This core is the province of architecting. Architects are not generalists; they are specialists in systems, and their models must refine into the technology-specific models of the domains in which their systems are to be realized.

Chapter 10 develops a detailed set of modeling methods, and guiding heuristics, for expressing the process elements of Chapter 9. Architects may choose to use many different modeling notations, but we provide here the specifics of a SysML (Friedenthal et al. 2014) centric approach. We define SysML-related products relevant for capturing each step and organizing them into a modeling chain that connects from general concepts to discipline-specific models. The chapter also discusses approaches to the views that do not have simple SysML notational representations and challenges in capturing things like trade comparisons over 10's to 1,000's of solution alternatives that are not well represented in current MBSE notations. Given the similarity of SysML to other systems-oriented modeling methods (like Unified Architecture Framework or UAF (Martin and Brookshier 2023)), the approach is easily extended to those other notations.

The result of the modeling is an architecture description (AD). Defining architecture frameworks (standards on architecture descriptions) has been the subject of a great deal of community work for more than two decades. Chapter 11 reviews elements of that work that are important to our architectural practices. First, it returns to the terminology of Chapter 8 with greater formality, using the IEEE 1471 and ISO 42010 standards to define the concept of an architecture framework. Second, it reviews some key concepts in popular frameworks and compares them to the processes of Chapters 9 and 10 to illustrate how the methods of this book can be adapted in situations where a specific framework is required. In general, the architecture framework is not a driver of good practices, and good practices can be used with almost any reasonably well-chosen architecture framework.

REFERENCES

Friedenthal, S., A. Moore, and R. Steiner (2014). *A Practical Guide to SysML: The Systems Modeling Language*, Burlington, MA: Morgan Kaufmann.

ISO (2022). *ISO/IEC/IEEE 42010 Software Systems and Enterprise - Architecture Description: 69*, New York: International Standards Organization.

Martin, J. N. and D. Brookshier (2023). Linking UAF and SysML models: Achieving alignment between enterprise and system architectures. *33rd Annual INCOSE International Symposium,* Honolulu, Hawaii, 24.

Case Study 7

SAR Drone

CONTEXT FOR THE CASE STUDY

The objective of Part III is to outline approaches for carrying out architecture studies, large and small, ultimately capturing and implementing the results in a digital engineering environment. Comprehensive case studies of architecting and engineering large systems are essentially absent from the published literature. A complete case study would be far too large to be published in traditional form (e.g., in a journal). Understanding the case study would require significant domain knowledge and most readers not in the domain would question its applicability. The effort involved in gathering all of the materials, editing to standards of readability for those not involved in the study, and verifying publishability would be enormous. If the goal was further to show implementation in a digital environment, there is the problem of publishing in a tool environment where access may be difficult.

Throughout the book, we have cited published references to case study material where possible. One thing missing has been citations to architecting captured in a digital engineering environment. There certainly have been such projects, but those known to the authors are unpublished. As a partial bridging of the gap we introduce the SAR Drone System case study here that the authors have used in some academic settings. The elements are familiar enough that they do not provide a severe barrier to wide understanding. The nature of the problem is clearly a system problem, and the key elements are mostly related to how one integrates and uses technical capabilities rather than on narrowly technical issues. The case is not wholly hypothetical, many SAR teams are using drones in some of the missions and approaches discussed here. But there are no proprietary and security barriers to having a full description.

SAR DEMANDS IN THE WASATCH MOUNTAINS

The Wasatch mountains, the sub-range of the Rocky Mountains in the United States, run north-south from near the Utah-Idaho border, past the Utah cities of Ogden, Salt Lake City, and Provo before dividing and merging into the complex ranges of southern Utah. The section known as the Wasatch Front adjacent to the Ogden, Salt Lake City, and Provo metropolitan complex is one of the most heavily used mountain areas in the world. Famous for their snow, they hosted the 2002 Olympic Games and will host again in 2034. The winter Wasatch host numerous destination ski resorts and are a major destination for backcountry skiing. In the summer, hikers and campers traverse the whole range. The favorable geology of canyons hosts thousands of rock climbing routes, with some areas accessible 12 months of the year.

The diversity of terrain, the complex sports commonly pursued, proximity to densely populated areas, and environmental complexity (avalanche activity is very high in the winter, and the very steep terrain makes for communication dead zones even very close to populated areas) means the demand for SAR services is quite high. Hikers get lost year-round, skiers are injured in falls or trigger avalanches, campers get sick, and climbers fall or are struck by falling rock. Every county has an established SAR team and specialty teams exist as well.

Unsurprisingly, SAR teams are interested in technology to support their operations. Their concerns are diverse, from improving outcomes (reducing the number of people killed or seriously injured) to lowering the load on typically volunteer organizations. In the following section consider how SAR stakeholders might articulate their concerns and interests around employing drone technology to achieve their objectives. We present the stakeholder discussion in the form of abbreviated conversations not as specific requirements. Hopefully at this point in the book, it should be clear that we expect that stakeholders express themselves as they see fit, not in pre-digested systems terms. It is the architect's challenge to interpret their concerns appropriately and develop the structured models and decisions needed to bring systems to bear to achieve their objectives. Those familiar with the SAR domain will note we have made some simplifications and left off some details. The intent here is to provide an interesting laboratory for exploring architecting concepts, methods, and models and not to run an actual SAR drone study.

SAR STAKEHOLDER/SPONSOR CONCERNS

As background, the SAR team is a volunteer organization working in an assigned geographical region of mountains adjacent to the urban Wasatch Front region. The team has a small facility in the valley with an operations center (communications centric) and a staging area to store and maintain key team assets. These include a few vehicles, including pickup trucks, and specialized rescue gear like stretchers. Team members maintain their own personal gear and are required to be prepared to deploy to the mountains at any time of the year. Conditions can vary from hot summers to winter snows, where snowshoes or skis will be required.

The team partners with other groups and capabilities as appropriate and needed. In winter situations, there is always coordination with the ski patrols at the resorts. If helicopter or aircraft capabilities are needed they can be requested from Sheriff's departments or even from the military (through a chain of authority). The team tries to deal with situations on their own; how many others need to get involved will be a function of circumstances.

TEAM LEADER PERSPECTIVES

Search and rescue pretty much encapsulates our two missions: Search and rescue. There are lots and lots of scenarios for both, varying from pretty cookie-cutter cases that happen frequently to really unique events. While we need to be prepared for anything, I spend most of my time when I'm trying to improve how we operate by looking at the things we do a lot.

Most search missions are lost hiker or lost child cases. We get a call (somebody gets a call, and it gets routed to us) that somebody went on a hike and didn't come back or walked out of camp and didn't come back. We take the information on who and where, make a plan based on location and conditions, do a suitable team call-out, and then send people out to look. We can and do call for overhead search, but it takes authority, and we don't generally do that immediately. Aircraft and helicopters have other jobs, and somebody has to pay for operating time if they are called out.

Rescues are much more complicated mix. In the summer months, we get injured or sick hikers. Definitely injured climbers from taking a fall or rock fall. In winter, accidents at the ski resorts are dealt with by the ski patrol. We will get called on back-country accidents. Of special importance are avalanche incidents because that is what generates fatalities with a much higher probability. Also, the nature of the response to an avalanche often determines fatality or not. Unfortunately, it is extremely difficult to get professionals on site of an avalanche within the 15- to 20-minute survival window unless the avalanche happens at a ski resort, and that pretty much doesn't happen since much of the ski patrol's job is to make sure conditions are controlled.

When I look at new equipment, and especially new technology, I'm looking with a couple of goals in mind. First, we want to improve outcomes, especially saving lives and reducing the severity of injuries. Speed matters more than any other variable we can conceivably control. The sooner someone with the appropriate training and equipment gets to the injured/sick person, the more likely we can mitigate the outcome, and the sooner we can evacuate, the better. The fast way out is by helicopter, and we can do that pretty fast here; the geography is favorable. But we don't pull somebody out via helicopter unless we know we really need to, and that usually means having a trained person on site. Pulling somebody out of a mountain environment by helicopter is risky and involves having trained people at the site to rig for helicopter operations.

Second, I care about the load on the team. Suppose we get the message "Hiker hurt their leg at 10,000 feet on the XYZ trail and needs help." Do they have a minor ankle sprain and can walk out with assistance, or do they have a broken leg and need evacuation? If the latter, unless we can do a helicopter rescue, we need around a 20-person team to do a carry out. If I call out 20 people and send them in and it turns out that it was a sprained ankle and with a decent splint and some poles, they can walk out, that is a big burden on the team. In some places, if they get that call, they'll just send two people up to make contact and find out what's up. Of course, if it turns out to be the broken leg, then the timeline has really been set back. Fortunately for us, we have a lot of volunteers, but it is best not to abuse that when you don't have to, and in some conditions, like deep winter, the more people you send in, the more things can go wrong.

Could drones help us with either of these issues? Seems reasonable that they might. It has been suggested that drones can speed up the process of search and first contact. That seems possible, though exactly how that would work needs to be worked out. Obviously, you can do a visual search by flying a drone around. Knowing exactly where they are can be pretty important. There are lots of places in these mountains where 100 feet matters, if the 100 feet is up so they are one cliff line removed from the trail instead of on the trail. These mountains are popular, and that has to be taken into account. When you see a party, is it the one we're looking for? Parties in trouble frequently move (even if they shouldn't). How are you going to talk to a party you see from the drone? If they had a cell phone in range and with charge we'd already be talking to them. Being able to deliver payloads could mitigate problems, in at least some cases. Getting a lightweight kit to people could mitigate hypothermia and might save lives, or at least make things do downhill a lot slower, giving us more time to get there. First aid supplies would sometimes be useful, though the parties who know how to use them probably have them.

If we are deploying the drone from a truck after we drive into the mountains, then time to get there is always part of the equation, but you don't have to fly very far. To really cut the timeline down you'd have to be able to fly from permanent locations, but then the distances are long and the support tail impact may be considerable.

Third, anything new we bring in I must look at what does it does to the support and training load. We can't use anything that we aren't trained on, don't have available, or haven't maintained. I've only got so many hours I can get out of volunteers. Every time we add something that requires training do I have to leave something out? There are a lot of skill demands. I'd like to have many more people with high-level first aid certification or being Emergency Medical Technician (EMT) certified. I could really use more people with full avalanche training. We could use more climbing and high-angle rescue skills. All of those are complex skills with considerable training courses. Then there is the cost. I'm particularly sensitive to cases where we are offered some new technical thing and it comes with a long maintenance and support cost tail. People will donate things to us, or donate to buy a thing, but getting people to donate to the year-after-year sustainment cost is a very different thing.

I look at the potential use of drones through this lens. How do they change the timeline of my missions? Is there an opportunity to really shorten the timeline, especially for the cases where time really equates to outcome. Second, how does it change the burden on the team to respond to an event? Can it reduce the number of people I need, or does it increase the number? Does it reliably help me distinguish the situations requiring a full call-out from those that can be handled by a small team? Finally, what would it take to support the capability, in money, training, and personnel? How much of a back-end burden does it create? Optimally, it would let me reduce something I'm already supporting.

8 Architecting Models and Processes Concepts

By relieving the mind of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the [human] race.

Alfred North Whitehead

DOING ARCHITECTING: THE INTERACTION OF PROCESS AND MODELS

The previous chapters provided an intellectual framework for what architecting is, how it relates to conceiving and creating systems, and laid out significant cases. We defined the overall concept of what systems architecting is and examined cases of it being carried out, emphasizing valuable heuristics for commonly encountered situations. The emphasis was on how the varying characteristics of architecting problems and the systems being architected (e.g., builder architected and collaborative assemblage) impact how we approach the problem. In Part 3, we make a detailed elaboration of architecting processes and description models. The ultimate goal of Part 3 is to define architecting methods that fit in the broad framework of model-based systems engineering (MBSE).

The interaction of process and description models is an important concept. Processes (the tasks and how we carry them out) and description models are dependent and interactive. A process is pointless if it does not produce something. We only know we've completed a process by having a product. In a design activity what we produce are models of the system (whether of a system to be built or one that already exists). To have a "design" of a system means, concretely, that we have a set of design descriptions. These design descriptions define some aspect of the system of interest, typically via some model representation.

Design descriptions are abstractions of the system itself. Abstractions at various levels can all be useful, albeit at particular times and in particular phases. The overall process of design is to produce a set of design artifacts sufficiently complete and consistent to be used to build the system of interest, whether in a single copy or in a production run. This requires great detail in description late in the process when we are ready to fabricate and assemble. But very different types of detail are needed early when we are deciding whether to go forward or not, or are reconsidering overall decisions about the system's structure.

Processes and methods must be understood within the context of the overall system development process. The natural way to think of systems architecting is at the initiation of system development. At the initiation of a project to build a system, we have to make essential decisions. We presumably have a process in place to make those core early decisions. But that doesn't always happen. Early architecting may have happened partially by accident or as the result of some fiat from sponsors rather than through a disciplined process. It may be necessary to return to those architectural decisions later or at a point of crisis. The discussion in this chapter must be placed within a framework of systems development and architecting scenarios. At the level of the overall system development process, we have:

1. Systems developed and delivered singly or in small production runs. The process begins, focuses on the specific system to be built, builds it, and moves on moves to sustainment of the systems produced.
2. Systems designed, built, and delivered repeatedly in a functional spiral of increasing capability. Most of these are software systems or at least software-intensive, but some also have major hardware elements.
3. Collections of systems, whether a family-of, portfolio-of, or system-of where there are multiple systems involved each with a significant degree of independence in development and production. Here the architecture challenge divides between architecting (selecting the architecture of) the collection versus a single system within the collection.

Architecting naturally should happen when one of these efforts is first initiated, but some architecture projects are after the fact. This may happen because there was no deliberate architecting at the beginning and the perception of a need for architectural understanding has arisen, or because needs have changed and the architecture as it exists is inadequate. From how architecting occurs, we get scenarios of forward, reverse, over-arching, and prospective architecting. These may be purpose-driven, as we identified in the nominal case in Chapter 1, or in some cases driven by speculative technology innovation.

THE CENTERPIECE: ARCHITECTURE DECISIONS

The process of design can be envisioned as a process of sequential decision or stepwise refinement. Choosing functions to include and exclude, choosing physical or logical partitions, and choosing implementation technologies are all decisions. One of the fundamental observations about architecting, that early decisions have disproportionate impact on cost, value, and risk, leads to the conclusion that there are "architectural decisions" and that those architectural decisions are of great importance. This sequential decision process is equivalently a form of stepwise refinement or stepwise movement from an abstract understanding of the system of interest to a concrete and physical understanding.

What is an architectural decision? Most simply put, an architectural decision is one that disproportionately drives the value, cost, or risk of the system of interest. If the result of the decision is a major factor in cost, in value to sponsors, or in risk to develop, then the decision is "architectural." There is no universal catalog of what those decisions are, they depend on the system's context (business or operational) and the dominant implementation technologies. Part of the architect's personal toolkit must be an experience base of what constitutes architectural decisions in the environments where the architect works. While we cannot catalog all possibilities, we offer case studies and heuristics.

In the DARPA Grand Challenge case that opened the book, teams had to decide on the design scope of their vehicles. All had to choose whether to build or buy a vehicle, whether to make the vehicle the subject of extensive design trades or to work with conventional vehicle capabilities and put all effort into automated driving. That choice of vehicle approach necessarily drove subsequent decisions with large cost, value, and risk impacts. Some teams drastically narrowed the focus, working only with lightly modified commercially available vehicles, restricting themselves to off-the-shelf sensors, and putting all their team efforts into algorithms and software. This limited vehicle development cost and risk but concentrated risk in algorithm and software development. The possibility of using mechanical innovation to compensate for failure in algorithm or software space was foreclosed. Other teams took a more "full-court-press" approach, being willing to heavily modify vehicles and develop new sensor systems. Greater design space in the vehicle potentially increased vehicle costs and associated risks but opened up additional pathways to solve specific problems of safely traversing the course.

The full-court-press group gained extensive design flexibility and the ability to customize an approach to the race operational conditions. Those in the commercial-vehicle-only group, on the other hand, gained focus. They had no need to build and maintain teams or facilities to support development activities in the areas they had foreclosed. They also ensured that their solutions were likely to be more generally applicable to many different (and valuable) operational scenarios other than the race challenge itself. The appropriateness of deciding either way ultimately lay in the strategic positions of the teams and what fundamental objectives (fundamental to the teams and the team's sponsor, not the DARPA sponsor) they had.

In the DC-3 case study (Case study #2 before Chapter 3), we emphasized the significance of aircraft size, carrying capacity, and performance margin under single engine failure as distinguishing between Boeing's and Douglas' approach. Here there were key interactions between cost, value, and risk. A larger airplane with more powerful engines but lower relative weight and better aerodynamic performance would clearly cost more and require different risks (of supplier partnerships and incorporated technologies) to be undertaken. On the value side, it delivered a different value proposition, the ability to realize value in a different operational concept rather than a more optimized delivery in the legacy operational context.

The architects of the GPS (Case study 6 before Chapter 7) made very clear architectural decisions during the "Lonely Halls" meeting, decisions that clearly drove cost, value, and risk. They choose to focus on an inherently global, 3-dimensional positioning system able to provide continuous positioning. This created the greatest value for its users but committed the program to deploying a very large minimum viable system, in contrast to some of the other, rejected, alternatives. The architects chose the alternative with atomic clocks on the satellites and an all-digital signal structure, something that was very technically aggressive (risky) at the time of the choice but made for a more robust system where the cost of receivers could be aggressively driven down as Moore's Law drove down the per transistor cost of chips on a long-term exponential curve.

The readers are invited to review the other case studies, and their own experience, to identify the architectural decisions, those that drove cost, value, and risk.

The practice of architecting is an interplay between decisions and descriptions. Much of the next few chapters will be architecture descriptions, specifically writing models that describe or define the architecture of a system. Picking up the fundamental concept here, the concept that architecture is a set of decisions, it follows that the role of architecture descriptions and the models that make them up is to represent the architecture decisions. As we work through the range of architecture modeling methods, the notations, and the architecture description frameworks, it is important to keep the end in mind. The end is to make and document the architecture decisions, and to make those architecture decisions good decisions, meaning to make architecture decisions ultimately leading to a successful system of interest.

As an obvious example, we'll see that an architecture description essentially always contains a physical view, a representation of the physical components of the system of interest and their interfaces to things outside the system boundary. It is extremely difficult to come up with a case for a system of interest where its physical components are not important to cost, value, or risk (taking software to be a physical component). If the reader can come up such an example it is likely it proves the rule by how odd it is. Likewise, architecture descriptions usually contain some kind of functional view and some kind of cost view as repositories of the models defining those respective properties.

When making architecture decisions the rationale for those decisions should be recorded along with the decisions. The decision rationale may be more enduring than the decisions themselves. Circumstances may change, the process of design may result in learning that changes our assessments, but the rationale for building the system of interest, and thus for making the architecture decisions, is likely more enduring. This is just a recognition that problems are typically more enduring than solutions. Essential problems in domains, whether business or military, typically endure much longer than the particular system solutions we develop to address those problems. Deep knowledge of the problem domain, and how that translates into decision rationale, is often one of the architect's most valuable assets.

DEVELOPMENT SCENARIOS AND ARCHITECTING

If architecture is centered on a set of architecture decisions, when are those decisions made? The answer above was "early in the development process," but it isn't that simple. Not all development processes have a single, easily recognized starting point. We do systems architecting work in various scenarios. We won't try to systematically list every relevant possibility, but the most common, that link to the concept of architecture orientation (first mentioned in Chapter 2), are important.

Purpose-Driven, Forward Architecting

In this case, the architect has the big three available:

1. A recognized problem, seen as inadequately addressed. There is substantial value to be realized from providing a system solution.
2. A sponsor, a person or organization with resources to spend against the problem and authority to do so. The sponsor's resources are at least roughly aligned with what would be needed to build a relevant solution.
3. Users who are prepared to use any provided system to realize the value and who are engaged with the sponsor.

This scenario is most similar to the civil architecting metaphor. From an architecting perspective, it is the least complex, even if the resulting system might be enormously complex. The complexity of the resulting system is solution-domain complexity; the simplicity here is of the problem domain. That said, there may be factors beyond how the development problem is presented, making the problem-domain complex. For example, the problem presented may be interconnected with others with many stakeholders over whom the sponsor has little influence, but who can impact any development efforts. It may be very difficult (and uncertain) to build any system that realizes much of the value available, and solutions may involve very low-maturity technology. Escaping from solution-domain complexity might require problem-domain tradeoffs that the sponsor is unwilling to make.

Technology-Driven, Forward Architecting

The key difference in this case is the lack of present and available users. One essential leg of the big three is missing. In this case, the sponsor often no longer has anything approaching the resources necessary to realize a solution. The canonical case is where someone has technology and perceives a new application, but users are not engaged (and won't be engaged until there is some system to engage with), and resources may be limited. With limited resources, the architecting effort can not just transition to building the system, it has to transition to selling the system, really selling the program to build the system.

Most architecting techniques are the same as in the previous, but there will be significantly different heuristics and practices. Because the user base in the previous scenario is available and engaged, the architect does not have to speculate on what is acceptable or useful to users, they can be asked directly. In the technology-driven case the architecture team commonly proceeds with a concept of what is wanted, but that concept may or may not be well received by users when it is eventually built. Even if it is well received, the user base may find dramatically different things to do in the system rather than what was intended. This can lead to the well-known phenomenon where the "fast follower" beats out the technology innovator by using the innovator's product as market research.

Reverse Architecting

Reverse architecting happens when a system (or systems) has already been built, but sponsors believe they are architecturally flawed and want to do something about it. Common examples include:

- The original system fails to evolve well or has been evolved repeatedly to the point it now looks like a hard-to-sustain amalgam rather than something that effectively accommodates the needed evolution.
- A group of systems have been interconnected and do valuable things jointly, but the interconnections lack a coherent set of structural decisions (an "architecture") that makes interoperation and evolution reasonably easy. The sponsor believes this can be fixed.
- There is a need to replace a successful system but there is no consensus on what needs to remain intact in the replacement to build on the current value.

Reverse architecting depends on both gaining a solid understanding of the current system, specifically its defining decisions, and determining why it does not realize the value the sponsor perceives could be realized. Usually to start you build an "as-is" architecture model of the existing system. This led to a common failure mode:

> When studying the architecture of an existing complex system it is always possible for the as-is study to expand to consume all available budget and schedule.

There are always more questions that can be asked, and answered, about the existing system or systems. There are always more perspectives and more details. Some teams may find it more comfortable to keep studying something well-defined, even long past the point of diminishing returns, rather than wade into the more uncertain work of trying to find out how to fix the issues. Reverse architecting teams need to exercise hard discipline in developing just enough information on the as-is system to enable definition of what the future should be.

Interlude: Mapping the Case Studies

To make this more meaningful, and as a segue to collections of systems, consider how this applies to some of the case studies. The Global Positioning System showed a history that went through multiple architecting scenarios. The first predecessor system, Transit, was purpose-driven and forward-architected, though motivated by a technology insight. The user base, the ballistic missile submarine fleet, had a specific problem: how to get daily position updates to correct inertial navigation drift. The technology solution (in the late 1950s) was unclear. When the possibility of satellite-based positioning was discovered from Sputnik observations DARPA was able to form a program to demonstrate it in the maritime position update context. This quickly evolved, once demonstrated, to a program office that could architect the Transit system in purpose-driven fashion with direct involvement of its primary user base.

Moving to the 1960s, the new efforts (Project 621B and Timation) were mostly technology-driven but still forward-architected. The scenario was transitioned into much more technology-driven than purpose-driven. There were sponsors for the component programs, but none of them had a set of users with sufficient influence on resource allocation to form a plausible program. The success of the program, as outlined in the "Lonely Halls Meeting," required finding the sponsor, and the nascent user base, to gain the resources necessary to build the initial system. GPS then became an outstanding illustration of how the most successful technology-driven system become so successful by changing user operational concepts. One of the slogans of the GPS program, "Drop five bombs in the same hole," was fully achieved. But the enormous success of GPS came from its use in so many other applications, from day-to-day civilian navigation to time synchronization of wide area networks.

The DC-3 case might be thought of as technology-enabled, purpose-driven. New technology in aerodynamics, structures, and engines enabled aircraft performance well above what was previously possible. The question was how to exploit this increased performance. The Boeing team built the 247, directly addressing the known need and superior performance on customer-defined airmail routes. Douglas, with encouragement from his customers, aimed past the point of optimality for current operations and achieved greater success when that point proved to be profitable without the support of air mail-carrying contracts. A key to this was understanding the real, underlying objective of safe operations did not require three engines, but did require single-engine-out survivability challenging operational situations (over western US mountain ranges).

Portfolio and Family Architecting (Forward, Reverse, and Sustained)

One thing these case studies suggest is what we aren't just looking at architecting single systems but groups of systems. Aside from the basic scenarios for architecting a single system, we may look at architecture spanning multiple systems. In looking at groups of systems it is helpful to identify group categories that share common challenges in architecting and development. We've found three categories as particularly useful: Families-of-Systems, Portfolios-of-Systems, and Systems-of-Systems (or "Collaborative Systems") (Maier 1998, 2019).

Family-of-Systems: A group of systems related by shared design or production features. Shared features are usually chosen to gain economy-of-scale advantages in production cost while retaining the ability to specialize in smaller customer niches.

Portfolio-of-Systems: A group of systems related by common management and resource allocation, usually managed against goals for one or more missions.

System-of-Systems (Collaborative Systems in Chapter 7): A group of systems such that,

1. Operate together to produce capabilities not possessed by any individual member.
2. The individual systems have sufficient operational independence to provide valued operational capability when not part of the group.
3. The individual systems are managed, at least in part, for their own purposes rather than the purposes of the group (integration into the group is at least partially voluntary and contingent).

The task of architecting the group is not the task of architecting all members of the group. In the sense described here, to architect the group means to make architectural decisions that pertain to the group *as a whole rather* than specifically to one member of the group. This typically means decisions on shared capabilities, features, or design elements. A major challenge in architecting at the family, portfolio, or SoS level is choosing which aspects to deliberately architect and which to defer to those who will architect individual members of the group.

Thought Exercise: A Family of Printers

As a thought exercise, consider the whole family of printers made by a company. What is the architecture of an individual printer versus what is the architecture of the printer product line (the family-of-systems)? Following the heuristic of architecture decisions being those things that define most of the value cost, and risk, we can identify differences in the architecture of individual printers versus the whole product line.

At the level of individual printers, value, cost, and risk will be driven by:

- The customer facing performance and features, like page rate, resolution, color, and integration of scanning. Of particular importance will be how those features compare to competitive offerings.
- Specific parts make up the product, especially very high-cost parts like the core print engine.
- Whether or not any low-maturity technologies or parts are integrated into the design.

At the level of the whole printer product line, there is a distinction between choosing the things that are shared and choosing the structure of the product lines.

- Part of the family-of-systems architecture is what is shared across all of, or part of, the product line. For example, are specific printing technologies used in multiple offerings to gain economies of scale in production? Are external interfaces, at the physical and software level, deliberately chosen in common so different members of the family can be seamlessly connected to varying customer networks?
- The other part of the architecture of the family is choosing its overall structure, how is it divided into clusters of products with sharing within but not without the cluster? Does the company choose to address certain market segments, say high-volume printing service providers, with different products than those targeted at other segments, say low-volume office use? This is an obvious "yes," but why? Presumably the product segments are chosen based on a combination of problem-domain clusters (user groups with similar needs to each other but distinct from other user groups) and solution-domain clusters (technology-driven recognition that certain performance/feature points are effectively addressed by similar or distinct technology and design choices).

R&D, Product Pipelines, and Architecting

An extension of the portfolio architecting scenario is the research and development (R&D) or product-pipeline scenario. This is a portfolio, a group of systems related by common management, but specifically a group related in time and progression. The idea is that at any given time we want systems in multiple phases of development. We want a continuously changing group of early phase concepts being explored. A subset of those are promoted to more advanced development, while others that were early phase studies are abandoned. Some of those promoted to more advanced development are further promoted to full-scale development, while others are found unpromising and do not move on.

Various R&D and product pipelines are present in many industries. Well-known examples are in pharmaceuticals, electronics, and military equipment. Thinking through the value, cost, and risk defining choices in structuring an R&D pipeline is a good exercise in architecting a non-physical system, treating the pipeline itself as the system, a system that produces physical systems.

- Where does the pipeline start and stop? At what level does something enter the pipeline and how does it exit? In some pipelines, the entry point may be fundamental research in relevant scientific areas (some pharmaceutical pipelines include basic research). In others, the entry point must be an identified problem and solution concept (the US Defense Advanced Research Projects Agency (DARPA) is like this). Is the exit of the pipeline part of the organization's operational delivery, or is the intent to spin-out successful concepts to other organizations?
- What are the entry and exit criteria? How is funding managed? If funding for a phase of the pipeline is fixed then any entry must correspond to an exit, phase-by-phase. If return-on-investment criteria are used for entry and exit then funding must be variable.
- What is the risk profile, individually by project and collectively? An advantage of managing a portfolio is that the risk of the whole can exploit averaging of the risks of the elements, assuming the risks of the elements are uncorrelated and the objectives of the whole are met by average portfolio performance and not the worst case. Effectively exploiting investments in high-risk elements may require a willingness to let those risks play-out, which may require a tolerance for substantial cost risk, which may itself conflict with other portfolio structures like a fixed budget by phase. These architecture decisions may be complex and interlinked.

THE SYSTEM DEVELOPMENT LIFECYCLE AND ARCHITECTING

The lifecycle of a system can be simple or quite complex. It may be possible to locate architectural decision making very specifically in time, or those decisions may be spread over many phases or be obscure. We previously introduced key ideas of how architecture decisions are made primarily early in the lifecycle, while recognizing that lifecycles may themselves cycle, perhaps many times as in spiral and incremental models. As discussed above, there are important scenarios where there are multiple systems and multiple layers of architecture decisions, say those at the level of a family-of-systems as a whole and then again for each member of the family.

Whether the overall process is primarily a once-through waterfall, or a spiral aiming at objective systems, or even an agile process where the exact final system is left to be decided by the process of development, all system development processes that invest significant resources contain some essential elements.

- There has to be an overall concept of what system to build. We always leave some detailed aspects to downstream design, but we also always have some overall concept, or there is little likelihood we could ever have gotten a commitment of resources to proceed. There is usually effort put into a solution-independent description of the problem (and certainly should be), but it is always colored by some concept of what the resulting system will be.
- The concept has to be refined into a design that can be built. The design is dominated by physical, functional, and informational perspectives. The physical corresponds to the actual physical entities that make up the system of interest. The function must be expressed in the operation of physical, software, and human procedural elements. And the informational, typically implemented in software, in how the system gathers, organizes, and persistently retains data.
- The design tree needs to end up specifying of individual elements that can be built. This eventually becomes a bill of materials, construction drawings, fabrication instructions, or individual software programming directives.
- We have to assemble the elements, verifying construction as we go. As the system is more fully assembled, there are test processes, including feedback loops, to fix what does not work as expected. Eventually, we have to certify for use, meaning we have sufficient confidence in what we have built that we can use it for the intended purpose. Certification can be anywhere from very informal to extremely formal and intensive, depending on the resources on the line and the consequences of failure.
- If we build significantly more than one copy, there will have to be a defined manufacturing process. How detailed the process is, the extent to which it is a system itself, and the level of certification it gets will depend on the complexity and additional requirements of that manufacturing process.

The ISO/IEC/IEEE 42020 standard on architecture processes defined six processes, of which numbers 3, 4, and 5 are considered "core." See ISO (2018), particularly Table 8.1 and linked sections.

1. **Architecture Governance:** This concerns how an enterprise governs its collection of architectures to gain mutual alignment among them and with the enterprise's goals, policies, and strategies.
2. **Architecture Management:** The concerns implementing the architecture governance directives.
3. **Architecture Conceptualization:** This, the first of the "core processes," is very much the primary subject of this book. The formal statement in the standard is "Characterize the problem space and determine suitable

solutions that address stakeholder concerns, achieve architecture objectives, and meet relevant requirements."

4. **Architecture Evaluation:** This is where we assess how well an architecture does what it is supposed to do, meet objectives and requirements and address stakeholder concerns.
5. **Architecture Description:** This is the writing of an architecture description document (which may be a collection of digital models) to meet the intended uses. Depending on the lifecycle model, the intended use might be to support building the system, or to provide guidance in a system-of-system integration, or constraint a portfolio, or something else, depending on the lifecycle objectives and stakeholder concerns.
6. **Architecture Enablement:** This is where we are building and maintaining the capabilities we need to do the other five processes.

Three points are of particular importance. First, the core concern of this book is what the ISO standard calls "architecture conceptualization." The ASAM portion of the APM-ASAM is all about characterizing the problem space (including determining the objectives) and generating suitable solutions. In Chapter 9, we focus on heuristics for each aspect of problem space characterization and solution generation. Later in this chapter, we review a generic multi-view model for representing problem space and solution spaces and how they are jointly evaluated. The process of design, from architecting to detailed disciplinary engineering, is represented by a series of models. The models are progressively refined and reduced in abstraction, as the process proceeds. Each step of refinement is marked by making decisions that reduce the space of possible system implementations. At some point, we pass out of architecture conceptualization and are engaged in detailed design. Finding that border will be dependent on the larger lifecycle process. In some cases, it will be formally gated, as with design reviews and deliberate stepping between phases. In some contexts, it will be better to see the design process as a learning process where we cycle based on our increased knowledge of the program and solution domains, gained by the activities of design. Architecture decisions will be heavily weighted to the front, but can be continuously encountered as we cycle, learning more about the problem space (with user needs are the most uncertain) or solution alternatives (when solution risks are foremost) through those cycles.

Second, the core processes depend not just on knowing what the system of interest is about but also on the purposes of the architecture and any architecture description generated. The purpose of the architecture depends on the overall lifecycle model. We reviewed cases in the discussion of the orientation phase of APM-ASAM in Chapter 2, and will return in more depth in Chapter 9, for the purpose of the architecture. If the sponsors can fund the system of interest on their own the architecture purpose is to drive the decision to develop or not, and then drive the detailed development. If they cannot fund the development, then the primary purpose of the architecture is likely to be to raise funds or sell the development program. This is distinct from driving the program. If the scope is multiple systems only under partial control of the sponsors, then the purpose is likely be to influence those other systems, identifying and focusing on the smallest set of things needed to influence to accomplish the mission purposes.

Third, we recognize that different scenarios lead to very different levels of formality and process "weight." Long-lived, high-consequence systems with no "do-over" options require deep up-front work and rigorous development processes. The development of other systems would be choked to death by attempts at up-front, rigorous development. When the uncertainty of user response is high, consequences of mistakes are low, and learning rate is essential, then it is better to repeatedly prototype and create a virtuous learning cycle. This is itself an architectural choice, one embedded in both the program and system structure. Recall the discussion of the DARPA Grand Challenge case study. At the time of the case study, the teams did not, and could not, know what set of features would yield a successful run on race day. It had never been done successfully, so the required combination could not be known empirically, and there was no reliable theory to predict. The way to maximize the likelihood of success was to enable experimentation, gathering as much useful data as possible. This meant a system and program architecture that best enabled early, frequent, and safe experimentation while gathering enough diagnostic data to learn fast.

ROLES, VIEWS, AND MODELS

This chapter opened with an emphasis on architectural decisions. We extend that emphasis her to models and the application of MBSE to architecting. While doing architecting, the primary purpose of modeling is to support making architectural decisions and capture the results of those decisions. A complete set of MBSE models will extend well beyond the architecture boundary, as it should. MBSE is a lifecycle concept in which linked models extend from conception to manufacturing and support. Some common usage calls a large fraction of the model set, the fraction with system models, "architecture models." In our terms, most of the content will be on the engineering side of the three-column concept (Table P.1 in the Preface). Architecture models should be reserved for those models directly addressing the value, cost, and risk-driving aspects. In practice, the transition will likely be invisible and driven not by the use of specific models but by how they are used and what information they cover.

Models should be the primary means of communication with clients, builders, and users; models are the language of the architect. Models enable, guide, and help assess the construction of systems as they are progressively developed and refined. After the system is built, models, from simulators to operating manuals, can help describe and diagnose its operation. This is the lifecycle vision of MBSE.

To be able to express system objectives and manage essential system design elements, the architect should be fluent, or at least conversant, with all the languages spoken in the process of system development. The language set extends from clearly architectural considerations at the beginning (e.g., objectives and alternative concepts) to disciplinary design, usually well outside the architecture boundary and firmly within engineering. By role, these languages are those of system specifiers, designers, manufacturers, certifiers, distributors, and users.

Three groups of models are primarily architectural and will be architecture concerns in essentially all scenarios:

1. Models of the value of the system, primarily to users but also to sponsors and possibly others.
2. Models of overall physical structure. Specifically, we will always need models of physical structure and content that enable cost estimation.
3. Models of the critical acceptance requirements for the sponsor.

The first drives why we want the system at all and becomes the source for the acquisition requirements, assuming we choose an acquisition model driven by pass/fail requirements. The last is a subset of the entirety of the requirements. In the middle is a subset of the complete, detailed system design. Because the architect is responsible for total system feasibility, the critical portions may include highly detailed models of components on which success depends but only abstract, top-level models of other components.

Models can be classified by their role or by their content. The role is important in relating models to the tasks and responsibilities not only of architects but of many others in the development process. Of special importance to architects are modeling methods that tie together otherwise separate models into a consistent whole.

Roles of Models

Among the systems architecting roles where we seek modeling support are:

1. Communication with client, users, and builders.
2. Maintenance of system integrity through coordination of design activities.
3. Design assistance by providing templates, and organizing and recording decisions.
4. Exploration and manipulation of solution parameters and characteristics; guiding and recording of aggregation and decomposition of system functions, components, and objects.
5. Performance prediction; identification of critical system elements.
6. Provision of acceptance criteria for certification for use.

These roles are not independent; each relates to the other. But the foremost is to communicate. The architect discusses the system with the client, the users (if different), the builders, and possibly many other interest groups. Models of the system are the medium of all such communication. After all, the system will not come into being for some time to come. The models used for communication become documentation of decisions and designs and thus vehicles for maintaining design integrity. While models for engineering likewise serve a communication role, what distinguishes the communication role in architecture is the combination of "up" and "down" communication. By "up" we mean communication with users and sponsors. By "down," we mean communication to the next level of implementers. Engineering modeling typically has a strong downward emphasis, emphasizing communication with each lower-level developer.

Communication with the client has two goals. First, the architect must determine the client's objectives and constraints. Second, the architect must insure that the system to be built reflects the value judgments of the client, where perfect fulfillment of all objectives is impossible. The first goal requires eliciting information on objectives and constraints and casting it into forms useful for system design. The second requires that the client perceive how the system will operate (objectives and constraints) and that the client can have confidence in the progress of design and construction. In both cases, models must be clear and understandable to the client, expressible in the client's own terminology. It is desirable that the models also be expressive in the builder's terms, but because client expressiveness must take priority, proper restatement from client to builder language usually falls to the architect.

User communication is similar to client communication. It requires the elicitation of needs and the comparison of possible systems to meet those needs. When the client is the user, this process is simplified. When the client and the users are different (as discussed in Chapter 5 on sociotechnical systems), their needs and constraints may conflict. The architect is in a position to attempt to reconcile these conflicts.

In two-way communication with the builder, the architect seeks to insure that the system will be built as conceived and that system integrity is maintained. In addition, the architect must learn from the builder those technical constraints and opportunities that are crucial in insuring a feasible and satisfactory design. Models that connect the client and the builder are particularly helpful in closing the iterations from builder technical capability to client objectives.

One influence of the choice of a model set is the nature of its associated "language" for describing systems. Given a particular model set and language, it will be easy to describe some types of systems and awkward to describe others, just as natural languages are not equally expressive of all human concepts. The most serious risk in the choice is that of being blind to important alternate perspectives due to long familiarity (and often success) with models, languages, and systems of a particular type.

Models, Viewpoints, and Views

Chapters 8 through 10 discuss this book's approach to modeling in systems architecting. Chapter 11 looks outward to the community to review other important approaches and draw contrasts. Unfortunately, there is a lot of variation in the usage of important terms. We focus on three terms that are important in setting up a modeling framework: *model*, *view*, and *viewpoint*. We use the definitions of model, view, and viewpoint taken from the Institute of Electrical and Electronics Engineers (IEEE) standards. These terms have undergone a change from the earlier usage in ANSI/IEEE-1471 to the more current ISO/IEC/42010. We have a preference, at least for discussing the concepts, for the original terms used in ANSI/IEEE-1471:

Model: An approximation, representation, or idealization of selected aspects of the structure, behavior, operation, or other characteristics of a real-world process, concept, or system (IEEE 610.12-1990).

View: A representation of a system from the perspective of related concerns or issues (ANSI/IEEE 1471-2000).

Viewpoint: A template, pattern, or specification for constructing a view (ANSI/IEEE 1471-2000).

These terms have been elaborated in later standards, such as the ISO/IEC/IEEE 420X0 series (ISO 2018, 2022). The differences between standards are largely technical, the older versions above will serve here. A model is just a representation of something; in our case, it is some aspect of the architecture of a system. The modeling languages of interest have a vocabulary and a grammar. The words are the parts of a model; the grammar defines how the words can be linked. Beyond that, a modeling language has to have a method of interpretation; the models produced have to mean something, typically within some domain. For example, in a block diagramming method, the words correspond to the types of blocks and lines, and the grammar are the allowed patterns by which they can be connected. The method also has to define some correspondence between the blocks, lines, and connections to things in the world. A physical method will correspond to physically identifiable things. A functional diagramming technique corresponds to more abstract entities—the functions that the system does.

A view is just a collection of models that share the property that they are relevant to grouped concerns of a system stakeholder. For example, a functional view collects the models that represent system functions. An objectives view collects the models that define the objectives to be met by building the system. The idea of view is needed because complex systems tend to have complex models and require a higher-level organizing element.

View is inspired by the familiar idea of architectural views. An architect produces elevations, floor plans, and other representations that show the system from a particular perspective. The idea of view here generalizes this when physical structure is no longer primary.

Viewpoint is an abstraction of view across many systems. A viewpoint consists of the concerns addressed (belonging to stakeholders) and the methods used (notations and heuristics). A good architecture description should either contain the definitions of its viewpoints or reuse standard definitions. The definition of viewpoints is important for model understanding, but may be implicitly understood by the time. The concept in detail is important primarily for defining standards for architecture description, so we defer full exploration until later.

These concepts are depicted schematically in Figure 8.1.

Classification of Models by View

A view describes a system with respect to some set of attributes or concerns. The set of views chosen to describe a system is variable. A good set of views should be complete (cover all concerns of the architect's stakeholders) and mostly independent (capture different pieces of information). Table 8.1 lists the set of views chosen here as most important to architecting. A system can be "projected" into any view, possibly in several ways. The projection into views and the collection of models by views are shown schematically in Figure 8.2. Each system has a purpose (achieves some objectives), has some behavior (abstracted from implementation), has a physical form, performs at some level, retains data, and is constructed and run in some manner (is managed). Views are composed of models. Not all views are equally important to system developmental success, and the set will not remain constant

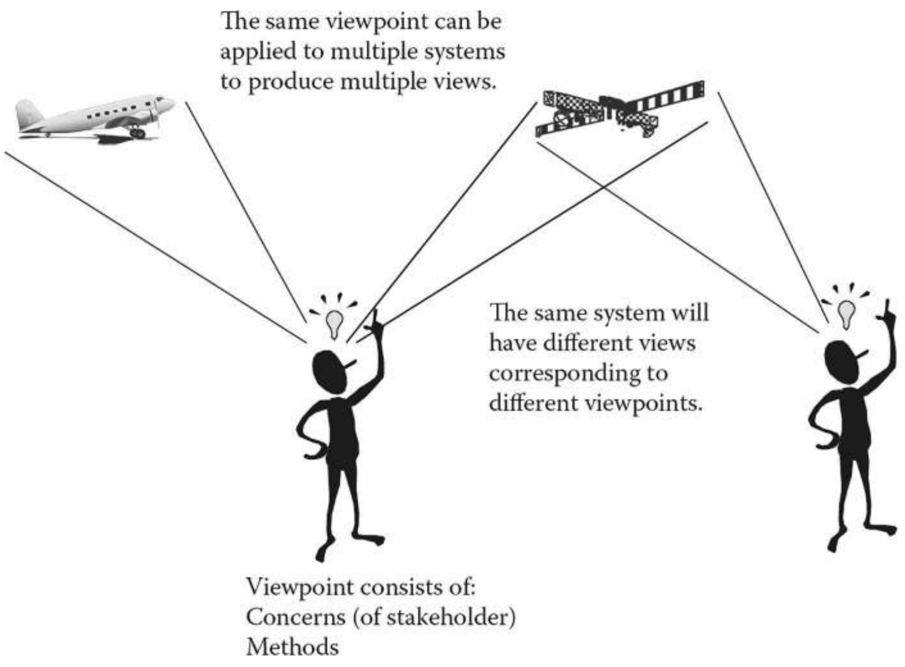

The same viewpoint can be applied to multiple systems to produce multiple views.
The same system will have different views corresponding to different viewpoints.
Viewpoint consists of:
Concerns (of stakeholder)
Methods

FIGURE 8.1 The concept of viewpoint and view. A view belongs to one system, a viewpoint is reusable across many systems.

TABLE 8.1
Major System or Architectural Views

Perspective or View	Description
Purpose/objective	What the client wants
Form	What the system is
Behavioral or functional	What the system does
Performance objectives or requirements	How effectively the system does it
Data	The information retained in the system and its interrelationships
Managerial	The process by which the system is constructed and managed

over time. For example, a system might be behaviorally complex but have relatively simple form. Views that are critical to the architect may play only a supporting role in full development.

Various standards define varying numbers of views; see Chapter 11 for additional details. Some of the variation is due to attachment to particular modeling methods. Some variation just represents different terminology for the same thing.

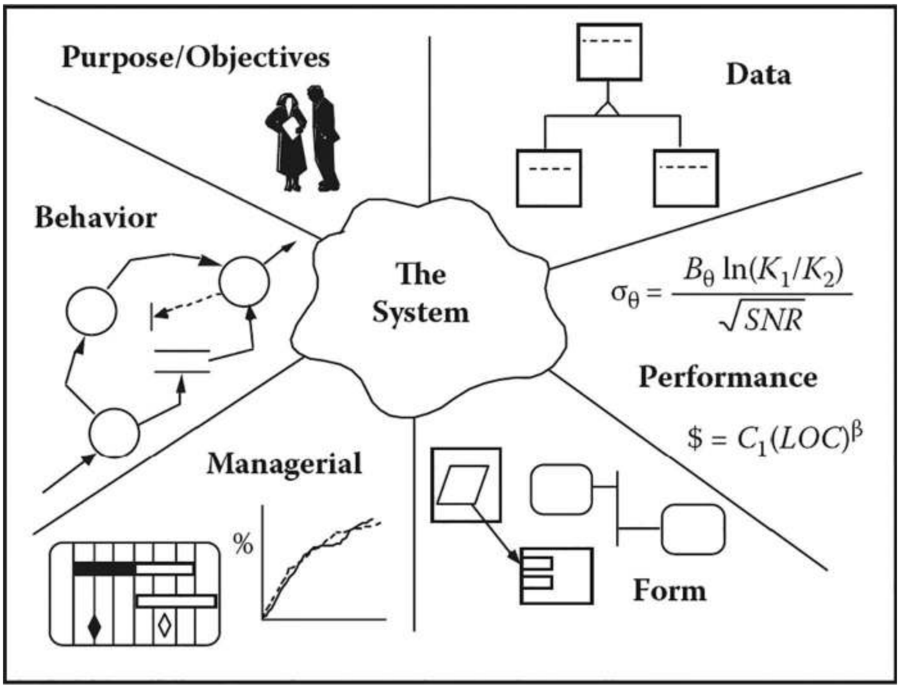


FIGURE 8.2 The six views. All views are representations of some aspect of the actual system. Each view may contain several models, as needed to capture the information of the view.

Some standard breakdowns repeat logically equivalent elements but in distinct ways, where there are behavioral models embedded in both the "Operational" and "System" views in several standards. In this book, we will build up the view concepts from scratch based on their role within our ASAM process framework. Subsequently, we will examine MBSE implementations within different architecture description frameworks.

Although any system can be described in each view, the complexity and value of each view's description can differ considerably. The architect must determine which views are most important to the system and its environment and be expert in the relevant models. The views are chosen to be reasonably independent in the sense that specific information appears in only one, but there is extensive linkage among views. For example, the behavioral aspects of the system are independent of the system's form in that they define different things, but they are inherently related. The system can produce the desired behavior only if the system's form supports it. This linkage is conceptually similar to a front and side view being linked (both show vertical height) even though they are observations from orthogonal directions.

The following sections describe models used for representing a system in each of the views of Table 8.1. The section for each view defines what information is captured by the view, describes the modeling issues within that view, and lists some important modeling methods. In Chapters 9 and 10, we will then concretely assemble a process for building architecture descriptions from the components first defined here and using standard notations. Part of the architect's role is to determine which views are most critical to system success, build models for those views, and then integrate them as necessary to maintain system integrity. The integration across views is a special concern of the architect.

Objectives and Purpose Models

The first modeling view is that of objectives and purposes. Systems are built for useful purposes—that is, for what the client *wants*. Without them, the system cannot survive. The architect's first and most basic role is to match the desirability of the purposes with the practical feasibility of a system to fulfill those purposes. Given a clearly identifiable client, the architect's first step is to work with that client to identify the system's objectives and priorities. Some objectives can be stated and measured precisely. Others will be quite abstract and impossible to express quantitatively. A civil architect is not surprised to hear that a client's objective is for the building to "be beautiful" or to "be in harmony with the natural state of the site." The client will be very unhappy if the architect tells the client that such goals "are not well-formed requirements" and to come back with unambiguous and testable shall statements to replace those abstract concepts. The client will go look for a competent architect, one able to translate conceptual desires into physical expressions.

One approach is for the architect to prepare models to help the client to clarify abstract objectives. Abstract objectives require provisional and exploratory models, models that may fall by the wayside later as the demands and the resulting system become well understood. Ideally, all iterations and explorations become part of the systems document set. However, to avoid drowning in a sea of paper, it may be necessary to focus on a limited set. If refinement and trade-off decisions (the creation of concrete objectives from abstract ones) are architectural drivers, they must be maintained, as it is likely the key decisions will be repeatedly revisited.

Modeling therefore begins by restating and iterating those initial unconstrained objectives from the client's language until a modeling language and methodology emerges, the first major step closer to engineering development. Behavioral objectives are restated in a behavioral modeling language. Performance requirements are formulated as measurable satisfaction models. Some objectives may translate directly into physical form, others into patterns of form that should be exhibited by the system. Complex objectives almost invariably require several steps of refinement and indeed may evolve into characteristics or behaviors quite different from their original statement.

As a low-technology example (though only by modern standards, the technology was often not mature then) consider the European cathedrals of the Middle Ages. A cathedral architect considered a broad range of objectives. First, a cathedral must fulfill well-defined community needs. It must accommodate celebration-day crowds, serve as a suitable seat for the bishop, and operate as a community centerpiece. But, in addition, cathedral clients of that era emphasized that the cathedral "communicate the glory of God and reinforce the faithful through its very presence."

Accommodation of holiday celebrants is a matter of size and floor layout. It is an objective that can be implemented directly and requires no further significant refinement. The clients—the church and community leaders—because of their personal familiarity with the functioning of a cathedral, could determine for themselves the compliance of the cathedral by examining the floor plan. The size of the crowds, their "operational concept" (who would be where and what they would be doing) would be clear. The way the operational concept and crowds could interact could be determined from examination of the floor plans. An expert, like the architect, would do better at this, but even people who are relatively untrained can compare sizes, look at how things scale, and do simple calculations.

But what of defining a building that "glorifies God?" This is obviously a property only of the structure as a whole—its scale, mass, space, light, and integration of decoration and detail. Only a person with an exceptional visual imagination can accurately envision the aesthetic and religious impact of a large structure documented only through drawings and renderings. Especially in those times when architectural styles were new, and people traveled little, an innovative style would be an unprecedented experience for perhaps all but the architect. This is not something where the ability of a planned object to fulfill the objective can be assessed absent extensive expertise.

In this example, we also see the interaction of heuristics and modeling. Models define the architect's approach to the cathedral, but heuristics would be needed to guide decision making. How does the architect know what building features produce the emotional effect that will be regarded as glorifying God and reinforcing the faithful? The architect can know only through induction (experience with other buildings) and the generalization of that induction through theory. Our own experiences with such building can suggest the elements of appropriate heuristics (e.g., great vertical height, visual and auditory effects, integration of iconography, and visual teachings).

Where the use of such heuristics gets really interesting is by considering their extensions and generalizations. We know what features are typically included in sacred architecture in societies with Western European culture, at that time. Are those universals, do they consistently occur in other cultures and other times? To a limited extent, yes. Architecture intended to impress shares a lot of features across cultures. But some reflection may suggest alternative formulations of the heuristics. For example, an underlying feature of many sacred spaces is the notion of sacred-secular separation. Sacred spaces are arranged in ways clearly distinctive from secular spaces. They are unique and the uniqueness signals to the user entrance to a different (and sacred) environment. Different cultures, and different times, experience the secular environment in different ways, contrast the spaces of an agricultural market city in pre-modern eras, to Times Square or the Ginza, and so will experience the sacred-secular distinction differently.

Refinement of objectives through models is central to architecting, but it is also a source of difficulty. A design that proceeds divorced from direct client relevance tends to introduce unnecessary requirements that complicate its implementation. Experience has shown that retaining the client's language throughout the acquisition process can lead to highly efficient, domain-specific architectures, for example, in communication systems.

> **Example**: Domain-Specific Software Architectures are software application generation frameworks in which domain concepts are embedded in the architectural components. The framework is used to generate a product line of related applications in which the client language can be used nearly directly in creating the product. For a set of message handler applications within command and control systems, the specification complexity was reduced 50:1 (Bergner et al. 2005).

One measure of the power of a design and implementation method is its ability to retain the original language. But this poses a dilemma. Retention implies the availability of proven, domain-specific methods and engineering tools. But unprecedented systems by definition are likely to be working in new domains, or near the technical frontiers of existing domains. By the very nature of unprecedented system development, such methods and tools are unlikely to be available. Consequently, models and methodologies must be developed and pass through many stages of abstraction, during which the original relevance can easily be lost. The architect must therefore search out domain-specific languages and methods that can somehow maintain the chain of relevance throughout.

An especially powerful, but challenging, form of modeling converts the client or user's objectives into a meta-model or metaphor that can be directly implemented. A famous example is the desktop metaphor adopted for Macintosh computers. The user's objective is to use a computer for daily, office-oriented task automation. The solution is to capture the user's objectives directly by presenting a simulation of a desktop on the computer display. Integrity with user needs is automatically maintained by maintaining the fidelity of a desktop and file system familiar to the user.

Modern MBSE notations generally do not have formal models for purpose and objectives, though they do have some adaptable tools. SysML and methods built on SysML (Chism 2004, Aleksandraviciene and Morkevicius 2021) do support Stakeholder-Concern tables, a method we will take up in detail later. Use-cases, in SysML and other notations, can be used for purpose-objectives or for more detailed behavioral modeling. There are several standard documentation dialogs suggested by heuristics that are likewise useful for defining purpose and objectives.

A very direct method is through a quantitative value model (or utility model) (Keeney 1996, Hammond et al. 2015). We take up detailed value model construction within a performance view, but the foundation of the performance view is set by the top-level objectives or measures on capabilities. For example, range and payload are essential objectives of any commercial airplane, passenger or cargo specialized. At the objective level we set those as measures of the systems capability, and begin the mapping to the system's attributes. In the performance view we model how those capabilities are achieved in terms of lower-level properties. In SysML and related MBSE notations the top of this model tree will be capabilities with associated "measures of effectiveness" or MOEs. The MOEs will be computable from value properties associated with lower models of form or behavior.

Models of Form

Models of form represent physically identifiable elements of, and interfaces to, what will be constructed and integrated to meet client objectives. Models of form can be abstract in terms of subsystems or closely tied to particular implementation technologies, whether the concrete and steel of civil architecture or the less tangible codes and manuals of software systems. Less tangible physical forms may be relevant in some cases, such as communication protocol standards, a body of laws, or a consistent set of policies.

Models of form vary widely in their degree of abstraction and role. For example, an abstract model may convey no more than the aesthetic feel of the system to the client. A dimensionally accurate but hollow model can assure proper interfacing of mechanical parts. Other models of form may be tightly coupled to performance modeling, as in the scale model of an airplane subjected to wind tunnel testing. The two categories of models of form most useful in architecting are scale models and block diagrams.

Scale Models

The most literal models of form are scale models. Scale models are widely used for client and builder communication and may function as part of behavioral or performance modeling as well. Some major examples include the following:

1. Civil architects build literal models of buildings, often producing renderings of considerable artistic quality. These models can be abstracted to convey the feel and style of a building or can be precisely detailed to assist in construction planning.
2. Automobile makers mock up cars in body-only or full running trim. These models make the auto show circuit to gauge market interest or are used in engineering evaluations.
3. Naval architects model racing yachts to assist in performance evaluation. Scale models are drag tested in water tanks to evaluate drag and handling characteristics. Reduced or full-scale models of the deck layout are used to run simulated sail handling drills.
4. Spacecraft manufacturers use dimensionally accurate models in fit compatibility tests and in crew extravehicular activity rehearsals. Even more important are ground simulators for on-orbit diagnostics and recovery planning.
5. Software developers use prototypes that demonstrate limited characteristics of a product that are equivalent to scale models. For example, user interface prototypes that look like the planned system but do not possess full functionality, non-real-time simulations that carry extensive functionality but do not run in real-time, or just a set of screenshots with scenarios for application use.

Physical scale models are gradually being augmented or replaced by virtual reality systems. These “scale” models exist only in a computer and the viewer’s mind. They may, however, carry an even stronger impression of reality than a physical scale model because of the sensory immersion achievable.

Block Diagrams

Physical block diagrams are ubiquitous in multiple industries. To be a model of form, as distinct from a behavioral model, the elements of the block diagram must correspond to physically identifiable elements of the system. Some common types of block diagrams include the following:

1. System taxonomy diagrams that show the elements of a system and their relationships. Relationships may be composition or association or specialization, depending on the nature of the system.
2. System interconnect diagrams that show specific physical elements (modules) connected by physically identifiable channels. On a high-level diagram, a module might be an entire computer complex and a channel might be a complex internetwork. On a low level, the modules could be silicon chips with specific part numbers and the channels pin-assigned wires. In current MBSE methods, as discussed in Chapter 10, interconnection is represented by associations or ports on definition diagrams or connectors on "internal block diagrams."
3. System flow diagrams show modules in the same fashion as interconnect diagrams but illustrate the flow of information among modules. The abstraction level of information flow defined might be high (complex messaging protocols) or low (bits and bytes). The MBSE equivalents, per chapter 10, are item exchanges on connectors on internal block diagrams. The port abstraction captures the high-low abstraction hierarchy when they are used to show the nesting of protocols and exchanges.
4. Structure charts, task diagrams, and class and object diagrams that structurally define software systems and map directly to implementation. A software system may have several logically independent such diagrams, each showing a different aspect of the physical structure. Take for example, diagrams that show the invocation tree, the inheritance hierarchy, or the "withing" relationships in an Ada program. Specialized block diagrams have been developed to capture these kinds of structures for different forms of software architecture (Gomaa 1994).
5. Manufacturing process diagrams are drawn with a standardized set of symbols. These represent manufacturing systems at an intermediate level of abstraction, showing specific classes of operation but not defining the machine or the operational details.

SysML, Unified Architecture Framework (UAF), and other notations have widely standardized approaches to block diagrams. Older methods that were foundational include Hatley and Pirbhai (Hatley 1994, Hatley et al. 2013). Given the wide use of some of these notations, we can say there are reasonably standardized methods for creating physical block diagrams, and we integrate those into our approach to architecture processes and architecture description creation in Chapter 10.

That said, there are still weaknesses in widely used notations and methods. Specification of data flow control is still weak in widely used tools, and patterns for capturing heavily layered interfaces are also lacking. System model linkage to hardware-specific and software-specific models leaves something to be desired. While software architecture does have its well-established discipline-specific models, they are not as widely used as is desired, and the linkage back to system-level and hardware-level models is incomplete. These are areas for future improvement.

Behavioral (Functional) Models

Functional or behavioral models describe specific patterns of behavior by the system, in logical terms. These are models of what the system *does* (how it behaves) as opposed to what the system *is* (which are models of form). By logical we mean in problem-domain terms rather than solution-domain terms. Behavioral models can be written in either logical or physical terms, and both may be appropriate. The key difference is what the functions in the models operate on. In a logical model, the input/output objects are problem-domain elements. In a physical behavioral model, the input/output objects correspond to physical inputs and outputs.

Architects need behavioral models as systems become more intelligent and their behavior becomes less obvious from the systems form. The models are also important as systems become software-intensive and where there is importance that the architect can analyze what the system does independently of its actual implementation. Unlike a building, a client cannot look at a scale model of a software system and infer how the system behaves. Only by explicitly modeling the behavior can it be understood by the client and builder.

Determining the level of detail or rigor in behavioral specification needed during architecting is an important choice. Too little detail or rigor will mean the client may not understand the behavior being provided (and possibly be unsatisfied), or the builder may misunderstand the behavior actually required. Too much detail or rigor may render the specification incomprehensible—leading to similar problems—or unnecessarily delay development. Eventually, when the system is built, its behavior is precisely specified (if only by the actual behavior of the built system).

From the perspective of architecting, what level of behavioral refinement is needed? The best guidance is to focus on the system acceptance requirements and to ensure the acceptance requirements are passable but complete. Ask what behavioral attributes of the system the client will demand to be certified before acceptance and determine through what tests those behavioral attributes can be certified. The certifiable behavior is the behavior the client will get, no more and no less.

Example: In software systems with extensive user interface components, it has been found by experience that only a prototype of the interface adequately conveys to users how the system will work. Hence, to ensure not just client acceptance but also user satisfaction, an interface prototype should be developed very early in the process. Major office application developers have recorded office workers as they use prototype applications. The recordings are then examined and scored to determine how effective various layouts were at encouraging users to make use of new features, how rapidly they were able to work, and so on. However, for many systems it will be desirable to provide multiple user interfaces into a given capability.

That is best achieved by having a logical model, independent of interface implementation, of what the interface manipulates.

Example: Hardware and software upgrades to military avionics almost always must remain backward compatible with other existing avionics systems and maintain support for existing weapon systems. The architecture of the upgrade must reflect the behavioral requirements of existing system interface. Some may imply very simple behavioral requirements, like providing particular types of information on a communication bus. Others may demand complex behaviors, such as target handover to a weapon requiring target acquisition, queuing of the weapon sensor, real-time synchronization of the local and weapon sensor feeds, and complex launch codes. The required behavior needs to be captured at the level required for client acceptance, and at the level needed to extract architectural constraints.

Behavioral tools of particular importance are mission threads or scenarios, data and event flow networks (functional flow models or activity models), mathematical systems theory, autonomous system theory, and public choice and behavior models. Each of these has a specific set of modeling methods, often several, available. There are advantages and disadvantages to each approach to behavioral description.

Mission Threads and Scenarios, Use-Cases

A thread or scenario is a sequence of exchanges and functions (carried out on the exchanges) between external actors and the system of interest. It is an ordered list of events and actions that represent an important behavior. One version of this is use-cases, a well-known technique in systems and software engineering. Each of these techniques describes the behavior of a system through examples. Rather than trying to comprehensively define the behavior, they provide examples of the desired (and possibly the undesired) behavior. One chooses to use threads/scenarios/use-cases over more comprehensive methods because of a belief that description via examples, or stories, is more a more effective communication vehicle.

To see this better, consider how different methods handle normal versus abnormal behavior. Thread or use-cases description focuses on an un-branched sequence of functions or actions in response to a given input, leading to outputs. The normal path does not contain branches; that is, it is a single serial scenario of operation, a stimulus, and a response thread. Branches can be added. Standard use-case templates provide for separate descriptions of branched paths, whether error-paths or other reasons. But regardless, the focus is on the main path. For many purposes, this is appropriate, it focuses the dialog with stakeholders on what they are trying to achieve rather than more rigorously going searching for a complete specification of behavior. Depending on where we are in the process (recall the three columns of Table P.1), it may be appropriate to ignore off-nominal cases or it may be essential to identify them. Are we at a stage where we are just trying to see what can't fit, or are we trying to run down every detail required for a defect-less implementation?

Threads can be of two types: The use-case and the anti-use-case. The first type is to require that the system *must* produce a given thread—that is, to require a particular system behavior to be present. The anti-use-case alternative is to require that a particular thread not occur—for example, that a function never occurs without a particular type of authorization. The former is more common, but the latter is just as important. Anti-use-cases are frequently created by writing from the perspective of an attacker or some actor trying to cause undesirable behavior. When structured this way, they do not address the preventive means (usually part of the physical design) but focus on what the attacker is trying to achieve (and thus what we want to prevent).

Threads are particularly useful for client communication. Building the threads can be a framework for an interactive dialogue with the client. For each input, pose the question "When this input arrives what should happen?" Or in reverse ask, "When you want X to happen, what input to the system should generate it, and how?" Trace the response until an output is produced. In a similar fashion, trace outputs backward until inputs are reached. The list of threads generated in this way becomes part of the behavioral requirements.

Threads are also useful for builder communication, although other descriptive methods may be more useful. Even if not complete, the threads directly convey the desired system behavior. They also provide useful tools during design reviews and for test planning. Reviewers can ask that designers walk through their design as it would operate in each of a set of selected threads. This provides a way for reviewers to survey a design using criteria very close to the client's own language. Threads can be used similarly as templates for system tests, ensuring that the tests are directly tied to the client's original dialog.

Use-case has become the popular term for behavioral specification by example. The term originally comes from the object-oriented software community, but it has been applied much more widely. The normal form of a use-case is the listing of an example dialogue between the system and an actor. An actor is a human user of the system. The use-case consists of the sequence of messages passed between the system and actor, augmented by additional explanation in ways specific to each method. Use-cases are intended to be narrative. They are intended to be written in the language of users and to be understandable by them. When a system is specified by many use-cases, and the use-cases interact, there are a number of diagrams that can be used to specify the connections.

Data and Event Flow Networks

A complex system can possess an enormous set of threads. A comprehensive list may be impossible, yet without it, the behavioral specification is incomplete. Data and event flow networks allow threads to be collapsed into more compact but complete models. Data flow models define the behavior of a system by a network of functions or processes that exchange data objects. The process network is usually defined in a graphical hierarchy, and most modern versions add some component of finite state machine description. Data and event flow networks are another term for functional decomposition. Current data flow notations are descendants either of DeMarco's data flow diagram (DFD) notation (DeMarco 2011) or Functional Flow Block Diagrams (FFBD). SysML uses an activity model notation whose heritage comes from the DFD and FFBD history. UAF is very similar to SysML activity models. Both the DFD and FFBD methods are based on a set of root principles:

1. The system functions are decomposed hierarchically. Each function is composed of a network of subfunctions until a "simple" description can be written in text.
2. The decomposition hierarchy is defined graphically.
3. Data elements are decomposed hierarchically and are separately defined in an associated "data dictionary" or other data and information model.
4. Functions are default assumed to be data triggered. A process is assumed to execute anytime its input data elements are available. Finer control is defined by a finite state control model (DFD formalism) or in the graphical structure of the decomposition (FFBD formalism).
5. The model structure avoids redundant definition. Coupled with graphical structuring, this makes the model much easier to modify.
6. Most methods using these graphical notations make a distinction between a logical behavioral model (one that uses problem domain concepts in the data and does not define the physical structure of the data) and a physical behavioral model. The latter is tied much more closely to the physical decomposition and structure of the system and the information exchanges become exchanges of specific data objects. Separation of the logical and physical behavioral models, and their interwoven development, is a signature feature of the Hatley-Pirbhai method (Hatley et al. 2013).

The activity model notation in SysML is similar to this tradition, though different in details. The semantics of what executes and when are more complicated and more aligned to enabling executable simulations. There is a clearer distinction between defining and declaring functional elements (as there is throughout the notation).

Mathematical Systems Theory

The traditional meaning of system theory is the behavioral theory of multidimensional feedback systems. Linear control theory is an example of system theory on a limited, well-defined scale. Models of macroeconomic systems and operations research are also system theoretic models but on a much larger scale.

System theoretic formalisms are built from two components:

1. A definition of the system boundary in terms of observable quantities, some of which may be subject to user or designer control.
2. Mathematical machinery that describes the time evolution (the behavior) of the boundary quantities given some initial or boundary conditions and control strategies.

There are three main mathematical system formalisms distinguished by how they treat time and data values:

1. **Continuous Systems:** These systems are classically modeled by differential equations, linear and nonlinear. Values are continuous quantities and are computable at all times.
2. **Temporally Discrete (Sampled Data) Systems:** These systems have continuously valued elements measured at discrete time points. Their behavior

is described by difference equations. Sampled data systems are increasingly important because they are the basis of most computer simulations and nearly all real-time digital signal processing.

3. **Discrete Event Systems:** A discrete event system is one in which some or all of the quantities take on discrete values at arbitrary points in time. Queuing networks are the classical example. Asynchronous digital logic is a pure example of a discrete event system. The quantities of interest (say data packets in a communication network) move around the network in discrete units, but they may arrive or leave a node at an arbitrary, continuous time.

Continuous systems have a large and powerful body of theory. Linear systems have comprehensive analytical and numerical solution methods and an extensive theory of estimation and control. Nonlinear systems generally don't have closed-form solutions, but many numerical techniques are available, some analytical stability methods are known, and practical control approaches are available. The very active field of dynamical systems addresses nonlinear as well as control aspects of systems. Similar results are available for sampled data systems. Computational frameworks exist for discrete event systems (based on state machines and Petri Nets) but are less complete than those for differential or difference equation systems in their ability to determine stability and synthesize control laws. A variety of simulation tools are available for all three types of systems. Some tools attempt to integrate all three types into a single framework, though this is difficult.

Many modern systems are a mixture of all three types. For example, consider a computer-based temperature controller for a chemical process. The complete system may include continuous plant dynamics, a sampled data system for control under normal conditions, and discrete event controller behavior associated with threshold crossings and mode changes. A comprehensive and practical modern system theory should answer the classic questions about such a mixed system—stability, closed-loop dynamics, and control law synthesis. No such comprehensive theory exists, but constructing one is an objective of current research. Manufacturing systems are a special example of large-scale mixed systems for which qualitative system understanding can yield architectural guidance.

Autonomous Agent, Chaotic Systems

System-level behavior, as defined in Chapter 1, is behavior not contained in any system component but which emerges only from the interaction of all the components. A class of system of recent interest is that in which a few types of multiply replicated, individually relatively simple, components interact to create essentially new (emergent) behaviors. Ant colonies, for example, exhibit complex and highly organized behaviors that emerge from the interaction of behaviorally simple, nearly identical, sets of components (the ants). The behavioral programming of each individual ant, and its chaotic local interactions with other ants and the environment, is sufficient for complex high-level behaviors to emerge from the colony as a whole. There is considerable interest in using this truly distributed architecture, but traditional top-down, decomposition-oriented models and their bottom-up integration-oriented complements do not describe it. Some attempts have been made to build theories of such systems from chaos methods. Attempts have also been made to find rules or heuristics for the local intelligence and interfaces necessary for high-level behaviors to emerge.

Example: In some prototype flexible manufacturing plants, instead of trying to solve the very complex work scheduling problem, autonomous controllers schedule through distributed interaction. Each work cell independently "bids" for jobs on its input. Each job moving down the line tries to "buy" the production and transfer services it needs to be completed. Instead of central scheduling, the equilibrium of the pseudo-economic bid system distributes jobs and fills work cells. Experiments have shown that rules can be designed that result in stable operation, near optimality of assignment, and very strong robustness to job changes and work cell failure. But the lack of central direction makes it difficult to assure particular operational aspects (for example, to assure that "oddball" jobs will not be ignored for the apparent good of the mean).

Public Choice and Behavior Models

Some systems depend on the behavior of human society as part of the system. In such cases, the methods of public choice and consumer analysis may need to be invoked to understand the human system. These methods are often ad hoc, but many have been widely used in marketing analysis by consumer product companies.

Example: One concept in intelligent transportation systems proposals (recall the discussion in "Case Study 4" on ITS before Chapter 5) is the use of centralized routing. In a central routing system, each driver would inform the center (via some data network) of his or her beginning location and his or her planned destination for each trip. The center would use that information to compute a route for each vehicle and communicate the selected route back to the driver. The route might be dynamically updated in response to accidents or other incidents. In principle, the routing center could adjust routes to optimize the performance of the network as a whole. But would drivers accept centrally selected routes, especially if they thought the route benefited the network but not them? Would they even bother to send in route information?

A variety of methods could be used to address such questions. At the simplest level are consumer surveys and focus groups. A more involved approach is to organize multiperson driving simulations with the performance of the network determined from individual driver decisions. Over the course of many simulations, as drivers evaluate their own strategies, stable configurations may emerge.

Performance Models

A performance model describes or predicts how effectively an architecture satisfies some objective, either functional or not. Performance models are usually quantitative, and the most interesting performance models are those of system-level functions—that is, properties possessed by the system as a whole but by no subsystem.

Performance models describe properties like overall sensitivity, accuracy, latency, adaptation time, weight, cost, reliability, and many others. Performance requirements are often called "nonfunctional" requirements because they do not define a functional thread of operation, at least not explicitly. Cost, for example, is not a system *behavior*, but it is an important property of the system. Detection sensitivity to a particular signal, however, does carry with it implied functionality. Obviously, a signal cannot be detected unless the processing is in place to produce a detection. It will also usually be impossible to formulate a quantitative performance model without constraining the system's behavior and form.

Performance models come from the full range of engineering and management disciplines. But the internal structure of performance models generally falls into one of three categories:

1. **Analytical:** Analytical models are the products of the engineering sciences. A performance model in this category is a set of lower-level system parameters and a mathematical rule of combination that predicts the performance parameter of interest from lower-level values. The model is normally accompanied by a "performance budget" or a set of nominal values for the lower-level parameters to meet a required performance target.
2. **Simulation:** When the lower-level parameters can be identified, but an easily computable performance prediction cannot, a simulation can take the place of the mathematical rule of combination. In essence, a simulation of a system is an analytical model of the system's behavior and performance in terms of the simulation parameters. The connection is just more complex and difficult to explicitly identify. A wide variety of continuous, discrete time, and discrete event simulators are available, many with rich sets of constructs for particular domains.
3. **Judgmental:** Where analysis and simulation are inadequate or infeasible, human judgment may still yield reliable performance indicators. In particular, human judgment, using explicit or implicit design heuristics, can often rate one architecture as better than another, even where a detailed analytical justification is impossible.

Formal Methods

The software engineering community has taken a specialized approach to performance modeling known as formal methods. Formal methods seek to develop systems that provably produce formally defined functional and nonfunctional properties. In formal development, the team defines system behavior as sets of allowed and disallowed sequences of operation and may add further constraints, such as timing, to those sequences. They then develop the system in a manner that guarantees compliance with the behavioral and performance definition. Roughly speaking, the formal methods approach is as follows:

1. Identify the inputs and outputs of the system. Identify a set of mathematical and logical relations that must exist between the input and output sequences when the system is operating as desired.

2. Decompose the system into components, identifying the inputs and outputs of each component. Determine mathematical relations on each component such that their composition is equivalent to the original set of relations one level up.
3. Continue the process iteratively to the level of primitive implementation elements. In software, this would be programming language statements. In digital logic, this might be low-level combinational or sequential logic elements.
4. Compose the implementation backward up the chain of inference from primitive elements in a way that conserves the decomposed correctness relations. The resulting implementation is then equivalent to the original specification.

From the point of view of the architect, the most important applications of formal methods are in the conceptual phases and in the certification of high-assurance and ultraquality systems. Formal methods require explicit determination of allowed and disallowed input and output sequences. Trying to make that determination can be valuable in eliciting client information, even if the resulting information is not captured in precise mathematical terms. Formal methods also hold out the promise of being able to certify system characteristics that can never be tested. No set of tests can certify that certain event chains cannot occur, but theorems to that effect are provable within a formal model.

Various formal and semiformal versions of the process are in limited use in software and digital system engineering. Although a fully formal version of this process is apparently impractical for large systems at the present time (and is definitely controversial), semiformal versions of the process have been successfully applied to commercial products.

A fundamental problem with the formal methods approach is that the system can never be more "correct" than the original specification. Because the specification must be written in highly mathematical terms, it is particularly difficult to use in communication with the typical client.

Model-Based Systems Engineering Terms

Within MBSE, the performance models are typically captured as "capabilities" and "Measures of Effectiveness (MOE)." Capabilities are the things we measure on the system that correspond to value, and the MOEs are models that compute/estimate the capabilities from lower-level properties. If the system of interest is a space launch vehicle, then an essential capability is its ability to place something into a desired orbit while being launched for an available launch facility. We would likely measure that capability by the mass it can place into orbits of interest. We might have other measures of the capability as well, for example the accuracy of placement and the reliability of delivery to the designated orbit. The MOEs on these capability measures would come from standard launch physics, coupled with measures of performance of each stage, of the guidance system, and reliability by component.

As we'll see in a subsequent chapter, MBSE methods have places to capture these but no real graphical notation. The guts of the models are mathematical or computational. We will see things in SysML in parametric diagrams with analogs in other notations.

An important larger aspect of these models is whether the architect forms quantitative models of overall value from the elements of performance. These are known as Value Models or Utility Models. In the decision analysis literature, a value model and a utility model are distinguished by how uncertainty is incorporated (if it is incorporated), but the terms are often used interchangeably unless the effort is a very formal, detailed one. Forming a whole-system utility model can be a very valuable effort. Ultimately, decisions on what to build, or whether to build at all, hinge on the value. Recall the emphasis on value, cost, and risk as the essential elements of architecture. So, logically, we should seek a model of overall value. The important caveat is that sometimes the effort required to form a full self-consistent and complete model of overall value is so great that it exceeds what is achieved by doing it. Less complete models may be more useful. This is a type of judgment call the architect must make.

Data Models

In the previous discussion, we noted that both behavioral and physical models are built around data exchanges. The "blocks" in the models are either physical or functional entities. The lines indicate connections and things are exchanged along those lines. Exchanges can be other than data (physical objects or energy, for example), but data exchanges are common. The data models (contained in a data and information view) define those exchanges. But the most important aspect of a data and information view is not exchange, it is retained data. What data does the system retain and what relationships among the data does it develop and maintain? Many large corporate and governmental information systems have most of their complexity in their data and the data's internal relationships.

The most common data models have their origins in software development, especially large database developments. Data models are of increasing importance because of the greater intelligence being embedded in virtually all systems and the continuing automation of legacy systems. In data-intensive systems, generating intelligent behavior is primarily a matter of finding relationships and imposing persistent structure on the records. This implies that the need to find structure and relationships in large collections of data will be determinants of systems architecture.

Example: Manufacturing software systems are no longer responsible just for control of work cells. They are part of integrated enterprise information networks in which real-time data from the manufacturing floor, sales, customer operations, and other parts of the enterprise are stored and studied. Substantial competitive advantages accrue to those who can make intelligent judgments from these enormous data sets.

Example: Intelligent transport systems are a complex combination of distributed control systems, sensor networks, and data fusion. Early deployment stages will emphasize only simple behavioral adaptation, as in local intelligent light and on-ramp controllers. Full deployment will fuse data sources across metropolitan areas to generate intelligent prediction and control strategies. These later stages will be driven by problems of extracting and using complex relationships in very large databases.

Data modeling methods trace from entity-relationship diagram methods developed to support relational database design to the more general concepts and diagram sets in unified modeling language (UML) and its various progeny. These methods are largely rooted in object-oriented concepts and modeling techniques. An object is a set of “attributes” or data elements and a set of “methods” or functions that act upon the attributes (and possibly other data or objects as well). Objects are instances of classes that can be thought of as templates for specific objects. Objects and classes can have relationships of several types. Major relationship types include aggregation (or composition); generalization, specialization, or inheritance; and association (which may be two-way or M-way). Object-oriented modeling methods combine data and behavioral modeling into a single hierarchy organized along and driven by data concerns. Behavioral methods like those described earlier also include data definitions, but the hierarchy is driven by functional decomposition.

One might think of object-oriented models as turning functional decomposition models inside out. Functional decomposition models like data flow diagramming describe the system as a hierarchy of functions and hang a data model onto the functional skeleton. The only data relationship supported is aggregation. An object-oriented model starts with a decomposition of the data and hangs a functional model on it. It allows all types of data relationships. Some problems decompose cleanly with functional methods and only with difficulty in object-oriented methods, and some other problems are the opposite.

Data-oriented decompositions share the general heuristics of systems architecture. The behavioral and physical structuring characteristics have direct analogs—composing or aggregation, decomposition, and minimal communications. There are also similar problems of scale. Very large data models must be highly structured with limited patterns of relationship (analogous to limited interfaces) to be implementable.

Managerial Models

To both the client and architect, a project may be as much a matter of planning milestones, budgets, and schedules as it is a technical exercise. In sociotechnical systems, planning the system deployment may be more difficult than assembling its hardware. The managerial or implementation view describes the process of building the physical system. It also tracks construction events as they occur.

Most of the models of this view are the familiar tools of project management. In addition, management-related metrics that can be calculated from other models are invaluable in efforts to create an integrated set of models. Some examples include the following:

1. The waterfall and spiral system development meta-models—the templates on which project-specific plans are built.
2. Program Evaluation and Review Technique/Critical Path Method (PERT/CPM) and related task and scheduling dependency charts.

3. Cost and progress accounting methods.
4. Predictive cost and schedule metrics calculable from physical and behavioral models
5. Design or specification time quality metrics—defect counts, post-simulation design changes, rate of design changes after each review

The architect has two primary interests in managerial models. First, the client usually cannot decide to go ahead with system construction without solid cost and schedule estimates. Usually, producing such estimates requires a significant effort in management models. Second, the architect may be called upon to monitor the system as it is developed to ensure its conceptual integrity. In this monitoring process, managerial models will be very important.

CONCLUSIONS AND EXERCISES

An architect's work revolves around models. Because the architect does not build the system directly, its integrity during construction must be maintained through models acting as surrogates. Models will represent and control the specification of the system, its design, and its production plan. Even after the system is delivered, modeling will be the mechanism for assessing system behavior and planning its evolution. Because the architect's concerns are broad, architecting models must encompass all views of the system. The architect's knowledge of models, like an individual's knowledge of language, will tend to channel the directions in which the system develops and evolves.

Modeling for architects is driven by three key characteristics:

1. Models are the principal language of the architect. Their foremost role is to facilitate communication with client and builder. By facilitating communication, they carry out their other roles of maintaining design integrity and assisting synthesis.
2. Architects require a multiplicity of views and models. The basic ones are objective, form, behavior, performance, data, and management. Architects need to be aware of the range of models that are used to describe each of these views within their domain of expertise, and the content of other views that may become important in the future.
3. Multidisciplinary, integrated modeling methods tie together the various views. They allow the design of a system to be refined in steps from conceptually abstract to the precisely detailed necessary for construction.

EXERCISES

1. Choose a system familiar to you. Formulate a model of your system in each of the views discussed in this chapter. How effectively does each model capture the system in that view? How effectively do the models define the system for the needs of initial concept definition and communication with clients and builders? Why did you choose the modeling method you did? How well does it allow you to integrate information across views.

2. Repeat exercise 1, but with a system unfamiliar to you, and preferably embodying different driving issues. Investigate models used for the views most unfamiliar to you. In retrospect, does your system in exercise 1 contain substantial complexity in the views you are unfamiliar with?
3. Investigate one or more popular model-based systems engineering tools. To what extent do they support each of the views? To what extent do they allow integration across views?
4. A major distinction in behavioral modeling methods and tools is the extent to which they support or demand executability in their models. Executability demands a restricted syntax and up-front decision about data and execution semantics. Do these restrictions and demands help or hinder initial concept formulation and communication with builders and clients? If the answer is variable with the system, is there a way to combine the best aspects of both approaches?
5. Models of form must be technology specific (at least eventually) because they represent actual systems. Investigate modeling formalisms for domains not covered in this chapter, for example, telecommunication systems, business information networks, space systems, integrated weapon systems, chemical processing systems, or financial systems.

REFERENCES

Aleksandraviciene, A. and A. Morkevicius (2021). *MagicGrid Book of Knowledge*, Kaunas: Dassault Systemes.

Bergner, K., et al. (2005). Dosam–domain-specific software architecture comparison model. *International Conference on the Quality of Software Architectures*, Berlin: Springer.

Chism, J. (2004). "Overview and status of the object oriented systems engineering methodology OOSEM." *Insight* **7**(2): 31–33.

DeMarco, T. (2011). Structured analysis and system specification. In: *Software Pioneers: Contributions to Software Engineering*. E. Denert (Ed), Berlin: Springer, pp. 529–560.

Gomaa, H. (1994). "Software design methods for the design of large-scale real-time systems." *Journal of Systems and Software* **25**(2): 127–146.

Hammond, J. S., R. L. Keeney, and H. Raiffa (2015). *Smart Choices: A Practical Guide to Making Better Decisions*, Brighton, MA: Harvard Business Review Press.

Hatley, D. J. (1994). "Current system development practices using the hatley/pirbhai methods." *The Journal of NCOSE* **1**(1): 69–81.

Hatley, D., P. Hruschka, and I. Pirbhai (2013). *Process for System Architecture and Requirements Engineering*, Boston, MA: Addison-Wesley.

ISO (2018). *ISO/IEC/IEEE FDIS 42020 Enterprise, Systems, and Software - Architecture Processes*, New York: International Standards Organization.

ISO (2022). *ISO/IEC/IEEE 42010 Software Systems and Enterprise - Architecture Description: 69*, New York: International Standards Organization

Keeney, R. L. (1996). *Value-Focused Thinking: A Path to Creative Decisionmaking*, Cambridge, MA: Harvard University Press.

Maier, M. W. (1998). "Architecting principles for systems-of-systems." *Systems Engineering: The Journal of the International Council on Systems Engineering* **1**(4): 267–284.

Maier, M. W. (2019). "Architecting portfolios of systems." *Systems Engineering* **22**(4): 335–347.

9 A General Architecting Process

DECISIONS AND DESIGN: AN ARCHITECTURE LIFECYCLE

Architecture decisions, as discussed in the previous chapter, are one element of a comprehensive design process. From conception to implementation design is a series of decisions. Some are of great importance and impact, and some are routine. The decisions that drive value, cost, and risk are architectural. That is not to say that lower-level design decisions are not important, made badly they may kill the system. Per the right-hand column in Table P.1 of the preface, when we build and deploy a system, even a minor mistake can kill the system, something most experienced developers are only too familiar with.

Conceptually, we can think of design as a progression, specifically a progressive removal of abstraction. Progressive refinement of design is one of the most basic patterns of engineering practice. It permeates the process of architecting from models to heuristics, information acquisition, and management. Its real power, especially in systems architecting, is that it provides a way to organize the progressive transition from the ill-structured, chaotic, and heuristic processes needed at the beginning to the rigorous engineering and certification processes needed later. All can be envisioned as a stepwise reduction of abstraction, from mental concept to delivered physical system.

In software, the process is known as stepwise refinement. Stepwise refinement is a specific strategy for top-down program development. The same notion applies to architecting but is applied in-the-large to complex, multidisciplinary system development. Stepwise refinement is the progressive removal of abstraction in models, evaluation criteria, and goals. It is accompanied by an increase in the specificity and volume of information recorded about the system and a flow of work from general to specialized design disciplines. Within the design disciplines, the pattern repeats as disciplinary objectives and requirements are converted into the models of form of that discipline. In practice, the process is neither so smooth nor continuous. It is better characterized as episodic, with episodes of abstraction reduction alternating with episodes of reflection and purpose expansion.

Before treating the conceptually difficult process of general systems architecting, look to the roots. When a civil architect develops a building, does he or she go directly from client words to construction drawings? Obviously not; there are many intermediate steps. The first drawings are rough floor plans showing the spatial relationships of rooms and sizes and external renderings showing the style and feel of the building. Following these are intermediate drawings giving specific dimensions and layouts. The construction drawings with full details for the builder follow on after. The architect's role does not have a universally applicable stopping point, but the normal case is based on the needs of the client. The client hired the architect to accomplish a specific portion of the overall development and construction process. When the designs are sufficiently refined (in enough views) for the client to make the decision to proceed with construction, the architect's conceptual development job is complete. The architect may be busy with the project for some time to come in shepherding the conceptual design through detailed design, overseeing construction, and advising the client on certification, but the initial concept role is complete when the client can make the construction decision.

In a different domain, the beginning computer programmer is taught a similar practice. Stepwise refinement in programming means to write the central controlling routine first. Anywhere high complexity occurs, ignore it by giving it a descriptive name and making it a subroutine or function. Each subroutine or function is "stubbed" — that is, given a dummy body as a placeholder. When the controlling routine is complete, it is compiled and executed as a test. Of course, it does not do anything useful because its subroutines are stubbed. The process is repeated recursively on each subroutine until routines can be easily coded in primitive statements in the programming language. At each intermediate step, an abstracted version of the whole program exists that has the final program's structure but lacks internal details.

Both examples show progression of system representation or modeling. Both examples embed strategy, in terms of ordering of decisions in ways that meet client or sponsor needs, into the progressive development process. In building, the sponsor needs to set up a distinct decision point where financing is resolved and a building contractor is hired. In software development, top-down stepwise refinement assembles a program in a fashion that facilitates continuous testing and incremental delivery. The building sponsor needs a development process (and a relationship with the architect) that supports the customary financial arrangements and the limitations of the contracting industry. Software developers, especially in commercial markets, prefer mechanisms that facilitate incremental delivery and provide full program level "test harnesses."

Progression also occurs along other dimensions. For example, both the civil architect and the programmer may (should) create distinct alternative designs in their early stages. How are these partial designs evaluated to choose the superior approach? In the earliest stages, both the programmer and the civil architect use heuristic reasoning. The civil architect can measure rough size (to estimate cost), judge the client's reaction, and ask the aesthetic opinion of others. The programmer can judge code size, heuristically evaluate the coupling and cohesion of the resulting subroutines and modules, review applicable patterns from catalogs, and review functionality with the client. As their work progresses, both will be able to make increasing use of rational and quantitative evaluation criteria. The civil architect will have enough details for proven cost models; the programmer can measure execution speed, compiled size, and behavioral compliance, and invoke quantitative software quality metrics. Programmers will also have improved cost models as progression continues. Software cost models are predominantly based on code size, and the progressive development of the top-down structure supports improved estimates of code size.

As first discussed in Chapter 1, architecture decisions are embedded in an overall system development lifecycle. That lifecycle might be once-through, building one system one time (the classic waterfall). It may be some form of spiral cycle leading to production. It may be an incremental process from prototypes and "minimum viable products" leading to no determinate endpoint, with no endpoint required when increments are sufficiently profitable or operationally valuable to be self-supporting. We make architecture decisions in each case. In all cases there are some up-front architecture decisions. At a minimum the choice to pursue development on one problem/technical line by expending resources and not pursuing a different line is an architecture decision. Rapid prototyping is sometimes thought of as the antithesis of overly rigorous up-front analysis, but to do a prototype one has to choose among approaches. Many prototypes were possible, each possible prototype emphasizes one set of concerns and ignores others. We make the decision regardless of how little attention it is given.

As briefly discussed in Chapter 1, and to be discussed in depth in Chapter 12, architecture decisions can be up-front or can be episodic. In the classic waterfall they are primarily up-front, they are embedded in the choice of requirements and top-level design features. In spiral and incremental processes "up-front" occurs episodically. Every time we go through a spiral cycle or decide to produce another increment we are making architecture decisions of the sort identified in the previous chapter. We are deciding what portion of the problem space to work on, and which portion to ignore. We are deciding that solution approach to use. Either explicitly or implicitly we are laying down stable forms that we will need in this and subsequent cycles.

Even in agile development much the same thing happens, though the compression of cycles leads to some differences. Each agile cycle chooses some features to develop, be those features user-facing or under-the-hood kinds of things seen as necessary for continued progression. The features typically come off of a backlog that itself represents the slower resolution of decisions as to what longer-term direction the system is to take. There is also the concept of "architecture runway" in agile where the team needs to make decisions, and implementations, of infrastructure needed ahead of when features that will require it come off the backlog.

Decisions that span multiple cycles or architecture runway are more specific examples of the heuristic of stable intermediate forms. This heuristic states: Systems evolve more effectively when they possess stable intermediate forms than when they do not. The original source for this heuristic (Rechtin 1991) took it from construction where we understand the need for structures to be self-supporting (as much as possible) as they are assembled. When we want the system to not just be assembled but to evolve then we have to consider the stability of all of the evolutionary pathways. The original meaning was primarily physical, the idea that the structure should be physically stable at each stage. But many other meanings are also relevant. For example, a system can be regarded as economically stable in evolution if it produces profitable cash flows at each stage. A system would be politically stable in evolution if it has a sufficient constituency at each stage, both for its operation at that stage and for the transition to the next stage.

An alternative way to visualize the lifecycle is as the progression of models. This is a very useful visualization as it leads directly to our approach to architecture modeling and the application of MBSE. We discussed this concept earlier in the chapter looking at the paired analogy of civil architecture practice and stepwise refinement in software development. We can envision the development of the plans (architecture) for a building as a stepwise progress from abstract to specific and detailed. The initial steps provide very general, abstracted representations, the sketches and low detail models. These progress to very specific, detailed models. Along the way the architect generates and compares multiple concepts. Each concept expressing some different combination of features or methods. The client should be engaged, not to judge the feasibility of the features represented but rather the value perceived to be provided by those different combiantions of features.

The software progression outlined with essentially the same, although the way the client interacts with the intermediate representations is likely to be different. In software the tradition of "mocking up" alternative designs is somewhat different. Examining different user-interface concepts side-by-side is commonplace, and fits with the analogy to civil architecture practice well. Mocking up different data structures or algorithms is probably less common, but still known and useful. The evaluation of those kinds of different internal concepts would likely not be done with clients, they are fundamentally internal design choices, albeit choices with architectural concepts. The civil architecture analogy would be the examination of alternative structural or construction approaches. Sometimes comparison of such internals would be essential as the structural and construction methods may be essential to realizing the overall concept of a building. It is easy to think of buildings both centuries old and very modern whose vision, and user experience, was enabled by innovative construction. Whether a Frank Gehry building or Brunneleschi's dome in Florence the concept of a building is sometimes tied deeply to internal construction enabling methods.

In Chapter 10 we show implementation of the progressive reduction of abstraction concept in MBSE. Here the implementation of the pattern is somewhat different than it is seen in the civil architecture and software development comparisons. MBSE models are all abstract in the sense of being non-specific and representing abstractions of physical things. Conversely, MBSE models can be detailed even at the top-level. The elements of MBSE models are things like blocks, classes, and objects. These are abstracted representations of physical things (in the case of blocks) or are inherently abstract concepts, in the case of classes. However, they may contain considerable detail, for example value properties or representations of physically realized interfaces, even when considering things at the top level of a long system-subsystem-component hierarchy. The process of vertical decomposition in a physical system-subsystem-component hierarchy is less a progression of abstraction to detail as it is a progression of aggregation to physical component.

At the bottom of the hierarchy there is no doubt the elements are specific, identifiable, realized physical entities belonging to specific engineering disciplines (e.g., electrical or mechanical). The goal of the MBSE physical hierarchy is to reach disciplinary design, identifying and controlling the arrangements above the disciplinary levels. The design progression as reduction of abstraction applies in MBSE, but is more embedded in the overall process than in the model specifics. Stepwise refinement may be a property of an MBSE method, not of the MBSE models themselves. The models allow for definition of precise detail at nearly any level in a hierarchy, the refinement process is defined by the method not the notation.

Some MBSE methods, as will be discussed in Chapter 10, deliberately build in some notion of stepwise refinement from abstract to concrete through the use of parallel hierarchies. For example, in UAF if one starts with the Strategic View, then the Operational, then the Resource (the physical decomposition element in the UAF scheme) then there will be a logical pattern of stepwise refinement. The "blocks" in the strategic view will be very abstracted capabilities. In the operational view they will be "operational performers" that do not (in general) have any one-to-one relationship with physical elements. Only in the resource view do the blocks/usages have a clear and essential relationship to physical things bought or built.

The effort put into design must be balanced with the time required. Careful design may reduce defects, but the more rigorous the design process the longer it will take. Time spent designing is opportunity cost for what could have been built

On the Role of Speed

An occasional critique of systems engineering, and systems architecting, is that it is too time consuming. In current times speed is essential and we are better off going faster, even if we frequently get things wrong, then going more slowly and carefully. It is important to consider the role and value of time (or speed if you prefer). Significant up-front thinking does not imply going slowly, and going fast does not imply getting anywhere you care to be faster.

A general, and powerful approach, is to first consider the time-value of whatever system is under consideration. Will having the system sooner necessarily deliver more value? Sometimes there will be an emphatic "yes," in many different domains. In a competitive market where it is known that others are developing innovative products in the same space then being first may have very important value. For example, in some electronic chip markets the price of a chip made with a new technology (and thus denser than the previous generation) is expected to drop by halves on a regular schedule. To not be first to market means to miss out on the early period of higher prices and to only be able to sell in the later, lower, part of the price curve. In military operational domains, for example electronic countermeasures and counter-countermeasures, advantages are often fleeting and will be lost if late to need.

However, there are often also countervailing factors. First, being first to market or operations with an advanced capability may not be valuable if you cannot produce enough. There are very few "silver bullet" capabilities in either commercial markets or military operations where very small numbers can make a large overall difference. In the military sphere consider the last years of World War II where many advanced capabilities were delivered (jet aircraft, long range guided missiles, nuclear weapons). Only the latter had close to a "silver bullet" effect operationally with low numbers.

Second, the most innovative systems typically do not show their greatest impact until they have been digested by users and those users have significantly changed their operational concepts, or entirely new markets have been found. Consider the nexus of technologies that make up modern individual computing (personal computers, networking, laser printing, windowed user interfaces, etc.). All of those features were available in integrated, marketed systems (e.g., Xerox Star) several years before the market took off with systems supplied by other companies. Much of the key to the takeoff was the emergence of new user applications and new users who were not engaged by the earliest system providers.

Third, effective deployment of a system may have important pacing elements that have little to do with the system's development. This is especially likely where a capability does not deliver user value until it is integrated into a longer chain, or where capabilities are not widely deployed until they are certified for use by some authority whose processes may be very different from those of the developer. As an example, consider developing new space-based environmental remote sensing capabilities (Volz et al. 2016). Most of the time those capabilities are not delivered directly to end-users (though sometimes they are). Most of the time the remote sensing data goes through a long chain of processing and assimilation into other products. In a very important case, weather forecasting, new data is useful either when it is fully assimilated into numerical weather models or integrated into human forecasting processes (Anthes et al. 2019). The process of changing models and processes to assimilate new data, and then (very importantly) validate the increased accuracy of forecasts after that assimilation is not simple and typically requires several years. Even if assimilation processes could be accelerated the process of validation is inherently lengthy. Much as in drug trials, retrospective validation is always suspect, the gold standard is prospective trials, and those trials have to be long enough and wide enough to validate whatever is the increased margin of performance. If the performance increase is modest, as it usually is given the maturity of the field, then only a long and complex trial will fully validate it.

THE APM AND ASAM FRAMEWORK

Chapter 2 first introduced the Architecture Project Method (APM) and Applied Systems Architecting Method (ASAM) and it has been successively elaborated in the immediately preceding chapters. The APM is a method for structuring an architecture project. It assumes you are faced with identifying and making (or facilitating the making) of a set of architecture decisions. Canonically this is starting a new program, or more precisely, deciding whether or not to start a new development program and choosing exactly what sort of program to start. This carries the assumption, really observation, that we consider starting far more development efforts than we actually carry out, and that considering and rejecting most hypothetical development efforts is a good thing. Most superficially clever ideas just aren't that good, and strong pruning of possible start-ups is a good thing. The ASAM is the core architecting process. It is how we work through the stakeholder analysis, problem domain analysis, solution brainstorming, and related analyses to decide to go forward or to drop the effort with relatively little investment.

Our goal is to define a generic, broadly applicable set of methods that can be implemented in varying levels of formality and are tool agnostic. Relative to the ISO/IEC/IEEE 42020 (ISO 2018) architecture process standard we seek to address primarily Architecture Conceptualization, the process of problem understanding and generation of alternative solutions, at an architectural level. Architecture synthesis necessarily includes a degree of architecture evaluation, another 42020 process. We cannot select an architecture if we don't know that it is suitable, and have selected it from among alternatives. We also want to describe the architecture, so our process includes architecture description, also a 42020 process. What we do not include from the 42020 processes are governance, management, or capability development. These are processes of the enterprise that is responsible for architecting rather than architecture processes, as we see them. They are important, but not the subject of the generic process here.

If the situation warrants the method should be implementable with formality in an MBSE tool environment. In Chapter 10, we will review a detailed implementation in SysML as supported in tools popular at the time of this writing. We will outline implementation in other notations and tools. Alternatively, the methods can be implemented in non-specialized computer tools or on large sheets of paper. The latter, large sheets of paper, is quite effective and should be considered for early cycles, we use it often in educational settings and in some real working sessions.

If you are architecting with markers on large sheets of paper, you are most likely engaged in a "charrette." Charrettes are an important concept and being able to perform one is a very valuable skill for an architect. The term charrette is borrowed from civil architecture practice where it was inspired by 19th century architecture students in Paris and their late term cram-sessions with canvases. We use the term to cover short terms (days to 2 weeks) end-to-end design cycles conducted on a forced schedule-certain basis. We use the charrette concept to get away from a reliance on slow, tool heavy cycles that focus too much on getting every element right and not enough on learning by doing, where the doing must extend through to the end of the design process.

The description of APM-ASAM may superficially emphasize slow and careful analysis. Often, it will be used that way and architecting often gets explicit attention in high consequence environments where do-overs are very expensive or impossible. We have emphasized that there are other programmatic templates, like incremental development and the risk spiral where prototyping and rapid, iterative development is deliberate and emphasized. The sponsor and architect must choose. In some environments being careful and right will beat being fast. In others being faster wins. Some questions can be best resolved (resolved more accurately and quickly) by prototype experience than by analysis. Some prototypes look exciting but yield nothing that could not have been learned cheaper (and faster) via better analysis. We discuss in the structure of the APM and ASAM where prototyping efforts can be utilized and where they make more sense than analysis processes.

WALKING THROUGH THE APM

In the combination of the APM and the ASAM, illustrated in Figure 9.1 we first address the APM. The APM is a generic method for conducting architecture projects, that is projects where a set of architecture decisions is the desired outcome. Exactly what decisions depends on the scenario. The most common case is where there is a go/no-go decision, we should proceed to the next development step or drop the idea as unfavorable. Assuming we want to go ahead, then more cycles will result in decision on what to build.

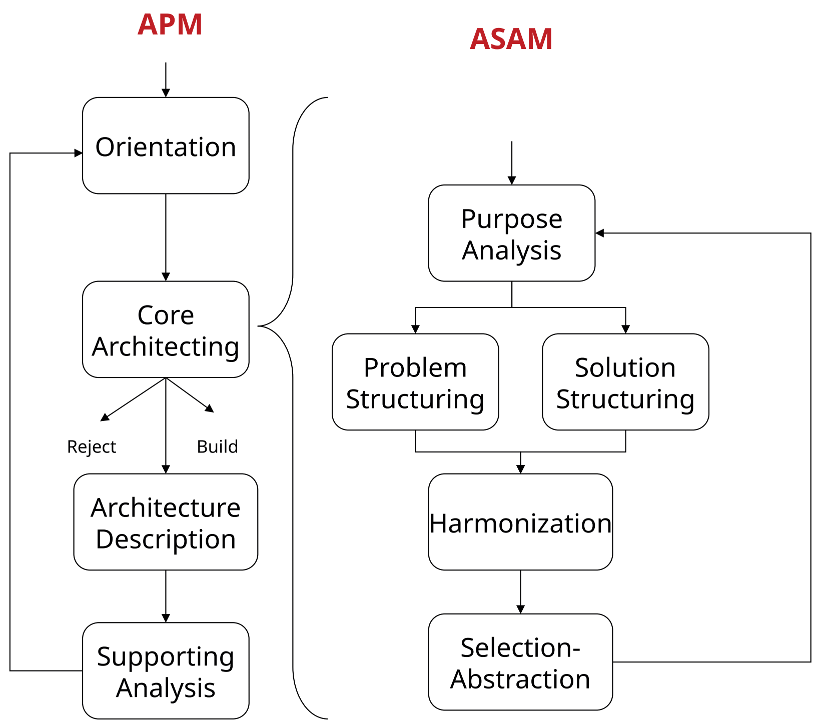


FIGURE 9.1 The activities in the APM-ASAM. The APM organizes the overall process flow, the ASAM is how we define core architecting activities.

Going back to the opening case study on the DARPA Grand Challenge, each team had to make a decision to try and build something (meaning forming a team, seeking funding) or to bow out. Having decided to proceed some technical and programmatic decisions logically follow. What kind of vehicle should be select? What sensors should we start with? Are we building a more complex vehicle that we can only test later in the development cycle or are we forcing ourselves to test early and often even with something very simple?

In other scenarios, like reverse architecting or the future architecture study, the go/no-go is not present (except if we assess that the conditions of the study are so unfavorable that proceeding is not worth it) but we still have to decide what kind of recommendations to develop. Are we recommending a single, decisive configuration or a class of configurations from which a subsequent team will choose?

However this works out, it is not likely to be resolved in a single pass through the APM. We expect that multiple passes will be necessary. This acceptance leads us to one of the first heuristics:

In an architecture study do not rely on a single pass process. Always conduct the study in cycles with the option to repeat cycles several times before deciding to drop the pursuit or move to a more detailed stage of development.

As with all heuristics this comes from inducting on experience rooted in understanding of general principles. A challenge in assessing some general system-of-interest concept is that typically both purpose and solution are intermingled with each effecting the other. There is some assumed purpose up-front (or nobody would be willing to sponsor the effort) but that purpose may be poorly chosen. It may be poorly chosen either for problem domain reasons, like stakeholders don't really value it very much, or because it is a poor match for available technology and developers. By way of analogy, in commercial business a product is successful because it addresses some market niche where there is substantial unrealized value, and a product in that niche can be developed with high quality, low cost, and effective performance. A given organization may have identified a very valuable unserved market but be incapable of delivering a product that effectively addresses it. The organization may be able to develop excellent products, but not ones for which there is a corresponding market segment. Sometimes the relationship between the two is apparent only when both are studied and allowed to interact.

In the authors' experience some architecture studies fail because they study the problem domain deeply and effectively, look at solutions carefully, and realize there is no match only too late in the study schedule to do anything about it. They realize they were studying a poorly framed problem, their solution is irrelevant, and what they learned about the problem domain from the solution study they no longer have time to apply, the allotted study time has nearly run out. Had they understood this earlier the study could have been redirected in time to be effective. Our heuristic on not relying on single pass architecture study forms is a specialization of the very popular "first day" heuristic:

> All the really important mistakes are made the first day

Running architecture studies in cycles gives you multiple "first days." Each pass through the process is an opportunity to have a "do-over" of a sort. At least in a study the cost of a do-over is always bounded to a fraction of the study cost, itself a tiny fraction of what it will likely eventually cost to build a system. A structured process for the beginning of the APM, or essentially the "first day," we call *orientation.*

Orientation

Orientation is the process of understanding the context of an architecting project. Orientation is the set of activities necessary to make a preliminary definition of the project (although the system-of-interest presumably emerges). A simple heuristic to guide orientation is to ask the following questions:

1. What is the system-of-interest to the architecting effort (at least, the assumed system-of-interest)? What sort of system does the sponsor believe will eventually emerge?
   a. Be specific on this, not vague. What do the sponsors think they are going to buy or build? Is buying something immaterial (like services) as acceptable as buying something material?

2. What is the scope of the system-of-interest (and of the overall effort)? Is the system-of-interest a narrowly defined system with a single mission, a complex multimission system, or some assemblage of multiple systems (for example, a family of systems or a collaborative system)?
   a. What things in the space of interest does the sponsor control? The key distinction between a regular system and a system-of-systems is the scope of control. In an open or virtual system-of-systems (Chapter 7) the components are not subject to full central control, by the sponsor or anybody else.
3. What is motivating the investigation into constructing the system-of-interest? Is it the sponsor/user needs, a new technology believed to be able to create value, or some other reason?
   a. A good way to look at this is by characterizing the sponsors and users available for the study. If the sponsors have the resources and authority to build the system-of-interest and the users are present and involved, you have a purpose-driven system. If sponsors are available with funding but you don't have users most likely you have a technology driven system.
4. What is the apparent technology level? That is, is everything needed to accomplish the basic purpose well within current state-of-practice, pushing state-of-practice, or well beyond it?
   a. Two effective ways of assessing this are to look for similar systems and to do a very rough feasibility analysis. If many other systems of similar capabilities exist, then the technology level is within current state of the practice (if not necessarily within state of the practice for the sponsors). If nothing like it exists, why is that? Are there identifiable hard constraints in technology that make the assumed concept infeasible?
5. What hard constraints are believed to exist (like a fixed delivery date)? Are they really hard constraints or just assumptions?
6. What resources are available for the effort, can more be acquired, if so how, and what are the expectations for interacting with the sponsor? Is the sponsor prepared to engage in value discussions with the team, and is sponsor time available to have the discussions?
   a. A sponsor who does not have the resources to build and who is not deeply involved in figuring out how to obtain those resources is probably not going to be able to make any progress. The architect must decide if it is worth adopting somebody else's dream without their help making it real.
7. When the architecture effort is complete, what will be done with its products? Will they be used to start a system acquisition, to guide other acquisitions, to guide research and development (R&D) activities, to mark off completion of a bureaucratic requirement, or for some other purpose?
   a. This should be a fairly obvious question to ask, and yet it often is not asked. Or if it is asked the answer is unrealistic. For example, there are many studies where the team (and sponsors) seriously underestimates the number of people who will have to be convinced at the end in order

to take any action to build a new system, specifically the number and power of other parties beyond the study's sponsor whole will have to be brought on-board.

8. Are the purposes of the system-of-interest, the architecting effort, and the architecture documentation to be developed all consistent with each other?
    a. If they are not consistent it is likely to be a problem. One case of common disjunction is where architecture documentation is being requested primarily to fulfill bureaucratic requirements not to drive design and development. This is rarely a valued adding activity.

Core Architecting

This is the application of the ASAM to the problem area, as elaborated by the orientation process. We take this up in the next major section of the chapter.

Decision-Making

Point #7 in the orientation question list above, "When the architecture project is complete what will be done with the products?" is integral to this element. The reverse is also true, there can be no definition of done if we don't know how the products of the architecture effort will be used. Exactly what kind of decision-making is involved will depend on the organizational context. Some typical examples are:

One-Time Events: This is where an architecture study is set up to make decisions for a single situation. For example, the organization may already have an effective commitment to go ahead with something, but the question is exactly what. An organization could have a legacy system or systems that need replacement, but the replacement approach must be selected. When the study produces a replacement concept that is assessed as passing criteria, or is seen as optimal with sufficient confidence, they can proceed to the next stage. Looking at our examples and case studies, several are of this type. The DARPA Grand Challenge (Case Study 1) is almost a canonical case. The race is a one-time event, the decision to build a vehicle and what sort of vehicle to build is a one-time event. There are, of course, long-term consequences and spin-offs, but the architecting work is generated by a one-time event. The NSOSA case (Maier et al. 2020) concerns a discrete event, the decision to build a set of next generation systems. The organization can anticipate it will re-occur in the generation following, but the study is for a one-time case.

Pipelines and Portfolios: Here the organization maintains a formal pipeline of system concepts, from earliest concept exploration stage to full-scale development. At any given time, some number of concepts are being explored, some have advanced to more detailed exploration, some are in full development, and some are in sustainment or have been spun out to other organizations. This kind of structure is common in both Government and commercial organizations. Advancing from one stage to the next, or retirement to a pool of studied-but-rejected concepts will be based and value, risk, and cost/affordability criteria. One key distinction is whether affordability is assessed in terms of budget availability (a budget allocation model) or return-on-investment and ability to attract or justify new funding. See Maier (2019) for a more detailed example of types of criteria used and examples from the authors' experience.

Research and Development Investments: This is like the previous case, but the resulting investment is in enabling technologies rather than full development. It is accepted that the systems-of-interest will be too technologically immature to build immediately but may be worth investments in enabling technologies. There is a considerable literature on this subject, see Shishko et al. (2004), Gray et al. (2005), Wicht and Szajnfarber (2014) for the relationship to portfolio construction. A consideration at the intersection of architecture and R&D is the role of architecture in identifying real-option paths where R&D advancements would enable substantial capability gains.

Reverse Cases: Here the system-of-interest already exists, and the architecture was initiated because of dissatisfaction with the current architecture, or sometimes because leadership feels the current architecture is not understood. The finish condition is when sufficient understanding has been obtained that the team can make confident recommendations on what to do about the current architecture. This may drive a new study to replace the system, or a development effort to fix the current system, or a decision to accept the limitations as the best mix given the overall objectives of the system and the costs of alternative courses of action. The NSOSA example has been used in this text several times in forward architecting, but it was proceeded by a reverse architecture study. That reverse or as-is architecture description is documented in Maier and Wendoloski (2020).

Architecture Description

Architecture description moves from collections of working models to more formalized groupings structuring as reference documents. Architecture models are organized into a formal architecture description, often using an "Architecture Framework," a concept defined subsequently. The key here is to avoid confusing the architecture description from the work process that precedes it. An architecture description is (or should be) a consequence of good architecting work. We formalize the architecture description in part during the previous core processes (the ASAM component of APM-ASAM) but are likely to want to produce a full description only after we have made the requisite decisions on where to go next. The description can then be tailored to whatever the next stage is, based on the decisions made. If the next stage is a competitive acquisition the desired description will be different from if the next stage is an internal full-scale development and different if the next stage is just put a record in the library.

Outside Information Gathering

In practice, effective architecting often depends on relatively specialized and in-depth data. Recall the heuristic of variable technical depth. Good architecting is typified by deep investigation of particular, narrow areas in subsystems or subdisciplines (the heuristic of *Variable Technical Depth*). These deeper investigations are often done separately from the core process of architecting. A cycle of architecting reveals the areas needing in-depth investigation. The architect sets up the in-depth study, and those results are fed back into further architecting. Typical activities in targeted information gathering include:

- Direct stakeholder interaction. For example, site visits to with users targeted by the proposed system or to where similar or existing systems-of-interest are used. The site visit would include observing activities, focused interviews with key users/managers, reviews of plans, and other formal activities to gather stakeholder information.
- Targeted studies needed to get specific information. Examples include studies of specific technology maturity in critical areas, market surveys for competitive products or the availability of key components, and design studies of specific concepts of interest with sufficient depth to generate defendable cost estimates.
- Rapid prototyping efforts where the prototype is targeted for information gathering either on user value or technological feasibility.

Moving On to Engineering

In some cases, there will be a hard transition out of architecture and into engineering, as where the resulting architecture description is used for a competitive acquisition and contracted system development. This is common in Government system developments and the acquisition may be competitive or sole source. In other cases, the transition may be softer. Recall the spiral and incremental program models discussed previously. If the system is developed in one of these iterative then architecture development is episodic and will re-occur during the process.

Consider the risk driven spiral first described in Chapter 1 (more details are in Chapter 12 and Boehm and Hansen (2001)). Each cycle of this process is intended to resolve the highest remaining risk to achieving the targeted capability level for the system-of-interest. A cycle may resolve its risk as expected, in which case the next cycle can proceed according to the original plan. But some cycles may either have an unfavorable risk resolution (nothing tried worked as well as needed) or an exceptionally favorable resolution (things tried worked better than planned). Either case might have rippling consequences into other parts of the architecture of the system-of-interest. Those rippling consequences may result in architecture reconsideration and then a restructuring of the subsequent spirals, to possibly include abandonment of the effort.

One other element of moving on from architecting is the recognition that sometimes one group's implementation is another group's architecture, and sometimes not. The point about "sometimes not" is that as objectives and design decisions at a higher level in vertical decomposition are allocated down sometimes they leave a wide option space for trades, and sometimes very little. Sometimes the next level down will be faced with their own value-cost-risk driving decisions, and sometimes they will have been precluded. Hence, sometimes there will be architecting to be done recursively, and sometimes it will have done been, de-facto, at the higher level. The idea of vertical decomposition, inherent in systems thinking, recognizes that means-objectives or implementation objectives at an upper level are strategic objectives at the lower level. Objectives, requirements, design elements moved downward, being refined and made more specific as they go. Design is a continuous process of refinement and reduction in abstraction as design moves downward in a system-subsystem hierarchy. We take this up in more detail later in this chapter. Software is especially likely to contain its own architecture decisions. Depending on the point in the design hierarchy where hardware and software divide, the tradespace available to the software may be quite large. When the software is not statically divided onto individual processors in the larger system, but retains a unified identity then there will absolutely be a partially independent software architecture effort.

Architecture decisions are part of this pattern. A value-cost-risk driver identified at one level may well come from a consideration much lower down. Value-cost-risk at the whole system level might be heavily driven by a technology choice found much farther down the hierarchy. For example, the mission effectiveness of a surveillance system might be driven the performance of one sensor type, and the performance of that sensor type might be driven by the performance of one component. The overall value of an electric vehicle might be heavily driven by the energy density of the battery, itself driven by detailed considerations of the battery chemistry. Thus, in architecture we may find the occasional dependence of things at the "top-level" on things much lower down, nominally much farther down in the chain of vertical decomposition.

Within the APM, Figure 9.1, we identified core architecting as itself addressed by what we call the ASAM processes. In implementing the overall APM-ASAM we first summarized the APM elements above, passing over the core architecting stage as something to address in more detail. That detail is the ASAM.

ASAM SUMMARY

The ASAM consists of five activities: Purpose Analysis, Problem Structuring, Solution Structuring, Harmonization, and Selection/Abstraction. Each activity can be elaborated in terms of activity-specific heuristics, analytical methods appropriate to it, and models that document the activity's results or help guide how it is carried out. In this section we give a high-level summary of each activity and methods for organizing them, highlighting the charrette method. The remainder of the chapter goes into detail on heuristics by activity and generic models for representing the results of the activity. Generic models means here models implemented with less formality and in many formats (non-specialized computer tools, large sheets of paper). In Chapter 10, we re-cover the ASAM now defining an approach based on specialized MBSE notations and computer tools. ASAM is intentionally structured to be implementable at multiple levels of formality and with varying levels of tool support. It is entirely possible to implement ASAM, even on a complex problem, on large sheets of paper. There are benefits, of course, to implementation in a more rigorous, tool-supported environment. There are likewise benefits to carrying out the process with a minimal set of tools, just large sheets of paper and marker. Lightweight methods encourage exploring broad alternatives and make it easy to throw things away. Lightweight methods focus the team on the problem at hand rather than the tools used to describe it. Heavier weight methods may draw too much attention to tools over content and encourage premature commitment when the tool environment does not support easy representation of multiple disparate alternatives. As usual, it will depend on the situation.

The five elements of the ASAM, with an emphasis drawn on the activity focus for each element is illustrated in Figure 9.2.

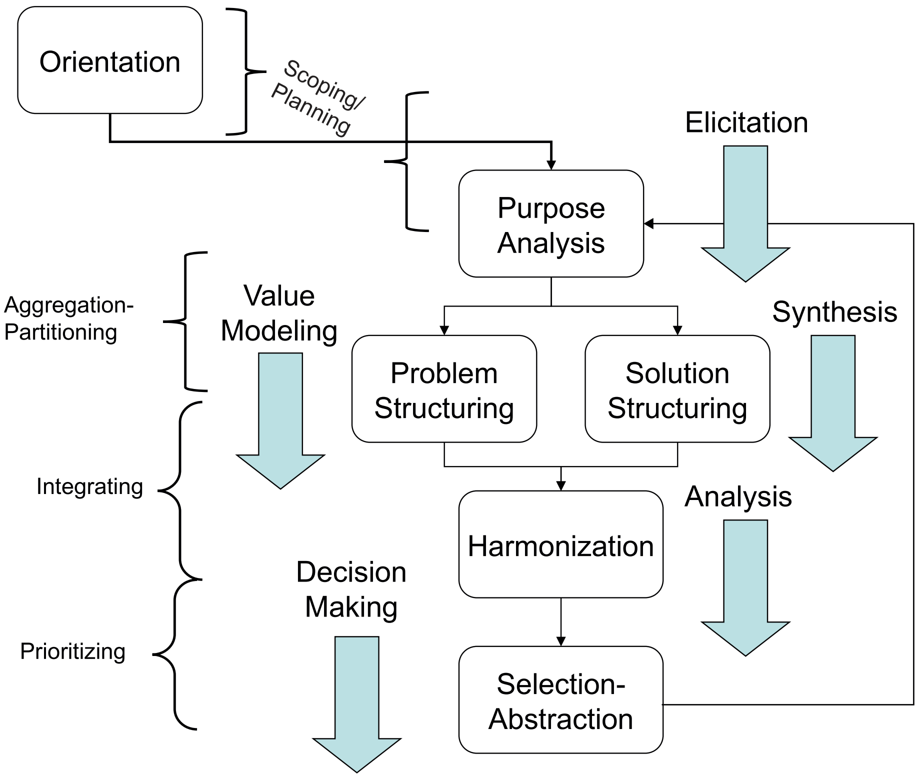


FIGURE 9.2 ASAM in more detail showing the emphasis of each of the elements. The heuristics and tools for each element draws on the established literature from the emphasis areas. We will discuss heuristics for each element, drawing on Appendix A sources as indicated above.

Purpose Analysis

Purpose analysis is the process of determining what the system-of-interest is for, and why the sponsor wants it (or at least believes he or she wants it). The focus is on what value, what value will stakeholders realize from the system. Purpose analysis is a broad-scoped investigation in the system-of-interest. It does not consist of just trying to discover and record assumed requirements or objectives. The intent is to delve more deeply into why the sponsor believes that having the system-of-interest will create value. Purpose analysis explores and challenges the scope of the system, what is included and what is not. It is common, and good practice, to maintain multiple choices for scope and to assess consequences of alternative scoping.

An important concept in architecting and purpose analysis is that we are choosing the problem, not just choosing the solution. This is sometimes hard to accept, there is a bias in engineering that the problem is a given, it comes from some other process, it comes from above the practicing engineer, either the customer or management. The bias is, to an extent, built into engineering education and mindset.

Engineers are typically rewarded for being good problem solvers, and the quality of a solution to a given problem is relatively easy to assess. However, most engineering practice is embedded in a network of problem selection, and the quality of that selection is a key to success.

To best understand this, consider a thought experiment the authors have presented to organization leaders in space exploration and other government and commercial systems. We ask, suppose that your organization was seen as so successful, as delivering so much value, that Congress or other higher authority decided to increase your budget 20% a year for many years running. How long would it be until you ran out of worthwhile things to work on ("worthwhile" interpreted as delivering value commensurate with money expended on previously justified projects)? A common answer is something like "I don't know, but we'd run out of other resources (qualified people, industrial capacity, other facilities) before we ran out of good things to work on." Virtually every time the leaders of the organization believe they could deliver worthwhile return-on-investment with at least double their current budget.

When this is the situation that organization has a problem selection problem. For everything they are told to work on, or choose to work on, there are several other things of equivalent value they are not working on. Most of these organization leaders did not seem themselves as selecting problems, they saw themselves as assigned problems, generally by still higher level managers or outside customers. We've gotten this view even when the person questions was at a very high-level position, at a level one might think would be master of their own fate rather than reactive to outside demands. The point is better problem selection (say problems whose solution has better cost-benefit ratio) could improve overall effectiveness much better than any improvement in solution efficiency. Whether or not that better problem selection was within control of the organization or only higher levels does not matter to the observation, better problem selection might be key to your performance even if you do not have the power to do it.

Problem Structuring

Where purpose analysis is broad based and inclusive, problem structuring seeks to narrow. Purpose analysis accepts the full range of stakeholder inputs, whether precisely stated or not, whether unambiguous or not, and whether feasible or not. If you've done purpose analysis well you deeply understand stakeholder perspectives, both within and beyond what will eventually become the system's scope. Purpose analysis results in a "rich picture," problem structuring results in an ultimately precise model of what problem the system-of-interest will address.

Problem structuring seeks to convert the rich picture of stakeholder concerns from purpose analysis into more rigorously structured models. The conversion process may not be immediate or smooth, it may be necessary to spawn multiple problem descriptions. There fundamentally may not be a single problem to solve; perhaps the best representation of the problem space is as multiple problems that will eventually be separately (if at all) addressed. The most useful heuristics and techniques in problem structuring include problem framing, expansion and contraction heuristics, use-case analysis, and functional decomposition.

A particularly important part of problem structuring is the construction of value models. A value model is an explicit model of how the capability and performance of the system-of-interest maps to measure of value for stakeholders. It will typically have multiple dimensions corresponding to multiple stakeholder concerns. Depending on the needs of the situation it may be quite formal and able to produce explicit cost-benefit analysis, or be informal and more of just a tool to explicitly represent value statements and their consequences.

Solution Structuring

In parallel with problem structuring, we can synthesize solutions. We can do this in parallel (that is, without full knowledge of objectives or requirements) because we assume that exposure to solution alternatives will affect sponsor beliefs about the nature of his or her problem. This is one of the basic tenets of ill-structured problem solving, that exposure to solutions changes perceptions of problems, and because we wish to embrace that change, we must let the two sides of the process influence each other. Moreover, it is a very rare sponsor who does not know what they think they are going to get, in solution terms, and investigating the assumed solutions is inevitable. One of the best ways to introduce alternative solution concepts sponsors may not have considered is comparatively, and that means investigating the expected as well as the unexpected. Solution structuring makes use of the synthesis heuristics, including those for aggregation and partitioning. The products of solution structuring are models of the system-of-interest, and so usually include block diagrams in all their forms and other models of form (discussed elsewhere in this book).

Problem structuring and solution structuring are where the methods of MBSE are most directly suited. The assumption of modeling methods like SysML is biased for well-structured situations. Some tools, like use-cases, are applicable at multiple levels of abstraction and formality and so will see their use even in purpose analysis.

Harmonization

Harmonization is where we match up problem and solution descriptions to determine what can go together and what cannot. Harmonization is analytical, if not always rigorously so. The most useful techniques in harmonization are thus analytical and include functional walkthroughs, performance analysis, and executable simulations. A full analysis in harmonization would include model consistency and completeness analysis and generation of efficient frontier analysis (Varian 1992, Keeney and Raiffa 1993) to identify the achievable trades between cost and capability.

Selection/Abstraction

At some point, we must make choices. One choice might be to drop the whole pursuit (perhaps a very wise choice in some circumstances, and one best made early). If the fundamental purpose is to emerge from architecting and arrange for construction of the system-of-interest, at some point we must select the configuration desired or a class of configurations. One way to consider the outcome of selection is to make an authoritative choice and what is tradespace versus non-tradespace for subsequent design efforts. We will not, in architecting, completely design the system. Following the civil architecture analogy and the concept of architecting presented here, we defer to downstream design and development activities things that do not impact the fundamental value-cost-risk trades. The non-tradespace specification should be of those things that did drive value-cost-risk, but this is normally a small set of the complete design space that must be fully worked through to build a complex system.

One way to formulate the selection is that we select a class of solutions. Hence an architecture can be thought of a class of solutions, a set of solutions sharing key features in common, but still having a substantial associated decision space. The reference our NSOSA example (Maier et al. 2020) illustrates this idea in a significantly complicated case where extensive analysis was brought to bear. The architectures, for example in the reference's Figure 6, are constellations of satellites that share features in how they would be acquired and how functions are allocated to orbits and types of satellites, while leaving open the full mix of capabilities selected.

In family-of-system and collaborative system cases, "abstraction" may be a better concept rather than selection. By abstraction we mean selecting from the family or collaboration the things that are common, and likewise leaving out the things that are not common and leaving those to performing individuals or programs. This can be thought of as an extension of the above idea of architectures as classes of solutions with some elements fixed and others deliberately allowed to vary.

Charrettes, Long Cycles, and Exit Strategies

Cycling through the ASAM (all five elements above) can be done very flexibly. It can be done in a day, a week, a month, or a year. The difference is in the depth of study applied at each step. The ability to do effective cycles of the ASAM in varying times, from fast to as long as it takes to get the requisite depth, is an important architect's skill. The ability to do fast cycles is the ability to look at open-ended problems and quickly sift the promising from the unpromising. If a group can only do cycles taking months to a year, then the minimum investment will be quite large. It will only be possible to investigate a few problem areas. Given the typically highly uncertain nature of early architecting problems this will greatly increase the likelihood effectively studying the right problem area and expensive architecting studies will be expended on unpromising areas and thus be wasted. We find by experience that the least promising can be weeded out relatively quickly most of the time.

Short ASAM cycles are known as "charrettes." The term is borrowed from civil architecture practice and dates to the 19th century. The Wikipedia article (Wikipedia 2024) has a fine history including the appearance of term in the works of Emile Zola. The charrette is an intensive, time-bounded design effort (often kept to days or a week) where designers and stakeholders directly collaborate on both problem and solution exploration. We use it here as shorthand for ASAM cycles that are time bounded (the duration is fixed in advance) to a relatively short time (typically days to 1–2 weeks), and where the schedule and pace are structured to force full end-to-end consideration. Key charrette principles are:

1. Execute the whole ASAM, no part can be left out. Both problem and solution spaces must be investigated.
2. Be time bounded. The schedule is set in advance.
   a. A week is a workable schedule, if all players are present and familiar with the process.
   b. It is often effective (more effective than the fixed week) to use staged periods. For example, a month where the group is not together but devoted to stakeholder interviews and problem space information gathering, followed by a week of intensive ASAM work, a break to let people think (1–4 weeks), and then another week to do the ASAM steps again.
   c. "Middle weight" structures lasting a few months are very useful after an initial charrette and leave room for significant investigation outside of the ASAM process, for example concept design studies aimed at generating defensible cost estimates. The NSOSA program referenced earlier used charrette like cycles of 2–4 months before the full team was engaged on a 6 month design cycle.
3. It is essential that all interests, problem and solution, we represented and be represented by people who know the area and are empowered to speak for the area. If issues are met with "Well, we'll need to go back get an appointment with the bosses" the charrette will not work.
4. Charrettes never produce final answers, they illuminate what to do next. The goal is to identify unpromising and promising avenues, not to settle on a final system concept.
5. Solution and problem exploration should be broad. Do not generate five similar solutions. If you have five similar solutions drop four and start generating more that do not look anything like the ones you have. Similarly, if there are not multiple problem statements convergence is probably too fast.
6. In spite of #5, sometimes it is useful to set up a charrette to explore a targeted area, deliberately narrowing the scope.

While ASAM is the core of both short charrette cycles and long cycles the difference is in the depth of analysis and the type of tools. For example, purpose analysis in a charrette will rely on local expert knowledge of the problem area. In a long cycle there should be a formal process of site visits, stakeholder interviews, and full documentation of findings. When doing solution generation in a charrette brainstorming techniques are appropriate. In a long cycle there should be use of concept design centers (Aguilar and Dawdy 2000), combinatorial generation and optimization (Crawley et al. 2015), and similar methods.

Good use of the APM-ASAM iteratively normally involves making each additional cycle longer, more in-depth, and more focused. Early cycles (like charrettes) don't dive deep and look broadly at the problem and solutions spaces. Each cycle narrows down the problem and solution spaces and allows for greater analytical depth. Most engineering organizations don't have a problem doing thorough, in-depth analysis. The challenge is usually being able to compress the cycle time while maintaining an appropriate and effective level of rigor. In the sections to come on each ASAM element we provide some guidance on the heuristics and tools appropriate to the scenario.

The APM-ASAM must be conducted with an exit strategy in place. Ultimately the exit strategy is to either build something (via whatever contracting or in-house mechanism is appropriate), buy something, or drop what is being studied and move on to something else. The exact form of this should have been understood from the orientation questions discussed earlier. Note that if there is no clarity on what it means to exit, no clarity on what "done" means, then the effort is likely wasted, hence the importance of that part of the orientation process. In the case of portfolio construction it is a good assumption that far more things will be studied than will ever be developed. Most things will be dropped in early cycles, preferably with good documentation of why and the conditions under which it may be useful to reconsider. Study of early phase concepts is assumed to be ongoing, as the team drops one they pick up another. This is rather like a venture capital firm or a movie studio, things are always "in development," and dropping one of them just means picking up another. The portfolio feeding is continuous. In once-through situations the outcome is more binary, we develop the concept to the point we proceed to full-scale develop or drop the whole effort.

INTERLUDE: PROGRESSION AS A FUNDAMENTAL PATTERN

Design of a system, from its earliest conception to construction, can be thought of as a progression, or a series of progressions. One progression is the reduction of abstraction or the creation of detail. The system is seen with very few details in the beginning and as it progresses more and more detail is added. Think of the civil analogy. A build concept starts as shapes and arrangements, lists of features, artistic rendering. As the process progresses the drawings become precise, scaled representations of the exact sizes and physical relationships, and include details of features like electrical circuits and plumbing.

One pattern of progression is from logical behavioral model to technology-specific behavioral model to physical model. There is also hierarchical decomposition within each component. The technology of modules is indeterminate at the top level and becomes technology specific as the hierarchy develops. The progression chain eventually may go to machine tool settings on the factory floor.

There are several types of progression or patterns of model refinement especially important to architecture. One is evaluation refinement, linked to the idea of heuristic progression. The criteria for evaluating a design progress or evolve in the same manner as design models. In evaluation, the desirable progression is from general to system specific to quantitative. For heuristics, the desirable progression is from descriptive and prescriptive qualitative metrics to domain-specific quantitative metrics. This progression is best illustrated by following the progression of a widely recognized heuristic into quantitative metrics within a particular discipline. Start with the partitioning heuristic:

> In partitioning, choose the elements so that they are as independent as possible — that is, elements with low external complexity and high internal complexity.

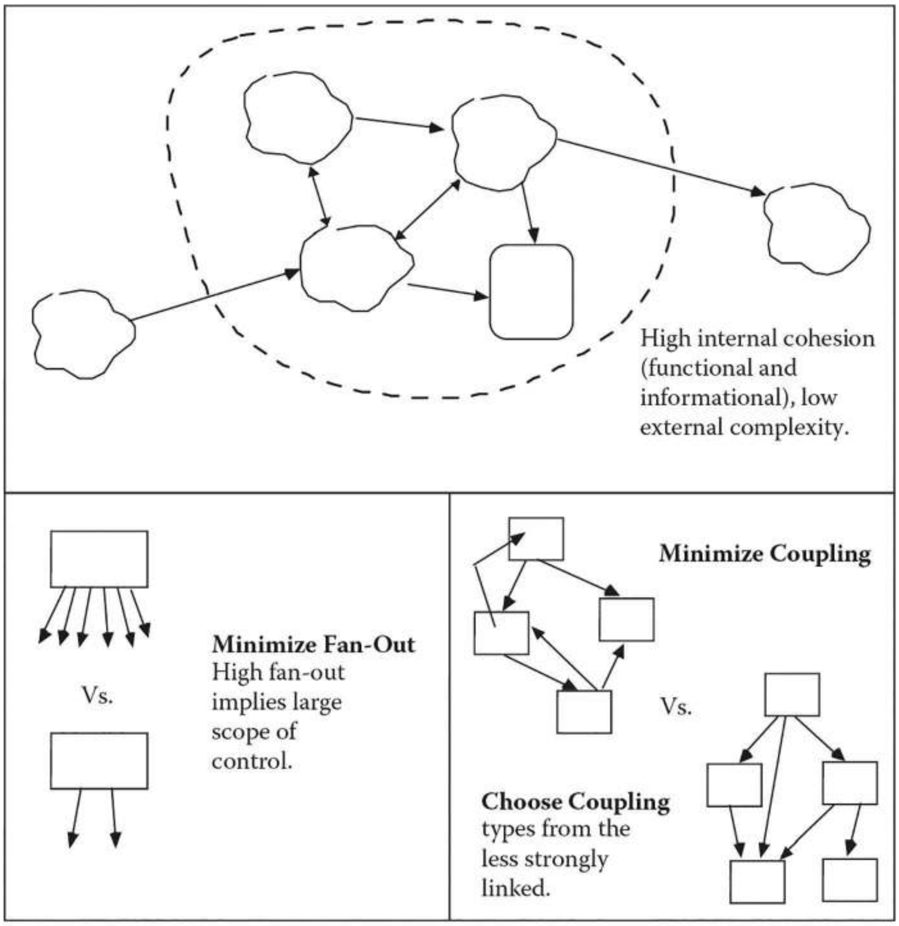

High internal cohesion (functional and informational), low external complexity.
Minimize Fan-Out
High fan-out implies large scope of control.
Vs.
Minimize Coupling
Vs.
Choose Coupling types from the less strongly linked.

FIGURE 9.3 Software refinement of coupling and cohesion heuristic. The general heuristic is refined into a domain-specific set of heuristics.

This heuristic is largely independent of domain. It serves as an evaluation criteria and partitioning guide whether the system is digital hardware, software, human driven, or otherwise. But, the guidance is nonspecific; neither independence nor complexity is defined. By moving to a more restricted domain, computer-based systems in this example, this heuristic refines into more prescriptive and specific guidelines. The literature on structured design for software (or, more generally, computer-based systems) includes several heuristics directly related to the partitioning heuristic (Yourdon and Constantine 1979). The structure of the progression is illustrated in Figure 9.3.

1. Module fan-in should be maximized. Module fan-out should generally not exceed 7 ± 2.
2. The coupling between modules should be, in order of preference, data, data structure, control, common, and content.

3. The cohesion of the functions allocated to a particular module should be, in order of preference, functional/control, sequential, communicational, temporal, periodic, procedural, logical, and coincidental.

These heuristics give complexity and independence more specific form. As the domain restricts even farther, the next step is to refine into quantitative design quality metrics. This level of refinement requires a specific domain and detailed research and is the concern of specialists in each domain. But, to finish the example, the heuristic can be formulated into a quantitative software complexity metric. A very simple example drawing on (Yourdon and Constantine 1979) is:

> Compute a complexity score by summing: One point for each line of code, 2 points for each decision point, 5 points for each external routine call, 2 points for each write to a module variable, 10 points for each write to a global variable.

Early evaluation criteria or heuristics must be as unbounded as the system choices. As the system becomes constrained, so do the evaluation criteria. What was a general heuristic judgment becomes a domain-specific guideline and, finally, a quantitative design metric.

Progression of Emphasis

On a more abstract level, the social or political meaning of a system to its developers also progresses. A system goes from being a product (something new), to a source of profit or something of value, to a policy (something of permanence). In the earliest stages of a system's life, it is most likely viewed as a product. It is something new, an engineering challenge. As it becomes established and its development program progresses, it becomes an object of value to the organization. Once the system exists, it acquires an assumption of permanence. The system, its capabilities, and its actions become part of the organization's nature. To have and operate the system becomes a policy that defines the organization.

With commercial systems, the progression is from product innovation to business profit to corporate process (Bauermeister 1995). Groups innovate something new, successful systems become businesses, and established corporations perpetuate a supersystem that encompasses the system, its ongoing development, and its support. Public systems follow a similar progression. At their inception they are new, at their development they acquire a constituency, and they eventually become a bureaucratic producer of a commodity.

Concurrent Progressions

Other concurrent progressions include risk management, cost estimating, and perceptions of success. Risk management progresses in specificity and goals. Early risk management is primarily heuristic with a mix of rational methods. As prototypes are developed and experiments conducted, risk management mixes with interpretation. Solid information begins to replace engineering estimates. After system construction, risk management shifts to postincident diagnostics. System failures must be diagnosed, which is a process that should end in rational analysis but may have to be guided by heuristic reasoning.

Cost estimating goes through an evolution similar to other evaluation criteria. Unlike other evaluation criteria, cost is a continually evolving characteristic from the systems inception. At the very beginning, the need for an estimate is highest and the information available is lowest. Little information is available because the design is incomplete and no uncertainties have been resolved. As development proceeds, more information is available, both because the design and plans become more complete and because actual costs are incurred. Incurred costs are no longer estimates. When all costs are in (if such an event can actually be identified), there is no longer a need for an estimate. Cost estimating goes through a progression of declining need but of continuously increasing information.

All of the "ilities" are part of their own parallel progressions. These system characteristics are known precisely only when the system is deployed. Reliability, for example, is known exactly when measured in the field. During development, reliability must be estimated from models of the system. Early in the process, the customer's desires for reliability may be well known, but the reliability performance is quite uncertain. As the design progresses to lower levels, the information needed to refine reliability estimates becomes known, including information like parts counts, temperatures, and redundancy.

Perceptions of success evolve from architect to client and back to architect. The architect's initial perception is based on system objectives determined through client interaction. The basic measure of success for the architect becomes successful certification. But once the system is delivered, the client will perceive success on his or her own terms. The project may produce a system that is successfully certified but that nonetheless becomes a disappointment. Success is affected by all other conditions affecting the client at delivery and operation, whether or not anticipated during design.

Episodic Nature

The emphasis on progression appears to be a monotonic process. Architecting begins in judgment, rough models, and heuristics. The heuristics are refined along with the models as the system becomes bounded until rational, disciplinary engineering is reached. In practice, the process is more cyclic or episodic with alternating periods of synthesis, rational analysis, and heuristic problem solving. These episodes occur during system architecting and may appear again in later, domain-specific stages.

The occurrence of the episodes is integral to the architect's process. An architect's design role is not restricted solely to "high-level" considerations. Architects dig down into specific subsystem and domain details where necessary to establish feasibility and determine client-significant performance (see Chapter 1, Figure 1.1 and the associated discussion). The overall process is one of high-level structuring and synthesis (based on heuristic insight) followed by rational analysis of selected details. Facts learned from those analyses may cause reconsideration of high-level synthesis decisions and spark another episode of synthesis and analysis. Eventually, there should be convergence to an architectural configuration and the driving role passes to subsystem engineers.

ASAM HEURISTICS, METHODS, AND MODELS

To use the ASAM effectively to architect we need methods to carry out the goals of each element, and models to capture the decisions or information contributing to decisions in each element. We want to do this in a scalable fashion where we can implement in short, fast cycles (as in the charrette concept) or with in-depth methods appropriate in later cycles when decisions with very high consequences will be made. In general, the in-depth, analytical methods are well documented in other sources. There are likewise a wide range of modeling methods available in the existing literature. What we strive for here is what is less available, a set of organized heuristics for each element and modeling adaptations appropriate for the faster, more agile work done in charrettes. For heuristics we survey a wide set, with several getting detailed attention, as shown in Figure 9.4.

Chapter 10 takes up the details of modeling implementation with MBSE tools. In this chapter we concentrate on more generic modeling representations that can be used in any tool environment from integrated MBSE to large sheets of paper. Particularly significant modeling methods by ASAM element are shown in Figure 9.5.

Purpose Analysis

The objective of purpose analysis to gather a "rich picture" of stakeholders values and preferences, problem scope, solution scope, and other driving assumptions. While requirements analysis is often cited as the opening element in the systems engineering process, requirements gathering and analysis is not part of purpose analysis. At least, requirements gathering is not part of it if you take requirements in the narrow sense, of binary shall statements where satisfaction is either zero or complete. Binary requirements in that narrow sense are typically consequences of a decision on how to buy the system-of-interest, not fundamental properties of that system. Requirements in the broad sense, of stakeholder concerns, needs, and objectives, is very much a part of purpose analysis.

What typically constitutes the "rich picture?" There is no universal list, the contents are what is needed to understand the problem and solution spaces. There are typical products of wide applicability:

- A stakeholder identification list, results of stakeholder interviews
- A stakeholder/concern table
- The list of top-level use-cases or mission threads
- A list of anti-use-cases or other threat scenario representations
- Results of Problem Framing analysis
- As-Is and commonly expected solution models. Results of current deficiency analyses (if they exist).
- Context diagrams and top-level block diagrams, especially those set up to show alternative system-of-interest and enterprise boundaries.

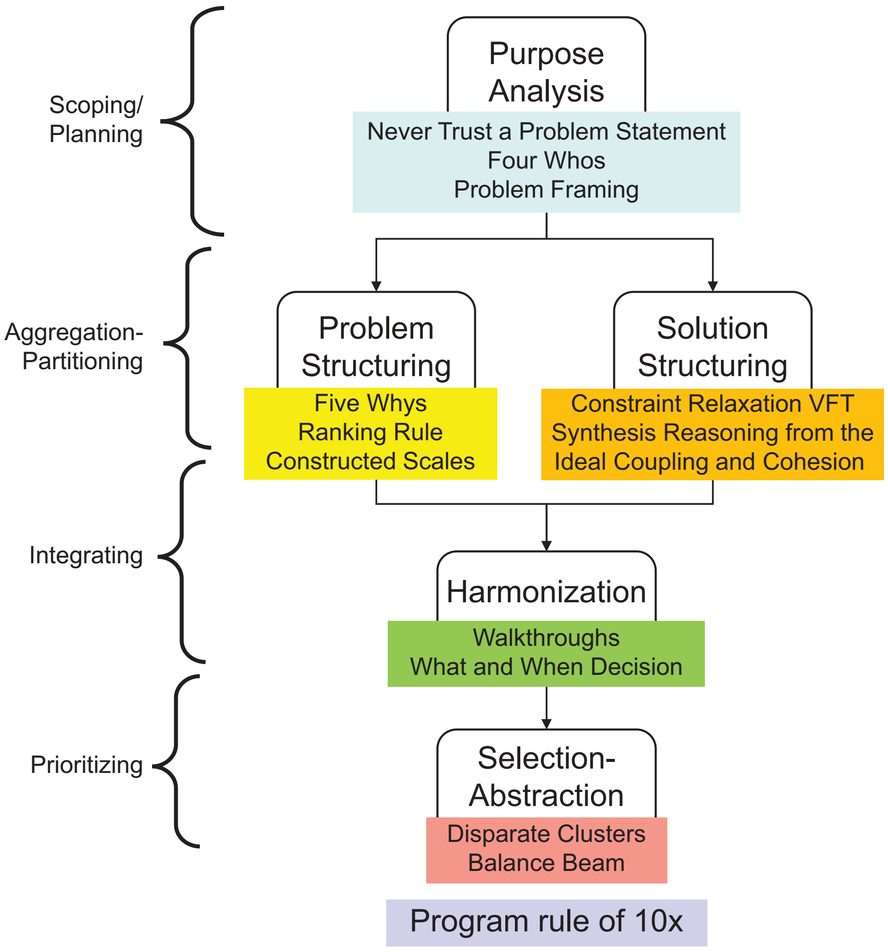


FIGURE 9.4 Orientation and ASAM with specific heuristics discussed and mapping to Appendix A heuristic categories for additional sources.

Heuristics are particularly useful guides to generating these products. Analytical methods are relatively less useful as there simply isn't much analytical in purpose analysis to hang on to. Purpose analysis hangs on stakeholder desires and preferences. While there are analytical methods for quantifying and managing preferences, one first has to know what the preferences are. That is an unstructured task. Given the unlimited scope of purpose analysis (potentially) normative sets of rules are likewise a poor guide. Among the most useful heuristics are:

- The First Day (and related scoping heuristics)
- The Four Who's
- The Five Whys (needs variation)
- The use/support/deploy/test use-case pattern
- The Outside-In heuristic

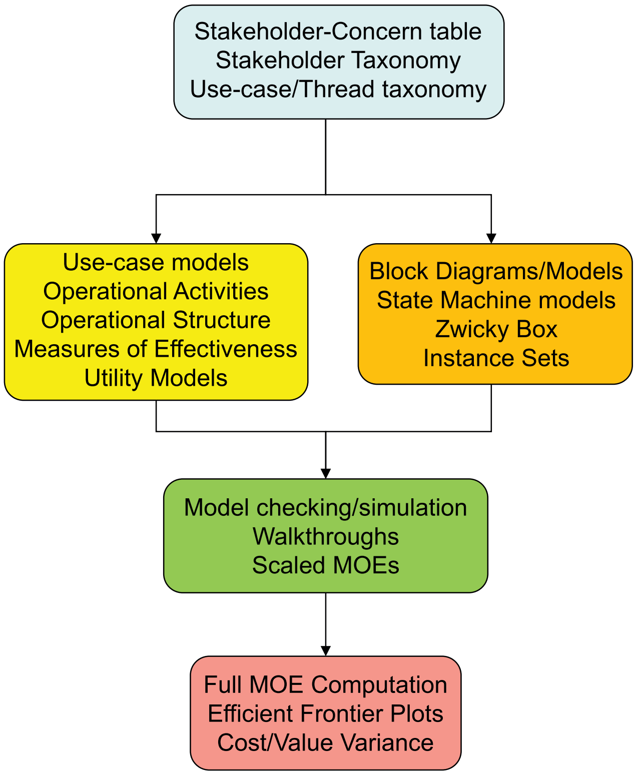


FIGURE 9.5 Models and other description techniques used to describe the results of each ASAM element.

- Problem Framing

Purpose analysis is structured by understanding four intertwined questions, each linked to description products.

1. Who are the stakeholders and what are they concerned about?
2. What do stakeholders (especially sponsors and users) think they are going to get from the system-of-interest, in logical and physical terms?
3. How will the system-of-interest deliver value, and connected is that value to the expectations of what the system-of-interest will be? How does this compare to what there is now (if anything)?
4. What is inside and outside the boundary of interest? Is the boundary of the system-of-interest tightly understood with little variance, or is the scope one of key trade variables?

Stakeholder Identification and Analysis

Stakeholders are those with significant influence over the systems development or use or those impacted by the system-of-interest. One of the most useful heuristics to identify stakeholders is “The Four Who’s.”

To identify a system's stakeholders, always ask: Who benefits, who pays, who supplies, and who loses.

Beneficiaries are obvious stakeholders. The group essentially always includes system users, both direct users and indirect users (those to receive data or services from the system even if they do not use it directly). Who pays is roughly equivalent to sponsors, but sometimes the funders are at a degree of remove from those directly pushing the architecture effort. An important aspect of suppliers as stakeholders is to understand why they are engaged. Are suppliers engaged just to get paid, or for bigger reasons? Is the system-of-interest the supplier's first priority, or their fifth? More than one sponsor has selected a supplier based on the anticipated performance of the supplier's A-team only to find that the supplier's priorities did not warrant more than their C-team. The presence of possible losers can be very important if the losers have strong influence and derail any development effort.

Architecture standards, such as (ISO 2022) recommend consideration of at least a minimum list of stakeholders. In addition to users, sponsors, and suppliers the lists may include acquirers, maintainers, regulators, operators, and architects.

Concerns are what stakeholders care about. Typical concerns include how the systems will deliver value or achieve the stakeholder's objectives, cost and feasibility of system construction, feasibility, risks of development or operation, maintainability, and the ability to evolve or upgrade the system after it is developed and deployed.

By convention in purpose analysis we say stakeholders have "concerns." Concerns are general and wide-ranging. Stakeholders may assert they have requirements, and certainly should be recorded among the concerns. When stakeholders assert requirements it is obviously important to track them, but equally a mistake to assume that anything asserted as a requirement really is, at least without deeper study. At this phase we have accepted that any given stakeholder will be part of the system, much less than any concern they have expressed at this point will be obligatory on the system. That can only be concluded after looking at the full range of stakeholders and concerns and comparing to what is feasible to do.

Stakeholder and concern analysis can be captured as a table, in structured text, or in some adapted diagrammatic forms (some discussion of this is Chapter 10). In building the table the rows are the stakeholders. One column can contain concerns in text form. Some will prefer to add additional columns, for example "role" (Sponsor, user, etc.) or "status" to hold indicators of what level of interaction there has been with the stakeholder (formally discussion with multiple people, information discussion, speculation). In a charrette there is little point to trying to organize the concern material into other than text. In later cycles it may be desirable to breakout the concerns into trackable objects, for example things that can be tracked in a requirements database.

For an example of the stakeholder table for a significant real architecture project see Maier and Wendoloski (2020).

What Will Stakeholders Get (Logical and Physical)?

It is rare for an architecture effort to be initiated with no idea on the sponsor's part of what was going to result. There is almost always some idea of what sort of system-of-interest is expected. The expectation may be more logical, say in terms of more abstract capabilities, or specifically physical. It is important to know what these expectations are. Not because the architect's job is to faithfully implement the expectations, but because the expectations are a strong guide to real preferences. Knowing the expectations allows them to be challenged and redirected.

The physical expectation is the easiest, it is understanding what physical system the stakeholder (especially the sponsor) expects to get. Do they expect to get an airplane, a satellite, a computer system, a software application? The logical system expectation is the abstraction into capabilities or services or information of the physical. An airplane provides something, passenger movement, cargo transfer, a profit stream, a military capability. A satellite likewise provides something, perhaps a communication service or some type of observation. As we'll discuss in sections to come, whatever is provided fits into some value chain or other chain of product and service delivery. Really effective architecting may need to re-envision or re-organize that chain, but we need a point of entry. Most of the detailed restructuring of the chain will be part of Problem Structuring within the ASAM, but it has to rest on a foundation of understanding established in Purpose analysis.

Some heuristics to support this process are obvious, starting with "just ask." Ask what system they expect to see, ask about its features, its inputs and outputs, and how it compares to anything they have now. Coupled to asking, a powerful heuristic is the "Five Whys" in the "wants" version. The Five Whys heuristic, classically defined in Ohno (2019) and Rechtin (1991), is about analyzing defects or failures.

> To find the root cause of a failure or defect you must as "why did it happen?" at least five times.

In the "wants" version you ask, "why do you want that?" at least five times. The idea is to get past a superficial understanding of why some feature of a system is valuable (this links strongly with the value proposition ideas of the next section) and to the deeper reasons. In particular, you are looking to get beyond "I want it because [it leads to these other things]" and instead get to "I just want it," or "That's my root mission," or some other expression of fundamental preference. This is one way to get at the concept of means-objectives versus fundamental objectives (Keeney 1996). Means-objectives represent the preference for things that are a means to some desired end. Fundamental objectives are the ends, there is nothing deeper, they represent the fundamental things desired, at least within the frame of the stakeholder. The deeper reasons may allow the solution space to be significantly rethought. Put another way, this is part of clarifying what problem we are solving and where fixation on a solution concept is masking a better interpretation of the problem. The classic heuristic on this is:

> Often the most striking and innovative solutions come from realizing that your concept of the problem was wrong.
>
> *Raymond (1999).*

Problem Framing is a heuristic or mini-process best applied at during purpose analysis. Problem Framing suggests asking two coupled questions, which can be elaborated through a slightly more involved process.

- How could you get what you are asking for but not get what you want? Put another way, why might getting your assumed solution be unsatisfactory?
- How could you *not* get what you are asking for but still get what you want? Put another way, how could failing to get the assumed solution still lead to success?

One of the authors was once involved in an assessment of a solution to a business problem involving very complex database technology. The technology was being considered to synchronize databases across globally distributed sites. It was important to the users that the databases each user saw in his own location be accurately synchronized with the databases seen by other users elsewhere. Because of size and continuous use this was very challenging (at the time of the assessment). After extended discussion, one of the outside participants asked "How did we get into this mess? Why don't you just have one database and have everybody access it?" The reason that was considered "not possible" was insufficient international communications capacity, a single database would not provide sufficient performance at all of the global sites because of communication network limitations. The natural question was why not buy more network capacity? Granted, international capacity is expensive, but the database solution being considered was likewise expensive and technologically risky as well. The answer was "International communication capacity comes out of a different budget, and we can't trade that budget for this." In this case, the ultimate owner was a single, commercial company, and so those budgets could be traded, if one went high enough in the corporate hierarchy.

Problem framing revealed that the problem was not database synchronization; it was the operating pattern of the researchers and their inability to purchase certain types of assets because of internal rules. They could have bought complex database technology and the result might still have been unsatisfactory performance on the actual business use-cases. Alternatively, there were other things that could have been bought not resembling the assumed solution at all that might have yielded excellent performance against the real, underlying problem.

Problem framing with a narrow focus on specific assumed solutions can reveal useful alternatives blocked by poor assumptions. Problem framing with a wide consideration of the whole situation may reveal the whole premise of the architecture study to be flawed. Whether or not this is a welcome insight depends on the situation. Either way, this is an example of the essential "first day" heuristic.

All the really important mistakes are made the first day.

Typical examples of serious "first day" mistakes are choosing the wrong problem to solve, ignoring better alternative problems, and poor scoping. We take up the scoping challenge in a subsequent section.

Assuming problem framing does not entirely shift the perspective on the situation, having any notion of assumed solutions can help start development of a use-case or mission thread model. We use the two terms, use-case and mission thread, synonymously. They refer to descriptions of end-to-end threads, from input to output, of things the system-of-interest should do. In Problem Structuring we take up developing detailed use-case descriptions, and in Chapter 10, we take up SysML and other modeling methods for use-cases or mission threads. In SysML a use-case is a specific modeling element and may be used somewhat distinctly from modeling elements used to cover mission threads. At the Purpose Analysis stage the primary goal should be to develop lists of end-to-end threads, either from a logical or physical perspective or both.

A useful heuristic for identifying threads is Outside-In. The Outside-In heuristic is:

> To document a thread through the system choose an input-output pair and trace the pathway (logical or physical) between them, starting from both ends and working toward the middle.

The distinction between the logical and physical pathways is whether we consider the inputs and outputs in problem domain or solution domain terms. Consider a command-and-control system. It consumes external information on the status of things of interest, and commands from higher authorities. It provides consolidated, analyzed status information to those authorities and processed commands to the entities being commanded, whatever they might be. A command or a status can be considered either as the information it contains (a logical perspective) or as the physical message that communicates it (a physical perspective).

A logical thread would consist of connecting information elements defining status and commands in problem domain terms, terms relevant to the sponsors and users, to each other through a series of other information elements and transformations. A physical thread would follow physical messages, encoded for physical communication systems, into and out of the system. Following the spirit of purpose analysis, the logical description is strongly preferred, it should be the description most relevant to the primary audience. However, the physical description may be as important, especially when the situation is dominated by legacy systems and any new system will realistically be required to interact with the legacy systems.

An informal representation of the set of use-cases can be as simple as a textual list, each entry annotated with a brief purpose and description. The formal approach will be SysML use-case or activity models or equivalent forms in other notations. We define patterns for these in Chapter 10.

An important special case of use-cases is anti-use-cases. An anti-use-case is a use-case written from the perspective of an attacker of the system-of-interest. It describes what an attacker is trying to do to the system rather than how that threat may be blocked. These are typically quite challenging to write well and require deep domain knowledge, but they are very important. Well-written anti-use-cases define functionally what an attacker is trying to accomplish and force comprehensive consideration of defensive measures rather than generic checklists of security or defensive measures. Anti-use-cases are founded in threat analysis and an understanding of the operational context of the system. Anti-use-cases are equally as valid for military, other governmental, and commercial systems. All systems face threats of theft, disruption, or damage, though the attackers, their resources, and motivations will all be different.

The Value Proposition

Building a system requires resources, resources that could have been expended on something else. We choose to build a system because we believe the value in having it will exceed the value of the resources expended to build it. Architecting supports a resource allocation decision, and its foundation is estimation of value. Broadly speaking, systems deliver value through the functionality they provide and through the performance attributes of those functions. A commercial airplane provides certain functions, like being loaded, taking off, flying the mission profile, landing, etc. Of course, a modern commercial airplane is very functionally complex, but at the user-facing, sponsor-facing level the value-delivering functions are simple. Those functions are provided with some performance attributes such as range, capacity, takeoff and landing constraints (indicating what airports it can use), reliability, maintenance hours per operating hour, and various others. The foundation for understanding the value provided by a system-of-interest is understanding this relationship between the functions and the performance attributes. For some systems value will be dominated by the set of functions they provide, and the challenge will be including greater functionality. For others the function set will be well-defined, but the attributes of those functions will be complex and will dominate the value proposition.

In Problem Structuring we will seek, eventually, to build a quantitative value model. The foundation of the value model in Purpose Analysis will be to know what attributes to measure. Linking to the outside-in development of threads discussed above, for each thread ask "what are the criteria for measuring how well this thread is accomplished?" In the simplest case the answer is binary, either the thread is present or it is not, and if it is present then we are satisfied. More likely stakeholders can identify how success on a thread is measured. In Purpose Analysis our goal is to capture how stakeholders think about success measures in their language. Whether that fits conveniently with, for example, requirements formats or utility models is not relevant, only that the architects understand in problem domain terms. It is not the task in Purpose Analysis to rewrite stakeholder concerns into proper requirements or multi-attribute utility models. It is the task in Purpose Analysis to understand so that can be translated into well-structured from in subsequent steps. This information will typically be captured as annotations on the thread list.

Attention to performance attributes is synergistic with attention to acceptance criteria. This builds on the idea that what defines a system development's success (the system's acceptance criteria) is a proxy for the success of the whole effort (for better or for worse). A heuristic that captures this is:

Ask early about how you will evaluate the success of your efforts.

Hayes-Roth and Waterman (1983)

Scope and Boundaries

The last of the considerations in Purpose Analysis is understanding the scope or boundaries of the system. What is inside the system-of-interest and is subject to design and trade, and what is outside the system and should be considered invariant, or at least not subject to design choice and control? Making a mistake in scope can be one of the biggest mistakes in architecting. Echoing the famous real-estate saying, a heuristic is "The three most important things in an architecture are scope, scope, and scope." A well-scoped system is one that is both desirable and feasible to build, the essential definition of success in systems architecting.

One thing that distinguished systems architecting from systems engineering is how we treat scope. Systems engineering typically treats scope as determined and fixed. It is essential to carefully define the scope, define the external interfaces that cross the internal-external boundary, and rigorously control those interfaces. In systems architecting we treat scope and the internal-external boundary as something that should be traded and as a point of great leverage. A common reaction to finding a problem and solution are overly complex is to descope, to try and resolve something simpler. But this is not necessarily the best course of action.

Moving to a larger purpose widens the range of solutions

Nadler and Hibino (1998)

A bigger scope can open the solution space, possibly to more feasible solutions. A more general lesson here is that Purpose Analysis should not attempt (initially) to identify the single, best problem statement, it should identify problem statements, emphasis on the plural. Treat the problem statement as something else on which deliberate choice will be exercised. As you carry multiple solution concepts to choose among, carry multiple problem statements to choose among. The ultimate choice might be more than one, just as we sometimes build more than one solution, if carrying different concepts to implementation is warranted by the value proposition and the risk posture. While an architecture effort might start by assuming there is a singular problem in need of solution, more than one might emerge. If each has a sufficient stakeholder base with sufficient resources, then this might be seen as an even greater success that resolving the initial problem statement.

Scope wraps back to the beginning, stakeholder analysis, as we ask who will judge success. An architecture project might start with well defined, closed team. But consideration of scope and value might challenge that. Another classic heuristic is:

Success is defined by the beholder, not by the architect.

Many systems are ultimately judged by an audience quite different than that for the conception of the system. Some architects have been wise enough to realize this. Consider the story of GPS in Chapter Case Study 6. As conceived it was a U.S. Air Force program, with a Defense Department wide audience and user base. As implemented today the user base is multiple billions of people, most of them not U.S. citizens much less U.S. Department of Defense personnel. The GPS architects, exploiting the history of the Transit system, realized the possibility for a far wider audience and included features to foster it. In effect, they allowed for the ultimate scope to radically change and for the system to have features to enable and exploit that.

Application to Examples

A Purpose Analysis documentation package can be substantial, if it includes things like interview transcripts for all key stakeholders. On a major project, one where hundreds of millions to billions of U.S. dollars are on the line, there should be a documentary record of stakeholder background. If for no other reason than over time as the project team gets larger and larger that information will be lost unless there is a way to refer to it. For the purposes of this book we have some vignettes of applying the methods above to real problems. These are deliberately vignettes of Purpose Analysis, not comprehensive accounts. They illustrate how Purpose Analysis methods or data elements can be realistically gathered and then used on problems of real scope.

Consider again the DARPA Grand Challenge with which we opened the book. Imagine you are part of a University based team considering entering the first race, before there is any history of such races to refer to. The obvious initial problem statement is simple, design and build an autonomous vehicle to complete an approximately 100-mile unknown course in as fast a time as possible. From a purpose analysis perspective, should we participate, and if we do participate, how should we structure our entry to maximize the value be get for resources expended?

The stakeholder analysis turns on where the resources (mainly money) needed is coming from. Let's assume there is enough internal (University) discretionary money available to at least get started, but that we'll also have to raise money to make an entry. The stakeholder analysis can then focus on two groups: Those who will do the work (faculty, students, staff) and those who will provide money or other material donations. In this case, given the nature of the DARPA challenge, we won't be seeking government grants, the sponsors will primarily be companies. This immediately tells us that winning the race is a means objective not a strategic objective. By itself winning the race, while nice, does not achieve any strategic objectives of any of the stakeholders. It does not, by itself, advance an academic career, provide more research funding, or advance a corporate objective. It would be fine publicity, for sure, and that should be factored in as addressing a concern such as "increase University reputation" that might be held by a University administrator stakeholder.

Problem framing very helpfully clarifies some of these issues. Consider the following thought experiment. Suppose we built a vehicle that, based on innovative mechanical engineering which enabled it to traverse very complex terrain, could win the race even though it had very limited navigational capabilities. Suppose it just followed a GPS track rolling over or around any obstacle encountered. Such a vehicle could fulfill the nominal problem statement, but how would it deliver long-term value? That is, would it "win" but not achieve our objectives? Assuming that a solution that "solves" the autonomous vehicle problem by rethinking what a vehicle is, and further assuming that the vehicle rethinking is not generalizable to making a conventional vehicle with large carrying capacity able to perform missions, we see that it likely would not have much carry over. By doing that we might make a short-term win in a few papers and reputation but would not meaningfully advance autonomous vehicle technology in general.

In contrast suppose we built a very smart system for machine vision and autonomous decision-making that did not win the race, but that demonstrated very effective autonomous driving of a conventional vehicle, albeit too slow to win under DARPA's constraints. That would be recognized as a major advance with wide possibilities for further development and improvement. In terms of advancing our position as providers of autonomous vehicle technology this would be a large win, even though it did not win the race.

In the introduction of the DARPA Grand Challenge case study we identified some key technical architectural decisions (choice of vehicle, choice of sensors, choice of program architecture) that would follow a decision to participate. While those decisions are not part of Purpose Analysis, the discussion of the data generated in Purpose Analysis should show that it would be highly relevant to helping us to make those decisions.

The NOAA Satellite Observing System Architecture study (NOAA 2019) provides examples in using the purpose analysis heuristics. We previously cited the publication of the associated stakeholder table. The weather satellite constellation is also an example of choice in where to model the value proposition, eventually resulting in quantitative value model (Anthes et al. 2019). At the initial point there was a well-vetted set of requirements available, known as the Consolidated Observational User Requirements List (COURL). We could conclude that the purpose of the weather satellite constellation is to fulfill those requirements, and thus use fulfillment of the COURL list as the value model. However, this was found to be misleading. The sponsor (owner) of the U.S. weather satellite constellation is the National Oceanic and Atmospheric Administration (NOAA). NOAA has a variety of missions to fulfill. The COURL observations are means to fulfilling those missions, not the mission themselves. For example, providing weather forecasts, and very specifically watches and warnings of dangerous events, is a core mission. Satellite observations are very useful to that mission, but satellite observations do not directly fulfill that mission. Only by combining satellite observations with other observations, assimilating into numerical models, and carrying out other processes are high-quality forecasts delivered (Bauer et al. 2015). To succeed at the sponsor-mission level it is insufficient to better satisfy COURL requirements than the current constellation satisfies them, it is necessary to improve observations in areas that lead to forecast improvements. The relationship between observational quality and forecast quality is non-linear and very complex (Bauer et al. 2015, Palmer 2020). Moreover, there are multiple types of forecasts (e.g., hurricanes, space weather, winter weather, severe storms) relevant to warning and watch quality and each is tied to distinct, but sometimes overlapping, sets of observations.

The outcome of this analysis was a collection of mission threads, each tied to the delivery of a class of space-based observation (Anthes et al. 2019). Because space observation of environmental phenomena is relatively simple (most observations are provided by sensors operating in a vacuum-cleaner mode rather than a focused commanded mode) each thread is functionally simple but the quality measure on the threads are not so simple. Global weather imagery and global soundings are both collected in a functionally very similar way, but the measures of their quality (what determines their utility in forecasting models) are not nearly so similar or simple.

Problem Structuring

Purpose Analysis takes a broad view and considers both problem space and solution space. Its ambition is to broadly and deeply understand who the stakeholder are and what they care about. The rich picture idea emphasizes embracing the stakeholders' perspectives as they are, with whatever ambiguities and inconsistencies real stakeholders have. Problem Structuring, the next elements, reverses the emphasis. The emphasis narrows to the problem space and seeks to drive out ambiguity and imprecision. Because we are in the architecting phase it may not be possible, or desirable, to immediately narrow down to a single conception of the problem. It may be best to carry multiple problem descriptions for several study cycles with the understanding that we are ultimately selecting the problem as much as we are selecting a solution (or class of solutions, given that we are architecting).

Two heuristics provide a good lead-in to understanding problem structuring.

> Problem structuring has been done well when it is possible to objectively rank any solution alternative in the value system of each stakeholder

> Hang on to the agony of decision as long as possible

The first recognizes the inherent diversity in values for stakeholders. Values are possessed by stakeholders, they are not abstract. Something that is valuable for one may be non-valuable for another. The second reminds us to avoid premature decision-making in architecting. The ambiguity and uncertainty inherent in architecting are very frustrating to many people. It induces a desire to make some decisions, even if they are poor, just to make forward progress. Time is valuable of course, and not making decisions in the vain hope that somebody else will make some is a failure pattern. What the heuristic really says is to be aware of when you really need to cut off options. When you really must cut off options, or when the answer becomes clear, go ahead.

Sometimes the least expensive way to resolve some large uncertainty is to try something and see how it works. A prototype may be a cheaper path to resolution of something in the problem space than still more analysis. Hence the heuristic:

> Firm commitments are best made after the prototype works
>
> *Rechtin (1991)*

This is a further reason to emphasis an episodic approach to architecting. We may not fully understand the problem space until we have experience with what a system delivers, until user-stakeholders have experience with what a new system can deliver.

Problem structuring normally works along two inter-related tracks: Functional elaboration and value model construction. Functional elaboration builds on the use-case/thread list discussed in purpose analysis and fills in the relevant details. A value model expresses the relative or absolute value of an alternative solution in terms of attributes or values provided by that solution. The two tracks inter-relate in that the possession of a given function is itself an attribute of the system-of-interest that carries value. If a system concept possesses the function, it gets the value, if it does not possess the function it does not get the associated value. Many value properties are measures on functions, for example the quality or quantity of the data provided by the function.

One of the challenges in purpose analysis in architecting is identifying and restricting study to the important, value-delivering aspects and not getting lost in engineering details that should be deferred to later phases of development. Consider the NSOSA weather satellite constellation example referred to before. Each weather satellite is a complex object possessing many functions. Most of those functions relate to the safe and reliable operation of the satellite. These are things in deployment, initialization, safing, day-to-day operations, fault detection, and so on. From the perspective of the mission, which is providing data to support weather forecasts, the important functions are to collect and transport observation data. The value comes from what observation types the satellite collects, the quality of the data collected, the timeliness with which it is transported, and the reliability with which the data is provided. All the other functions, necessary as they may be to the operation of the satellite and thus gaining the valued data, are still means to the end, not the end itself. Means to an end don't carry direct value. Functions related to the operational mission, collecting observations, are the concern in architecting. Quality metrics on those observations are the concern of architecting. Functions relating to non-observational operations and housekeeping are not architectural concerns.

Some readers may object that the satellites must, at least eventually, be fully functionally described, and poor functional description may cause the satellite to fail providing no observations. This is true, but we still argue these are not *architectural concerns* because they do not drive value, cost, or risk. Not driving value is the argument just made, the value is in environmental observations. Not driving cost or risk is contextual, we see it because weather satellites are relatively mature. We have built and flown weather satellites effectively for decades. There are no essential challenges in their design and construction. The cost of weather satellites is driven by the observational instrument quality not by risks in the satellite bus (there is no reason to use immature technology to build satellite buses that have proven effective for decades). In other contexts this might not be true. In some contexts the operations of deployment, initialization, and so forth might be greatly challenging and a major source of cost or risk. The architect must have a level of domain knowledge to make this judgment.

As an example, some types of observations might best be made operating in a very hostile environment (say a high radiation zone). Operating in that zone itself is not value-delivering, only the consequences of being there, getting the desired observation is valuable. In this case the decision to operate in that environment could be architectural, but because it drove cost or risk not value. Cost and risk driving choices certainly can be architectural, even when the linkage to value is indirect.

Returning to the NSOSA example, the value model contains many relatively simple functions each with an attached set of quality metrics. Each observation function is essentially just "continuously collect observation XYZ and transport the data to user assimilation networks." Whether the observation is imagery, vertical sounding, radio occultation, or in situ radiation measurements the function is essentially the same. The quality measures that tell us how valuable the data collected is do differ considerably. In assessing the complexity of the system, and determining value, it is the specific observation performance values that are at issue, not any complexity in this function itself.

Some systems will be characterized by functional complexity where having large collections of use-cases/threads with extensive elaboration will be very important. Data assimilation, tracking, and planning systems are often of this type. In others the value will be driven by the quality/performance achieved on relatively simple functions. The NSOSA case is very much the latter where value is driven by quality/performance on a set of simple functions.

A framework for constructing value models with either emphasis is multi-attribute utility theory. The literature on MAUT is large, and much of it emphasizes aspects that are rarely of architectural concern. Our approach is a pragmatic mixture of heuristic and rigor and is greatly influenced by the work of Keeney (Keeney and Raiffa 1993, Keeney 1996, Hammond et al. 2015). One of Keeney's observations we emphasize is that structuring the decision problem before trying to find an optimal decision often renders the optimization unnecessary (Keeney 2004). Optimization is over-rated in decision analysis, deep analysis and understanding of the problems if under-rated. The classic MAUT framework tells us to:

1. Carefully define the problem requiring a decision. Structure the problem description by identifying a set of "objectives," where each objective is something that delivers value as performance on that objective is maximized or minimized.
2. Build scales of measure for each objective. Scales can be natural (like mass or range) or artificially constructed, as would be a scale for "flexibility" or "scalability." Establish on those scales a trade-able range, the range of values of interest for the decision problem.
3. Define the alternative set, the set of possible decisions. For our purposes that will be the subject of solution structuring.
4. Analyze the consequences of each decision choice. This means analyze how each alternative identified would perform on the objectives.
5. Assuming the answer is not obvious at this point (although an obvious answer often has become apparent by this point) quantify tradeoffs between higher and lower performance levels on objectives where performance differs.
6. For the fullest rigor, determine a utility function, a function that combines all the attributes and represents overall client satisfaction. Weighted, additive utility functions are commonly used, but not required.
7. Include uncertainty by determining probabilities, calculating the utility probability distribution, and determining the client's risk aversion curve. The risk aversion curve is a utility theory quantity that measures the client's willingness to trade risk for return.
8. Select the decision with the highest weighted expected utility.

Keeney makes a number of heuristic observations about this framework that simultaneously can call the framework into question, and help use versions of the framework in practice (Keeney 1982, 1996, 2004, Hammond et al. 1998,).

- Clarifying the problem (purpose analysis) and developing objectives (part of problem structuring) frequently make the best course of action obvious without doing the full analysis.
- Expanding and improving the alternative set (better solutions) often yields better solutions than trying to find the optimum from the initially chosen set of alternatives (which may all be poor).
- The scale of real problems (number of objectives and alternatives) is often large enough that heuristic simplifications are necessary to make it tractable. The quality of the heuristic simplifications then become drivers in the quality of the decision over optimization procedures.
- Constructing additional, better alternatives once the consequences are known is often much more effective than trying to carefully quantify tradeoffs.
- Tradeoff analysis is fundamentally limited by the likelihood that stakeholders will have different values.
- Extensive analysis to find an optimal solution is only of high value when the optimum is hard to find (non-obvious even after all of the problem structuring) and enough better than nearby alternatives to exceed uncertainty bounds. This is pretty rare in technical design situations.
- While uncertainty is technically quantitative through probability, the reality is usually highly subjective. Situations are often unprecedented or with data much thinner than the probabilities of relevance. Risk aversion curves are an intellectual construct and eliciting real people for them often generates inconsistent results (if you can get people to sit through the elicitation at all). The whole process may look analytical, but largely be a gloss.

One clear benefit of the decision theory framework is that it makes the decision criteria explicit and, thus, subject to direct criticism and discussion. This beneficial explicitness can be obtained without the full MAUT framework. This approach is to drop the analytic gloss, make decisions based on heuristics and architectural judgment, but (and this is more honored in the breach) require the basis be explicitly given and recorded.

The problem that with multiple stakeholders there will be multiple value systems and multiple preference orders is real. Single clients can be assumed to have consistent preferences and, hence, consistent utility functions. However, consistent utility functions do not generally exist when the client or user is a group, as in sociotechnical systems (Mueller 2003). Even with single clients, value judgments may change, especially after the system is delivered and the client acquires direct experience.

In practice there is again an advantage here in the MAUT framework making explicit what is otherwise hidden. We can be explicit in attaching different tradeoffs to different stakeholders. We can see how different choices are more, or less, appealing to different audiences. This can lead to practical but still analytically based discussions of collective strategies (prioritize multi-way satisficing, emphasize the occasional “home run” improvement?).

Rational and analytical methods can produce a gloss of certainty while hiding highly subjective choices. No hard and fast guideline exists for choosing between analytical choice and heuristic choice when unquantified uncertainties exist. Certainly, when the situation is well understood and uncertainties can be statistically measured, the decision theoretic framework is appropriate. When even the right questions are in doubt, it adds little to the process to force quantification. Intermediate conditions call for intermediate criteria and methods. For example, a system might have as client objectives “be flexible” and “leave in options.” Obviously, these criteria are open to interpretation. The refinement approach is to derive or specialize increasingly specific criteria from very general criteria. This process creates a continuous progression of evaluation criteria from general to specific and eventually measurable.

Heuristics and Guidelines for Problem Structuring

Problem structuring is the canonical problem in bridging domains. Purpose analysis requires understanding the user/customer domain with only loose understanding of solutions. Some solution understanding is required, certainly, but not in-depth. Solution structuring, to come next, is requires deep understanding of what can be built or bought. It is rooted firmly in engineering. Problem structuring primarily addresses the problem domain, but it has to do so in a way that is relevant to the eventual definition of solutions. It is about the bridging from problem to solution. One signature of this is that the intellectual span is often quite large. The architect needs to understand the problem domain side (military operations, weather forecasting, a commercial market) as well as technical tools for describing solutions (data models, finite state behavior, technology limits on performance).

Guideline: Choose a Top-Down or Bottom-Up Value Focus

Problem structuring is about understanding the value of different alternatives to stakeholders. There two broad ways to do this, top-down and bottom-up. In top-down you start with an overall mission objectives, “air superiority,” “weather forecasting improvement,” “zero emission transportation,” and work out specific objectives that correlate to achievement of the mission. In bottom-up you start with users/customers and build on what their specific objectives are within the general problem area.

Guideline: Model the System Boundary

What is inside the system-of-interest, and what is not and thus outside the responsibility and power of the system developer, is an essential choice. There is always a boundary, every system that can be built has to leave much to the environment. In problem structuring always build a model of what is inside and what it outside the system-of-interest. If inside/outside subject to trade, then build a model that encompasses the trade space, or several models each one representing a discrete choice of system boundary.

Representing the system boundary informally one can use simple block diagrams or just text listings. More formally there are several MBSE diagram types, including context diagrams, SV-1 diagrams, and several forms of taxonomy diagrams. We go through these in Chapter 10.

Guideline: Build a Logical Model of What the System Does

A logical model means a model of system-of-interest inputs and outputs in problem domain or user terminology and concepts. This model should capture the system's desired or required functionality, described in terms relevant to the users and sponsors, as opposed to engineering terms. So, for example, inputs and outputs would be in terms of the information conveyed not the bits and bytes by which the information is conveyed, unless the users specifically care about the specific bits and bytes. Occasionally that will be the case.

The beginnings of this model were discussed in purpose analysis. During Purpose Analysis, you build lists of use-case or mission threads and do some detailed development. During problem structuring the team should prepare detailed use-cases for all significant behaviors corresponding to stakeholder concerns. As discussed in Chapter 10, a more complete use-case contains a dialog of how external entities interact with the system and what the system does or produces in response. Alternatively, the team may choose to develop other kinds of logical models including functional decomposition or object-oriented. These more technical model types are covered in Chapter 10.

Guideline: Model the System's Environment

Models of the system itself will be concentrated in the solution structuring phase. Aside from a logical model of the system (see above) we need a model of the system's environment. This should include what the system interacts with, what supports the system, and what attacks the system (whether deliberately or in the sense of environments that hinder operation). The point of this effort is to further define the environment within which the system will have to operate. The environment can be modeled in the same fashion as described above for modeling the system's boundary and logical function. Use-cases written from the perspective on an attacker are referred to as "anti-use-cases" or "abuse-cases."

Heuristic: Objectives are More Important than Alternatives

In this work we use the term "objectives" in the sense used by Keeney (Keeney 1996). An objective is some property we want the system to have, accompanied by some scale of satisfaction. Performance on objectives leads to value. Something is an objective because a stakeholder values what it represents. Objectives can be thought of as a midpoint along a continuum from concern to objective to requirement. Consider this in one of our examples, of weather forecasting. Everybody has probably seen a weather radar picture on television or a web site when severe storms are in the vicinity. It is an objective of the program that provides the U.S. network of weather radars to deliver radar products to users. We want those products because they are known, under the appropriate conditions, to provide a reliable indication of the presence and behavior of severe storm cells and tornados, leading to the weather service issuing warnings on which people can take lifesaving actions. This is the delivered value, the ability to provide information allowing reliable actions to further a defined mission (reduce weather caused death/injury and property damage).

There is a "direction of preference," we want higher quality radar images, more frequently, of more of the atmosphere, with higher reliability. As the products get better (in the direction of preference) warnings get better and more value is delivered. Every feature of the radar product has a scale of measure. We can characterize the resolution size (in meters), the wind velocity direction and speed accuracy, the minimum altitude visible, the precipitation resolvable, the update rate, and so forth. There is a relationship between the quantitative performance delivered and the quality of the warnings and a relationship between the quality of warning and people's ability to take actions. The relationships posited here are complex and only partially known, but the direction of them is known along with something about the magnitude and the likely role of points of diminishing return.

This takes us part of the way down the chain. At the top of the chain we have broad concerns: Issue warnings that allow mitigation of life-risk and property damage due to severe weather. These become structured objectives including measures of performance. Eventually we may declare a requirement: The system shall provide range resolution of XYZ meters with an update rate of ABC times per minute. Most of the time imposing a binary pass-fail value on an objective is a consequence of how we want to acquire the system, *not* an inherent problem domain concept. The requirement might be 100-m range resolution, but forecaster users would probably prefer 80 or 50 m, and a product with 150 m resolution would have some value (probably). Most of the time there are no hard pass-fail thresholds in actual use, but they are imposed as part of the acquisition process. As architects doing problem structuring we should be focusing on understanding what that value scale really is. Is 100 m a point where incremental value from finer resolution starts to really flatten out, and thus might be a good place set a requirement? Suppose we find out later than technology easily supports 200 m of resolution but that 100 m would be a serious stretch requiring investment in much less mature technology? Did we gather enough information during problem structuring to know what the value trade off would be?

We say that objectives are usually more important that alternative solutions for several reasons.

- Objectives drive the problem being solved. Trying to rank, or even articulate, solutions in the absence of objectives is meaningless.
- If you focus on alternatives you tend to find "little" solutions. This means solutions that incrementally adjust what we do now. A focus on objectives can lead to a more radical re-interpretation of the situation and much more radical (and more effective) solutions.
- The objectives are often more lasting than the solution. Solutions come and go with different technologies and funding, but objectives that characterize an important problem are likely to be good for generations of solutions trying to address those problems.

Guidelines: Writing Objectives and Finding Fundamental Objectives

Identifying objectives, and selecting good objectives, is helped by a number of principles. First, we need some conceptual terminology. Following the conceptual picture of value focused thinking (Keeney 1996), which we largely do throughout here, an objective has a context, an object, and a direction of preference. If we were working on a cargo carrying drone we might have:

Context: Delivery of cargo to remote areas
Object: Distance from cargo availability point to delivery point or mass of cargo or both
Direction of Preference: More is better (we prefer more distance, more cargo mass)

The fourth element is a value measure on the objective. We take that up in the next section.

We want to distinguish "fundamental objectives" and "means-objectives." As we discussed earlier, we can use the "Five Whys" heuristics in its "Five Wants" form to move from a means objective to a fundamental objective. A means objective is not valued on its own, it is valued because it is seen as indicating that we are getting something else. A fundamental objective is valuable to achieve on its own, not because it leads to something else. What is fundamental and what is means varies from context to context and stakeholder to stakeholder. It is dependent on the decision situation. Suppose we are architecting cargo delivery drone operations in the context of supporting wilderness search and rescue operations. Is delivering cargo to remote areas a fundamental objective or a means objective? In the broad context of supporting wilderness area search and rescue the fundamental objectives are things like "Reduce the loss of life in remote area accidents," and "reduce risk to SAR personnel." Faster delivery of critical supplies to a distressed party without requiring ingress by a crew is a means to that end. Drone delivery is thus a means to an end. If the scope of the study is "how can we employ technology to improve SAR

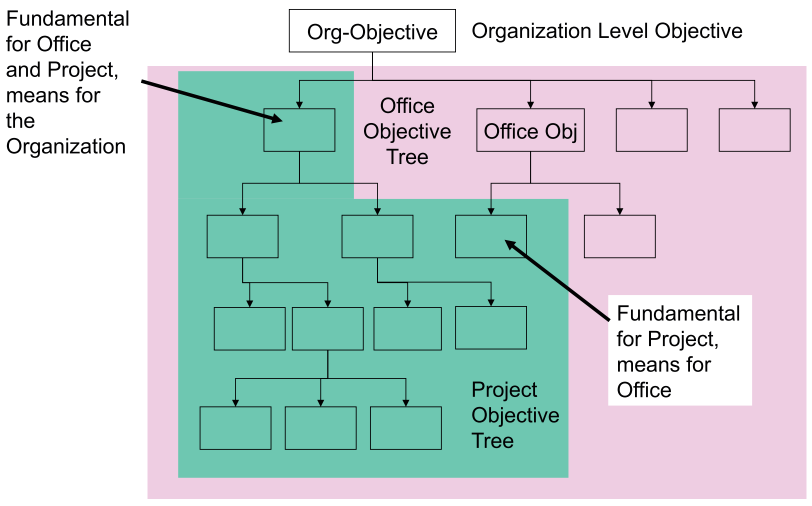

Fundamental for Office and Project, means for the Organization
Org-Objective
Organization Level Objective
Office Objective Tree
Office Obj
Fundamental for Project, means for Office
Project Objective Tree

FIGURE 9.6 Objective contexts and the appearance of fundamental objectives. Objectives are normally fundamental only in context. What is fundamental in one context is likely a means objective in a higher context.

outcomes?" Then drone operations are just one option among many. Perhaps they will turn out to be very effective and cost-favored or perhaps not, but they would be compared to other, disparate, alternatives. If the study scope has already been restricted to "Rapid delivery of supplies to distressed parties" then drone operation specific objectives may be considered fundamental rather than means.

Architects should carefully examine various contexts for architectural problems, consider varying contexts, and develop fundamental objectives in those contexts. This is illustrated in Figure 9.6. There is a large context, always larger than any maximum context for a given architecting problem. This belongs to the organization as a whole. Within that there are contexts for variations on the basic goals for the system being architected. Usually there is at least one more level before the project, say the office within the organization or the portfolio of projects which the system-of-interest is a member. Different contexts yield different (but overlapping) fundamental objectives. What is fundamental for one context (at the top of the tree, requiring no justification for "Why do you want that?" other than "It's our mission") is likely a means objective one level higher, having been derived from some other, higher level, objective.

The most common relationship between logical functions and objectives is the objectives attach to the functions as measures on the function. In the DARPA Grand Challenge example the highest level function is to drive the defined course. The attached objectives could be average speed achieved and probability of completion. In the SAR drone the function is delivery of a cargo payload to a designated point. Attached objectives could be vertical and horizontal separation possible and cargo mass delivered. If the function were more complex, for example you can't deliver to a point you have to deliver with some kind of search or real-time command and control, both the function and objectives would be altered.

Guideline: Develop Value Measures for All Significant Attributes

We usually want a value measure on an objective. The simplest measure is "natural units." This means things like kilometers or kilograms. Careful consideration of the context might alter this, however. For example, suppose the context is "delivery of cargo from search and rescue teams to stranded parties in the wilderness." Now linear distance might be a poor proxy for ability to span the distance required, since vertical elevation difference and a need to fly a complex course due to obstacles might be dominant. As a different example, consider range in the context of a military combat aircraft. A useful mission is (often) not to fly from a point of origin to a different point and back by the most direct route. The ability to reach an operational area from a base may require a whole mission profile of high and low altitude operation, and maneuver, and operational value on arrival may require endurance and energy in the operational area. Maximum range might be a good surrogate for "maximum effective combat range in the offensive counter-air mission," or it might not.

When a simple natural scale is insufficient you can develop "constructed scales." A constructed scale is a form of ruler where you define points on the ruler (that is measure points) by constructed examples. The idea is to build examples that span the range of extremely high value performance (fully satisfying the objective) down to an example of minimum useful satisfaction of the objective. In the case of the cargo drone or the military aircraft the best-case construction would be the launch point and operational delivery points as far apart (in distance and other factors) and the most demanding operational profile that stakeholders believe is relevant. The easy constructed point would be the least demanding origin and destination points and the least demanding operational profile. Then fill in intermediate points to a density that is sufficient for distinguishing alternatives of interest.

Guideline: Grind the Ambiguity Out of All Buzzwords

Stakeholder: "I'm concerned about my system being flexible. Flexibility is an objective, make it more flexible."

This fits the basic model, "more flexible" is a direction of preference, but whatever does "more flexible" mean? This is a common problem when dealing with concerns like flexibility, resilience, scalability, availability, security, and other characteristics (often referred to as -ilities). It is very important to confront this during problem structuring. The terms are ambiguous and we cannot reliably assess (or synthesize for) those properties without removing the ambiguity. Constructed scales are a way out of this problem, and also form an effective method for stakeholder dialog.

Architect: "Okay, flexibility is important to you, you must have good and bad experiences in flexibility. Tell me about the system you have experienced that best exemplifies the kind of flexibility you want. Tell me about the system you have experienced that best exemplifies the kind of inflexibility you are trying to avoid."

People in a position to specify complex and expensive systems will have examples. This is the dialog referred to above for building a constructed scale. The examples will (start) to serve as marks on a ruler for assessing flexibility. For one stakeholder this might distill to the time and effort required to deploy a new feature on their product, or a new product in their product line. For another stakeholder it might be the ability to change suppliers or ramp up or ramp down production rates. Flexibility by itself is ambiguous and means very different things in different contexts. The same is true of attributes like resilience, availability, and others (even where they sometimes have standardized definitions). The examples also serve to suggest ways of achieving those attributes. If others have built systems that do very well, or very poorly, on those attributes then look at what they did and either try to extend, or avoid, the lessons.

Heuristic: Work Out Specialized and Generalized Versions of the Problem

Scoping the system was a key consideration in purpose analysis. The issue is not settled in purpose analysis, the exploration continues in problem structuring and is not resolved until selection/abstraction. In architecting we choose the problem as well as the solution. The distinction for scoping in problem structuring is we are not satisfied with understanding the stakeholder perspective, we seek a precise description, or descriptions where multiple scopes are in-play. A useful heuristic for stakeholder dialog on this is the specialized/generalized heuristic. This is like the previous heuristic noting that sometimes a larger version of the problem/system is easier to deal with then what immediately presents itself. The specialized-generalized heuristic suggests dialog, and analysis on versions of the problem at hand narrowed to special cases, say only one sensor type or one restricted environment. Likewise, extend the other way by expanding to more general cases, more general data, more general environments, or more general threats.

Heuristic: Look for Evocative Metaphors and Analogies

Some difficult problems have simplified in the architect's mind when the problem was seen to be similar or analogous to some other problem. The other problem, for which there were well-understood solutions, or at least approaches, becomes a metaphor for the difficult problem at hand. This is often useful in two ways. First, on a purely practical level, the techniques from the field of the metaphor way be directly applicable. One of the authors was stymied by a complex problem in sensor scheduling and had spent a great deal of effort on complicated mathematics to try and assess feasible and infeasible performance. A newcomer to the project, on being briefed on the problem, remarked "seems very similar to the problem of scheduling jobs in a real-time operating system." The analogy turned out to be close and there was a very rich set of algorithms and analysis procedures from operating system schedulers that were directly applicable.

The second value of looking for such metaphors is that they create an easy to communicate idea of where the system is going and expand people's thought paths. Taiichi Ohno, widely regarded as the father of lean production, said (Ohno 2019) that a key inspiration for just-in-time production came from supermarkets in the U.S. and how the process of restocking allowed consumer demand to "pull" products from the supply chain and into distribution. Whether or not the specific management techniques of supermarkets would apply to parts managements on a production floor was less important than the easily conveyable concept and the derivation of the idea of demand pull over production push.

Heuristic: Model in Both the Customer's and the Architecture Consumer's Language

We've paid primary attention to this point in problem structuring to capturing things in user/customer/sponsor terms. Since the objective here is understanding in the problem domain that is appropriate. However, consider also who is going to consume the architecture products produced and how they will be consumed. Will the products of problem structuring be used by developers (who are also stakeholders)? Then it is appropriate to ask if the products of problem structuring have content useful to developers and are in a form consumable by developers. To know this you need to know the needs of developers, just as you need to know the needs of users or customers.

Guideline: Try Relaxing Technology Constraints

A now 50-year history of continuous improvement in semiconductor technology has rendered one computational constraint after another on building systems irrelevant. A driving constraint 1 year becomes an irrelevance a few years later. While few other fields have seen the same level of continuous exponential change, long-term continuous improvement has been present in many other fields. One consequence is that is emphasizes the importance of separating the constraints that reflect what is currently available in rapidly changing implementation technology from those constraints that are more fundamental. This leads us to the heuristic suggesting we assume away or relax technology constraints and see where that leads. By experience this has two useful effects on problem structuring.

First, relaxing immediate technology constraints brings focus to more fundamental constraints. The speed of light, Shannon capacity in communications, conservation of energy, and the like are still fundamental constraints that cannot be assumed away. Second, if you relax the technology constraint to get away from the question "will it work?" you are still left with the question "Once it did work, would it matter, would it be valuable?" Looked at in detail this can be illuminating, and daunting. Something that technically works might still be a long way from being operationally or commercially valuable. A related heuristic the authors have seen is "Count the Technical Miracles."

> Count the number of technical miracles required to make some concept work and be valuable. If it is one or two it's worth further study. If it's five or more probably not worth bothering.

Relaxing technical feasibility is a way of getting to "so what" questions. Often times the road from being able to make it work to being able to make it valuable to others is long and complicated.

Solution Structuring

With solution structuring we arrive defining the system-of-interest. "It's about time" some may be remarking. This is a significant point, the emphasis in ASAM, and in architecting in general, is on problem understanding and problem selection. It is not that solution understanding and selection is ignored or unimportant, after all the point of the effort is to identify a system-of-interest to buy or build. But the previous discussion has explained why we place so much attention on problem understanding. Our understanding of the problem space frames how we conceive of solutions. We don't conceive of solutions to problems we do not recognize. A solution-first approach tends to lead to small solutions, solutions that only fit within a current operational context. These may be very valuable, and accreting incremental solutions can be very powerful, but they do not typically move beyond existing architectures. By engaging in architecting we are, at least implicitly, prepared to challenge problem contexts.

The goal of solution structuring is to generate a basket of solutions that can be assessed and used to (ultimately) make architectural decisions. The goal is not to find and select the most desired solution and build it. That is the goal of subsequent engineering. In the spirit of progressive design discussed above, architecture segues to engineering as we make value-cost-risk determining decisions and move into implementation.

Needing a set of solutions people naturally start brainstorming. There are many books and courses and consultants on brainstorming. We find there are a number of heuristics that are particularly applicable when brainstorming in the architecture context.

> In a group brainstorming exercise, first work individually and alone, then work in a group.

When the group is responsible for brainstorming, start by having the individuals independently and alone generate solution concepts. Then work in the group to generate more. Do this in cycles. There is a thinking phenomenon known as "anchoring" where the first idea presented in a group becomes the "anchor" around which people then generate variations. This tends to limit the scope of concepts generated to those in the neighborhood of the anchor making it less likely to get genuinely disparate ideas.

> Always include things you know stakeholders are biased to want. Also include things you know they are biased to not want. You are a stakeholder too.

Most architecting efforts start from a position that stakeholders know what they think they are going to get. Coupled with this is knowing what they think they don't want. This should not be ignored. There is a risk that by picking up quickly on what stakeholders are biased to expect that the effort becomes simply feeding back expectations. Other heuristics we'll discuss are there to battle that so don't avoid the expectation. Likewise, it is often useful to study things stakeholders are biased to not want. First, understanding why they think they don't want those alternatives may be very helpful in identifying otherwise hidden objectives. Hidden objectives need to be fed back into purpose analysis and problem structuring. Just because objectives are not immediately and directly articulated does not mean they are not vitally important.

> If everything in your solution basket looks similar, throw out most of them (leaving 1–3 behind) and deliberately generate things that break the patterns in your basket.

It tends to be easy to generate a lot of solutions, at least a lot of similar solutions. It can even be mechanized, sometimes known as "combinatorial design," or the Zwicky box. We do not emphasize generating the maximum possible number of alternatives, we are more concerned that the set we do generate is sufficiently disparate. Combinatorial generation of alternatives may be useful when there is sufficient analytical support. The simplest heuristic we have found for this is the discipline of limiting the number of similar alternatives and deliberately introducing larger variation. This means tracking the contents of your solution basket, classifying them through similarity (see the next heuristic on architectures as classes of solutions) and then culling down. If one is disciplined enough to do the culling then the adjunct is to deliberately create disparate ones. The other heuristics provide some guidance on that. Look at the performance characteristics of the similar set and try to push the envelope, one performance attribute at time if necessary. Try to generate alternatives that challenge stakeholder notions of an acceptable alternative.

> Architectures define classes of solutions, not single solutions. Expand a solution by determining what class or classes it belongs to, or contract multiple similar solutions into a single class.

One of the basic principles of this book is that architectures are sets of decisions. Many specific system instances can share a common set of architecture decisions. Hence, they form something of a class, the class of solutions sharing the relevant architecture decisions. This principle is very useful during solution structuring. For expanding solutions it suggests asking what are the architectural decisions embedded in a particular solution idea, and what else could we do within that decision set? For contracting it asks can we identify the common decisions underlying a number of different, but related solution concepts that could be collapsed.

Referring to some of the examples we've tracked, consider the DARPA Grand Challenge vehicles and the NOAA satellite constellation. In the DARPA vehicle case, one key decision a team had to discuss was whether a team would build on a lightly modified commercial vehicle or build a vehicle from scratch. If starting from the commercial vehicle base they are tied to the standard level of robustness to driving errors and the environment (slopes, obstacles, etc.). The sensing and driving capability thus needs to be similar to what a moderately skilled human driver would do. If the vehicle were built from scratch, it could (potentially) be designed to be far more tolerant of poor driving decisions and difficult environmental conditions. We could see these as two classes of solution each reflecting a different value proposition and set of developmental risks.

In the NOAA case two key decisions are the choice of orbits and the choice satellite acquisition model. The current architecture chooses a specific set of orbits (sun synchronous LEO, certain geostationary positions, and the Sun-Earth L1 point) and a Government-owned acquisition model. There are clear orbit choice alternatives (e.g., all-LEO, all-MEO like GPS, other high orbit mixes) and acquisition alternatives like payload hosting. These likewise defined classes of alternatives with their own behavior in value-cost space.

> Use values to drive solutions and solutions to illuminate values. Spending a lot of effort to find the optimal choice in a set of bad alternatives is likely to be much less effective than trying to find some better alternatives.

A key tenet of value focused thinking is that values are not just to assess alternatives, they should drive the creation of alternatives. A way to do this is to frame some design challenges in terms of articulate values. For example, if stakeholders say (as captured in problem structuring) that they would value performance on some attribute up to level well beyond what has ever been provided, take it as a challenge to come up with solution alternatives that actually would do that. Is it possible? What would be the downsides of such an alternative? Do the same for any minimally acceptable level of performance articulated. Come up with alternatives that achieve (just) those minimum performance values. How much money can you save by going that low in performance? Is it a lot or a little, relatively speaking? This can help fill in the trade space and show where other factors, like cost, are sensitive (or insensitive) to particular performance factors.

The reverse of using values to motivate solutions is using solution preferences to reveal values. If stakeholders dislike particular solutions find out why. The "Five Whys" heuristic is valuable here, as intimated in the discussion earlier. Just as with defects, where the root cause is rarely obvious and requires digging through layers to discover, root preferences are not always obvious either. This process can, and should, begin all the way back at purpose analysis. In purpose analysis we introduced the heuristic of problem framing. The stakeholder will often start from the position of "I want System X" or an equivalent like "I want a system with X features." "Why" questions, and their variants like the following try to get at the underlying need perceived to be satisfied:

- Why do you want that?
- What need does that satisfy?
- If you had that what would you do differently than you do now? What could you do that you don't do now?

A distinction on solution structuring is that this dialog may be best conducted when there is a richer set of solutions to include in the discussion. We can do the problem framing dialog not just with the stakeholder's assumed solution, but with alternative solutions, including solutions generated to challenge preconceptions about what an acceptable solution is. It can be useful to generate solutions that deliberately challenge some of the articulated stakeholder values so at to better clarify those values. It is often the case that people can't engage meaningfully in dialog about abstract values or characteristics but will get to those values in the context of concrete alternatives.

In the NOAA satellite case study the team deliberately constructed some examples to better clarify key values. For example, they created alternatives with content essentially identical to an articulated desired solution but achieved in very different programmatic ways. For example by only re-arranging responsibility among mission partners. These elicited back very direct feedback on desirability, or undesirability, and so clarified otherwise hidden objectives.

> Cultivate knowledge of a wide vocabulary of solution structures. You can't articulate concepts for which you have no descriptions.

Architects, and architecture teams, need a wide vocabulary of system solutions. It is very difficult to synthesize things in domains you are not familiar with. Without a broad solution vocabulary you will be very limited in the kinds of solutions you can conceive. Case Study 5 illustrates this with respect to layered versus hierarchical software. The traditional systems engineering conception of system-subsystem relationships as being primarily whole-part relationships does not well describe layered network and software systems. In that case study we saw two (at least two) architecture alternatives. One was made by choices on how software is structured, and what commonality exists, based on a vertical hierarchy and whole-part relationships. The other envisioned the system's software as a series of layers and identified commonality and sourcing decisions associated with that alternative view. The discussion on strengths and weaknesses (the values achievable and the impacts on costs and risks) showed that one was not necessarily dominant over the other, at least until we take more detailed knowledge of the business drivers into account. To get to the point of having the tradeoff discussion, you have to conceive of the alternatives. That required some knowledge of layered architectures. To really get to the point of having a tradeoff discussion, you also need means of describing the alternatives. In this chapter we are focused on the mental mechanics of developing solution alternatives, we turn to deeper methods of describing them in Chapter 10.

This connects as well with the second heuristic above. Some groups get fixated on optimization and do the optimization over a small set of poor solutions. This is a well-known form of decision bias where we tend to concentrate on the decision directly presented to us (choose A or B) instead of asking "If I can choose between A and B why can't I put C, D, and E into the mix as well?" In design situations there is rarely any reason to exclude a wider range of solutions, except lack of imagination or bureaucratic blinders. The NOAA case had multiple illustrations of this where parts of the stakeholder community operated from the assumption that choosing the future architecture meant making very restrictive choices on increments from existing capabilities. The narrow perspective would have focused on issues like adding one or two spectral bands to certain instruments or one additional orbit with an existing satellite capability rather than reconsidering the whole functional allocation.

There should be program solutions to go with system solutions. Plan to throw one away, you will anyway.

In addition to alternative concepts for the system solution generate alternative concepts for how the system-of-interest can be built (program solutions). In the Government world, the default assumption is Government contracting (usually to a limited set of contractors) for a cost-plus or fixed price development effort, with that development effort awarded on the base of a design competition against requirements. But consider that DARPA ran an open entry race with prize money in the DARPA Grand Challenge. Satellite launch vehicles, some on-orbit capabilities, and many computing assets are bought as services. Many environments could be competed on the basis of prototypes defined by very few requirements, only by operational objectives, and occasionally have been. Even within traditional Government acquisition programs there are many alternative structures based on more spiraled or incremental models.

The same concepts are present for commercial system developments. An important variable there is that the ultimate customer is typically not the sponsor. The sponsor speculatively develops the system then must find customers. The flexibility here is to break that speculative paradigm and look for co-development possibilities where things are developed in direct collaboration with some customers, who presumably get early and/or better access, and whose collaboration hopefully make the product more likely to succeed later.

Remember that the program templates introduced earlier emphasized various forms of iterative development. We attach the heuristic "Plan to throw one away, you will anyway" as a reminder on the desirability of using spiral processes at every level. An architecture study should always be a spiral so that insights into purpose and problem gained in solution structuring can be fed back and made use of in later cycles. In coming up with great solutions, we rarely, if ever, come up with one right away. We usually come up with some bad ideas, and do not address the problems with our ideas until we have explored them extensively. The more innovative the system concept is, the more likely it will have to be exposed to the market and users for an extended period before the desirable approach is revealed. The more innovative the solution is, the more likely it is that extensive, linked operational changes will have to be made before anything like full value can be realized. Those changes will often involve "throwing away" the early attempts. Consider some of the following examples:

1. Personal digital assistant (PDA) devices were on the market for many years before becoming popular. A wide variety of form factors and user interface styles were tried. It was not until the Palm Pilot® (Palm, Inc., Sunnyvale, California) hit the market with its key combination of features (form factor that fit into a shirt pocket, usable stylus-based input, and one-touch computer synchronization) that the market began growing very rapidly. Ironically, but not surprisingly, the pre-Palm leaders were generally unable to follow Palm's lead even after the Pilot had shown the market-winning combination of features. It was not until the iPhone that the whole PDA market collapsed into smartphones, something the PDA leaders were unable to follow.
2. The Global Positioning System (GPS) became a huge success only after two events not linked to the GPS program office occurred. First, commercial companies invested in GPS chipsets that drove the cost down, and volume up, for commercial receivers. Second, the U.S. military pioneered an entirely new bombing CONOPS based on locating targets to GPS coordinates with high precision and receivers so cheap they could be put on the weapons.
3. In the DC-3 story (Part II), we note that it was the DC-3 (and not the DC-1 or DC-2) that revolutionized the airline business. Intermediate systems had to be thrown away.
4. Although the original Macintosh computer could be said to have revolutionized personal computing, that revolution was dependent on extensive evolution. First came the Xerox predecessors that demonstrated essentially all of the distinctive capability, but at a price point and market focus with little interest by customers. Then were failed systems by Apple (the Apple III and Lisa) before the breakthrough. The original Macintosh had to be reengineered to effectively exploit the desktop publishing market (that it had almost single-handedly created). Third, the revolution truly began to be global only when the key interface ideas were ported to Microsoft operating system based personal computers.

These large-scale examples of the heuristic, where the program that surrounds the system "throws one away," either intentionally or not, are mirrored in small-scale design activity. It is rare that we can derive a best solution or representation, much like the first drafts of written works are rarely very good. We improve our designs, like we improve our writing, and like we improve our systems, by iterative development.

Harmonization

In the preceding steps, we have solution descriptions and problem description(s). From a standard systems engineering perspective we should be able to assess solutions for their own quality (consistency and completeness), assess solutions against the problem, find an optimum, and move on. To an extent that is exactly what we should do, but in the architecting context we need to realize that the problem is selected as much as the solution. Especially in an early cycle, as with charrettes, the problem space is at least partially open.

The first step is to look at the problem descriptions (in the plural) as a whole and the set of solutions (really solution classes, following the previous logic of architectures as classes of solution) and see what falls out. What you are looking for at this point is compatible groupings. Are there groupings in the problem space and corresponding groupings in the solution space? The situation can vary from simple to horribly complex. Some typical cases are:

- One cluster in the problem space, covered by one or several clusters in the solution space. This is the classic situation. There is reasonable consensus (if not complete agreement) among stakeholders on the problem space. There are one or more feasible approaches in the solution space. The concept of an "optimum" solution probably makes sense.
- There are multiple clusters in the problem space, but they relate to each other in some coherent way. The most common example is where the problem space clusters represent increasing capability levels, probably with jumps between them requiring a significant jump in solution content to reach each problem space jump. This may be resolvable by looking at the return-on-investment curve (is it worth it to stakeholders to invest more to get more?) or by an incremental program plan where you can move up the capability ladder over time.
- Multiple relatively unrelated clusters in the problem space is more difficult. This probably represents major splits in the stakeholder space (disjoint sets of concerns). Multiple system solutions (multiple problem-solution pairs) may be the best answer. In the commercial space this is "market segmentation" and is a good thing not a bad thing. There may be encompassing solutions but the situation where you need multiple stakeholders with significantly differing interests to all collaborate and stay on-board a difficult system development to make that development work is high risk.
- A situation where stakeholders break into not just into disparate but opposed groups of concerns, especially when coupled with high uncertainty in results is one classic definition of a "wicked problem" and the subject of many case studies of development failure (Metlay and Sarewitz 2012). Even if there are systems that meet each groups objectives the opposed nature of those objectives may make any stable development plan untenable.

In intermediate situations where problem space and solution space groupings overlap, architects should employ techniques from the above steps (Purpose Analysis, Problem Structuring, Solution Structuring) to try and refine understanding and see if more compatible approaches can emerge. Problem framing and related techniques look for win-win refinements. Expansion or contraction of scope may make finding more compatible sets easier.

For an example of the second situation above, see the case study on intelligent transportation systems before Chapter 5 and Maier (1997). The problem space analysis identified different groupings of functional capabilities that were roughly hierarchical (the higher levels containing all of the capabilities of the lower). There were also multiple communication alternatives identified. There was some alignment between the scale and cost of the communication alternatives and the levels of functional service. The lower levels of functional service could be satisfied by the simpler, smaller scale communication concepts while the higher levels of functional service would require the larger, more committing communication concepts. The challenge then was to see if this required a strong commitment early, for example to build the most committing communications network, or if there were more incremental roll-out concepts. This then generated some additional objectives (architecture should support incremental deployment) and a search for how pieces of the problem space might be identified that would facilitate the incremental nature of the roll-out. This is a typical situation in systems-of-systems problems (as discussed in the chapter) where part of the challenge is to abandon the view that the architect and sponsor can control the entire process but need to look for stable intermediates that support decentralized deployment and operation.

In the case cited on NOAA weather satellites (Maier et al. 2020), it notes that terrestrial and space weather observation are largely disjoint sensor types (with a few exceptions). There are solution concepts that have separate space platforms for space weather and terrestrial observation instruments, and concepts that combine them. The combined concepts generally have some cost advantages, but those cost advantages are sensitive to assumptions on how satellite integration costs grow as the number of instruments rises, and how having more instruments on a satellite effects how often it has to be flown to maintain required probabilities of mission capability. The balance of this trade is not obvious. Both would pass consistency and completeness checks (as defined below) but they address concerns over individual program cost, program risk, and political stability differently.

On a more mechanical level, we want to check that our problem descriptions and solution descriptions are compatible and complete. This is consistency and completeness checking. This subject is best taken up in the context of the specific modeling methods used to write the descriptions, and so is a subject of Chapter 10.

At a less formal, but more encompassing level, we can define consistency and completeness (of architecture descriptions) as follows:

Consistency: A set of architecture views or models is consistent if there could exist a physical system as represented in those views.

Completeness: A set of architecture views is *complete* if all stakeholder concerns have been covered in those views. Completeness is meaningful only in relation to purpose.

Within a view the goal of using MBSE methods should be that consistency is guaranteed by the modeling notation's rules. This holds up more or less well depending on the modeling language chosen and the rigor of the method. Between views there is some support for consistency checking within modeling methods but a fair amount of expert judgment is still required. For example, a simple, but still powerful, method of checking consistency between a functional or logical view is a walkthrough. This works well enough at catching serious inconsistencies. If you cannot logically walk a use-case through a physical model of the system then something is wrong and they are not mutually consistent. But having a plausible walkthrough, or even an executable simulation of the functional model on a physical model is not a guarantee it would work that way if built. Walkthroughs are more an indicator of "lack of notable inconsistencies."

Walk throughs should be done for both use-cases and anti-use-cases, ideally with stakeholder involvement. Put the physical system description up, bring up the use-cases, walkthrough step-by-step what should happen. Often there will be a mismatch between the level of detail in the use-case and the level of detail in the physical model. Fill in as appropriate (recording what was said), or leave it for later implementation decision, as is appropriate. For anti-use-cases walk through the steps the attacker is trying to accomplish. Note at each step what stops the attack from working, and what plausible capability an attacker would have to have to make the step work. The walkthrough assessment can then be fed back to additional iterations of a design or as notes for changing the approach in a subsequent cycle.

For completeness the first requirement is that all stakeholder concerns be represented and documented and in a form that can be checked. If they are then we can ask if views are present that provide information relevant to those concerns. If the answer is "no" then there is obvious incompleteness. If the answer is "yes" it still might be the case that the information provided is relevant to the concern but does not resolve it. If concerns are sufficiently well quantified then we a stronger case can be made, but there is still required judgment. In Chapter 10, we examine examples of how choice of MBSE method enables some types of completeness analysis.

Selection/Abstraction

We finally come to selection/abstraction, the point of making a decision. The first key point here is understanding what decisions to make, and what not to make. Purpose Analysis should have helped us define what kind of outcome is required.

- Is the goal to select a specific system configuration (beyond an architecture and being more specific) and proceed with it? If we are architecting in order to submit a proposal, then perhaps so.
- More likely the goal is to make a limited set of architectural decisions, constraints on the configuration of the system-of-interest, and proceed to development where the far more numerous detailed design decisions will be made.
- It may be that the sponsors don't want a single architecture decision, they want a set from which to choose, presumably a small set that has distilled out the easily rejected choices and that focus on the essential remaining trades. Proper practice here would be to focus on the remaining alternatives to choose among those distinguished by different value trades, value trades being the classic domain of the architecture client.
- The term "abstraction" is used here as a reminder that in some circumstances, especially where the system-of-interest is really a collection of systems (a portfolio, family, or system-of-systems) that the architecture of the whole won't fully define any one system, it will be constraints or common choices across the collection.

- If the ASAM cycle we are completing is an early cycle, a typical charrette and plans for more cycles, a major purpose is to feedback information into another cycle. What problem clusters and alternative concepts are especially promising and deserve deeper exploration? Which ones are very unpromising and either need a major fix or can be removed from consideration?

No supposedly optimal selection process can yield anything valuable if the domain of choice is poorly chosen or the time is not right. Too soon and too late are both bad. Keep in mind the heuristic of variable technical depth from Chapter 1. Good architectures, viewed as decisions into the design, are rarely broad and shallow (that is, top-level only). Good architectures are usually uneven in depth, going in depth to make decisions about low-level things that are critical to value, cost, or risk. For example:

- Cost is often driven by narrow considerations. Small changes in specified performance of an observational instrument can make a very large difference in cost. Differences in how something is certified to operate can radically change the cost of development.
- Risk likewise often driven by narrow considerations. A single low maturity technology critical to the operation of the system can drive a whole program. Recall in the GPS case study how the choice to put atomic clocks in orbit dramatically changed the value of GPS and was the driving technical risk.

With the importance of proper decision scoping underlined we can consider mechanisms for making selections. From problem structuring and solution structuring we should the elements of a classic decision analysis. From problem structuring we should have a set of value objectives including a rating scale. As we'll see in Chapter 10, a very well-structured value model includes objectives with performance ranges. A best practice is to define each performance range with closed ends, meaning minimum (least valuable) and maximum (most valuable) values. Conceptually these equate to a "threshold level" and a "maximum effective" (often called the "objective level"). This allows us to construct an explicit value model and assess the relative worth of any alternative on each objective individually. It also facilitates doing objective versus objective tradeoffs. But, before going to that effort the situation should be considered as a whole. Following the logic of Keeney (2004), is it worth invoking the whole machinery of decision analysis?

Before trying to generate tradeoffs between objectives it is best to build a rank table. To build a rank table you start with a consequence table. The consequence table has a column for each alternative of interest and a row for each objective (or the opposite if an MBSE tool defaults to that) and the corresponding alternative-objective score in each cell. The rank table is the same but substitutes the relative rank (assessed across the row) for each score. The rank in a cell is assessed only on that objective. With the relative rank information, one can do several useful analyses without doing any tradeoff analysis, that is, without trading performance on one

TABLE 9.1
Rank Table for some Intelligent Transport Communication Alternatives

	Alternative			
Evaluation Factor	**Wide Area Digital Broadcast**	**Digital Cellular Packet Radio**	**Digital Wide Area (Satellite) Radio**	**Vehicle Roadside Beacon**
Functional support	4	2	3	1
In-vehicle cost	2	3	4	1
Infrastructure cost	1	2	3	4
Deployment mode	1	1	1	4
Self-supporting?	1	1	1	4
Triage class	1	1	1	4
Within historic public budgets?	1	1	1	4
Useful in multiple architectures?	1	1	1	1
Supports multiple ITS services?	4	2	2	1
Cost scaling?	3	2	3	1
Supports non-ITS services?	3	1	1	4

objective for performance on another objective. See Table 9.1 adapted from Maier (1997) for a rank table example from a real case.

- If one column has equal or higher rank scores than another column in every row then that alternative dominates the alternative with the worse ranks. A dominated alternative can be removed from consideration. Note that in Table 9.1 the column for wide area satellite radio is dominated by the Digital Cellular radio column (alternative).
  - Before removing it be sure and consider if its poor performance is inherent in the decisions defining that alternative or if it could be rectified.
  - Also before removing an alternative because it is dominated, consider if there are missing objectives. If the dominated alternative was proposed probably people had some reasons to like it. Are those reasons reflected in the objectives?

- If all alternatives perform equally (or nearly equally) on some row (some objective) either that objective is not useful (does not distinguish alternatives) or the alternative set is too constrained. What would happen if you synthesized alternatives to deliberately do better on the objective that is otherwise equal? In Table 9.1 the "Useful in Multiple Architectures" row does not distinguish alternatives.

- If you have an alternative that is almost dominant (or almost dominated) consider the small set of objectives in which it does relatively poorly.
  - Are they important enough to any stakeholder to swing the success or failure of the system? Who cares about them and why? Might that cause you to focus on, or drop the alternative in question?
  - Can you create a new alternative that does better on the objectives where it is inferior? Would that become dominant?
- Where there are two alternatives whose performance is the same on many objectives but strikingly different on others:
  - Can you build a new alternative combining the best of both, something systematically better than either?
  - Does each naturally appeal more to a different stakeholder set or a different notion of the core purpose of the system? Is there an opportunity to do both or is there a forced choice?
  - If systematic improvement is impossible and there is a forced choice then you have identified on what you need to focus the study of tradeoffs.
- If an alternative is dominant except for cost-related objectives it may be more useful to arrange several alternatives along an axis of increasing value for increasing cost. Ideally this leads to an efficient frontier analysis (see further discussion below).
  - Whenever considering cost objectives carefully consider who the stakeholders for cost are, and what they really care about. Lifecycle total cost is often used in analysis but it usually has the worst stakeholder correlation. There is rarely any customer who directly cares about lifecycle cost. Government acquisitions are more driven by annual cost (what has to be appropriated each year) rather than lifecycle cost. Commercial decisions are driven more by return-on-investment than raw cost.

One thing is never a valid purpose for a rank table. The purpose of a rank table is never to set up the computation of a weighted rank score to compare to others. Weighted ranks are not concepts in MAUT. Weighted combinations of objective scores that went into ranks may be good things to compare, they can be and often should be the basis for quantitative tradeoffs, but weighted ranks never are.

A useful extension of the rank table method is the Pugh Concept Selection method (Pugh and Clausing 1996). In the Pugh method one column is selected as the base and then row entries are rated relative to the base entry, as either worse, same, or better. In a similar fashion one can look for systematically better or worse alternatives (systematically better might be candidates to be the new base). The structure should be used to inspire attempts to synthesize new alternatives, along the lines of "Alternative 5 is equal to or better than base on all but two attributes, can we generate a solution to work those two specifically?" In Table 9.2 we reformat Table 9.1 into Pugh form with the Digital Cellular alternative as the base.

This format visualizes major differences and should inspire additional work in investigating the objectives and solutions examined. Compare the base and the wide area broadcast. Wide area broadcast is same or worse on everything except cost

TABLE 9.2
Rank Table Restructured into Pugh Format

Evaluation Factor	Wide Area Digital Broadcast	Digital Cellular Packet Radio	Digital Wide Area (Satellite) Radio	Vehicle Roadside Beacon
Functional support	–	S	–	+
In-vehicle cost	+	S	–	+
Infrastructure cost	+	S	–	–
Deployment mode	S	S	S	–
Self-supporting?	S	S	S	–
Triage class	S	S	S	–
Within historic public budgets?	S	S	S	–
Useful in multiple architectures?	S	S	S	S
Supports multiple ITS services?	–	S	S	+
Cost scaling?	–	S	–	+
Supports non-ITS services?	–	S	S	–

S, same; −, means worse than base; +, means better than base.

factors. How important are cost factors in this application? In this application the costs are largely paid by customers who voluntarily purchase and use commercial products, and by private companies who choose to deploy infrastructure. Costs matter only to the extent the costs are a deterrent to action. If the return-on-investment in value received (financial or otherwise) is sufficient then the costs will be paid, so do they really matter in the choice? Also note how the beacon alternative differs from the base in almost every objective level. We could engage in a complex trade analysis, how much of each is worth how much of the others, but given how dramatically different they are would that be a trustworthy trade analysis? Perhaps it would be better to look more deeply at what operational concept each supports, and does not, and see if there some other basis for judging a choice.

Note the logic at work here. Selection/abstraction in architecting is not just grinding out the apparent optimal solution, it is using the search for value, cost, and risk to find good and better alternatives. We embrace the closed loop study of purpose, values, and alternatives to progressively approach better and better architectures.

The idea of efficient frontier analysis is to collapse multiple objective scores into a single (or sometimes multiple) composite and look at how many alternatives compare on cost versus composite score. The "efficient frontier" is the convex hull of the set showing the highest composite score that can be achieved for a given cost. The local slope of the efficient frontier is the local trade of "value points" per unit additional cost achievable. The elements of the efficient frontier plot are shown generically in Figure 9.7.

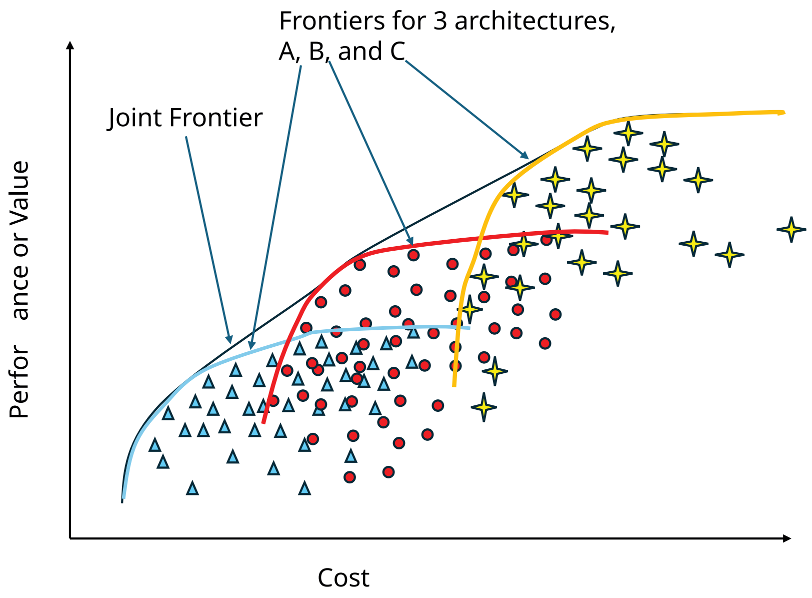


FIGURE 9.7 A generic efficient frontier chart showing the differing efficient frontiers for three alternative architectures. Similar features appear in real analyses of this type, though typically the number of architectures is larger and the breaks may not be so apparent.

The real value of efficient frontier analysis in architecting is not locating an "optimal" solution on the efficient frontier, it is extracting information that reveals important trades between value and cost, possibly stakeholder dependent, that can then be used to reassess values and generate new alternatives. Consider first the use of efficient frontier analysis in comparing and separating architectures. We say that a subset of alternatives has the same architecture if all of the alternatives in the subset resolve a given decision the same way, although they may vary in other decisions. We can compute the efficient frontier separately for each architecture and visualize it the diagram. Since many design decisions are open within an architecture its subset will still contain many alternatives at a wide variety of value-cost positions and its own efficient frontier line. Consider the generic efficient frontier plot in Figure 9.7 with the three subsets for three hypothetical architectures imposed. What is immediately clear is that Architecture A provides the best value-cost combinations at relatively low cost, Architecture C performs best at relatively high cost (and cannot achieve low-cost numbers), and Architecture B can outperform both at intermediate cost numbers.

This by itself is a useful insight. It may be further refined by, for example, restricting the value metric to a subset of stakeholders. With careful normalization you can generate overlay plots that show the value-cost regions for different stakeholder sets on the same plot. This may reveal important positive or negative relationships between the objectives of different stakeholders. For a dramatic real example see Figure 9.8 (a schematic representation of Figure 3 from Maier et al. (2020)). Figure 9.8 shows an efficient frontier plot for a subset of NSOSA stakeholders, those

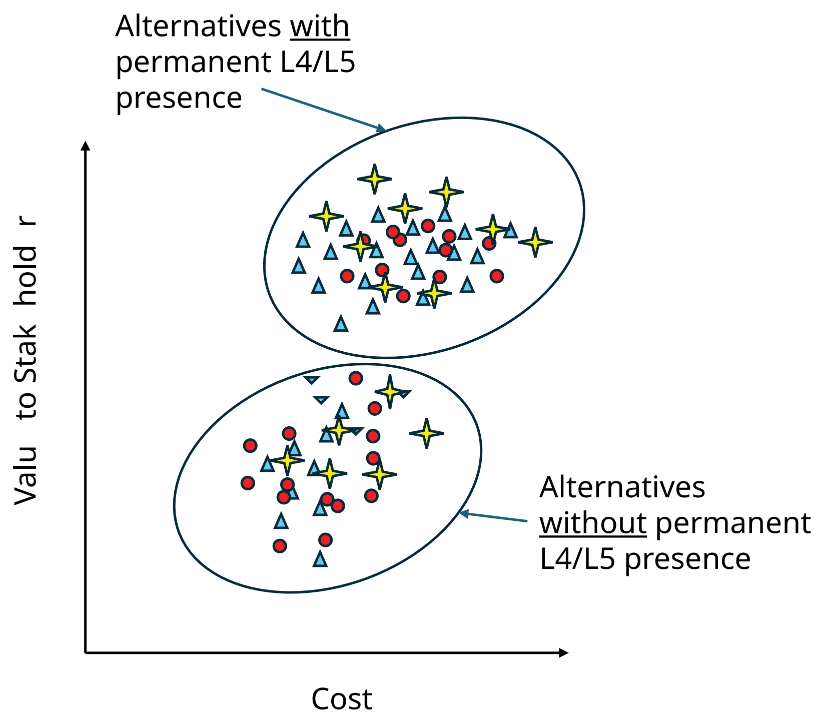


FIGURE 9.8 Efficient frontier plot (schematic) for a subset of stakeholders showing where the efficient frontier breaks down because the impact of a single architectural design choice is so large. See Figure 3 in Maier et al. (2020) for the original and more detailed version.

concerned with space weather forecasting. The overall value metric used in NSOSA was split between those objectives relevant to terrestrial weather stakeholders and space weather stakeholders, but in analysis they can be split out. In the analysis restricted to space weather stakeholders we found two clusters, those alternatives that contained a permanent presence at the Earth-Sun L4/L5 point for solar observation, and those alternatives that did not. The size of the assessed value difference due to that capability was large enough to separate the clusters. What we see is the permanent occupancy of the relevant orbit positions makes a large, positive difference in space weather value (while incurring some cost). These observations carry large value, and in the design analysis it was found that once you had made the investment in a permanent L4/L5 presence there was no point in going halfway, there should be a full investment in observing capabilities at those locations (marginal cost of adding one more observation was low).

If we looked at an efficient frontier plot for terrestrial weather only the points would all be mixed, but if we plotted the efficient frontier for those alternatives with the L4/L5 presence it would be systematically below that for alternatives without the L4/L5 presence. That is, permanent presence at L4/L5 is systematically advantageous for space weather but is systematically disadvantageous (in a value-cost sense) for terrestrial weather. The obvious reason is that those orbital points contribute unique, and valuable observations of key space weather sources on the sun, but do nothing for terrestrial weather, and required dedicated observation platforms to deliver those observations. By itself this does not tell us what to do, but when this occurs it clarifies a highly stakeholder specific trade in the selection of alternative architectures. In a commercial setting we would probably note this as a clear marker for possible market segmentation.

Another form of analysis derivable from the efficient frontier is order-of-buy. The idea here is that we can isolate alternative-to-alternative differences (e.g., inclusion of element X, use of approach Y over approach Z) and look at the impact to the value metric versus the cost metric for alternatives along or close to the efficient frontier. If we start at the low-cost end of the efficient frontier we can see what gets added first, second, etc. to give the value-cost tradeoff along the efficient frontier. This is the "order-of-buy" or the optimal order in which to add things to the mix. The analysis also offers an important caution, it may not be valid as you pass points on the efficient frontier where the individual efficient frontier curves cross each other. As you move from the region where one architecture is preferred to another the order-of-buy content may shift, perhaps in a dramatic way. These are breakpoints where the basic form of the preferred solution changes.

Uncertainty and Sensitivity

So far, we have discussed the selection analysis as if we know all of the factors with certainty. That is never the case in a real situation. Quantification of uncertainty can be an enormous task, and to some level it is essential, but we need to keep in mind that uncertainty analysis is important only to the extent that it shifts architecture decisions. To assess the impact of uncertainty we need to understand and organize it. There are distinct types of uncertainty and different types may lead to different responses.

- Cost estimates that we have to use in this analysis, are notoriously uncertain. If we are concerned if a given alternative is affordable within a fixed budget, then the absolute cost uncertainty is very important. But if we are concerned with the relative cost-benefit of two alternatives, then the relative cost uncertainty is important. The cost variances of alternatives A and B might be larger than the difference in their means but the variations might be so correlated that difference in means is very significant regardless of the absolute size of the variances.
- Stakeholders judge value based on what they think they will get if they have the alternative under consideration. If the capabilities involved are new this may be a very uncertain judgment. As we have discussed before in this book, revolutionary change comes from changes in the concept of operation, typically meaning that prior estimates of value are very flawed.
- Stakeholders may have very diverse judgments of value. We may not know whose judgment will prevail if/when the system is deployed.

- Value judgments might be accurate given stakeholder assumptions about the operating environment might be accurate, but the operating environment may not be stable. The context for the system may shift during the development and deployment period.

There is an extensive literature on characterizing, quantifying, and responding to uncertainty. For structuring the uncertainty in the operating environment the four level (projected, discrete, continuous, and ambiguous) model is well known and widely used (Courtney 2001). The four-quadrant model (Metlay and Sarewitz 2012) is related and similarly useful. The example of the weather satellite constellation was quantitatively examined by the authors for both value and cost variance in an example of full quantification of architecture models and integrated into efficient frontier analysis (Maier and Wendoloski 2020, Maier et al. 2022). See Crawley et al. (2015) for other examples of uncertainty thinking in architectures.

Model refinements are taken up in the next chapter. What we want to emphasize here are the heuristics and strategies. Faced with uncertainty there are some large-scale strategic responses that have direct architectural consequences. These can be generally characterized (building on Courtney (2001), Metlay and Sarewitz (2012)) as:

1. Look for no-regrets choices and do them (possibly passing up bets in order to do them)
2. Project and make big bets (and live with the results)
3. Project and spread your bets
4. Buy real options
5. Follow smarter and faster

The pre-condition to understanding these strategies is you need to have developed some kind of model of future variation. In purpose analysis and problem structuring terms that means explicitly identifying how stakeholder objectives (or the stakeholders themselves) would be different in different future scenarios. At a simple level all objectives might be the same but their performance ranges and relative importance might be different. In the more complex cases perhaps stakeholders come in and out and have entirely different sets of objectives in different future scenarios. On the solution side the scenario might impact the cost of alternatives, or enable/disable whole classes of alternatives. The methods of works like (Courtney et al. 1997) are useful to building alternative future models. These suggest some general strategies in the face of uncertainty.

A **no-regrets** choice does well (at or near the top of value-cost performance) in most or all of the alternative futures you have identified. It is rare to have a comprehensive solution that is no-regrets, but common to find no-regrets elements.

The **big-bets** strategy means you assess what you believe will happen in the future, assess what choices do best in that projected future, and commit to those with best expected value. You accept whatever downsides will occur if the future does not pan out as projected (hence the “big bet”). The justification a big-bets approach is typically a large expected return-on-investment relative to the uncertainty. A bet doesn't have to be certain to be worth making an all-in investment, it just has to have a large enough payout if it works and a cost that is acceptable to the organization making the bet. Organizations (Government and commercial) make big bets on, for example, technologies working out all the time. That not all of them work out does not necessarily indicate they were bad bets.

Project-and-spread is the classic portfolio strategy references. If you make a number of bets with large expected returns and *uncorrelated* variances the collective will have a high payout ratio and relatively lower variance. Essentially this is making many "big bets" with sufficiently uncorrelated outcomes. Obviously, those only works when each bet is not too big. Research and development organizations and venture capital funds use this approach all the time, and often with great success. From their experiences we can abstract a number of useful heuristics and guidelines:

- The variances in returns must be uncorrelated or the whole portfolio may fail (or win) together. There is a danger of herd-mentality setting in among managers resulting in too many investments in too similar areas.
- It is important to be able to identify failing elements of the portfolio early and accurately and terminate them before they have run through too much expense.
- Investing in high return/high variance things means most will fail. This requires careful attention to the incentives for managers of the portfolio elements so that identifying that their piece is failing early and shutting it down is rewarded.
- Conversely, when something is successful the portfolio management needs to be able to exploit it and realize the benefits. This may be challenging for things that pay off very well but not in the ways expected.

To buy a **real option** (Courtney et al. 1997, Shishko et al. 2004, Gray et al. 200) means to pay up-front to make your system more responsive to the value of changing later. You know things will change and what would be better choices will be evident later. Buying the real option means making an investment now that enables a more effective shift later when the better direction will be evident. In aircraft and missiles real options are things like design features that allow up-sizing or down-sizing future versions easily while other parts of the design can remain unchanged. In networks it can mean designs that allow protocol functions to be added later without requiring wholesale replacement and mechanisms to allow multiple versions of protocols to run in parallel. In business real-options can take very diverse forms, from paying for right-of-first-refusal on strategic property to managing a patent portfolio. The forms of real-options in systems can be likewise diverse.

When uncertainty is the greatest one of the powerful approaches is to be better at following, and following fast. If you really can't predict where opportunities will arise, and you know others are busily seeking them out but are probably not better than you at finding them, you may be better off positioning for follow up very rapidly when others identify the opportunities. From a systems architecting perspective this is unsatisfying as it probably does not involve any physical system choices, but it goes deeply to the architecture of the organization.

DETAILED DESIGN, IMPLEMENTATION, AND CERTIFICATION

The emphasis of the above has been on selecting, or abstracting, what to build. Once something has been selected to build architecting is not over, although other disciplines (systems and disciplinary engineering and management) move to the front. We do need to consider the architecting activities that remain in and through development. The most characteristics of the development phase architecting concerns is certification.

Certification

To certify a system is to give an assurance to the paying client that the system is fit for use. Certification is equivalent to "validation" in the verification and validation terminology. Certifications can be elaborate, formal, and very complex or the opposite. The complexity and thoroughness are dependent on the system. A house can be certified by visual inspection. A fly-by-wire flight control system might require highly detailed testing, extensive product and process inspections, and even formal mathematical proofs of design elements. Certification presents two distinct problems. The first is determining that the functionality desired by the client and created by the builder is acceptable. The second is the assessment of defects revealed during testing and inspection and the evaluation of those failures with respect to client demands.

Whether or not a system possesses a desired property can be stated as a mathematically precise proposition. Formal methods can track and verify such propositions throughout development, ideally leading to a formal proof that the system as designed possesses the desired properties. However, architecting practice has usually been to treat such questions heuristically, relying on judgment and experience to formulate tests and acceptance procedures. The heuristics on certification criteria do not address what such criteria should be, but they address the process for developing the criteria. Essentially, certification should not be treated separately from scoping or design. Certifiability must be inherent in the design. Two summarizing heuristics—actually on scoping and planning—are as follows:

> For a system to meet its acceptance criteria to the satisfaction of all parties, it must be architected, designed and built to do so — no more and no less.
>
> Define how an acceptance criterion is to be certified at the same time the criterion is established.

The first part of certification is intimately connected to the conceptual phase. The system can be certified as possessing desired criteria only to the extent it is designed to support such certification. Whether that certification is to be rigorous and formal or informal has to be included in the design and developed from the architecting considerations beginning with purpose analysis.

The second element of certification, dealing with failure, carries its own heuristics. These heuristics emphasize a highly organized and rigorous approach to defect analysis and removal. Once a defect is discovered, it should not be considered resolved until it has been traced to its original source, corrected, the correction tested at least as thoroughly as was needed to find the defect originally, and the process recorded. Deming's famous heuristic summarizes:

> Tally the defects, analyze them, trace them to the source, make corrections, keep a record of what happens afterwards and keep repeating it.

The defining problem in ultraquality systems is the need to certify levels of performance that cannot be directly observed. Suppose a missile system is required to have a 99% success rate with 95% confidence. Suppose further that only 50 missiles are fired in acceptance tests (perhaps because of cost constraints). Even if no failures are experienced during testing, the requirement cannot be quantitatively certified. Even worse, suppose a few failures occurred early in the 50 tests but were followed by flawless performance after repair of some design defects. How can the architect certify the system? It is quite possible that the system meets the requirement, but it cannot be proven within statistical criteria.

Certification of ultraquality might be deemed a problem of requirements. Many would argue that no requirement should be levied that cannot be quantitatively shown. But the problem will not go away. The only acceptable failure levels in one-of-a-kind systems and those with large public safety impacts will be immeasurable. No such systems can be certified if certification in the absence of quantitatively provable data is not possible.

Some heuristics address this problem. The Deming approach, given as a heuristic above, seeks to achieve any quality level by continuous incremental improvement. Interestingly, there is a somewhat contradictory heuristic in the software domain. When a software system is tested, the number of defects discovered should level off as testing continues. The amount of additional test time to find each additional defect should increase, and the total number of discovered defects will level out. The leveling out of the number of defects discovered gives an illusion that the system is now defect free. In practice, testing or reviews at any level rarely consistently find more than 60% of the defects present in a system. But if testing at a given level finds only a fixed percentage of defects, it likewise leaves a fixed percentage undiscovered. And the size of that undiscovered set will be roughly proportional to the number found in that same level of test or review. The heuristic can be given in two forms:

> The number of defects remaining in a system after a given level of test or review (design review, unit test, system test, etc.) is proportional to the number found during that test or review.

> Testing can indicate the absence of defects in a system only when: (1) The test intensity is known from other systems to find a high percentage of defects, and (2) Few or no defects are discovered in the system under test.

So the discovery and removal of defects is not necessarily an indication of a high-quality system. A variation of the "zero-defects" philosophy is that ultraquality requires ultraquality throughout all development processes. That is, certify a lack of defects in the final product by insisting on a lack of defects anywhere in the development process. The ultraquality problem is a particular example of the interplay of uncertainty, heuristics, and rational methods in making architectural choices. That interplay needs to be examined directly to understand how heuristic and rational methods interact in the progression of system design.

CONCLUSIONS AND EXERCISES

Conclusion

A fundamental challenge in defining a systems architecting method or a systems architecting tool kit is its unstructured and eclectic nature. Architecting is synthesis oriented and operates in domains and with concerns that preclude rote synthesis. Successful architects proceed through a mixture of heuristic and rational or scientific methods. One meta-method that helps organize the architecting process is that of progression.

Architecting proceeds from the abstract and general to the domain specific. The transition from the unstructured and broad concerns of architecting to the structured and narrow concerns of developed design domains is not sharp. It is progressive as abstract models are gradually given form through transformation to increasingly domain-specific models. At the same time, all other aspects of the system undergo concurrent progressions from general to specific.

The emphasis has been on the heuristic and unstructured components of the process, but that is not to undervalue the quantitative and scientific elements required. The rational and scientific elements are tied to the specific domains where systems are sufficiently constrained to allow scientific study. The broad outlines of architecting are best seen apart from any specific domain. A few examples of the intermediate steps in progression were given in this chapter. The next chapter brings these threads together by showing specific examples of models and their association with heuristic progression. In part this is done for the domains of Part II, and in part for other recognized large not domains not specifically discussed in Part II.

Exercises

1. Find an additional heuristic progression by working from the specific to the general. Find one or more related design heuristics in a technology-specific domain. Generalize those heuristics to one or more heuristics that apply across several domains.
2. Find an additional heuristic progression by working from the general to the specific. Choose one or more heuristics from Appendix A. Find or deduce domain-specific heuristic design guidelines in a technology domain familiar to you.
3. Examine the hypothesis that there is an identifiable set of "architectural" concerns in a domain familiar to you. What issues in the domain are unlikely to be reducible to normative rules or rational synthesis?
4. Trace the progression of behavioral modeling throughout the development cycle of a system familiar to you.
5. Trace the progression of physical modeling throughout the development cycle of a system familiar to you.

6. Trace the progression of performance modeling throughout the development cycle of a system familiar to you.
7. Trace the progression of cost estimation throughout the development cycle of a system familiar to you.

REFERENCES

Aguilar, J. A. and A. Dawdy (2000). "Scope vs. detail: the teams of the concept design center." *Aerospace Conference Proceedings, IEEE* **1:** 465–481.

Anthes, R. A., et al. (2019). "Developing priority observational requirements from space using multi-attribute utility theory." *Bulletin of the American Meteorological Society* **100**(September): 1753–1774.

Bauer, P., A. Thorpe, **and** G. Brunet (2015). "The quiet revolution of numerical weather prediction." *Nature* **525**(7567): 47.

Bauermeister, B. (1995). Personal Communication.

Boehm, B. and W. Hansen (2001). "The spiral model as a tool for evolutionary acquisition." *CrossTalk* **14**(5): 4–11.

Courtney, H., J. Kirkland, and P. Viguerie (1997). "Strategy under uncertainty." *Harvard Business Review* **75**(6): 67–79.

Courtney, H. (2001). *20/20 Foresight: Crafting Strategy in an Uncertain World*, Cambridge, MA: Harvard Business Press.

Crawley, E., B. Cameron, and D. Selva (2015). *System Architecture: Strategy and Product Development for Complex Systems*, Upper Saddle River, NJ: Prentice Hall Press.

Gray, A. A., et al. (2005). A real options framework for space mission design. *2005 IEEE Aerospace Conference*, Big Sky, MT: IEEE.

Hammond, J. S., R. L. Keeney, and H. Raiffa (1998). "The hidden traps in decision making." *Harvard Business Review* **76**(5): 47–58.

Hammond, J. S., R. L. Keeney, and H. Raiffa (2015). *Smart Choices: A Practical Guide to Making Better Decisions*, Brighton, MA: Harvard Business Review Press.

Hayes-Roth, F. and D. A. Waterman (1983). *Building Expert Systems*, Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

ISO (2018). *ISO/IEC/IEEE FDIS 42020 Enterprise, Systems, and Software - Architecture Processes*, New York: International Standards Organization.

ISO (2022). *ISO/IEC/IEEE 42010 Software Systems and Enterprise - Architecture Description: 69*, New York: International Standards Organization.

Keeney, R. L. (1982). "Decision analysis: an overview." *Operations Research* **30**(5): 803–838.

Keeney, R. L. (1996). *Value-Focused Thinking: A Path to Creative Decisionmaking*, Cambridge, MA: Harvard University Press.

Keeney, R. L. (2004). "Making better decision makers." *Decision Analysis* **1**(4): 193–204.

Keeney, R. L. and H. Raiffa (1993). *Decisions with Multiple Objectives: Preferences and Value Trade-Offs*, Cambridge, MA: Cambridge University Press.

Maier, M. W. (1997). "On architecting and intelligent transport systems." *IEEE Transactions on Aerospace and Electronic Systems* **33**(2): 610–625.

Maier, M. W. (2019). "Architecting portfolios of systems." *Systems Engineering* **22**(4): 335–347.

Maier, M., et al. (2020). "Architecting the future of weather satellites." *Bulletin of the American Meteorological Society* **102**(3): E589–E610.

Maier, M. and E. B. Wendoloski (2020). Weather Satellite Constellation As-Is and To-Be Architecture Description: An ISO/IEC/IEEE 42010 Example. Aerospace Technical Report. El Segundo, CA: The Aerospace Corporation.

Maier, M. W. and E. B. Wendoloski (2020). "Value uncertainty in architecture and trade studies." *IEEE Systems Journal* **14**(4): 5417–5428.

Maier, M. W., K. L. Yeakel, and E. B. Wendoloski (2022). "Cost variance analysis in constellation architecture studies." *IEEE Systems Journal* **17**(2): 1928–1938.

Metlay, D. and D. Sarewitz (2012). "Decision strategies for addressing complex, 'messy' problems." *The Bridge* **42**(3): 6–16.

Mueller, D. C. (2003). *Public Choice III*, Cambridge, MA: Cambridge University Press.

Nadler, G. and S. Hibino (1998). *Breakthrough Thinking: The Seven Principles of Creative Problem Solving*, Los Angeles, CA: Prima Lifestyles.

NOAA (2019). "Future Satellite Architecture." Retrieved 24-December, 2019, from https://www.space.commerce.gov/business-with-noaa/future-noaa-satellite-architecture/.

Ohno, T. (2019). *Toyota Production System: Beyond Large-Scale Production*, New York: Productivity Press.

Palmer, T. (2020). "A vision for numerical weather prediction in 2030." arXiv preprint arXiv:2007.04830.

Pugh, S. and D. Clausing (1996). *Creating Innovtive Products Using Total Design: The Living Legacy of Stuart Pugh*, Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Raymond, E. (1999). "The cathedral and the bazaar." *Knowledge, Technology & Policy* **12**(3): 23–49.

Rechtin, E. (1991). *Systems Architecting: Creating and Building Complex Systems*, Englewood Cliffs, NJ: Prentice Hall.

Shishko, R., D. H. Ebbeler, and G. Fox (2004). "NASA technology assessment using real options valuation." *Systems Engineering* **7**(1): 1–13.

Varian, H. R. (1992). *Microeconomic Analysis*, New York: WW Norton & Company.

Volz, S., M. Maier, and D. Di Pietro (2016). The NOAA satellite observing system architecture study. *2016 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, Beijing, China 10–15 July 2016, IEEE.

Wicht, A. and Z. Szajnfarber (2014). "Portfolios of promise: A review of R&D investment techniques and how they apply to technology development in space agencies." *Space Policy* **30**(2): 62–74.

Wikipedia (2024). "Charrette." Retrieved 15-September, 2024, from https://en.wikipedia.org/wiki/Charrette.

Yourdon, E. and L. L. Constantine (1979). *Structured Design. Fundamentals of a Discipline of Computer Program and Systems Design*, Englewood Cliffs, NJ: Yourdon Press.

10 Architecting with Digital Engineering

INTRODUCTION

Chapter 9 defined how to carry out a general-purpose architecting process (APM-ASAM) in terms of steps, heuristics, and other guidelines by phase. It also defined typical products by phase. This chapter discusses the carrying out the process within an overall framework of digital engineering. The importance of digital engineering integration varies with the overall organizational commitment to digital engineering. If the organizational intent is that any program resulting from architecting has to be run in a digital engineering environment, then the desire to capture architecting information in compatible formats will be high. If there is no commitment to overall digital engineering, then the need to do so in architecting will obviously be much lower. Even in that case there are very positive aspects to using model-based, digital methods for representing architecture description products and the intermediate work products. These aspects will often drive our choices regardless of the program template we follow and our overall commitment to digital engineering.

The nominal model-based systems engineering and digital engineering (MBSE/DE) chain, following SysML standards (Friedenthal et al. 2014) and reference methods like OOSEM or MagicGrid (Aleksandraviciene and Morkevicius 2021, Estefan and Weilkiens 2022), consists of:

1. A use-case model of the user-facing desired functionality of the system-of-interest. This implicitly includes identification of user stakeholders and other stakeholder classes directly interacting with the system.
2. System-level measures of effectiveness (MOEs), captured in parametric models and linked to lower-level performance properties of the system. This may include definition of the computational links between high-level MOEs and lower-level performance.
3. In-depth definition of system functionality with other model types, eventually allocating to single-discipline (e.g., mechanical, electrical, software) elements that are either built or bought. The models types are diverse, including activity models, state machines, and others, possibly including fully executable models at different levels of system abstraction.
4. Definition of the physical hierarchy of the system's components, beginning with the context of the system, what is inside and outside the designated scope. The terminal nodes of the physical tree are single-discipline elements to be built, or components that can be supplied as whole from external

DOI: 10.1201/9781003483595-20

sources. This will look like a block hierarchy at the top with the tree eventually transitioning to disciplinary models at lower levels as the components become single discipline or enter the domains of focused computer-aided design tools.

5. Definition of the system development approach, eventually in high detail, resulting in the construction and delivery of the system. The exact form of this will depend on the program template (e.g., build-all-at-once, protoflight) as discussed earlier in this text. This will include cycling and repeated delivery to operations or sales, assuming that is relevant to the system of interest.

Architecting concerns the top of the process and the value–cost–risk driving choices throughout the chain of digital products. So the challenge in integrating systems architecting with MBSE/DE is capturing the architecting work (outlined in the previous chapters) in models compatible with the breakdown of 1–5 above.

As discussed in a subsequent chapter, we also look to standards compliance, specifically with the ISO/IEC/IEEE 420X0 series of standards. The most relevant is 42010, the standard on architecture description. Our approach in this chapter follows the concepts of 42010, implemented in MBSE models. It is possible to meet 42010 compliance without using SysML or other related notations, as long as one properly defines the notations used in viewpoint specifications (Emery and Hilliard 2009, Maier and Wendoloski 2020b). Architecture standards compliance and MBSE/DE are not causally joined, though the relationship is strong.

While going through model-based methods to represent architecture decisions, it is essential not to lose sight of the fact that architecting and design refinement are learning process not a static point of arrival. Depending on the program template, we will at some points cut off the refinement process, cut metal, and commit, but we reach that point deliberately. This learning cut off may be a single hard point, or there may be many strung out along a cycle of continuing development. However, it is the learning process that is central to architecting.

APM-ASAM IN DIGITAL MODELS

In the subsequent sections, we go through the elements of APM-ASAM, as discussed in Chapters 8 and 9, but now with a focus on defining the products of those process steps with MBSE descriptions. Suitably arranged and coupled with decision-making toward a system development, we have the elements of an ISO/IEC/IEEE 42010 compliant architecture description, and we have performed the Architecture Conceptualization and Architecture Elaboration activities in ISO/IEC/IEEE 42020. We also almost certainly have performed most of the elements of Architecture Evaluation from 42020.

In this chapter, we present three examples. First, we present MBSE materials from the SAR Drone case study concept at the opening of Part IV. This is not specifically a real project, but it easily could be and we developed as an instructional case study. Second, we use a major project of one of the authors, the NOAA Satellite Observing System Architecture (NSOSA) study (Anthes et al. 2019, Maier and Wendoloski

2020b, Maier et al. 2020). And third we return to the DARPA Grand Challenge introduced at the beginning of the book. Familiarity with the NSOSA reference will be a help in reading this chapter, but it is not necessary. Likewise, it may be useful to go back and refresh your memory on the first case study on the DARPA Grand Challenge, and remember that it was written from the perspective of a team brought together to conceive and build a vehicle. Many products presented in this chapter are from the SAR Drone case that has also been discussed in Chapter 9.

To level set with regard to NSOSA, the study developed alternatives for the U.S. civilian weather satellite constellation after flyout of the programs in use at the time of the study (2015–2019). The satellites addressed by this study have began development in the early 2020s and will fly in the 2030s, with an operational period from the early 2030s to the late 2040s (approximately). The main issues considered in this study were:

1. Whether to maintain the long-standing agreements on orbital occupancy by the United States and its European and Japanese partners, or to seek to modify those agreements.
2. Whether to maintain continuity of the U.S. civilian orbital occupancy, augment those orbits with additional orbits, or to replace them in part or as a whole with different constellation concepts.
3. The performance objectives for future observational instruments, below, at, or beyond current performance levels. Whether or not to add observational capabilities for phenomenologies not now observed or to concentrate on incremental changes to existing observational capabilities.
4. How to partition observational capabilities among satellite platforms. Whether to maintain the current level of aggregation or to significantly disaggregate.
5. What business and acquisition arrangements to be used in future generations, in particular to what extent to rely on data buys or hybrid systems for mission capabilities versus the accepted approach of U.S. Government missions for all core capabilities.

ORIENTATION MODELS

Orientation typically has little model-based content. Its centerpiece is answering the orientation questions, the eight questions in the "Orientation" section of Chapter 9. Recalling the "All the really important mistakes are made the first day" heuristics, one purpose of these questions is to highlight those "first-day" assumptions that may turn out to be the big mistakes. Some may be glaring and obvious from casual discussion, like a sponsor who cannot realize the vision for the system without making large changes to systems owned and operated by others, and who has no authority over other systems. Others may be subtle and require investigation and analysis.

For each question and its initial answer, be sure to challenge that answer, specifically, where alternative answers are acceptable to the sponsor and where they are not. For example, the sponsor may assume, in response to orientation question number one, that a physical system will result. If it is possible to accomplish some identified purpose without building anything, is that a good result? Some may say it is and be glad to avoid developing a new system, whereas others may have reasons why building a new system is an essential part. Either way understanding and recording the resolution is part of the orientation process.

While the questions–answers are the heart of orientation, some other results will rapidly feed purpose analysis (the next step) and can and should be recorded during orientation. We define MBSE descriptions for these in the following section on purpose analysis.

- Determining who are sponsors and users relates to using the Four Who's heuristic. Within Who Benefits, Who Pays, Who Supplies, and Who Loses the sponsors are those who (at least in part) pay and users presumably must benefit or they will not likely be users.
- The value proposition for doing the system of interest must touch on identifying the stakeholders.
- The scope of the system draws a boundary of possible stakeholders.

PURPOSE ANALYSIS

Purpose analysis has been done well when we have a rich understanding of the reasons, primarily sources of value, for the system of interest, in the language of the stakeholders. Chapter 9 specified identification of stakeholders and their concerns, problem framing analysis, and identification of needs and objectives (functional and measure oriented) as the key elements of conducting purpose analysis. The Four Who's, Problem framing, Five Whys (problem oriented), and a few others were the primary heuristics for guiding the process.

Stakeholder/Concern Table(s)

A stakeholder/concern table is a straightforward summary of who the stakeholders are, what they are concerned about, their role, their influence, and other factors we might want to track. The simplest representation of this is just a table with stakeholders in a column, a textual description of their concerns in a column, and add on any other information we like (e.g., role, influence, necessity) as we wish. See Table 10.1 for an example for the DARPA Grand Challenge case, following the challenge question at the beginning of the book to look at it from the position of a university team tasked to look at entering.

To follow MBSE principles, this table should not be a static textual object, it should be constructed from model components. A straightforward way to do this within SysML is as follows.

1. Define the stakeholders as a taxonomy of "actors" (from use-case models) or as blocks. The list of generic stakeholder types in ISO/IEC/IEEE 42010 can be prompts to consider. The Four Who's discussed in Chapter 9 is a strong source and was used in Table 10.1.

TABLE 10.1
Notional Stakeholder and Concern Table for Case Study 1, DARPA Grand Challenge

Stakeholder	Role	Concerns	Influence
University administration (VP Research)	Benefits (possibly), pays (initially)	What is the return on investment? Is this an effective use of discretionary funds (can it yield large benefits)? Will it further university mission? Can we get something out of it whether we win or not?	Can say "yes" to starting, probably not finishing.
Team leads (faculty)	Benefits, supplies	How can we structure an effort that both produces credible performance and publishable results? To be successful we need students to contribute, but they have to in a framework that produces something that works. How will what we build contribute to my longer term research agenda (both results and funding)?	Does not have to participate, but absent participation will be pointless for university.
Team members (Grad students, staff)	Benefits, supplies, could lose if distracting	How does one year of work on this get me one year closer to my degree? Opportunity to get in on the front end of something very exciting. Opportunity to prove capability in both research and development.	May not have a lot of choice, enthusiasm very valuable
Potential industry partner/sponsor	Pays, supplies, wants benefits	How does participating via university further my business goals? How does participating further public/brand image? Bad result would be embarrassing and could hurt brand.	If none choose to participate university probably cannot proceed.

a. The 42010 defined set (clause 6.2) to consider is users, operators, acquirers, owners, suppliers and vendors, architects, designers and developers, implementers, maintainers, regulators, testers, public-at-large, adversaries, and competitors.

2. Create a SysML requirements package of stakeholder concerns. Use stakeholder language to fill in the text for each stakeholder concern. Break the concerns into discrete elements to facilitate later tracking.
3. Each requirement in the stakeholder concern package should have a "trace to" relationship to one or more stakeholders.
4. Fill in additional reference information for each requirement in the stakeholder concern package. This could include source documents (e.g., interview or site visit reports), assessed role, risk, or others. Each group can customize what additional information is used by stereotyping a desired requirement type in the notation.

Figure 10.1 illustrates a hierarchy for stakeholders for the Wasatch SAR Drone concept introduced in the Part III introduction.[1] Figure 10.2 shows a stakeholder-concern table for the Wasatch SAR Drone concept built using the SysML elements and process above.

Concerns can translate into problem or solution aspects in many ways. There is no one-to-one mapping from concerns to decisions or specific modeling elements, but there are typical patterns.

1. Concerns that the system do something, that it have some function. The translation path in APM-ASAM will primarily be a concern to use-case or concern to activity (in SysML). Given that we treat concerns as "requirements," even though there usually won't be a formal shall statement, this is a "refine" relationship and setting it up allows for each tracking of concern coverage.

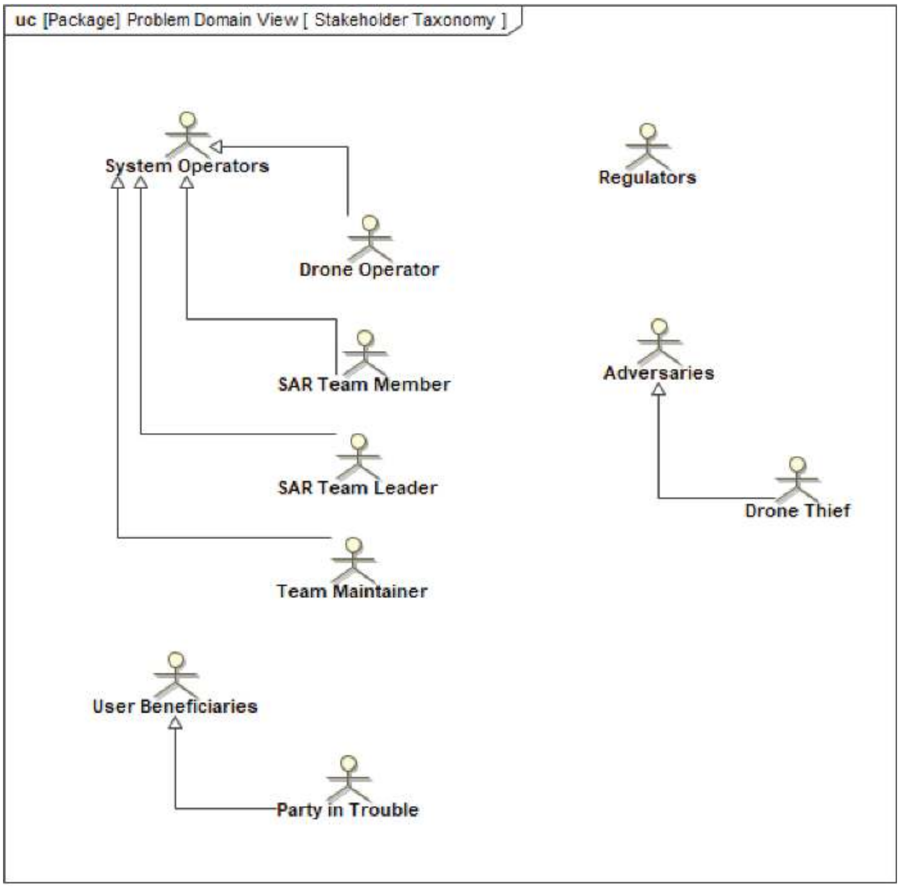
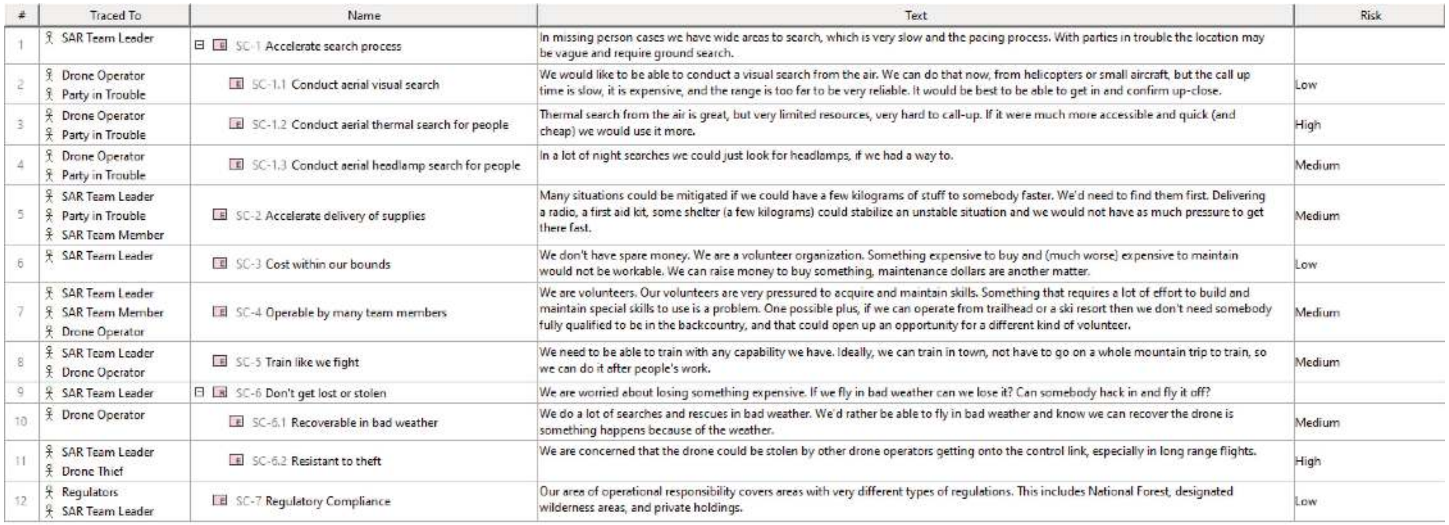


FIGURE 10.1 Stakeholder taxonomy for the SAR Drone concept case study as rendered in SysML.



#	Traced To	Name	Text	Risk
1	SAR Team Leader	⊟ SC-1 Accelerate search process	In missing person cases we have wide areas to search, which is very slow and the pacing process. With parties in trouble the location may be vague and require ground search.	
2	Drone Operator Party in Trouble	SC-1.1 Conduct aerial visual search	We would like to be able to conduct a visual search from the air. We can do that now, from helicopters or small aircraft, but the call up time is slow, it is expensive, and the range is too far to be very reliable. It would be best to be able to get in and confirm up-close.	Low
3	Drone Operator Party in Trouble	SC-1.2 Conduct aerial thermal search for people	Thermal search from the air is great, but very limited resources, very hard to call-up. If it were much more accessible and quick (and cheap) we would use it more.	High
4	Drone Operator Party in Trouble	SC-1.3 Conduct aerial headlamp search for people	In a lot of night searches we could just look for headlamps, if we had a way to.	Medium
5	SAR Team Leader Party in Trouble SAR Team Member	SC-2 Accelerate delivery of supplies	Many situations could be mitigated if we could have a few kilograms of stuff to somebody faster. We'd need to find them first. Delivering a radio, a first aid kit, some shelter (a few kilograms) could stabilize an unstable situation and we would not have as much pressure to get there fast.	Medium
6	SAR Team Leader	SC-3 Cost within our bounds	We don't have spare money. We are a volunteer organization. Something expensive to buy and (much worse) expensive to maintain would not be workable. We can raise money to buy something, maintenance dollars are another matter.	Low
7	SAR Team Leader SAR Team Member Drone Operator	SC-4 Operable by many team members	We are volunteers. Our volunteers are very pressured to acquire and maintain skills. Something that requires a lot of effort to build and maintain special skills to use is a problem. One possible plus, if we can operate from trailhead or a ski resort then we don't need somebody fully qualified to be in the backcountry, and that could open up an opportunity for a different kind of volunteer.	Medium
8	SAR Team Leader Drone Operator	SC-5 Train like we fight	We need to be able to train with any capability we have. Ideally, we can train in town, not have to go on a whole mountain trip to train, so we can do it after people's work.	Medium
9	SAR Team Leader	⊟ SC-6 Don't get lost or stolen	We are worried about losing something expensive. If we fly in bad weather can we lose it? Can somebody hack in and fly it off?	
10	Drone Operator	SC-6.1 Recoverable in bad weather	We do a lot of searches and rescues in bad weather. We'd rather be able to fly in bad weather and know we can recover the drone is something happens because of the weather.	Medium
11	SAR Team Leader Drone Thief	SC-6.2 Resistant to theft	We are concerned that the drone could be stolen by other drone operators getting onto the control link, especially in long range flights.	High
12	Regulators SAR Team Leader	SC-7 Regulatory Compliance	Our area of operational responsibility covers areas with very different types of regulations. This includes National Forest, designated wilderness areas, and private holdings.	Low

FIGURE 10.2 Stakeholder-Concern table generated as a SysML requirements table.

2. Concerns about system attributes, like cost, time, personnel skills, weight, or anything else. In MBSE/SysML terms, these attributes map to value properties of model elements or MOEs.
3. Analysis of concerns may lead to combinations of the above and other requirements. This would be the "derive" or "trace" relationship and would be captured in additional requirements models within a SysML model.

Remember that the point of recording stakeholder concerns is to gain an understanding of those concerns, to capture in the stakeholder's own language their concerns, not to suppress that understanding in search of more modeling rigor. Capturing the concerns as elements in a stakeholder-concerns/needs requirements model works well for this, taking care to note that following the classical strict rules of requirements (e.g., using shall statements, careful disaggregation) is probably entirely inappropriate for expressions at this level. Using elements of a requirements model in this case would be a convenience for providing tracking so that as they are refined and decomposed (presumably eventually to become traditional requirements) the traceability path is not lost.

As alluded to above, a major path for refining stakeholder-concern understanding is via use-cases and MOEs (both elaborated below). Stakeholders often translate into actors in use-cases, so some stakeholder concerns are well expressed by their participation in use-cases. A measure of effectiveness is relevant for a system if and only if some stakeholders care about it. The point of such measures is that they are measures of things stakeholders care about. When formulating the MOEs, we should be doing so in response to understanding how satisfying a concern is measured. This implies a linkage, although that linkage may not show up until a later stage (Problem Structuring). That linkage is easily represented by connecting the stakeholder block to the MOE holding block as it is developed.

Before we dig into use-cases, it is essential to consider exactly what use-cases we are considering and what hierarchy it implies. The fundamental use-case for the Grand Challenge was to autonomously drive the required roads. Accomplishing that task, from a technical perspective, breaks down in a slew of other tasks, from measuring GPS position to sensing obstacles to adjusting speed along some risk/return strategy (it being a competitive race). But the stakeholders are primarily not concerned with just doing the driving task, they are concerned about doing that task specifically on the race day and doing the task in a way that leads valuable follow-ons. If they were solely concerned with winning the race, and second place was no different from last (only first gets the money), then it wouldn't matter how the task was accomplished, only that they won. But the concerns clearly show that winning is not the priority, it is the consequences of participating that really matter. Winning would be great, but primarily to the extent showcases the success of technical approach, and not at the expense of a technical approach that does not scale to other applications. It would be easy to imagine that most of the stakeholders would prefer finishing fifth, if they could do so very reliably and with very extensible technology.

We conclude that in the DARPA Grand Challenge case, the concerns reveal that the value of the system is as much in how it performs the task as it is in how well it performs that task. Contrast that with the implications of the concerns expressed in the SAR Drone case. One of their leading concerns is with how quickly and effectively they can conduct a search for a missing or in-trouble but poorly located party. They see drone operations as a means to the end. If there were a technical or operational approach that worked well and did not involve drones, it would be perfectly fine (assuming it did not clash with other concerns, like number and training of personnel). The contrast in valuing what is done versus how it is done is important to the architect and cannot be lost in the quest to build models. Architects have to perceive the ends desired by stakeholders, not just what they ask for.

Use-Cases: Lists, Taxonomy, Purposes, Selective Dialogs

Detailed use-case development, including the decomposition into detailed activity models, is part of problem structuring. The foundations for that detailed functional definition of the system happen in purpose analysis and derive directly from stakeholder-concern identification. Use-cases have an established model-based structure and are supported in most MBSE tools. They are a standard part of SysML. The key to effective use-cases in purpose analysis is exploitation of the heuristic of variable technical depth in architecting (recall the discussion around Figure 1.1 in Chapter 1). Both breadth and depth are necessary, but not together and only selectively, where one or the other drives value, cost, or risk.

During purpose analysis use-case development should have the following goals:

1. Identify the breadth of use-cases that deliver value to key stakeholders or are likely to drive cost or risk in the design. They do not necessarily need to be developed in depth but strive for breadth of understanding. Using some reference taxonomies is helpful.
2. Understand some use-cases in depth, especially those that are on the context edge of the system and in-tradespace/out-of-tradespace choices will be of high importance. Since you should be interviewing key stakeholders anyway on their concerns, take the opportunity to understand their in-depth perspectives on use-cases, assuming they attach high concerns to some.

A good taxonomy for use-cases is Operations; Direct Support; Test, Certification, and Assessment; Construction or Deployment; Exploitation; and Anti-Use-Cases or Threats.

Operations: These are the use-cases associated with core operational value, what the system does while in operations, doing what it was acquired for. For the DARPA Grand Challenge, this was driving the racecourse. For the elements of the NSOSA system, this is observing environmental conditions and delivering the data to weather forecasters. For the SAR Drone, these are the operationally valuable things the drone can carry out, such as conducting a visual search, conducting a thermal search, or dropping a survival supplies payload.

Direct Support: These are the use-cases associated with keeping the system in operation during its lifetime. This will include maintenance and logistics, protecting or securing the system, monitoring it and reporting on its status, configuring it, upgrading it, and similar tasks. For the DARPA Grand Challenge vehicles, these would be very limited because operations is a one-time deal. However, as we discuss in a moment, test and deployment in contrast were a very different situation. The SAR Drone would have a much more extensive set, including things like deploying from storage to the operational area, detecting failures and being maintained, and having software upgrades.

Test, Certification, and Assessment: These are the use-cases associated with operating the system prior to entering into operations. For some systems, test and especially certification can be value, cost, or risk drivers. It is impossible to fully test a spacecraft prior to operations whose operating environment is a moon of Jupiter. It is similarly impossible to fully test a weapon system for use in a maximally hostile combat environment. Any time you build a system in a single copy, you face the risk of damaging it so severely in test that the overall program fails. Failing to test adequately may kill the system, testing too vigorously may kill the system, and you only get one try. These use-cases differ from the operational use-cases in that they have to define the functions that surround the operational functions being tested, certified, or assessed. For the DARPA Grand Challenge, there would be use-cases here defining how the system runs representative courses, performance is recorded and diagnosed, and improvements identified. The cases here would deal with how humans interact with a robotic system during test and development (human drives while something records deviations between machine-directed behavior, human–machine interactive driving with learning?).

Construction and Deployment: These are the use-cases associated with how the system is assembled and placed into operations. Often this is routine and not architecture driving, but sometimes may be driving. A good example is where a new system has to replace an old in-operation system with no break in operations as the new one comes up and the old one is retired. Obviously, this has often been accomplished, but there are cases where it has serious complexities that may drive architectural choices. For example, in data processing and archiving systems if the new system has to take over for the old, then all of the old data must be ported over, and that places full backward compatibility constraint on the new system, something that can have deep consequences. But if there is no backward compatibility, then the old system will have to be kept running in parallel until everything on it can be abandoned. That likewise carries great programmatic consequences.

Exploitation: This refers to getting value from the system not by what it directly does but through secondary consequences of having built the system. The most common example is technology exploitation. The primary system delivers value, in part, by how it is possible to exploit the technology developed and proven in building the primary, but in applications other than the primary.

Anti-Use-Cases or Threats: These are use-cases written from the point of view of attackers of the system. They are typically things like steal information, steal system components, disrupt system operation, corrupt system operation, and so forth. Use-cases are things that deliver value to the system's stakeholders, and so we want to enable them. Anti-use-cases deliver value to threats to the system and so we want the system to NOT do them. Anti-use-cases should provide a guide to design for prevention. Anti-use-cases presume an intelligent, hostile threat. We usually describe passive threats (like environmental threats) in the Direct Support Section rather than here.

As part of the stakeholder-concern exploration process, the team should build lists of use-cases in each of the taxonomy categories. It is quite likely many of these will be very simple, nothing more than a use-case title and perhaps a few sentence statement of the use-case purpose will suffice to know it can be specified and developed in due course. But some will bring up larger issues and be worth undertaking detailed dialog development and doing so while already interacting with the stakeholders is convenient and avoids having to come back later to do what could have been done early.

In the author's experience, direct interaction with stakeholders who have direct involvement in the problem domain is essential to good architecting. One cannot rely on official requirements documents. In the author's experience, every occasion where we have had direct interaction with key stakeholders talking through key use-cases in detail, we have learned things that appeared nowhere else.

Example: When building a new system that receives command, control, and tasking from existing systems, there are almost always significant quirks in the existing command, control and tasking systems that are not reflected in official documentation. "Exceptional" behavior may actually be common, normal behavior may be ruled by custom and practices as much as by the rulebook. For example, many command and control systems are far more reliant on human-to-human real-time problem-solving than they are advertised to be. By itself that is not necessarily bad, but if much of the most valuable information in system is stored in chat logs rather than databases, then a new system connecting with it may be in for a big surprise. On the opposite side, if the new system delivers data to be exploited by an existing system, one with extensive human interaction, the way data is exploited may be subject to a wide range of customary operator/analyst practices that are not apparent from looking at the nominal design of the current exploitation system.

A generic use-case taxonomy, implemented as a package hierarchy in SysML, with a selection of representative use-cases listed (for the SAR Drone case) is shown in Figure 10.3.

An MBSE alternative to the use-case taxonomy is an operational activity taxonomy. The elements of this taxonomy will be the same as for the use-case taxonomy given above. We discuss some nuances on activity taxonomies and mission threads in the section on problem structuring. The only real distinction is that whether we use the use-case construct or move directly to activity models. In UAF rather than SysML, the operational activity taxonomy is a more common approach. A generic mission thread taxonomy in UAF notation is in Figure 10.4, implemented as an operational process diagram.

Objectives/Measures of Effectiveness

Detailed development of MOEs is a problem structuring activity, but their roots should be in purpose analysis. Anything that becomes an MOE should be traceable to stakeholder concerns. An important element of establishing MOEs at this phase is decision-making on where in the complex objectives hierarchy the decision for the system of interest should be made.

The NSOSA case provides a detailed example of the complexity of this issue. The ultimate purpose of collecting satellite environmental observations is to support

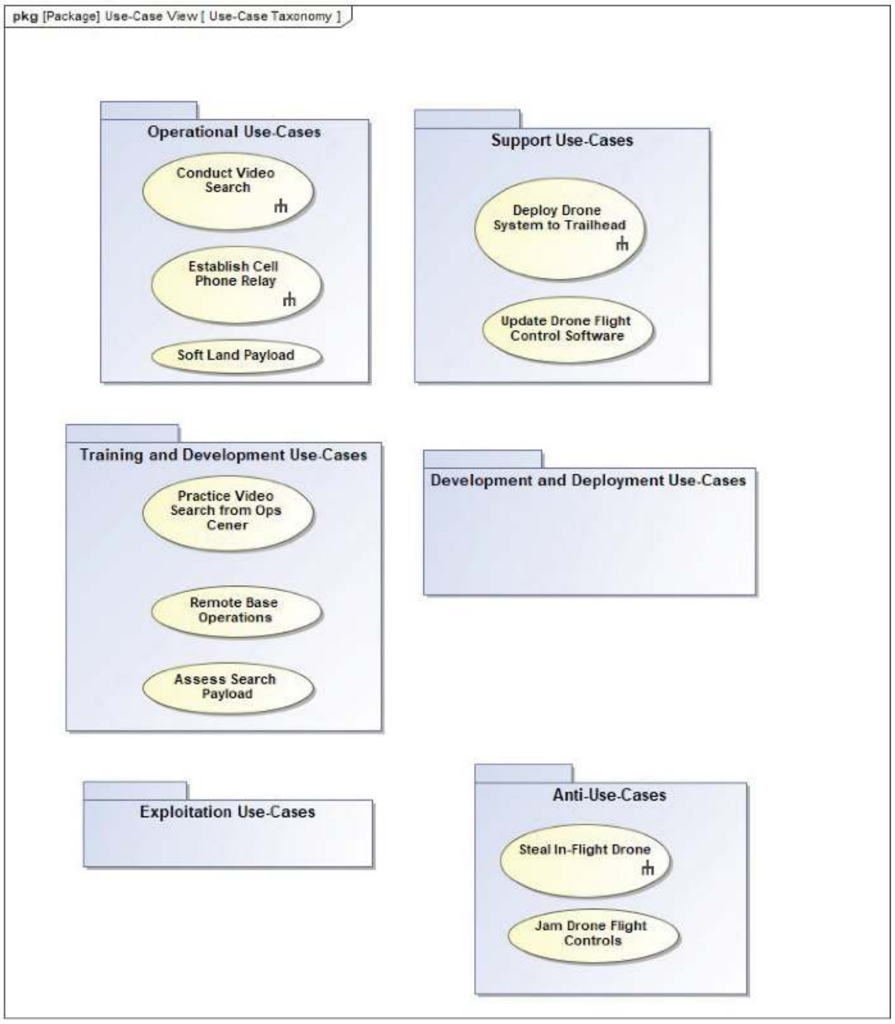

pkg [Package] Use-Case View [Use-Case Taxonomy]
Operational Use-Cases
Conduct Video Search
Establish Cell Phone Relay
Soft Land Payload
Support Use-Cases
Deploy Drone System to Trailhead
Update Drone Flight Control Software
Training and Development Use-Cases
Practice Video Search from Ops Cener
Remote Base Operations
Assess Search Payload
Development and Deployment Use-Cases
Exploitation Use-Cases
Anti-Use-Cases
Steal In-Flight Drone
Jam Drone Flight Controls

FIGURE 10.3 Representative use-case taxonomy for the Wasatch SAR Drone case example. Development and deployment use-cases and exploitation use-cases are empty because no architecturally relevant use-cases in those categories were identified.

actionable forecasting. Hurricane track and intensity forecasts, snowfall forecasts, geomagnetic storm watches, and warnings are all examples of actionable forecasts rooted in environmental observation (largely but not exclusively from space) combined with numerical modeling and human data exploitation. The U.S. National Weather Service maintains a set of overall metrics, with decades of history, on the effectiveness of those forecasts (NWS 2018). It would be natural and logical to assess alternative observational capabilities against their impact to those metrics as they are the root of mission effectiveness. However, this has very significant difficulties and was not chosen as the point of effectiveness measurement (Anthes et al. 2019).

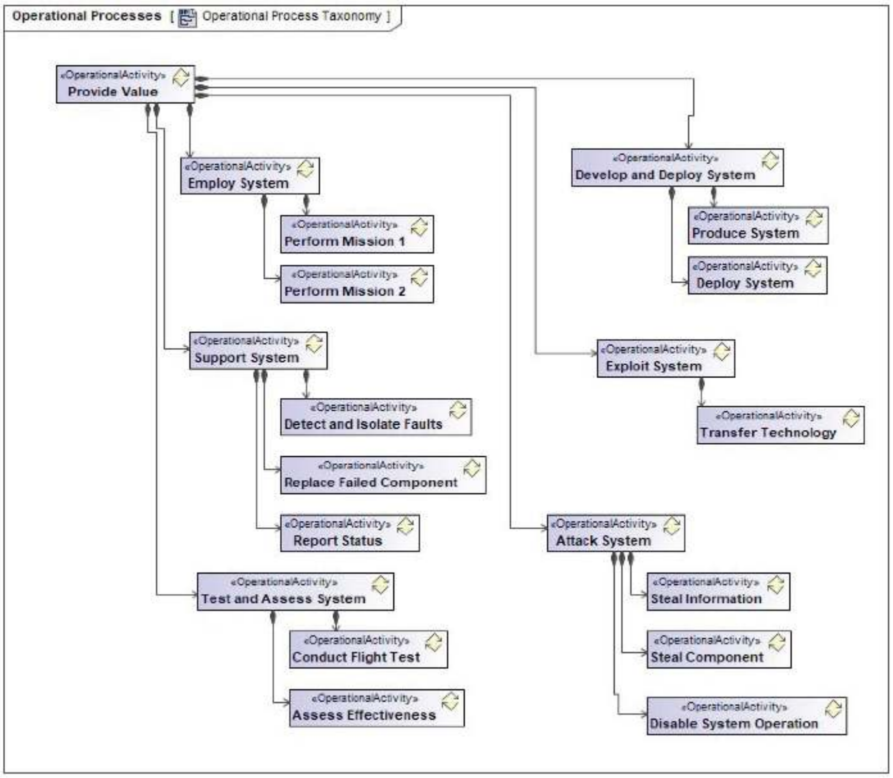

Operational Processes [Operational Process Taxonomy]
«OperationalActivity» Provide Value
«OperationalActivity» Employ System
«OperationalActivity» Perform Mission 1
«OperationalActivity» Perform Mission 2
«OperationalActivity» Support System
«OperationalActivity» Detect and Isolate Faults
«OperationalActivity» Replace Failed Component
«OperationalActivity» Report Status
«OperationalActivity» Test and Assess System
«OperationalActivity» Conduct Flight Test
«OperationalActivity» Assess Effectiveness
«OperationalActivity» Develop and Deploy System
«OperationalActivity» Produce System
«OperationalActivity» Deploy System
«OperationalActivity» Exploit System
«OperationalActivity» Transfer Technology
«OperationalActivity» Attack System
«OperationalActivity» Steal Information
«OperationalActivity» Steal Component
«OperationalActivity» Disable System Operation

FIGURE 10.4 Mission thread taxonomy but implemented as an operational activity structure in UAF.

First, while we know there is a linkage between observational quality and quantity and forecast quality, the quantitative nature of the link is unknown. Forecast quality is a combination of observation quality, numerical model quality, computation available for the models, and human interpretative skill. Each factor co-evolves with the others. The whole system at any given time is configured and tuned to do as well as it can with the observational data and computational capacity available. Models do not automatically do better when fed better data, almost paradoxically they do worse because they are no longer operating at their design point (Bauer et al. 2015, Palmer 2017). Models generally cannot be fully developed and tuned for new data until that new data is actually available. Likewise, human forecasters learn on the data and tools they have available, not on speculative data. At any given time, we can see how well we have done and the impact of the observational data we have learned to assimilate and predict with, but that does not linearly scale with what improvements we might realize if the quality and quantity of existing observational data types were improved, much less if an entirely new category of observational data were available.

Second, there are direct users for many observation types independent of their assimilation into larger models. The quality and continuity of individual data matter. There are users who would use entirely new sources if available, even though the full assimilation of new sources into models and forecast procedures would be an extended process.

As a result, the NSOSA project decided to frame the problem quantitatively in terms of hybrid measures. They were at a level of abstraction below user-facing forecasts (for the reasons noted above) but at a level above the large number of legacy-specific observation requirements. This decision was made to both ground the value model in user needs while providing trade space well beyond legacy requirements (that did not include new measurements with well-supported operational value or larger quality–quantity variation). This is representative of the typical kind of value model decision-making required in purpose analysis, a decision on what level of abstraction to choose to state quantitative objectives. The MBSE realization of this decision will depend on how the highest-level MOEs are formulated in problem structuring.

Detailed MOE generation is a task in problem structuring, but we would expect this kind of analysis in determining what MOEs should exist to be part of purpose analysis. The product would be an MOE list but without detailed development of the components and relationships. We could capture this as constraints in SysML or UAF, but there is no good reason. Basic list construction with notes on rationale are more important at this point in the process.

Problem Framing

The primary point of the problem framing process and heuristic is to challenge the statement of the problem. Following the heuristic that the first statement of the problem is never the best, and is frequently seriously flawed, part of purpose analysis is to look to reframing the problem. From an MBSE perspective, what constitutes the problem statement? Aside from a textual answer, it is embedded in the collection of stakeholders and concerns considered, what use-cases are considered in-scope and out-of-scope, and what physical element are in-scope and out-of-scope. To carry alternative problem statements, we need to have under consideration more stakeholders, more use-cases, and more/different objectives than we will necessarily adopt and need to be able to show alternative bundles corresponding to alternative problem conceptions.

There are no standard ways of doing this in MBSE notations, but it can be done easily enough by starting with a wider net deliberately documenting alternative scopes, with rationale, before eventually down-selecting the problem. The concept of developing many alternative solutions and down-selecting is broadly accepted, and we have relatively standard trade presentation mechanisms, like efficient frontier plots. The concept is the same in problem selection although we cannot expect to have the same level of integrated "optimal" problem choice.

The standard approach in APM-ASAM is to fill out problem framing sheets following the Implied Solution/Desired State of Affairs/Gap/Trap/Alternatives model. See Table 10.2 for an example from the SAR Drone case.

TABLE 10.2
Problem Framing

Implied Solution
A truck deployable drone system with a variety of payloads. Payloads to include cameras and droppables. Deployed teams will use the drones to locate parties-in-trouble and provide quick response lightweight payloads (e.g. survival supplies).
Desired State of Affairs
Missing persons and parties in trouble with poorly reported locations are found much faster than today, and with much less team callout. A high percentage of cases where death or disability could be mitigated by optimal emergency response are successfully mitigated.
Gap
The time and effort to find, reach, and provide supplies to parties-in-trouble are the pacing elements in better search and rescue outcomes, especially avoiding the worst outcomes (death, major disability). Today, the emergency response time is a key difference between remote and urban response and is an important factor in poorer outcomes in remote area response. Shortening the time to first contact and first supply will have a large impact on outcomes. It will also have a large impact on the SAR team effort and impact required to effect search and rescue (number of people called out, call out duration).
Trap
The time to contact and supply delivery size feasible with the truck-drone concept may be insufficient to make a material difference in worst outcomes, and not reduce call out numbers. The truck deployed concept requires a substantial call out itself, and mitigatable worst outcomes may be materially impacted primarily by very short response times (<15 minutes for avalanche burial) or only by provision of heavy supplies (e.g., medical aid, storm survival).
Alternatives
Base drones in locations that enable response without SAR team callout (e.g., ski resorts, mountain fire stations, possibly selected dedicated sites) with remote operations.
Invest in continuously available communications that fills-in coverage gaps so that parties can always be contacted for accurate location and status information (potentially mitigating callouts and enabling remote medical consultation).
Investigate partnerships or regulatory with ski resorts to increase year-round ski patrol presence as a quick reaction capability to search and rescue callouts. Adjust policies and incident management to allow tailored response by different groups (quick response versus large callout to evacuate persons).

Purpose Analysis Models Summary

Purpose analysis should result in a rich picture of stakeholders and their concerns and the broader context of the system of interest. Purpose analysis has been done well when the architecture team, and its sponsors, emerge with personal understanding of that rich picture. In terms of concrete products to capture that rich picture, we have identified:

- A stakeholder-concern table identifying all stakeholders in the current analysis and their concerns. This can be in the form of a textual table, as in Table 10.1, or generated from a model of the stakeholder hierarchy and associated concerns as in Figure 10.2.
- Documentation of stakeholder interviews and other interactions. The more these record what users and other direct stakeholders say and do the better. These records should be available for members of the team who join later to review and become familiar with the original motivating concerns.
- Use-cases, focusing on identifying the primary value-delivering use-cases within a taxonomy of functional areas; including selective development of dialogs for those use-cases believed to be driving of value, cost, and risk; identifying threats, as anti-use-cases, is also appropriate.
- Initial lists of objectives, though detailed development of measures on those objectives is not necessary at this stage.
- Problem framing sheets/tables. Problem framing is an analysis technique rather than a model, but we have a standardized capture form. The results are best recorded textually and then reflected in generating alternative problem statements to be carried through the early architecting cycles.

PROBLEM STRUCTURING

While purpose analysis concentrates on building understanding of the problem domain in all its richness, complexity, and possibly ambiguity and uncertainty, problem structuring seeks to refine and clarify. Some complexity and even ambiguity is inherent in the problem domain and cannot be eliminated but only clarified. The role of the problem structuring processes and heuristics is to provide that clarity, even where it may require breaking up problem descriptions into disjoint parts. There are three principal components of the problem description: The non-tradespace logical and physical context of the system of interest, the value-delivering functional content of the system of interest, and the measure of value or effectiveness for any system of interest we end up building or buying.

Context Models: Physical, Logical, Informational, Interface

Effective development of the system of interest requires that we know, precisely, what is inside the system and what is not. The inside/outside question has logical, physical, and informatic aspects. It also requires that we identify all interfaces between the system of interest and the outside or its "context." Here we refer to that which surrounds the system of interest as the context. Hence, we will want "context diagrams" which we choose diagrammatic representations.

Context can be physical, logical, or informational. It can concern the exchange of logical information or physical resources between the system of interest and the outside world, or it can concern the channels by which information or other resources are conveyed between the system and the outside world. Consider the following basic context relationships.

1. The system of interest interacts with designated physical external systems, and has defined physical interfaces with them (e.g., brackets, connectors, plugs, networks). The context definition is between those outside the system of interest (and thus non-tradespace) and what is physically required of the system of interest.
2. The interactions between the system of interest and the outside are governed by what functions are fixed to the outside and what functions are required to be provided by the system of interest. This is the logical context, the functions inside and outside and the exchange of information between them.
3. The logical context is also an informational context. Some information needed by the system of interest has home outside the system of interest, and the system of interest provides value by providing certain information.
4. There may be physical or logical elements whose status as inside or outside is uncertain. The sponsor might make incorporation of some existing things an option, subject to trade, or they could be replaced.

Sometimes these contexts are aligned, sometimes they are not. Consider a system that must connect to an existing external command and control (C2) system to receive tasking and mission essential information. There are two aspects to this connection, the logical and the physical. The logical connection is what tasking and mission information the system gets from the existing C2 system. The physical connection is how it gets the information (network, radio). Suppose the C2 system already exists and the new system is to connect to it. The logical/physical split could go several ways.

- The new system could be required to be backward compatible. In this case, it connects on the same physical channels and responds to the same tasking and the same mission information as the old system.
- The new system re-uses the physical channels but requires new information from the C2 system because the system-of-interest has new capabilities that have to be tasked and supported. This maintains the old physical-functional split but requires changes in the C2 system. It may be that the new system developer will also be responsible for developing new capabilities for the C2 system.
- The new capabilities in the new system-of-interest may be supported by changing the functional allocation between the C2 system and the system it supports. The new system-of-interest might take over some elements of functionality in the old C2 system.

The uncertain repartition logic above for C2 external systems is not at all restricted to C2 systems. Maintenance and logistics, data analysis and user product generation, data archiving, provision of customized user interfaces, and managing upgrades are all areas where trades may shift from one generation of system to the next. In each of the cases, they are likely to be represented by ambiguity on the external interface of the system of interest or unevenness between the physical and functional borders.

Since there are multiple contexts, context diagrams come in several forms. When using SysML or UAF, the usual approach is to use one or more coupled pairs. For the physical context, the first member of the pair is a block definition diagram (BDD) in SysML or a taxonomy diagram in UAF. This diagram captures the full context (all entities of interest, of which the system of interest is just one) as a hierarchical set of blocks. See Figure 10.5. Its twin is an internal block diagram (IBD) in SysML or an internal connectivity diagram in UAF. This diagram shows the same set of blocks (now in their guise as usages) along with interfaces and connections. See Figure 10.6 for an example.

These two examples were chosen to highlight some of the important choices in representation in systems architecting. The context BDD shows the set of things that compose, or are used by, the overall context, which includes the Wasatch SAR Drone system as a component. The overall context equates to the SAR Rescue Team, its personnel and equipment. Some of these contextual elements interface with the system of interest, and some do not. During systems architecting, it is useful to fully model the interactions among all of the context elements, even though many will not directly impact the proposed drone system. Broader modeling may be useful in understanding operations and identifying areas of ambiguity interfaces and allocation. It is common for mistakes to be made in understanding what is really inside and outside the system's boundaries, or for that judgment to evolve as the system design evolves, making a broader model valuable. The diagram may use SysML notations and color coding conventions (yellow for elements used by or interfaced to the context but not parts of it, and green for the system of interest, the one we are architecting).

If the blocks on the diagrams correspond to the physical entities of interest, it is a physical context diagram pair. In some frameworks (notably DoDAF and UAF, see Chapter 11), there is a distinction between the entities in the operational view and

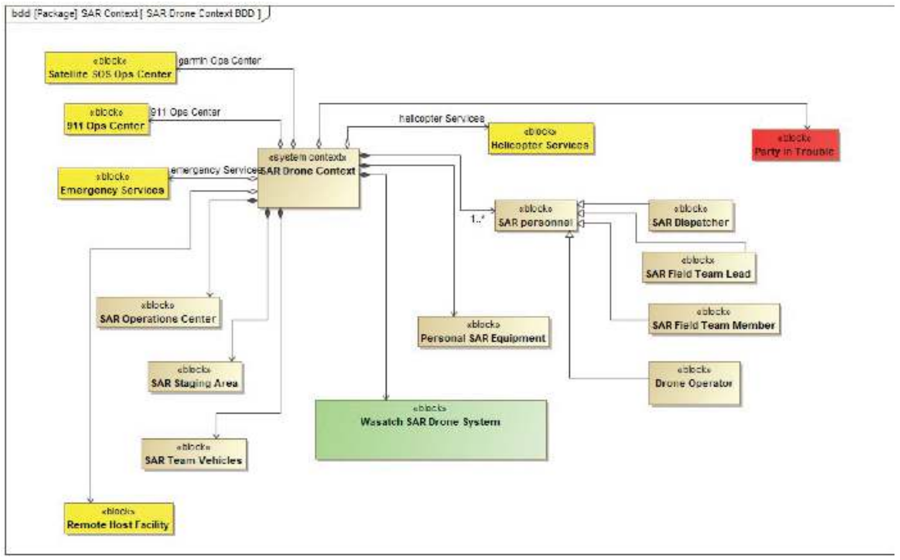

bdd [Package] SAR Context [SAR Drone Context BDD]
«block» Satellite SOS Ops Center
garmin Ops Center
«block» 911 Ops Center
911 Ops Center
helicopter Services
«block» Helicopter Services
«block» Party in Trouble
«block» Emergency Services
emergency Services
«system context» SAR Drone Context
1..*
«block» SAR personnel
«block» SAR Dispatcher
«block» SAR Field Team Lead
«block» SAR Operations Center
«block» Personal SAR Equipment
«block» SAR Field Team Member
«block» SAR Staging Area
«block» Drone Operator
«block» Wasatch SAR Drone System
«block» SAR Team Vehicles
«block» Remote Host Facility

FIGURE 10.5 Block definition diagram (BDD) for the wasatch SAR drone system context. This diagram defines the elements of the overall context within which the system of interest sits.

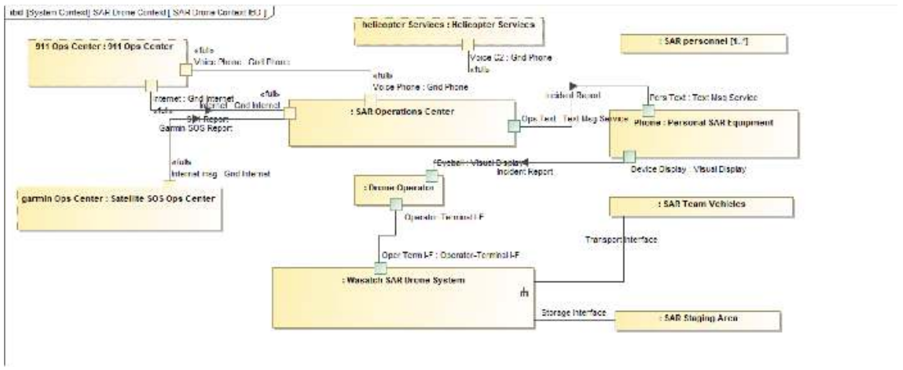

ibd [System Context] SAR Drone Context [SAR Drone Context IBD]
911 Ops Center : 911 Ops Center
«full»
Voice Phone : Gnd Phone
helicopter Services : Helicopter Services
Voice O2 : Gnd Phone
«full»
«full»
Voice Phone : Gnd Phone
: SAR personnel [1..*]
Internet : Gnd Internet
«full»
Internet : Gnd Internet
«full»
SPI Report
Garmin SOS Report
: SAR Operations Center
Incident Report
Pers Text : Text Msg Service
Ops Text : Text Msg Service
Phone : Personal SAR Equipment
«full»
Internet msg : Gnd Internet
Eyeball : Visual Display
Incident Report
Device Display : Visual Display
garmin Ops Center : Satellite SOS Ops Center
: Drone Operator
Operator Terminal I-F
: SAR Team Vehicles
Transport Interface
Oper Term I-F : Operator-Terminal I-F
: Wasatch SAR Drone System
Storage Interface
: SAR Staging Area

FIGURE 10.6 Context internal block diagram (IBD) showing the physical interfaces and exchanges between the system of interest and its context.

those in the systems or "resource" view. The identity of the entities in the systems or resources view is generally unambiguous, they are the physical systems or components in the environment of interest to the system of interest. In the operational view they are operational entities, and the degree to which they are distinguished from physical entities is not always clear and a matter of judgment for the architect. The intent is to identify logical entities (as they will be linked to logical functions or activities and information elements distinct from data representations) as distinct from physical realizations.

These context diagrams should accomplish the following:

- Clearly define what is non-tradespace (fixed and not changeable for the purposes of the architecture study and downstream design), what is tradespace (changeable in downstream design), and what is undetermined. The last category refers to elements that could be taken to be fixed, or delegated to downstream design trades, after the architecture study completes.
- Identify all external interfaces on the system of interest, again indicating which are non-tradespace and fixed, which are fully tradespace (can be developed independently as the system is developed), and which are open to design but under constraint.

It is also necessary to define the functional and informatic context, depending on the modeling style chosen for functions (see the next section) that may be captured with the functions or may be considered part of the context here. The classic form of a functional or logical context diagram has a single function for the system of interest, capturing all of the value-delivering capabilities intended for the system of interest, surrounded by functions provided by the external entities. Again, in the problem structuring phase, there may be ambiguity over whether certain functions are inside or outside the system of interest. There is no standard way of representing this, but there are convenient graphical formalisms in most modeling languages that can be bent to this purpose. See Figure 10.7 for a functional context diagram for the Wasatch

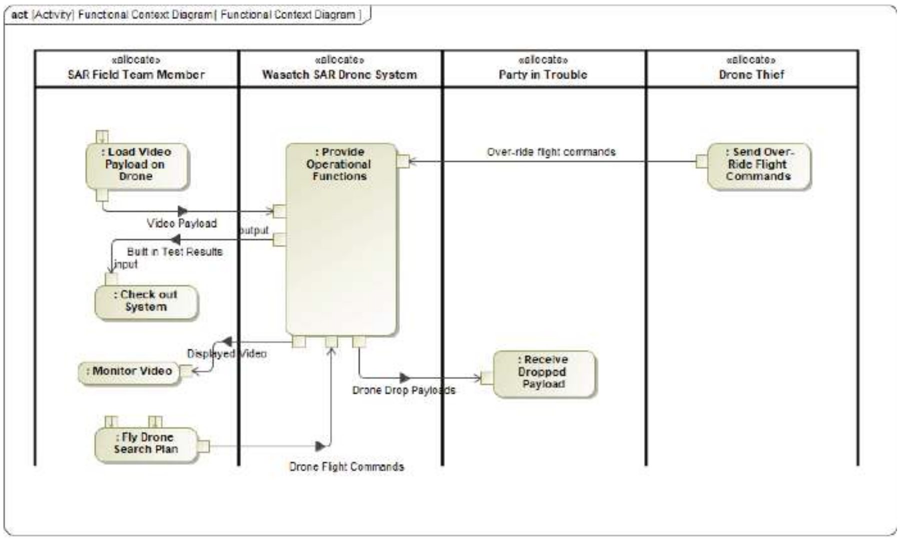

act [Activity] Functional Context Diagram[Functional Context Diagram]
«allocate» SAR Field Team Member
«allocate» Wasatch SAR Drone System
«allocate» Party in Trouble
«allocate» Drone Thief
: Load Video Payload on Drone
: Provide Operational Functions
Over-ride flight commands
: Send Over-Ride Flight Commands
Video Payload
output
Built in Test Results
input
: Check out System
Displayed Video
: Monitor Video
: Receive Dropped Payload
Drone Drop Payloads
: Fly Drone Search Plan
Drone Flight Commands

FIGURE 10.7 Logical context diagram.

SAR Drone system. This diagram is partial, as full logical context diagrams may be very busy and difficult to read. It is often useful to create several diagrams having each be thematic to some functional aspect of the system of interest. When there is an ambiguity, one can create a block at the context level with title TBD. Adding that to the figure as a swimlane is an example of how one might represent a situation with ambiguity on functional responsibilities. This should, of course, be marked as an issue to be resolved in the design process.

The case of ambiguous functional responsibilities typically happens when a new system with new capabilities is being developed and integrated with an existing system that will command, task, or otherwise exercise some control over the operations of the new system. This is the case outlined above for C2. Since the new system has capabilities that do not exist in older systems, the command–control–tasking systems presumably do not have the potential to exercise those new capabilities. Somewhere in the chain something will need to translate from high-level command, control, or tasking needs to the specifics of the new system. Will that happen in simultaneous upgrades to the command–control–tasking systems (outside the system of interest) or will those capabilities be brought into the system of interest? That is a very important choice and that should be determined in problem structuring, after appropriate trade analysis.

Part of developing the logical context is identifying and specifying the logical exchanges or the "information exchanges" in some usage. These are the information elements the system of interest exchanges with the outside world independent of their physical realization. Distinguishing logical information elements and physical data elements is both important and troubling. Depending on the situation finding the appropriate border may be quite difficult and determining how best to handle legacy situations may be complex. Distinguishing logical and physical realizations is essential, at least in the long-term, to interface modularization.

> The author once witnessed an exceptionally awkward web-based user interface for configuring some network communications equipment. The user interface required using a mouse to click virtual keypad keys with a tiny display of characters while using complex modalities to enter configuration information. It turned out that originally the only user interface was a keypad and tiny display physically on the front panel of the unit. Later it was possible to plug a terminal into the back of the unit, then later to network into the unit from a remote terminal, then from a web page. The original user interface software inside the equipment box was written on the basis of direct access to the keypad electronics and the tiny display. As each new connection capability became available it was used to mimic the original user interface, there being no abstraction of the configuration information available to directly control.

If there is an implementation-free information model then that can be realized in many different physical data formats, from intricately packed bits and bytes to textual streams to programming library controlled direct mappings to software constructs. The logical information model can be the invariant that enables wide variation in physical interface. In those cases where physical interface diversity is valuable this abstraction level is also very valuable.

All that said, sometimes other factors push in the opposite direction. For example, many command–and–control systems, and more importantly operational procedures, are built on specific, often text-based, physical formats. Users may be accustomed to exchanging text-based forms, possibly with loose syntactical rules, and have procedures and equipment based around that. Extracting the underlying information structured and then rigorously implementing it in new exchange systems may make machine-to-machine communication and automated processing easy but may disrupt long established and trained procedures and require wide system replacement. Trying for intermediate solutions that automatically process the loosely defined legacy formats may prove to be far more difficult than anticipated. This problem of how to move from existing loosely defined data exchanges to a rigorously enforced set of data standards (which using abstracted information models is built on) has brought down more than one major system development (Goldstein 2005).

Value-Delivering Functions

Problem structuring requires to fully define the functions to be implemented in the system of interest and to find how those functions exchange information with externals. There is a distinction between the functions that do the core things, the value-delivering things, that the system is being built for and the network of functions (mostly operating on and transforming physical data) that implement the system. This is the distinction between operational activities and system functions in the DoD Architecture Framework (Chapter 11 discussion) and the distinction between the "requirements model" and the "enhanced requirements model" (both functional decomposition models) in the Hatley–Pirbhai method (Hatley et al. 2013).

There are two primary standard model-based methods for defining these functions, which can be used singly, separately, or in conjunction and a couple of variations. One method is use-cases (already mentioned with purpose analysis) and the other is activity diagrams (and/or other functional diagram methods like StateCharts, Sequence Diagrams, Data Flow Diagrams, and others).

- **Complete Use-Case specification:** In this method, there is a full set of use-case diagrams covering all operationally desired functionality. Each use-case is defined by an operational activity with an associated detailed activity model (another set of diagrams).
- **Mission Thread Activity Model:** In this approach, we have a full taxonomy of operational mission threads (roughly correspondent to use-cases) with an associated activity model. The thread-actor mappings captured in use-case diagrams are instead captured using swim-lanes in activity flow diagrams.
- **Top-Down Activity Model:** Here we start with a functional context diagram (see Figure 10.7 previously) and then do a top-down functional decomposition on the single large function/activity allocated to the system of interest. When using this method it is common to also develop use-cases or mission threads as a way of elaborating the aggregated function of the system of interest.

Detailed development of these models is clearly on the transition from systems architecting to systems engineering, as we should expect. Rather than trying to draw a hard border between the two it is more important to understand what we emphasize and value on each side of the architecting/engineering boundary. At the root, we go back to the Three Column model in the Preface (Table P.1) at the beginning of this book. Architecting is about defining what is satisfactory and feasible, and defining enough to know that we have value, cost, and risk within the tolerances of the sponsor. Engineering is about completeness, making sure that every detail is correct within the parameters of our overall development strategy and flow.

A key element of transition from systems architecting to systems engineering to disciplinary engineering is the extension of activity models focused on externally facing needs to models that include design required functionality. In this process, the definition of the system of interest's external interfaces logically falls into architecting, while definition of the internal interfaces is engineering, except to the extent that some internal interface may be value, cost, or risk driving.

A very useful template for this is the Hatley–Pirbhai functional enhancement template. The procedure, updated to a more modern perspective on activity modeling with SysML, works as follows, see Hatley et al. (2013) and Maier (2022) for more detailed explanations. We start with a set of functionality assigned to the system of interest. In SysML or UAF, this will be a set of activity models, possibly a large set, containing many activities. For simplicity, we encapsulate that set as a single activity and note all of the external exchanges from the set of activities. These are the external interfaces of the system of interest represented at an information exchange level.

The Hatley–Pirbhai enhancement template is a five-box template (though two of the boxes are typically effectively merged), see Figure 10.8. The center box contains all of the activities allocated to the system of interest. The two side boxes, labelled inputs and outputs, contain functionality added to resolve how the information exchanges (logical elements) are carried on physical external interfaces. The top box is for human interface functions and the bottom box is for supporting functions not part of the main value-delivering functions. So, for example, returning to Figure 10.7, Drone Flight Commands is an external exchange from the SAR Team

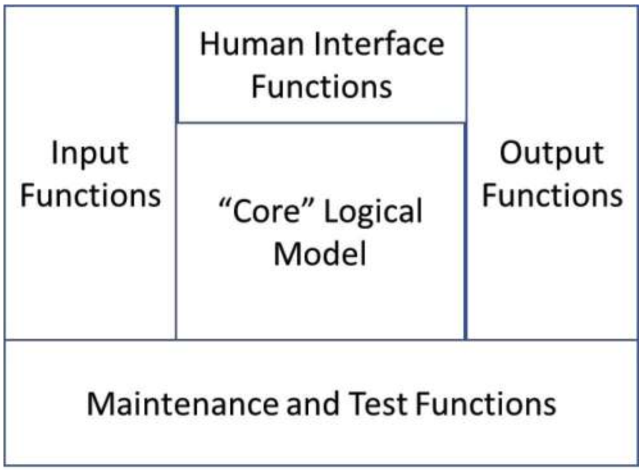


FIGURE 10.8 Hatley–Pirbhai functional enhancement template. The template prompts for human interface, general interface, and internal housekeeping functions to support the core allocated functions. The enhanced functions are dictated by the physical nature of the block in which the core functions are embedded.

Member to the system of interest. If the system as a whole requires functionality to carry out how the team member puts in the flight commands, that functionality fits in the template upper box. Most likely, for any complex system, the human interface is on some subsystem a few levels down from the system of interest as a whole. In a SysML model that will be carried down by a series of ports, first on the border of the system as a whole, then on subsystems until we get to a discrete hardware element. There the actual human interface functionality has to be realized.

The same logic carries for network interfaces. Suppose instead that drone flight commands came into the system of interest on the Internet (as in remote operation). Then the functionality necessary for network encoding/decoding would be carried out by the input/output boxes. Put another way, the enhancement boxes will be populated with functionality associated with implementing the relevant network protocols. The same idea holds for other external interfaces including those conveying physical objects. The bottom box has functionality for implementation-specific support to maintenance, monitoring, initialization/shutdown, and other housekeeping functions. Depending on how fully the support mission threads are built-out, there may be very little in this box.

This process described here is part of an overall process to support MBSE, starting from the architecture-level description and to carry down to the point of transition from multi-disciplinary system models to disciplinary (e.g., mechanical, electrical, software) models. As such it is not on the main path of systems architecting and so beyond the scope of this book's concerns, but we note it here as it assists in integrating between the perspective of this book and the larger MBSE processes. Of note, this outside-in, Hatley–Pirbhai inspired process of recursive functional enhancement fits very well with the DoDAF separation of OV-5 operational process models and

SV-4 systems functionality models. The operational activity model elements become realized by SV-4 functions, SV-4 functions are added at each level of decomposition to resolve the functional needs at each decomposition level, and the process repeats at each level of hierarchical decomposition until the engineers reach the point they transition to part supply or fabrication instructions.

As discussed in purpose analysis, the use-case or mission threads should organize into a taxonomy with the first element being whatever the core value functions of the system are to be (e.g., transportation, weapons effects, remote sensing), and other elements for support of the system, test/assessment/certification, deployment, and threats to the system. Detailed development of the activities is part of problem structuring where they are central to value, cost, or risk. The other threads are also developed in detail, eventually, but presumably this occurs somewhat downstream. There is little point in expending the effort on detailed development of low risk, purely supporting threads until we know that we want to proceed with development.

As we will discuss in the next section on objectives, requirements in the sense of hard, threshold criteria intended to be used in a contract are outputs of architecting and not inputs to it. We develop such requirements when we know we want to acquire the system and that contracted development in a hard criteria sense is the way we want to do it. In contrast, if we do agile development with many deliveries and assessment at each delivery, then hard requirements are much less useful. On the other hand, we still certainly need to know the functions to develop or the developers have nothing to work from. Likewise, where hard criteria exist, like we must implement certain functions and/or those function must meet hard criteria, then defining those is part of architecting.

Starting from context models and user-facing use-cases is part of a general outside-in orientation to architecting. This is typically most applicable when the system must fit into some pre-existing context, as very many systems must. If the system is truly standalone, then we may start from performance objectives or functions or another perspective. It will still be necessary to define the functions inside and eventually to clearly identify the internal/external interface boundary.

Objectives to Measures (to Requirements)

Developing and refining objectives is the process that leads from stakeholder concerns to quantitative measures of system quality and, ultimately, acquisition requirements. We can view the process schematically in Figure 10.9.

We start with stakeholder concerns, which are expressed however stakeholders see fit. They may be expressed qualitatively or quantitatively. They may be expressed in terms immediately related to things we can buy or build or may be expressed at many layers removed from things that are actionable in terms of buying or building a system. The first stage of refinement is to state the issue in Keeney's format (recall Chapter 9 on objectives) with an object and a direction of preference. This is where we dimensionalize the objective by stating it in terms that are, at least in principle, measurable. We refine that measure into something that is value-meaningful by identifying some set of low and high value along the measure. A two-point scale corresponds to the idea of "threshold" and "objective" values, the former the lower limit

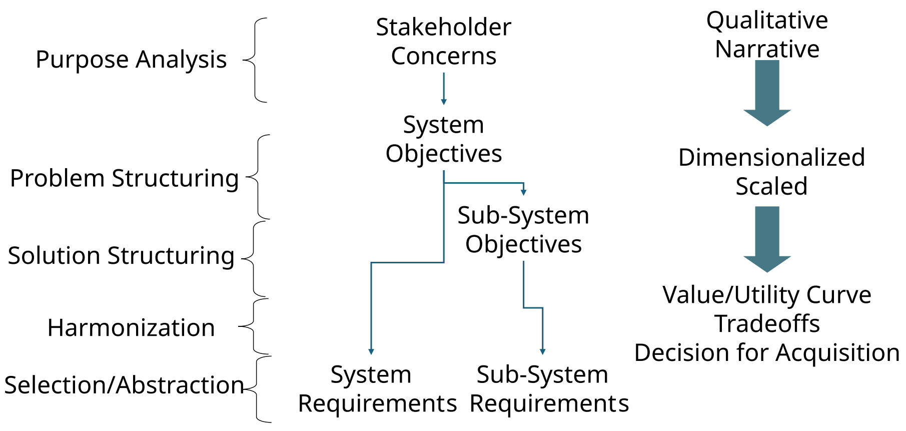


FIGURE 10.9 Schematic representation of the evolution of stakeholder concerns into objectives and requirements. We assume here that the term "requirements" is reserved for objectives with a binary complete-satisfaction or no-satisfaction scale, although we normally only proceed to the point of threshold-maximum effective objectives in architecting.

we will ever be willing to pay for or accept and the upper being the point at which we will no longer pay to improve. We often like a three-point scale and to not overload the word "objective" using:

- **Threshold:** The lowest performance value we will accept. In the context of an architecture study, all alternatives are required to do this well or better to be considered.
- **Expected:** What stakeholders think they are going to get, and what they want.
- **Maximum Effective:** The level of performance where we no longer are willing to pay anything to increase it. The point of strongly diminishing returns or the highest performance we can justify having a use for.

One can easily add some other useful reference points, for example current performance, the performance of a previous generation system of interest, or the performance of competing systems.

At this point we are imposing a value scale on a performance scale. The classic form of a requirement is a binary value scale, zero value below the threshold, and full value above the threshold. This smoother scale implied here corresponds better to real needs and concerns and facilitates doing broad trades on capabilities in ways that a binary scale does not. However, in an acquisition environment with enforceable contracts being necessary, one may prefer the inflexibility but easier judgment of binary value scales.

The final step is combining the scaled and value-coded performances across many objectives, usually in some weighted format, to an overall measure of effectiveness. As suggested in Figure 10.9 that step belongs to in later ASAM steps, in Harmonization and Selection/Abstraction, assuming doing so is necessary.

It may not be necessary or useful depending on the situation. We find that optimization is often over-rated in that finding a true optimum is really only valuable if the optimum point is better than nearby points by amounts that substantially exceed the uncertainty in the estimates of costs and value. Often there will be many design points closer together than the uncertainty estimates, if we can even quantify the uncertainties, and this may make it a deep effort at optimization not worth the time or expense compared to simply searching out some better alternatives or looking for better information.

Functions can also be objectives, and objectives may be measures on functions. Having a given function might be mandatory, in the sense that the system of interest cannot possibly deliver useful value without that function being present. Some functions might be value-providing but not mandatory, in that it is possible we'd trade away having the function to get something else. In many cases we'll treat the presence of the function as mandatory but will measure the performance of the function.

NSOSA Discussion

One of our examples is the NSOSA project. The process of developing the objectives is described in detail in Anthes et al. (2019). In the case of NSOSA, the use-case or functional model is elementary because the system of interest's purpose is to make and deliver environment observations. Functionally, each of them is the same: Sense the physical phenomena of interest and send the data from the satellite collecting it to the ground network for distribution. Some will argue that it has to be significantly more complex, with each observation type involving many steps of processing and subtleties in calibration and so forth. This is true, but it is not architectural. What we mean by that is the details do not affect the large-scale trades. Value, cost, and risk are driven by how good the observations are (better observations require better instruments, which means more cost) and how those instruments are distributed across satellites. The details are routine choices that flow naturally once one has selected the instrument performance level and mapped them to a satellite constellation.

Some observations are considered mandatory so they have threshold performance levels that are not "None." Some observations are considered optional and so have threshold levels of "None." In practice, it is usually necessary to assign the quasi-threshold performance level to nonmandatory objectives to structure the design process. If an observation is nonmandatory, that does not mean there is no lower bound to performance below which it is pointless to include the sensor that produces the observation. The point is that the optional observations were traded in and out as part of the study process, no instrument with performance below the quasi-threshold was considered for inclusion in a design. Some of the nonmandatory observations turned out to be consistently traded in almost all of the time because their cost–benefit was very favorable, more favorable than trading on higher performance on some mandatory observations. The observations (again, essentially functions) that were optional did have defined threshold performance levels meaning that if they were included in an alternative, the performance level would be at least the threshold, but they did not have to be included at all.

See Table 10.3 for examples of mandatory and nonmandatory objectives, with measures, adapted from the NSOSA study (Anthes et al. 2019) and NOAA/NESDIS (2019) with some simplifications for clarity.

TABLE 10.3
Examples of Mandatory and Non-Mandatory Performance Table for Some Objectives of Different Types

Observation Objective	**Measure**		**Units**	**Study Threshold**	**Reference (Legacy) Capability**	**Expected**	**Maximum Effective**
Global Vertical IR vertical temperature profile (mandatory)				Mandatory			
	Horizontal Resolution		Kilometers	50	14	10	2
	Temperature accuracy	per 1 km vertical layer	deg Kelvin	2.0 K	1.6 K	1.6 K	0.75 K
	Moisture accuracy	per 2 km vertical layer	Greater of % or g/kg	40% or 0.4	20% or 0.2	20% or 0.2	10%
	Update rate (to % global)		Hours	6	6	3	2
	Coverage density		%	80%	90%	95%	100%
	Data latency		Hours	2	2	1	0.5
Regional MW vertical sounding (non-mandatory)				**None**			
	Horizontal Resolution		Kilometers	50	N/A	25	5
	Temperature accuracy	per 1 km vertical layer	deg Kelvin	2.8 K	N/A	1.5 K	1.2 K
	Moisture accuracy	per 2 km vertical layer	Greater of % or g/kg	60% or 0.6	N/A	40% or 0.4	10%
	Update rate (CONUS)		Minutes	1	N/A	30	15
	Data latency		Minutes	10	N/A	5	1
Strategic objective			**Annual no-gap probability**				
Observation availability							
	Core observations			0.9	0.95	0.95	0.99
	Non-core observations			0.5	0.75	0.8	0.95

DARPA Grand Challenge Application Discussion

The DARPA Grand Challenge case provides a good illustration of how one may treat functions not part of the main operational group. The requirements on the DARPA Grand Challenge provided by DARPA were minimal (recall the discussion in Case Study #1 at the beginning of Part I of the book). The primary objective is to drive around the designated race course as fast as possible. But how to accomplish that is left entirely to the developers. Moreover, since the task had never before been accomplished, not even close to accomplished, nobody could know what functions were required to be successful.

It would have been clear to teams that functions associated with autonomously driving a series of GPS waypoints were required. Since a waypoint sequence is how the course was designated, there is no solution without the capability to navigate such a waypoint sequence. Also, it is clearly necessary to have an ability to detect, characterize, and steer around some obstacles that may appear along the driving course. Exactly what obstacles and how to go around them is unknown, but some capability is required. A team could send a team of sensor engineers and software designers to brainstorm numerous capabilities and functions that likely would be useful for accomplishing the task, but no one would know for certain what set of integrated functions would work to accomplish the task.

Suppose the team steps away from the brainstorming on sensors and algorithms for a while and considers other parts of the operational taxonomy, the parts associated with testing and deployment. The experience of the first race made it clear that nobody knew how to accomplish the task. There was no reason to believe that one could reason one's way to a working solution without extensive field trial-and-error experience. A successful system is going to have to be in the field, running (hopefully) representative courses, and learning from the experience. What does this tell us about the set of functions that should be included in the architecture?

How about the ability to have a human drive a vehicle around a designated course, recording what is happening, while a GPS waypoint steering algorithm is computing driving commands (but not executing them) and image recognition software is assessing what is open road and what is not? The algorithms see a left turn with clear field and plan to take it at 40 km/hr. The human slows to 20 before taking the corner. If the recording is good perhaps the human will note "road looked like sand on something smooth, and gently cambered the wrong way, not something to take doing fast." That would be very significant to know, especially as such observations built up. That would be a whole operating mode for the system that is useless and wasteful when considering the operational moment alone (the race), but very useful and value producing and risk reducing when considering the entire system lifecycle. The point is this would come from considering the macro-objectives like "maximize the likelihood of completing the racecourse" and "minimize the expected elapsed time on the racecourse" within an overall strategy for building and operating the vehicle. Our use-case and activity models and objective values and ranges can capture what we want to do, but we need to fold over that the architectural thinking on what we want to accomplish. The DARPA Grand Challenge case highlights this effectively because we cannot fall back on "I'll just work on meeting the requirements" because that is severely incomplete description of what the goals are.

Value Models in MBSE Tools

The standard format for incorporating value models into an MBSE model is through a combination of value properties, constraint functions, and parametric models. The process outlined above translates into MBSE elements as follows. The problem model is built as an implementation neutral logical model. Each function, mandatory or non-mandatory, equates in that model to a use-case and a refining activity. Associated with that function will be a set of value properties that correspond to the dimensionalized and measurable objectives in Figure 10.9. Those may be embedded in the functional model or are part of a standalone block model sometimes referred to as an analysis context. The information contained in those models will be equivalent to that shown (for a specific example) in Table 10.3. That information converts into MOEs via constraint equations embedded in a parametric model. Those constraint models would implement the trades between objectives and could include mandatory criteria through Boolean checks in the constraint models. In SysML, this would be a parametric model in a package structure relevant to the overall study plan. In UAF, one would most likely put all of the above into the operational view and use the operation-specific UAF model elements (including parametric models).

The second element of this process is the linkage between the lowest level of the objectives model and the performance estimates produced by the solution design models (produced in solution structuring). The overall structure is the same, value properties and constraint models, but now part of a second structure linked to the physical decomposition of the system.

The choice of exactly how to implement each element depends on the organizations' overall MBSE strategy and the capability of the tools they use. With a powerful tool set the entire process can be implemented in a single toolset. Tools are changing rapidly, and anything written here is in danger of becoming obsolete quickly. As an example current as of this writing (2025) the widely used Cameo and Magic tools (Catia 2024b) support capture of all elements listed above. The functions can be captured in SysML or UAF activities or blocks. The information like that shown in Table 10.3 is captured in generic or instance tables (where they can be used internally by the tools or exported to external tools). The translation of the measurable objectives to more abstract measures of value can be done in parametric models in SysML or UAF. Alternatively, exported data can be incorporated into other computational tools for analysis and visualization.

We can walk the process starting from Table 10.3. Collecting global infrared temperature and moisture profiles is an observation function that can be expressed independently of how it is implemented. Global infrared profiles can be collected by satellites in low, medium, or high earth orbits. There are many ways to accomplish it, but the information in Table 10.3 expresses the capability, with bounded measures, neutrally with respect to means of implementation. The information in the table corresponds to value properties in a SysML or UAF activity or block model (exactly how is a modeling style choice). The translation of a design performance point into normalized value ratings would be done by parametric models in SysML or UAF.

The physical view (from solution structuring) would translate the physical characteristics of specific constellation designs (number of satellites, orbital assignments, selected instrument performance) into the objective performance values in Table 10.3. In the MBSE tool environment, this is a parametric linkage from the physical view models to the value models in the operational view. The performance computation will not be implementation neutral; it will be deeply embedded in the discipline of the solutions. For example, the objective of Table 10.3 requires the global infrared profiles update rate. This is a relatively straightforward computation but depends on detailed models of both the instrument used to collect the imagery and the orbital dynamics. If the observation is collected by a low earth orbiting constellation with a fixed view sensor (as is primarily done today), the analysis requires computing orbital tracks and fields of view over the globe. If the observation were collected from medium or high orbits with scanning sensors, then the performance can be either orbit dominated or sensor operation dominated, depending on the exact constellation structure. In most realistic cases, this will require use of dedicated modeling and analysis tools with two-way interchange with the MBSE tools.

Summary of Problem Structuring Products

To carry out problem structuring, you should almost always focus on:

- Create context diagrams (physical, logical) that show the system of interest and everything that it interacts with. When there is trade space, when it is not clear what exactly should be inside and what should be outside build the diagrams to show that and prepare to trade on the choices.
- Working through a taxonomy of use-cases and anti-use-case types, prepare detailed use-cases (or mission threads) for every value driving, cost driving, or risk driving system behavior. Look carefully in the taxonomy for possible value, cost, or risk driving behaviors that may not be connected to primary operations (that instead show up in support, testing, deployment, or threat responses).
    - Either use-case notations or activity models are good formats for this. You can choose based on organizational practice or customer preferences.
- Start building out the objectives tree, aiming for the point where objectives are measurable and dimensionalized. The objectives tree can be captured in MBSE notations as value properties on elements of an operational, problem-domain model linked to effectiveness and utility models in constraint models. These can be fully captured in an MBSE tool environment, when the environment is extensive (Catia 2024b), or work through table export to external analysis tool environments. Most likely of elements of the analysis (e.g., visualizations) will be outside the tool environment.
- Don't expend large efforts on functional or objective descriptions where they are not value, cost, or risk driving. That work will most assuredly need to be done but can be deferred into systems engineering.
- Don't neglect how the development strategy, the program template, and other lifecycle considerations will impact the problem description and what

factors are value, cost, or risk drivers. If you have a protoflight system (you build it once and fly what you build) and you break it during development and test, you may fail entirely. The risk of failing your system during its development lifetime may be greater than the risk of premature failure while operating.

SOLUTION STRUCTURING

From a description point of view, there at first seems to be little to add to standard practices when it comes to solution structuring. We want to describe solutions, so use whatever notations are typically used and known to be effective given the domain of the system of interest. If the system of interest is an airplane, represent airplane designs. If the system of interest is an integrated network system, use something relevant to that. In the MBSE world, the default choice can be SysML or UAF practices as adapted to the domain of interest. Sometimes there are well-understood domain adaptations, sometimes not, but either way the issue is specifically architectural, it applies to all design work in that domain.

While this is true, there are other issues that are much more specifically architectural. In an architecture study, we should be generating and assessing many alternatives. We should be doing broad-based trade analyses. So the notations and methods we use for representing solutions should be supportive of making those trades. We need ways of representing the design alternative space and generating alternatives that we can assess within that space. Moreover, an "architecture" is often best seen as a class of solutions sharing some common attributes. We are not, in an architecture study, normally trying to downselect to the single best design, but we are trying to downselect to a class of alternatives sufficiently constrained (but just sufficiently) to facilitate detailed trades and development.

A core goal is to achieve something like what was shown schematically in Figure 9.7. The idea is to generate a large set of alternatives, which we can assess value and cost (folding risk into value or leaving it for the next dimension of the analysis). The value-cost position of each is plotted as a marker point. The "convex hull" of the points, the boundary of maximum value at given cost, is the efficient frontier, representing the best possible performance at a given cost. The digital engineering challenge is to accomplish it within the digital engineering toolset. Generating and interpreting efficient frontier charts is a topic for the next sections. The point here is that without work in solution structuring to generate the design points and evaluate them within the digital environment in a way that scales to the size of the evaluation space, we will be cut off from accomplishing the task within the environment. The concern in solution structuring is how to generate those points in a way that facilitates doing the analysis at a scale necessary to generate efficient frontier analysis. Good efficient frontier analysis usually requires generating, and analyzing, 100 or more alternatives. At that scale the generation and analysis process typically needs to be at least partially automated. But it must have the analytical fidelity in cost and performance/value for the results to be usable in decision-making and the method used to generate the alternatives has to be smart enough to generate a significant number of points that will end up near the efficient frontier.

Gaining the required fidelity requires the right tools, and those are not general, they are very application dependent. Cost tools, for example, for satellites, aircraft, and software are entirely different. A satellite cost tool with sufficient fidelity for one application would have nothing to do with a software cost estimating tool for another domain. So the fidelity question can only be answered by the architect understanding the particular demands of the domain.

A generally useful technique, which can be linked to digital engineering approaches, is the "Zwicky box" or a form of combinatorial design (Zwicky 1967). This approach involves choosing one from column A, one from column B, and so on, usually linked with some selection rules. Ideally this can be used to generate automatically large numbers of alternatives that can be assessed also at least somewhat automatically. Another term for this idea is "design vectors." We think of a design alternative as a vector in some space. The dimensions of the space define the space of designs that can be generated. A given dimension might be bounded continuous (any value can in principle be provided by a design) or discrete (there are finite alternative to choose from). In this concept, generating a Zwicky box of designs is like sweeping out the vector space, establishing the dimensions and the ranges of each dimensional variable.

This is easiest to illustrate by example. Consider a reduced case from the NSOSA study (Anthes et al. 2019). The study broke out 38 environmental observations from space. Two of those observations were global non-real-time imagery and global infrared vertical soundings. The former observations are used directly in multiple weather forecasting missions but especially relevant in seasonal and climate forecasting plus some short-term forecasting in areas not covered by real-time observations. The latter observations are key elements of global numerical weather forecasting (dominantly 3–7 day forecasts of all types including hurricanes and winter storms). There were also six strategic objectives. One is particularly important here: availability. Essentially the probability that, over an extended period, the capability would be provided at the threshold level of performance or better.

We can do the combinatorial form of generating alternatives for these two observation objectives alone in four parts:

1. Choose the orbital regime from which to do the observations. For these two they can be done from low earth orbit (LEO), medium earth orbit (MEO), or high earth orbit (HEO). We can do vertical IR sounding from LEO, MEO, or HEO. We can do non-real-time global imagery from LEO, MEO, or HEO. Today, we do both from LEO and put both sensors on the same satellite.
2. Choose the orbital occupancy and flight rate. This will determine the observation update rate (one of the performance parameters) and the availability. This has more continuous variables, but some analysis and concept development can narrow it down. In practice, we choose from a small set of orbital configurations that give the desired coverage and from a discrete set of flight rates that cover from threshold to a maximum effective availability.
3. Choose the instrument. There was an instrument catalog that had instrument reference designs (and costs) for instruments spanning the performance space, from minimally acceptable threshold through stakeholder expected to maximum effective. As it turned out, for these two observations

each observation had its own instrument required except at the high-performance end where there was a combined more complex instrument.

4. Choose the instrument aggregation level on the satellites. There are more observations and instruments. Those assigned to the same orbits can be combined on a shared satellite or disaggregated across dedicated satellites.

In some cases, there is also a fifth choice, the acquisition model. For some orbits and constellation forms, there are commercial hybrid and purchased-service alternatives aside from conventional Government satellite acquisition.

We have three orbital regimes, and within each regime we have 4 to 10 reasonable orbital sets and three launch rates. This makes around 60 alternatives so far. There are three instrument performance levels, raising the alternatives to about 180. To count aggregation patterns, you need to know the other instruments and orbits involved, but it is at least a factor of three. To estimate cost, you need to know the instrument costs, size a satellite bus for the instruments on it, and calculate the number of launches to achieve the desired occupancy, but the point is, the number is large. This is pretty common, but setting things up this way will yield large numbers. This essentially drives us to two paths. First, we can embrace the combinatorics and automate all of the generation and assessment steps. Then make use of modern computer power to work through the large space. Second, we can look for ways to prune. This usually means looking for why certain approaches dominate others (always have superior cost–benefit performance) and eliminate the systematically poor choices and build on the good choices.

This requires interaction with satellite discipline-specific performance and cost tools. In the NSOSA case, this was done in a semi-automated way utilizing design center capabilities (Aguilar et al. 1998, Aguilar and Dawdy 2000). If something similar were done but based on aircraft as the platforms, very different disciplinary tools would be used. One of the NSOSA challenges was to address the potential for radical change, that is abandonment of long-established architectural elements and their replacement by something very different.

One possible path to radical change would be replace current mixed LEO and GEO regimes with a highly proliferated LEO constellation, something resembling iridium or the U.S. DoD Space Development Agency constellation. This assessment requires serious reformulation of objectives. A radically changed constellation would not be compared on an apples-to-apples basis on observations, though the comparison might be manageable at a higher level. See Maier (2018) for a discussion of these points. This is an important challenge in architecture studies. Objectives may be only partially solution neutral. There might be a set of objectives that is solution neutral as long as the solutions are not too different from current systems, but not solution neutral if we entertain any solution. Trying to entertain any possible solution might require very different sorts of assessments. In the weather observation case, we would face issues such as:

1. Today real-time weather imagery (what we see on television) is taken from a single camera on a single platform. From a proliferated LEO constellation, the equivalent would have to be synthetically assembled from many

cameras on many platforms, all moving. If the update stayed the same (unlikely), would interpretability be the same? We have no way to know without spending the full cost of such a constellation to try it.
2. A proliferated LEO constellation could deliver a far larger volume of radio occultation measurements than today at the expense of vertical soundings. How would that impact the performance of numerical weather forecasting, given that we would only have numerical weather forecasting codes optimized to the occultation data, something that won't happen unless we make the decision to commit to the alternative constellation?

Comparing the DARPA Grand Challenge and the NSOSA case highlights how both may use digital engineering in their architecture work, but in very different ways. It also highlights how the choice of program architecture may impact the choice of system architecture. In the NSOSA case, we can build a SysML block model with extensive use of generalization for different kinds of instruments, satellites, and constellations. We could include interface representations, but the trade questions, the architecture questions, are not about interfaces. The architecture questions are about what performance points to target and how to distribute across orbits. From a SysML model perspective, the architectural information would be embedded in the network of parametric relationships from instrument performance and costs to satellite orbits and the cost impact of satellite disaggregation.

Consider the situation of a DARPA Grand Challenge team and their need for making trade studies and building models. They have to make a strategic decision early on the type of vehicle to base their entry on, as discussed in the case study at the beginning of the book. Assume they have decided to use a more or less off-the-shelf sport utility vehicle or pick-up truck, adapted to allow robotic control of steering, brake, and power. They still have to decide what sensors to use, how to install them (fixed or gimbaled) and what processing algorithms and approaches to use.

The sensor choice problem is easily envisioned in Zwicky box or design vector terms, although that may not be as productive. The dimensions are the different sensor types and their attributes (e.g., monocular video present/absent, then video resolution, frame rate, field of view). Much as in the NSOSA case, they could generate a broad set of possible design vectors and then do a detailed trade study. But on what basis, on what data, would the choices be made? Nobody has accomplished the task before, and there is no theoretical basis to choose. There is no set of established tools to put in a design center. Alternatively, the team could install a minimum set and start testing in environments that mimic race day (or more controlled environments if they prefer to more easily acquire well controlled data). This is trade-by-doing.

A classic systems engineering counter argument would be that trade-by-doing will encounter expensive rework costs when various guessed combinations of sensors and algorithms fail to work well together. This is true, but how would they discover that the poor combinations are in fact poor combinations without trying them? If the science of autonomous driving were mature (which it most certainly was not at the time of the DARPA Grand Challenge), then established knowledge should suffice. But when there is no established knowledge base, then that approach is mooted. Another way to look at this is the cost of fixing a defect is not the whole cost. The whole cost has to include the cost of discovering the defect. If the only way, or the cheapest way, to discover a defect is to build and try, then doing extensive up-front study may be a waste. It would not be a waste if it allowed defect discovery (and then remedy) at lower cost. But if defects cannot be reliably discovered until live test, then might as well move to live test as quickly as possible.

Solution Structuring Key Points

Exactly what you produce in solution structuring depends on what constitutes a solution for your problem (a physical system versus a nonphysical). Even within that variation, there are key points that apply all or most of the time:

- Clearly define the trade space of solutions, whether they are physical or nonphysical. Understand clearly what kinds of solutions you are generating and be explicit. Readers of this book are likely to usually be concerned with physical solutions, things you buy or build. Whatever maybe the solution type, be explicit about it, and be explicit about what it consists of and what the trade space consists of. Both physical systems and policy solutions have components and trade spaces.
- The solutions generated should address the stakeholder concerns identified in purpose analysis and problem structuring. If they don't then either the solution space or the problem space (or both) need to be adjusted.
- Choose a structured, documented notation to describe the solutions.
  - If the solutions in an identified, well-known domain, use a notation from that domain.
  - For general system solutions (things you buy or build), the default choice should be from among the notations widely used in MBSE.
- Try to break the solution space up into multiple dimensions or components with alternatives within those dimensions (the "Zwicky box"). See if you can mix-and-match alternatives from among those building blocks.
  - Try to capture the box dimensions in the notation selected.
- Use heuristics from Chapter 9 to push the edges of the solution space. Does your solution space touch the edges of the value model in problem structuring (threshold to maximum effective)? Does it include both legacy continuation and radical replacement alternatives (regardless of how desirable they may be)? If not, do at least some reformulation.
  - Radical alternatives may not be favorable, it may be easy to eliminate them from consideration. If so, do the elimination, build the case for elimination, don't just gloss over.
- If the architecture of the program is part of the solution, then be sure you are describing program alternatives as diligently as system alternatives.

HARMONIZATION

As discussed in Chapter 9, harmonization is the part of APM-ASAM that brings together problem and solution descriptions. We have two primary concerns in harmonization: Checking for consistency and completeness and preparations for decision-making. On the former we can check for consistency and completeness of the models we have prepared in the previous steps, and we can check more broadly for architectural consistency and completeness. The point here is to recognize a difference between consistency and completeness with respect to modeling rules and with respect to the overall architecture process.

This is clearer by example. Supposed we have defined part of our solution via a block diagram or sets of block diagrams written in SysML. If they were written in some other notation, the same logic would apply, but would need to be defined in the rules imposed by that other notation. Consider a diagram like that in Figure 10.6. If one block exchanges data (or matter or energy) with another block, then both sides of that exchange need to have compatible interfaces for that exchange (the port types should match). If data is consumed by one block, then it needs to be produced in the other block (data is produced and consumed in activities). The exchanges need to balance. If a block has a behavior and that behavior needs some input, then the input needs to be available from the other block.

This kind of model consistency checking extends to other views. If the system is specified to have some kind of behaviour, then that behavior needs to be defined and allocated to some block that can execute that behavior. If a behavior in block A exchanges information with a behavior in block B, then the information exchanged needs to pass between the two blocks realized as an object or signal exchange and mapped to a physical interface. Ideally the tool in which the models are built provides extensive checking of these kinds of consistency rules. An advantage of adopting a well-defined notation, like SysML, for architecture description is that will come with built-in roles for model consistency and tools for checking.

Completeness is a distinctly different property to assess. Using model-based methods in a digital environment again can facilitate, but completeness does not map so directly to the properties of a notation or things we can directly check. If we are following the architecture description terminology of ANSI/IEEE 1471 and/or ISO/IEC/IEEE 42010 (42010 was developed from 1471) (Maier et al. 2004), there is a core completeness check built-in that goes from stakeholders to concerns to description products. Stakeholders have concerns that are addressed in views (this in in the definition of a view). The question "Have all stakeholder concerns been addressed" transforms into whether or not all concerns have a corresponding view and does that view meaningfully address the content of the concern? Since concerns are not necessarily stated rigorously (we embraced the notion that concerns should be captured as stakeholders see them), the question of how well they are addressed, or not, is judgment not modeling rigor. Consider how this trace can play out in our examples and in different modeling types.

Various stakeholders for the weather forecasting enterprise are concerned that future weather satellites will support medium-term weather forecasting, preferably with significantly improved performance over today's. That concern maps to functions of the constellation and performance objective measures on those functions. All of the observation types (e.g., global infrared vertical sounding, global microwave vertical sounding, Global Navigation Satellite System Radio Occultation) that support medium-term numerical weather forecasting are included in the functional view. The objectives in the value model have measures on all of those observations (observation quality is known to be important in the quality of the resulting forecast, although the exact relationship is very complex and only partially known). The last step, setting a no-degradation criterion, could be accomplished by putting thresholds on those performance measures no lower than the performance of the constellation to be replaced. However, there are other concerns. One of them is that the new constellation be affordable within projected budgets. It is possible (though unknown until the analysis is done) that fitting into projected budgets and achieving zero degradation of any component performance measure is incompatible. The NSOSA study addressed this by setting lower performance thresholds with the option to raise after the question of worst-case affordability was resolved in order to maximize trade space.

This type of trace, from stakeholder concern to model element and choices on how the models are parameterized and even thresholded, is the completeness checking called for in the standards. The process and criteria are structured and should be fully documented in a complete architecture description, but are not one-to-one correspondent to modeling rules. The models can draw the linkages. If stakeholder concerns are modeled as objects, say a requirement type, and not just text in a table, then they can have a trace path to use-cases and parametric measures on those use-cases. In the sense of the thread to assess being embedded in the model, it ensures that it is available when following this guidance. But the assessment itself, the assessment of whether or not the actual functions and actual values resolves the concern or does not is a human judgment. That judgment can be focused by the walk through process just outlined, starting from each stakeholder concern and seeing where it appears in a corresponding view and how it is dealt with.

The same exercise can be done for the DARPA Grand Challenge case, and it illustrates the importance of problem and solution structuring not just addressing the obvious aspects of the immediate system of interest, but of addressing the full space of stakeholder concerns and all that impact them. Imagine two contrasting stakeholder concerns for a stakeholder either funding a team or a team looking to attract funding (and so be able to build a vehicle).

1. We need to win the race to ensure this was a worthwhile investment
2. We need a good return on investment in building a race vehicle. The sponsor should not regret investing in this over something else.

Assuming a robust competitive landscape, it is impossible to ensure winning. Just ask the sponsor of any Formula One racing team. Most don't win anything, and very successful teams often suffer long slumps when they are competitively overtaken. The only semi-reliable way to the top of a competitive scene is to out invest your competitors. In the case of the DARPA Grand Challenge that would have meant have a bigger team, developing and experimenting with more alternatives, testing with much greater depth, and iterating much faster.

For the second concern a similar strategy emerges, but without the emphasis on the size of the effort exceeding others. The way to get a bad investment outcome in pursuing the race is fail to deliver any vehicle at all, to deliver a vehicle that fails to operate, and/or to fail to resolve any significant question related to the feasibility of autonomous driving. Regardless of the concern we want to resolve, we see that the resolution is largely in the architecture of the program not the architecture of the vehicle. What's important is that the team have a low risk approach to getting a working vehicle together, and low risk of failing to learn something broadly useful to the domain of interest in the process of building the vehicle. This will certainly have important consequences in the vehicle. For example, the vehicle should avoid relying on things that require extensive, complex engineering efforts that break no new ground. You don't want to develop something that others already have and fail at it simply because of the volume of effort required. If you fail at some part of the system, it should result in some significant new discovery.

As noted earlier, a typical aim point in architecture study, in any study of trade-offs in system design, is efficient frontier analysis. To do efficient frontier analysis we must have the following:

1. A value model (from problem structuring) that computes at least one overall measure of effectiveness from the attributes of any alternative we develop.
   a. For more reliable conclusions, we need to also be able to measure the uncertainty in the measure of effectiveness. See Maier et al. (2022) for an example of how this was carried out at the architecture level for one study.
   b. If we want to be able to extract cost-effectiveness conclusions, then MOE differences from different sources that produce the same shifts should be mutually indifferent.
   c. It may be useful to have multiple value metrics for different stakeholder sets. There is nothing wrong with generating multiple efficient frontier analyses.
2. A cost model that estimates a cost metric of interest for any alternative we develop.
   a. For reliable conclusions, we need uncertainty bounds on the cost estimate. See Maier and Wendoloski (2022)) for an example of methods for making these estimates.
   b. Be aware that expected lifecycle cost may not be a cost metric of interest to stakeholders. The cost metric chosen needs to be decision-maker relevant. Decision-makers who are budget constrained and decision-makers who are return-on-investment (ROI) constrained will care about different metrics.
3. Processes and tools sufficient to compute the above on wide enough range of alternatives to give confidence in any resulting analysis. Looking at a small handful (less than 10) will not give much indication of the overall efficient frontier.
   a. Architecture analysis is best done when the alternatives can be arranged into classes (really shared architectures) that still provide the flexibility to have many alternatives at differing cost. This allows discovery of what there may be shifts in architectural preference.

Harmonization Key Points

Key elements of harmonization:

- Use the syntactic and semantic rules of your description notation to check for model consistency. Model consistency is a good thing, but don't mistake it for overall consistency and completeness.
- Walk through both the problem domain and solution domain models from stakeholder concerns through to assessment. Are all concerns addressed?
    - "Addressed" does not mean the stakeholder is happy with the result, it just means you've taken it into account. Some good alternatives may leave some stakeholders unhappy.
- Is there a lot of material in the problem and solution structuring models that does not map back to stakeholder concerns? If so it is likely you are moving from architecting to engineering. Engineering validly takes up many issues that are not directly visible to stakeholders, but do you need to take up those issues now, or can you ignore them until you've made architectural decisions?
    - Good architectures are no bigger than they have to be so as to leave as much further trade space as possible for detailed design.
- Assess your ability to do cost-capability analysis (efficient frontier). Do you have metrics of overall value? If two or more metrics conflict is there, a way forward to understanding trade-offs between them?
    - It is more important at this point to know that trade-offs can be done than to actually make them. Until the problem versus solution space has been analysed, we don't know if such trade-offs are actually necessary. It is better to not worry about how to make complex trades if those trades can be made unnecessary by cleverer choice of alternatives.

SELECTION/ABSTRACTION

Selection/Abstraction is the key point of APM-ASAM for decision-making. Within the ASAM framework, we've investigated the purpose, structured the problem and possible solutions, done harmonization checks, and now have to decide what to do. The subject of this chapter is integration with digital engineering, so the focus in on the models and representations. For selection/abstraction, the overall purpose of the architecture effort must always be at the forefront, if we lose sight of the purpose, clever use of digital engineering will be pointless.

The first foundation of architecture decision-making is to know what to choose and what not to choose. The goal is almost never to choose a single best configuration. That happens later through a detailed design process. The goal is to make the choices that have to be made early, or where failure to make early carries serious consequences for value, cost, or risk. With those essential choices made, the rest of the engineering process can proceed, whether that means putting the concept out for bid, conducting detailed design in-house, or some other acquisition strategy. It may mean deciding not to go ahead at all. There are many systems where the sponsor/

architect would have been better off just not proceeding and deciding that early when there was sufficient data available to show proceeding was a bad bet. When constructing portfolios (Wicht and Szajnfarber 2014, Maier 2019), there is a presumption (or should be) that most of the candidates for inclusion will be rejected. We should expect to examine many more early ideas than we forward to more detailed development. Nobody is shocked to find that a venture capital firm rejects most of the concepts presented; this is an example of that same principle.

We return again to the two cases, NSOSA and the DARPA Grand Challenge team. What choices does the DARPA Grand Team face, and how would models they prepare help them make those choices? The team would face the following, in a rough order of dependence from first to later:

1. Should we proceed at all? Are the reasonably likely returns against organizational objectives worth the required investment? If not, go do something else with the in-house resources needed for early start up.
2. If proceeding, what does the architecture of the program look like, and what gates would we have to judge continuing forward or dropping?
   a. If they assess they will need external support to achieve essential objectives, how soon does that support need to materialize to make the overall effort worth continuing? This would have to be based on a quick estimate of timelines for vehicle preparation and some number of cycles of testing and the minimum level of support needed to gain a reasonable return on the organizational objectives.
3. What is the vehicle approach? Should they plan to adapt an off-the-shelf vehicle or plan to build a vehicle?
   a. This decision will branch on either trying to obtain a suitable vehicle or open a design process on a vehicle to build, not to mention gathering the necessary mechanical engineering and construction resources.
4. What is the sensor approach? Specifically, in what order should sensors be investigated and worked for incremental integration onto the vehicle?
   a. The diversity of possible sensors is large and certainly exceeds the capacity of any reasonable university based team to meaningfully experiment with given the hard time line of a fixed race day. Given the lack of knowledge of what sensors will work in the scenario, and it being non-plausible to find a best mix other than by experiment, there needs to be chosen order of integration and experiment.
5. What is the software and incremental integration approach, including the approach to test instrumentation and learning by test.
   a. A key factor in success will be the ability to sequential integrate sensors onto the vehicle (assuming the idea of a custom go-anywhere mechanical system is dropped at decision 3), try them in the field, and learn from those trials. This will require a software approach in particular that is friendly to incremental integration and re-integration.
   b. Classic system verification testing is with the expectation that the system will work to specification, and if it does not, it needs to be fixed, and the root cause of the failure to work be identified and fixed itself.

But in this case, there is no expectation that anything will work in the operational environment the first time, and the reasons for the failure will likely only be known after the tests are conducted. This implies a significantly different strategy for data collection and analysis during test, something that needs to be developed early in the process as it is a key and enduring element of the overall approach.

How might digital engineering models support these decisions? For 1–3, in particular, it would be very high value if the team was able to do a charrette cycle that resulted in a traceable cost estimate. Even if the estimate was known to be wide variance, the ability to do a fast charrette and get cost out would be hugely informative for decisions 1–3. This is one of the major use-cases of concept design centers (Aguilar et al. 1998) and one of the major reasons both Government entities and corporations have invested in building them. A university would not likely have such a capability, but the point is that an integrated modeling environment, which included cost estimation, would be a key capability for the class of architecture decision we see in the DARPA Grand Challenge case.

Looking at the full set of decisions another thing that stands out is the value of being able to integrate across digital engineering and digital project management. These decisions heavily touch on schedules and management of development processes and not just the physical system itself. There is also a need to model at the integration level, being able to model modular interfaces that facilitate incremental development. Good digital engineering for this case will embrace incremental and agile development, not just waterfall systems engineering. There is no reason using notations like SysML and the associated tools should be severely biased to a waterfall development model, but that speaks to the surrounding method that organizes the tools and directs what specific models should be built and how they should be managed.

The NSOSA case highlights similar if also very different issues. The sponsor wanted recommendations for an overall architecture for future environmental satellite capabilities, meaning there isn't a proceed/don't-proceed decision at the top. But there is a top-level decision on radical alternatives. The decision hierarchy can be structured as follows:

1. Should NOAA replace the long-standing basic mixture of LEO and GEO satellites with a "radical alternative." Radical alternatives could include replacing everything with a proliferated LEO constellation (like the large communications constellations from SpaceX and others), or with a proliferated MEO constellation (like GPS), or through radically different acquisition such as data-as-a-service for most/all data.
   a. If the radical branch was favourable, it would then set off a whole series of other analysis efforts related to how to replace existing services.
2. If the basic constellation architecture is not replaced, then there are a series of smaller re-allocations possible. For example, whether to consolidate imaging in high orbit (using some non-GEO high orbits) or to maintain imaging in both GEO and LEO regimes.

3. The next series of choices have to do other orbital allocations, such as which orbits to use for space weather observation and whether to commit to additional space weather specialized orbits.
   a. Around this area, we can also look at significant acquisition alternatives less radical than full out-sourcing, like utilizing GEO rideshares; there are many commercial GEO satellites that sometimes do offer rideshare, rather than dedicated satellites.
4. Then we choose performance targets for the various observations, a process that is deeply dependent on how instrument cost increases with higher performance, whether or not new technology can be counted to lower the cost of higher performance, and to what extent higher level metrics, like forecast accuracy, are sensitive to instrument performance versus to data quantity from proliferating instruments.

Many of these choices are very amenable to classic efficient frontier analysis. One way to examine the radical alternatives question is to generate some radical constellation alternatives and compare them on an efficient frontier basis with conventional variations. The study found that radical constellations were not competitive, by large margins, on an efficient frontier basis under a robust set of assumptions. This paper (Maier 2018) explains in more detail for the interested reader. The key point for this chapter is not whether radical constellation alternatives are competitive, but that efficient frontier analysis was a strong approach to making that determination. Hence, the ability of a digital engineering representation to generate such analysis is the indicator of its worth in architecture studies of this type. Similarly, efficient frontier analysis allowed for assessment of the preferred performance regions for different constellation architectures and some acquisition alternatives.

Current notations (SysML and UAF), methods, and tools are of mixed utility in carrying out large-scale efficient frontier analysis. They bring some important capabilities to bear but are limited in important areas. The basic building blocks are present. As long as you can represent alternatives as instances of blocks (including in varying sets), then the value properties of those blocks, as set in the instances, can be chained through parametric blocks to MOEs. The weaknesses are tool rather than notation related. In real situations, the computation of MOEs from block values may involve dedicated tools (aerodynamic programs, rocket flight simulation, orbital coverage). Practically, they are unlikely to be brought fully inside the framework of an MBSE tool. The approach will probably be to pull data out of the MBSE tool, into computational tools, and then transfer to other visualization tools. This converts the problem from one fundamental to architecture notations into a tool-integration problem. In the larger scheme of MBSE, this is not surprising; any model-based engineering approach to large systems will inevitably involve integration among dedicated disciplinary tools.

Selection/Abstraction Key Points

Summarizing the key architecture description points and digital engineering integration for selection/abstraction:

1. Selection/abstraction must not make decisions that do not have to be made, given the stage of the development. Focus the selection effort on the specific decisions identified from stakeholder analysis that define the architecture, not the engineering decisions immediately adjacent.
2. Architecture decisions are as likely to involve the architecture of the development program as they are to involve the physical architecture of the system of interest. The digital engineering environment must include program architecture and be as capable of representing and assessing development alternatives as physical alternatives.
3. Efficient frontier analysis is commonly, and correctly, a goal for making selection decisions. Be sure that the infrastructure (MOEs, ability to generate significant numbers of alternatives, cost estimation capability linked to alternatives) is in place from problem and solution structuring.
4. Cost, or other measures of resource consumption (e.g., schedule, human resources), are essentially always needed to make selection decisions. The model for this need to be in-place.
5. Conducting efficient frontier analysis on large and complex systems is more demanding on tool integration than on notational limits. Plan for how to distribute the required models and computation over tools in a way that scales to the number of alternatives to study throughout problem and solution structuring. Realistically, hybrid combinations of tools are likely to be necessary rather than having everything in one tool.

ARCHITECTING AND THE DESIGN LIFECYCLE

A full commitment to MBSE and digital engineering is a lifecycle commitment. The majority of the effort will not be in architectural design, it will be in the main detailed design effort, production, testing, and deployment. That said, the lifecycle commitment to digital engineering will be greatly facilitated if the system development effort starts digital, if it is "born digital" as has been proposed (Roper 2021). In that circumstance we need to examine who builds the architecture-related digital engineering products, and what are built, and how it varies from one development scenario to the next.

We've outlined in this chapter the MBSE elements most associated with carrying out the APM-ASAM process. These are the most likely products of an architecture effort intended to lead to a digital-engineering-enabled system development. Who produces them depends on the overall development scenario. In a purpose-driven case, the architecture is most likely driven by the acquirer. In government acquisitions, this means the government acquiring organization will most likely build an architecture model to be used as the basis for a competitive (or sole source) system acquisition. The primary model here is a Government Reference Architecture (GRA) or, more generally, an acquisition reference architecture (ARA). The position of the GRA/ARA within the possible set of overall MBSE models is shown in Figure 10.10.

The acquirer is responsible (nominally) for:

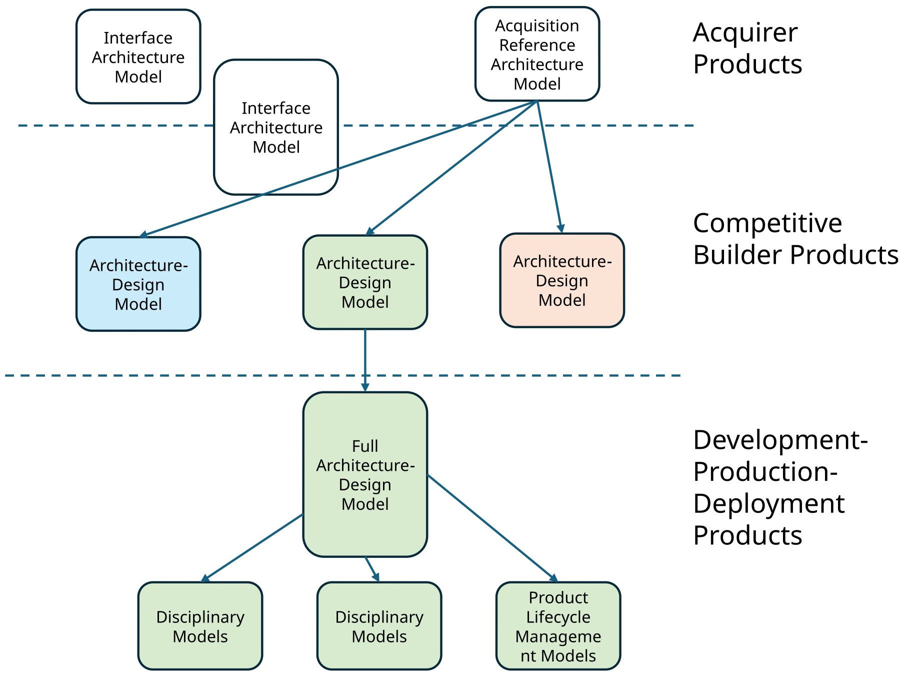

Interface Architecture Model
Interface Architecture Model
Acquisition Reference Architecture Model
Acquirer Products
Architecture-Design Model
Architecture-Design Model
Architecture-Design Model
Competitive Builder Products
Full Architecture-Design Model
Development-Production-Deployment Products
Disciplinary Models
Disciplinary Models
Product Lifecycle Manageme nt Models

FIGURE 10.10 Network of MBSE models in an ARA/GRA-driven system acquisition.

1. **The ARA/GRA**: This is a model of the non-tradespace/tradespace boundary. It includes the non-tradespace concept of operation, external boundary, and defines the external interfaces, at least up to where they may be subject to trade.
2. **External Interface Models**: In this approach external interfaces are defined via their own MBSE models shared between the offices responsible for the interfaces and user organizations.

Contractors/Suppliers build more detailed MBSE models on the supplied ARA/GRA. The acquiring program office may also build detailed models to cover the portions of the system the acquiring office is responsible for. For example, the acquiring office might be responsible for deployment systems or support systems once the system of interest is in operation. In a competitive acquisition, the competitors will build models at a level of detail sufficient for source selection, only the selected contractor will build a model that extends to full product lifecycle management and production.

In this approach the acquisition office is responsible for the ARA/GRA to include the non-tradespace concept of operations for the system of interest. Who actually authors the model will depend on how the acquisition office structures itself. Most likely they will need their own cadre of support staff to do the model development work. A similar model can be used for commercial systems whose characteristics (e.g., long life time, high consequence, strong regulatory constraints) mimic those of government-owned systems. When the system of interest is more like a typical commercial system and is builder-architected (Chapter 3) any ARA developed would be created by the builder organization as part of an integrated effort to build end-to-end MBSE models.

A few words are needed on some special cases, mainly analysis-focused versus development-focused efforts and architecture in agile development efforts. Some architecture projects are focused on the architecture of a specific, reasonably well-defined system of interest, and the clear intent is to build that system. The effort may be purpose-driven or technology-driven, but either way the clear intent is to build something. Other efforts may be analysis focused in that the intention is to enable a follow-on decision process that may or may not engage in developing one (or more, or none) system of interest.

Analysis Focused versus Development Focused Efforts

Analysis focused and development focuses architecture efforts primarily differ in whether or not there is a strong pre-supposition that the effort will result in building a system. As discussed with the APM, architecture efforts should always have a clearly identifiable decision. Deciding what to build is a clearly identifiable decision. Deciding whether or not to build anything at all is a decision. More analysis focused efforts may be harder to pin down as to what decision should be made.

In development-focused efforts, the products generated must include those driving development. Normally that will include models of the system of interest's physical configuration (a map of what must be built or purchased), the program to acquire and build, and cost estimates. In addition, there should be problem domain descriptions. Those can be bounded by how much of a problem domain trade space is really under consideration. In a digital engineering environment, the architecture product mix will be significantly determined by "working upward" from the development digital environment. The architecture effort needs to produce the products that feed the product development environment.

In analysis focused efforts, the mix of products is less clear. It is not automatically clear that the effort has to feed a product development environment. Analysis-focused efforts usually have a greater emphasis on problem domain models as usually one of the analysis goals is to identify if a system development is warranted at all, or the existing capabilities can work sufficiently, or if the return on investment is too low to warrant new system construction.

Agile Development and Architecture Runway

In an agile scenario, we deliberately intend to deliver many systems, each an incremental change from the previous. We embrace the notion that the destination of system development (or the study effort) should be adjusted as we go based on a learning process. How is architecture impacted in this situation? The agile community has the term "architecture runway" to refer to the decisions needed at any given stage that are impactful several stages ahead. Another way to look at this is via the notion of "architecture as invariants" or "stable intermediate forms," concepts discussed earlier in the book. Architecture runway is that set of things that are planned to be invariant, or at least intended to be invariant, several stages ahead of the current stage. Consider some of the following that are effectively architecture choices or decisions, even if possibly only implicit.

1. When we start building a solution, even a very incremental one, we are choosing the problem to "solve." We may leave important details of that problem for later elaboration, but the basic problem has been chosen. That is an important architectural choice.
2. The decision to proceed incrementally is an important decision, a point made earlier when we introduced program templates and is covered in more detail in Chapter 12.
3. In a software project as soon as we start developing software, we've made choices on programming language, execution environment, development tools, and so forth that are significant architectural choices. Unwinding those choices if they prove to be poor will be complicated and costly.
4. The way we sequence an incremental development effort is an important choice, and one that should reflect understanding of value, cost, and risk issues. Incremental development of function is likely to be effective at driving down uncertainty of user needs, but unlikely to driven down technology risks. A development path that rapidly confronts and drives down discrete technology risks may not engage with users. Which one works better depends on knowing where the risks are in advance of making the choice.

Some heuristics from systems architecting can be applicable in identifying and managing architecture runway. At the macro-level, the heuristics on architecting through invariants and stable intermediate forms are directly relevant, if not prescriptive. Effective runway is most likely to be found by investigating where invariance would have the highest impact on value, cost, or risk in a planned incremental development. The 4+1 guidance (Kruchten 1995) generically applicable to software is equally applicable in analyzing runway. Use-cases, the +1 element, are typically to be incrementally elaborated and advanced. So likewise we would expect the logical view to expand with each incremental stage, though it may be the hold place for important invariants. The concurrency model for the system is likely to be significantly harder to change after one as committed. If greater concurrency is likely to be needed as the system grows, then provisions for growing concurrency added very early will be valuable.

FUTURE WORK AND DIRECTIONS

This chapter provided a mapping of the APM-ASAM steps into typical products in MBSE/digital engineering environments. While the elements necessarily are all defined, there are four areas where further work is very desirable. First, architecting emphasizes starting with conceptual understanding with most (but not all) details suppressed. We are looking to "sketch-in" problem and solution descriptions, the progression process discussed in Chapter 9. SysML and UAF notations emphasise completeness and the eventual rigor (and checkability) of the resulting models.

This has obvious advantages for things in the right hand column of Table P.1 (the engineering column) but is a limitation in architecting. There are approaches for doing "sketches," such as incremental and deferred detail elaboration after setting up top-level elements, but we don't have widely accepted patterns for this usage.

The second area might be termed "patterns at scale." Many systems of interest consist of very large numbers of parts, with most of the parts being commercially supplied but formed into large units. Systems with substantial processing will have information processing subsystems with large numbers of computers, routers, networking parts, etc. Some of the information systems will themselves involve complex patterns of abstraction, for example cloud computing elements where large numbers of physical components interact to produce equally large (or larger) numbers of virtual components. We need some widely accepted patterns for representing and reasoning about elements like these that control the complication of high-level representations of the system of interest while also supporting full detailed lower level representations for construction. As discussed in Chapter 6, these patterns cannot rely solely vertical hierarchy (system-subsystem-component) structures, they need to work with layered structures and probably other compositional patterns better suited for heavy virtualization in designs.

We've emphasized the architecture is not just about the technical structure of a system of interest, it is also about how we build that system. Key value/cost/risk driving decisions are embedded in how we build (one-shot, incremental). This means that specification of the program concept may have a big role in architecting as specifying the technical concept. The consequence for digital engineering is the need to include project models (master-plan/master-schedule, cost) in the tool set used for architecture conceptualization, elaboration, and evaluation. Here a major challenge is the tool integration. There already exist well-established schedule tools, and most domains have well-established cost models and tools. If the technical modeling tools (like a SysML or UAF tool) is added to the set, and architecture information is spread across them, the tool integration challenge is obvious.

The UAF notation includes a project view and a set of project representations. This may ease tool integration, but the scalability of the project component in UAF tools is unproven as of this writing. Even if the tools provide outstanding support, most organizations are likely to already have built processes around dedicated planning and scheduling tools and may not be able to switch their approach to a single, integrated tool. Dealing with the various alternatives, and establishing widely accepted patterns for the different project templates, is an important area for future work.

Lastly, architecture is usually about making trades. The trade study (setting up objectives, creating significant numbers of alternatives, and doing analytical comparisons) is a standard element of architecting. If we have a narrow comparison (all alternatives share a roughly common block structure) and the number of alternatives is modest (single digits to tens), the ideas in current notations, like the use of instance tables and parametric rollups, are likely to be sufficient. There are difficulties, for example integrating the complex measure of effectiveness processes as discussed in the problem structuring section of this chapter, but that will largely be a matter of tool integration. What is more difficult is managing alternative generation and comparison when there are very disparate (share little or no block structure) or the number of alternatives reaches 100s or 1,000s so that managing tables becomes unwieldly. This is an area ripe for future work.

CONCLUSIONS

MBSE methods and tools have reached the maturity that they now extensively support the needs of architecture conception, elaboration (description), and evaluation. "Extensively" does not mean "completely," however. We can reasonably expect that most of purpose analysis, problem structuring, and solution structuring can be captured in current generation (2025) notations (SysML and/or UAF) and tools, plus simple traditional additions, like stakeholder interview records that will always be useful. There are areas where we must expect to have to work in other tool environments, which will generate a challenge in data exchange and managing large analysis projects.

The most important areas for external tools are likely to be discipline-specific analysis models needed to resolve technical deeper questions of feasibility or performance. Space systems, aircraft, power systems, manufacturing systems, and others have very powerful and established analysis environments. Those analysis results are often essential to assessing a concept value, cost, or risk, and so are part of architecting practice. In principle, it may be possible to reproduce the code or those tools within the simulation environments of large MBSE tools, or to encapsulate the established analysis tools and import them, but the effort level would be large and the return on investment is questionable given that there are only so many opportunities to do architecture work on systems large enough to require this level of effort. Well-defined export-import processes are likely to be more effective.

This conclusion stands in contrast to the situation in previous editions of this book where integrated modeling methods were available, but not with commercial tool environments capable of handling large systems. Well-defined integrated modeling methods capable to spanning from conception to detailed design now exist and are in wide use. The architect no longer has to work around the limitations, the challenge now is to make the best use of the rich set of digital engineering tools available.

EXERCISES

The exercises for this chapter are more like projects than discrete exercises. Depending on scope they constitute a semester (or longer, capstone) project.

1. For a system familiar to you identify the value, cost, and risk driving decisions. What models are commonly used to define such systems. Where do the value, cost, and risk defining decision appear in those models?
2. Using SysML, UAF, or some other integrated modeling method, built a model of a system familiar to you covering at least three views. If the models in any view seem unsatisfactory, or integration is lacking, investigate other models for those views to see if they could be usefully applied.
3. Choose an implementation technology extensively used in a system familiar to you (software, board-level digital electronics, microwaves, or any other).

What models are used to specify a system to be built? That is, what are the equivalents of buildable blueprints in this technology? What needs to be defined one level up (at the level of the system as a whole, not at the level of the implementation technology) to drive the implementation technology-specific models? How is that captured in system-level models?

NOTE

1 SysML and UAF graphics used in these chapters were made with Magic Systems of Systems Architect from CATIA, a division of Dassault Systems. We note at points here how features in comprehensive digital engineering environments such as Magic SSA can be used to support architecture methods.

REFERENCES

Aguilar, J. A., A. B. Dawdy, **and** G. W. Law (1998). The aerospace corporation's concept design center. *INCOSE International Symposium, Wiley Online Library*, Vancouver, BC, Canada, **8**: 776–782.

Aguilar, J. A. and A. Dawdy (2000). "Scope vs. detail: the teams of the concept design center." *Aerospace Conference Proceedings, IEEE* **1:** 465–481.

Aleksandraviciene, A. and A. Morkevicius (2021). *MagicGrid Book of Knowledge*, Kaunas: Dassault Systemes.

Anthes, R. A., et al. (2019). "Developing priority observational requirements from space using multi-attribute utility theory." *Bulletin of the American Meteorological Society* **100**(9): 1753–1774.

Bauer, P., A. Thorpe, and G. Brunet (2015). "The quiet revolution of numerical weather prediction." *Nature* **525**(7567): 47.

Catia (2024a). "Cameo Simulation Toolkit Documentation." Catia NoMagic Documentation. Retrieved 2024, from https://docs.nomagic.com/display/CST2024x/Cameo+Simulation+Toolkit+Documentation.

Catia (2024b). "Catia Systems Architecture and Specification." Retrieved 2024, from https://www.3ds.com/products/catia/systems-engineering/systems-architecture-specification.

Emery, D. and R. Hilliard (2009). Every architecture description needs a framework: Expressing architecture frameworks using ISO/IEC 42010. *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on, IEEE*, New York.

Estefan, J. A. and T. Weilkiens (2022). MBSE methodologies. In: *Handbook of Model-Based Systems Engineering*. A. M. Madni, Augustine, N. and Sievers, M. (Eds), Cham: Springer, pp. 1–40.

Friedenthal, S., A. Moore, and R. Steiner (2014). *A Practical Guide to SysML: The Systems Modeling Language*, Burlington, MA: Morgan Kaufmann.

Goldstein, H. (2005). "Who killed the virtual case file?[case management software]." *IEEE Spectrum* **42**(9): 24–35.

Hatley, D., P. Hruschka, and I. Pirbhai (2013). *Process for System Architecture and Requirements Engineering*, Boston, MA: Addison-Wesley.

Kruchten, P. B. (1995). "The 4+ 1 view model of architecture." *IEEE Software* **12**(6): 42–50.

Maier, M. W. (2018). Is there a case for radical change to weather satellite constellations? *2018 IEEE Aerospace Conference,* Big Sky, MT: IEEE.

Maier, M. W. (2019). "Architecting portfolios of systems." *Systems Engineering* **22**(4): 335–347.

Maier, M. W. (2022). Adapting the Hatley-Pirbhai method for the era of SysML and digital engineering. *2022 IEEE Aerospace Conference (AERO),* Big Sky, MT: IEEE.

Maier, M., et al. (2020). "Architecting the future of weather satellites." *Bulletin of the American Meteorological Society* **102**(3): E589–E610.

Maier, M. W., D. Emery, and R. Hilliard (2004). "ANSI/IEEE 1471 and systems engineering." *Systems Engineering* **7**(3): 257–270.

Maier, M. W. and E. B. Wendoloski (2020a). "Value uncertainty in architecture and trade studies." *IEEE Systems Journal* **14**(4): 5417–5428.

Maier, M. and E. B. Wendoloski (2020b). Weather Satellite Constellation As-Is and To-Be Architecture Description: An ISO/IEC/IEEE 42010 Example. Aerospace Technical Report. El Segundo, CA: The Aerospace Corporation.

Maier, M. W., K. L. Yeakel, and E. B. Wendoloski (2022). "Cost variance analysis in constellation architecture studies." *IEEE Systems Journal* **17**(2): 1928–1938.

NOAA/ NESDIS (2019). *Broad Agency Announcement (BAA) for Low Earth Orbit (LEO) Sounding Satellite and the Geostationary Orbit (GEO) Concept Exploration*, Washington, DC: NOAA.

NWS (2018). "Performance Management Homepage." Retrieved 2018, from https://verification.nws.noaa.gov/services/public/index.aspx.

Palmer, T. (2017). "The primacy of doubt: Evolution of numerical weather prediction from determinism to probability." *Journal of Advances in Modeling Earth Systems* **9**(2): 730–734.

Roper, W. (2021). *Bending the Spoon - Guidebook for Digital Engineering and e-Series*, Washington, DC: United States Air Force.

Wicht, A. and Z. Szajnfarber (2014). "Portfolios of promise: A review of R&D investment techniques and how they apply to technology development in space agencies." *Space Policy* **30**(2): 62–74.

Zwicky, F. (1967). The morphological approach to discovery, invention, research and construction. In: *New Methods of Thought and Procedure: Contributions to the Symposium on Methodologies,* F. Zwicky and Wilson, A. G. (Eds), Berlin, Heidelberg: Springer.

11 Frameworks and Standards

STANDARDIZATION CONCEPTS: FRAMEWORKS AND REFERENCES

The previous three chapters have developed the ASAM from the point of view of processes and heuristics and provided a guide to writing the resulting descriptions using MBSE methods. In this chapter, we review efforts at standardizing notations and methods. Standards efforts are an area of major progress over the editions of this book. In 1996 when the first edition appeared, there were almost no standardization efforts. Around 2000 the initial architecture description standard ANSI/IEEE 1471 appeared. This was refined in the ISO/IEC 42010 covering essentially the same ground, and the International Standards Organization (ISO) effort expanded to multiple areas with architecture, an effort that continues till the time of the publication of this, the fourth edition.

Against the backdrop of these standards efforts, we also had other efforts by the U.S. Department of Defense with the DoD Architecture Framework or DoDAF and Unified Architecture Framework (UAF) notation work by the Object Management Group. This latter did additional specialization work in MBSE notations, though as we saw in the last chapter, we can put general purpose notations like SysML to work in the architecture description role.

Tracing the history and cross influence of the various standardization efforts is primarily of historical interest, though it shows us how systems architecting was recognized within the history of systems engineering standards. What we need before describing the elements of the efforts of greatest importance to architecting is to understand the over-arching concepts and terminology. We have already introduced some, but we repeat here for completeness. We use here terminology primarily from ISO/IEC/IEEE 42010 (ISO 2022), which itself drew core terms originally from ANSI/IEEE 1471 (Maier et al. 2004), with a few additions and edits. The important terms to understand and relate here are architecture, architecture description, architecture frameworks, and reference architectures. The standardization efforts have primarily revolved around using these concepts as standardization vehicles.

Architecture: The working definition of architecture in this book has been “The set of decisions defining most of a systems value, cost and risk.” In a standards setting, we have definitions like “Fundamental concepts or properties of an entity in its environment and governing principles for the realization of this entity and its related lifecycle processes” (ISO 2022). For the purposes of this standards discussion, architecture is about the system of interest, not about models of that system or documents about that system. The architecture of a system is a *property* of the system.

DOI: 10.1201/9781003483595-21

Architecture Description: The succinct definition from ISO/IEC 42010 is "A work product used to express an architecture." The key point is that architecture descriptions (models, documents, etc.) are distinct from architectures much as drawings of a building are distinct from the key decisions about that building (decisions like how big? What rooms?) and from the building itself.

Architecture Framework: ISO/IEC 42010 defines "architecture description framework" (ADF) as "conventions, principles, and practices for the description of architectures established within a specific domain of application or community of stakeholders." An architecture framework is about how you write architecture descriptions and is some kind of standard for writing architecture descriptions within a domain or community.

Looking beyond ISO/IEC 42010 to follow-on standards in the 420XY family, we see "architecture XYZ Framework" as a generalized concept. Terms like "architecture modeling framework," "architecture analysis framework," and "architecture evaluation framework" can be used. Each refers to gathering "conventions, principles, and practices" for doing the XYZ activity (e.g., analysis, evaluation) in an architectural context.

Reference Architecture (Several Flavors): This is an emerging term used in distinct ways in different settings. One usage is where a reference architecture is a domain-specific standard on how to write architecture descriptions in that domain (that is, a type of ADF) and normative practices for what the architecture of systems should consist in that domain. For example, there is a banking industry reference architecture (Pavlovski 2013, Al-Fedaghi and Alsulaimi 2018). A second usage is where an organization that is acquiring a system of interest writes a reference architecture (often referred to as a "Government Reference Architecture" or GRA in defense acquisition circles) that defines the non-tradespace/tradespace boundaries for the system of interest. This type of reference architecture is specific to a system of interest and is intended to bound all proposed solutions for that system.

This conceptual background frames how we can explore architecture standardization. There are two primary threads. The first, and more common, is architecture description standardization. This seeks to standardize how architectures are described, much as blueprint standards work in construction. Most work in architecture standards is really on architecture description standards. Most of the discussion in this chapter will cover efforts in architecture description standardization. Architecture standardization, in contrast, is about standardizing elements of actual systems. This is analogous to building codes in civil construction.

A third leg could be architecture process standards. These may be elements of broader standards on system development, like ISO/IEC 15288 (EC 2002), or specifically targeted at architecture activities, as in ISO/IEC 42020 (ISO 2018).

While architecture description standards are the focus of discussion here, architecture standards, though not often called that, are hugely important. Consider the TCP/IP protocol suite that is the backbone of Internet communications. These standards are, in effect, architecture standards. They do not define a complete solution, and every equipment and software vendor can (and often does) develop their own compliant implementation. These standards define the communications services offered, and not offered, by compliant networks. The nature of the packet-switched architecture of TCP/IP greatly facilitates decoupling of applications from physical communications infrastructure and has allowed very rapid and decoupled evolution of both. Simultaneously, that decoupling makes provision of certain types of end-to-end services (e.g., some types of quality-of-service guarantees, some types of security) very difficult. The protocol set creates a clear set of trades on what kind of value delivering is easily supported and what is not.

ISO/IEC/IEEE 42010: STANDARDIZATION OF ARCHITECTURE DESCRIPTION

The object of this standard is the architecture description document (ADD). The ISO/IEC 42010 standard evolved from the earlier IEEE effort that resulted in the ANSI/IEEE 1471 standard. ANSI/IEEE 1471 likewise covered architecture description and not architecture, in the sense of the concepts discussion above. ANSI/1471 did place some constraints on how an architecture is developed by having normative constraints on stakeholders and concerns, but mostly concerned how an architecture is documented. By standardizing description methods, the hope was that descriptions would become easier to read, more consistently interpretable, and the methods and tools community would increasingly standardize on their descriptive side. A hoped-for side effect was better architecture, although the impact can only be limited. By analogy, hoping to improve architectures by improving architecture description is like hoping to improve building quality by standardizing how construction drawings are done. Drawing standardization is definitely relevant to the problem, but only touches on a limited aspect. Better drawings might make good and bad design choices easier to see and control, but it is the good and bad choices themselves that drive the quality of the buildings.

The approach of standardization of architecture descriptions, or more precisely the standardization of the elements and organization of architecture descriptions, was an outgrowth of the IEEE working group discussions (Maier et al. 2001) that led to ANSI/IEEE 1471. It reflected several realizations:

- Best practices in architecture (versus in architecture description) were virtually always domain-dependent. A good architecture made for a defense system was not the same as that made for a banking system.
- The definition of "good architecture" was domain and specific case-dependent. Good architecture was about addressing stakeholder concerns and delivering value in the context of the system. How you did that was entirely dependent on the context.
- On the other hand, there were some standardizable elements of architecture description that would facilitate assessing if a set of architectural choices was good or poor. While defense and banking systems would not (usually) share an architecture, they could share standard ways of describing and documenting an architecture, especially when both were describing the architecture of something implemented in a shared domain, like software.

ANSI/IEEE 1471 became the initial version of ISO/IEC 42010 and then evolved to its present form as practices evolved and additional concerns were incorporated. While ISO/IEC was not the first to assert its key principles for architecture description, they can now be considered the primary reference. ISO/IEC 42010 defines a set of description elements that are foundational:

1. Architecture descriptions should consist of administrative material, stakeholders, concerns, viewpoints, views, and rationale.
   a. The 42010 standard includes additional elements, but the above six are foundational and the focus of the discussion here.
2. Stakeholders and concerns are mandatory elements of the architecture description. Stakeholders and concerns are part of the architecture description space, and the architecture cannot be understood without them.
   a. There are very limited normative guidelines on what stakeholders and concerns must be considered.
3. Descriptive models are to be organized into discrete "views," and views are tied to stakeholder concerns. The connector is the viewpoint to which the view is conformant.
4. Each view has a corresponding viewpoint, and the viewpoint–view relationship is analogous to the type–variable or class–object relationship. Viewpoints must be documented with specific information, either in the architecture description or by reference.
5. The architecture decisions and the rationale for them must be provided. Rationale is a key part of the architecture description. This emphasizes the decision-centric nature of architecture over simple, providing a set of models and calling them "architecture."

The 42010 standard is somewhat larger than the points above that are primarily relevant to an architecture description being conformant to the standard. The standard also defines the conformance of an architecture description framework and a viewpoint description, topics of high interest to some groups. 42010 also includes some additional concepts, such as "aspects," but we choose to focus on the core set of standardization concepts here since they best align with the focus of this book.

The ASAM requires identifying stakeholders and concerns in purpose analysis. 42010 likewise requires their identification, but there is no associated 42010 process model. The recommended practice for ASAM is to document the identification in a stakeholder table. There is no required or recommended documentation form in 42010. 42010 also calls for "Stakeholder Perspectives" and "Aspects." These can be useful tools in elaborating the stakeholder analysis.

In ASAM we build models of the problem domain (Problem Structuring) and the solution domain (Solution Structuring). Chapter 9 provided a number of recommended practices and models (e.g., use-cases and anti-use-cases). Chapter 10 provided specific models in each ASAM basket, including use-cases in SysML form and the application of SysML behavior, block, and state machine diagrams to architecture description.

42010 requires that we organize our models into views, the views correspond to identified stakeholder concerns, and that each view be conformant to a documented viewpoint. What distinguishes the 42010 approach from most architecture frameworks is the explicit link to stakeholders and concerns. 42010 does not require any specific viewpoints or views. In contrast, some architecture frameworks have 30 or more viewpoint (or viewpoint equivalents). Those frameworks may not require using all those viewpoints, but they are viewpoint centric. The core of the framework standard is typically focuses on the models you use, not the concerns you address. It is an exercise for the user to decide how to prune the viewpoints for any particular situation. In contrast, 42010 essentially mandates that the architecture description author begin with stakeholders and concerns and use them to determine relevant viewpoints and views, and then develop the corresponding models.

This is compatible with the ASAM approach as elaborated in Chapters 9 and 10. The approach of Chapter 10 can be made conformant with the requirements of 42010 viewpoints and view by explicitly documenting the mapping from SysML (or some other MBSE notation) to the required elements of a 42010 viewpoint. For example, in Chapter 10, we use SysML block diagrams to document the physical structure of a system. This is a typical use for SysML block diagrams, but it not the only one. While SysML block diagrams are well suited to document the physical whole-part hierarchy of physical components, they can be as easily used to document other decompositions, such as a layered decomposition in software. From the perspective of 42010 standardization and conformance, whichever way you go it must be driven from the need to cover stakeholder concerns and be documented in a conformant viewpoint.

A 42010 conformant architecture description must show correspondence among view elements and between view elements and stakeholders/concerns. This is essentially the same as the consistency and completeness criteria we first introduced in Chapter 9 and explored in the MBSE implementation context in Chapter 10.

Finally, a 42010 conformant description must record architecture decisions and rationale (Section 6.10 of the standard). This is a further important distinction between the 42010 approach (rooted back to ANSI/IEEE 1471) and architecture frameworks that are description-only. The authors have at time seen long architecture presentations with many complex models but have been unable to find any answer to the question "But what are the architecture decisions?" The 42010 approach makes it clear that is not an acceptable outcome for an architecture description. The description must be rooted in the decisions it documents.

One currently available example of an architecture description for a significant system written to the 42010 standard is by Maier and Wendoloski (2020).

REFERENCE ARCHITECTURES

The term "reference architecture," as used practically and in in-development standards (Krob and Roques 2024, ISO To appear) is overloaded. There are three distinct concepts covered by the term reference architecture.

1. In an acquisition, a "Government Reference Architecture (GRA)" or "Buyer Reference Architecture" is a representation of the non-tradespace aspects of the system to be acquired. This would normally specify the required external interfaces, the required functional and nonfunctional properties, and required physical elements.
2. The reference architecture (ISO Committee Draft) for a domain specifies required elements that any system of interest in that domain has to possess to be compliant. A reference architecture in this sense is a capture of best practices for building systems in that domain. For example, see ISO (2022).
3. The third usage is a reference architecture that specifies how an organization requires architecture descriptions to be constructed, typically within a domain or acquisition in the responsibility of that organization.

The second and third usages are related but not the same. The focus of the second usage is the system of interest itself, not any ADD. The focus of the third usage is the ADD (or architecture description collection of models). The third usage overlaps with ADF. There is no point in drawing hard boundaries here, as realistic use-cases overlap. For example, the DoD Architecture Framework specifies (without hard mandates) the use of particular notations and views in writing a compliant ADD. A particular DoD organization responsible for acquiring multiple-related systems might (and some have) issue a reference architecture standard for any acquisitions within their domain. Taking the third usage above the organizations, an RA could further specify a more restrictive description framework than DODAF by requiring very specific notations and description products. In the second usage, they could further standardize not just descriptions but the resulting systems by, for example, requiring all architecture descriptions to be based on a shared data model including certain standard interfaces for internal or external connections of a particular type.

The focus of an RA of type three above is on standardizing architecture descriptions (properties of documents and models). The focus of an RA of type two above is standardizing architecture of classes of systems. The purpose of an RA of type one above is specifying the non-tradespace architecture for a single system of interest, a system that organization intends to acquire. All three usages of reference architectures, especially type one, is improvement to the processes of acquiring and building systems. An RA of type two or three is usually considered part of a standardization process as it will apply to a class of systems rather than to a single system. An RA of type one has a role in standardization, in that it standardizes the responses to an acquisition and the acquisition organization can impose some types of standards by their inclusion in the non-tradespace architecture description.

ISO/IEC/IEEE 42020 PROCESSES

In the ISO/IEC architecture standards set, 42010 covers architecture descriptions, 42020 covers architecture processes, and 42030 covers architecture evaluation. The strength of all three standards is concepts and vocabulary rather than specific models or processes. Much as 42010 does not standardize architecture or specific description

TABLE 11.1
APM-ASAM and MBSE Correspondences for DODAF Usage

APM-ASAM Element	ASAM-MBSE Model	DoDAF Correspondence
Orientation	Structured text material	Summary material
	"What stakeholders think they are going to get"	OV-1
Purpose analysis	Stakeholder table	Not called for, but essential
	Use-case diagrams or logical behavior taxonomy diagram	OV-6c (use-cases) or OV-5a (behavioral taxonomy)
	Problem framing analysis	Not called for, but important
	Logical context diagram	OV-2 diagrams
	Logical component BDD	OV-2 diagrams
Problem structuring	Use-case dialogs or behavior diagrams for threads	OV-6c dialog material or OV-5b diagrams (with DIV-2 materials)
	Operational state machines	OV-6b
	Value model (attributes and ranges)	Closest link is to Capability view, especially CV-2. Also SV-7 system measures
Solution structuring	Physical block hierarchy	SV-1 diagrams
	Physical internal connectivity diagrams	SV-2 diagrams
	Enhanced behavior model	SV-4 diagrams
	Parametric model (MOEs to performance)	SV-7 (measures only)
Harmonization	SysML-based consistency checks	No direct analog
	Use-case, mission thread walkthroughs	No direct analog, but see SV-5a and -5b below
	Enhanced behavior model embedding	SV-5a and -5b
	Parametric-value model MOE linkage	SV-7
Selection–abstraction	Efficient frontier analysis	No analog
	Remaining value analysis	No analog

languages but instead the structure and rules on ADDs, the process standard takes a relatively abstract approach to what within processes it standardizes.

42020 defines six inter-related architecture processes (ISO 2018) after Table 11.1:

Architecture Governance: Establish and maintain alignment of architectures in the architecture collection with enterprise goals, policies, and strategies and with related architectures.

Architecture Management: Implement architecture governance directives to achieve architecture collection objectives in a timely, efficient, and effective manner.

Architecture Conceptualization: Characterize the problem space and determine suitable solutions that address stakeholder concerns, achieve architecture objectives, and meet relevant requirements.

Architecture Evaluation: Determine the extent to which one or more architectures meet their objectives, address stakeholder concerns, and meet relevant requirements.

Architecture Elaboration: Describe or document an architecture in a sufficiently complete and correct manner for the intended uses of the architecture.

Architecture Enablement: Develop, maintain, and improve the enabling capabilities, services, and resources needed to perform the other architecture purposes.

Architecture conceptualization, evaluation, and elaboration are grouped as "core processes." The three core processes plus governance and management are also grouped. Looking at the processes defined in 42020 in the context of this book:

- The main concern of this book is how to carry out architecture conceptualization, with extensive attention also to architecture evaluation and architecture elaboration.
- Intellectually, the three core processes are linked. Conceptualizing an architecture, at least a good one, is dependent on being able to evaluate it. Conceptualization and evaluation are greatly facilitated by elaboration, since without some documentation of the architecture we are very limited in evaluation and even saying we have a concept.
- 42020 follows the idea first in 42010 and ANSI/IEEE 1471 of regarding "architecture" as conceptual and at least somewhat abstract, and distinct from the representation. This aligns with the idea borrowed from civil architecture that the architecture of a building is represented in drawings (and other models), but the drawings and model are not the architecture.
- Architecture governance, management, and enablement are of interest to organizations that need to architect multiple, inter-related things. The presumption is that there are many things worth architecting in the enterprise, and there is significant value in those architectures being jointly related and consistent. Recall from Chapter 7 some notions of collections of systems and how the architecture of the collection may relate to the individual architectures (systems-of-systems, families of systems, portfolios of systems). If an organization is concerned with one or more of these, then there are likely to be governance, management, and enablement concerns.
- In practice, if there are governance, management, or enablement concerns, it is important to understand if those are primarily exercised on architectures themselves or on descriptions. It is an observed problem in some organizations that they focus on governance and management of architecture drawings rather than the underlying content.

Returning to an example in Chapter 7, the Internet can be said to have an architecture in the TCP/IP and related protocol stack. Those elements are managed through the Internet Engineering Task Force (IETF) and associated bodies, with artifacts being the RFC series and other protocol documents. There are no SysML or DODAF products or other official "architecture framework" documents among them, but there is certainly an architecture and a governance process regardless. Unfortunately, the authors likewise have experience with organizations that have many architecture documents but little in the way of a useful architecture across a large set of inter-related systems.

The bulk of ISO/IEC 42020 is devoted to defining purpose, outcome, implementation guidance, activities and tasks, and work products for each of the five processes (clauses 6 through 10 of the standard). Clauses 6 through 10 constitute the bulk of the content to which an organization would need to be compliant if standard compliance were the goal. Many of the clauses refer to other relevant ISO/IEC standards. Within those clauses the activities and tasks section is the longest and contains the most content.

ISO/IEC 42020 aims to be applicable to a very wide range of systems, in terms of scale, complexity, and domain. As a result, the guidance in the clauses is generic and lack any domain specifics. This is by design and not by omission, and it is inherent in the broad applicability of the standard. The standard is thus best looked at as a source for standardized vocabulary and recommended practices. Aiming at compliance by itself cannot guarantee much in results. For example, under work products for architecture conceptualization (a major topic of this book), the ISO/IEC 42020 work products are [(ISO 2018), clause 8.5]: Architecture conceptualization plan, architecture conceptualization status report, problem space definition report, architecture objectives, quality model, and architecture views and models. That is a relevant list of work products and is consistent with APM-ASAM, but provides little specific guidance for any particular case. We take a more intermediate course in this book in providing specific approaches. APM-ASAM as implemented with SysML (Chapter 10) is not an ideal approach for every possible system, but for the broad class to which it applies, the guidance on architecture views and models it provides is much more specific.

DOD ARCHITECTURE FRAMEWORK

Introduction and Concepts

In the early 1990s, the U.S. Department of Defense (DoD) developed an architecture framework for Command, Control, Communications, Computing, Intelligence, Surveillance, and Reconnaissance (C4ISR) systems. The goal for this project was to improve interoperability across commands, services, and agencies by standardizing how architectures of C4ISR systems are represented. It also became a response to U.S. Congressional requirements for reform in how information technology systems are acquired.

The Architecture Working Group (AWG) published a version 1.0 of the framework (which became known as the C4ISR Architecture Framework) in June 1996. This was followed by a version 2.0 document in December 1997. The version 2.0 document was widely published and is available through the U.S. DoD Web sites, although it would now be considered obsolete. Subsequent to the publishing of the C4ISR Framework, it was further extended and designated the DoD Architecture Framework, now applicable to a much wider array of systems. The DODAF reached version 1.0 status in October 2003. A 1.5 version was released in August 2007. The most recent version is 2.02 from 2010 (DoD 2010).

Since then the intention is to transition from the DODAF to the UAF (OMG 2024) from the Object Management Group. At the time of this writing the transition is still in progress. There is considerable parallelism between DODAF and UAF. UAF was developed in part by merging concepts from many ADFs, so the content of the DODAF has large transitions, albeit with some name changes. The overall UAF structure is much more regular. It uses certain patterns of organization that repeat in a mostly uniform way. On the other hand, one consequence of the merger is that the number of discrete "views" or elements one write in UAF is quite large, more than 70. Doing all of them would almost certainly be counterproductive in any real situation. The user needs considerable guidance on what to write and how to pick and choose in any given situation. That guidance is not widely available. A related problem is that the scope of UAF is quite large, extending from the systems engineering elements like blocks, requirements, and activities to program elements like tasking and milestones, security representations, and even personnel. While some tools include support for all of these, no all-in-one tool is going to be best-of-breed in them all. Organizations are going to be attached to other tools, scheduling tools for example, and how to do the cross tool linkage without devoting most of the project staff to tool support is unclear.

While UAF is the future, we continue to discuss in DODAF terms for several reasons. First, most groups doing system architecting (as opposed to enterprise architecting) are working in DODAF terms rather than UAF. The concepts of DODAF are more directly applicable to the system scope. There is a fairly simple cross-mapping between DODAF elements and UAF elements, so one can adapt to UAF if there is clarity in the DODAF layout.

The DODAF requires the architecture description to be organized into summary information (also referred to as a "view") and three additional "architecture views," which are "Operational Architecture View," the "System Architecture View," and the "Technical Architecture View." These are often contracted in discussion to the "Operational Architecture" or "Operational View," the "System Architecture" or "System View," and the "Technical Architecture" or "Technical View." The "operational view of the architecture" is more consistent with the notion of view than any of the common contractions. Each of those views is composed of multiple models or "products" of various kinds.

The DODAF is a blueprint standard in that it defines how to represent a system's architecture, but it does not standardize any aspect of the architecture of the underlying system. It is possible to embed the equivalent of "building codes" using the DODAF mechanisms. For example, the Joint Technical Architecture (JTA) was a particular instance of a standards profile that could be incorporated as the technical architecture view. Although once there was an intent to drive compliance via inclusion of broader standards documents within system-specific architecture descriptions, it has not been successful. The JTA effort still exists, but it is not included within DODAF-compliant documents by mandate.

Users of this book will want to use the ASAM plus MBSE methods of Chapter 8–10, and the heuristics and insights from other chapters. How can that be combined with delivering architecture descriptions that are DoDAF complaint? Turned the other way, if DoDAF compliance is required (as it often is in US DoD programs), can the methods of this book be used to develop better DoDAF products that are more effective in the acquisition process? We find that good results can be achieved through the following, which we discuss how to accomplish in the following sections:

1. Use ASAM, and the associated MBSE templates of Chapter 10, to focus the development of DoDAF products and improve their quality. Within a typical U.S. DoD acquisition, one can use the methods to build a Government Reference Architecture (GRA) specific to the system of interest to define the tradespace/non-tradespace boundary and facilitate e-Program features (Roper 2021).
2. Capture an analysis of alternatives efficiently within DoDAF products, even when the number of alternatives considered is large (as we often want it to be).
3. Embed required U.S. DoD acquisition documents, like a Concept Definition Document or CONOPS, within a digital environment.

ASAM Mapping and other Correspondences

Working down these three areas, first there is a natural mapping from ASAM and MBSE-integrated practices (Chapters 9 and 10) to ISE/IEC 42010 elements and DoDAF products. The APM-ASAM methods provide a structured approach to generate DODAF products, or it might be better to say, they constitute a structured approach to architect a system and produce an architecture description within the DODAF standard.

The first required element in DoDAF is “Summary Information.” This is contained in the “all-view” and is denoted AV-1 Overview and Summary Information and AV-2 Integrated Dictionary. Both are simple, textual, or tabular objects. The first is information on scope, purpose, intended users, findings, and so forth. The second is definitions of all terms used in the description. The second is best captured in table form, something supported in most MBSE tools. Much of the summary information is a logical consequence of the orientation process defined in Chapter 9.

The second required element, the operational view, shows how operations associated with the system of interest are carried out through the exchange of information. This typically corresponds to an elaborated problem description, products of the Purpose Analysis and Problem Structuring phases of ASAM in Chapters 9 and 10. The DODAF operation view is defined as:

Operational View: A description of task and activities, operation elements, and information flows integrated to accomplish support military operations.

There are nine individual models defined within the DODAF operational view of the architecture. Each has a specified modeling language, although none of the languages is defined very formally. Some are entirely informal, as in the required High Level Concept Graphic (OV-1), while others (such as the Logical Data Model, DIV-2) suggest the use of more formalized notations, though they do not require it. The UAF structure, and its notation, aligns directly with the traditional requirements of the DODAF for the operational view. The defined elements are:

High-Level Operational Concept Graphic (OV-1): A relatively unstructured graphical description of all aspects of the systems operation, including organizations, missions, geographic configuration, and connectivity. The rules for composing this are loose with no real requirements.

Operational Node Connectivity Description (OV-2): Defines the operational nodes, and activities at each node, and the information flows between nodes. The rules for composing this are more structured than for OV-1 but are still loose. The natural correspondence in SysML is a block model (both block definition and internal block diagrams) where the blocks are the operational nodes. How the concept of an operational node corresponds to systems, people, facilities, or organizations is left to the architect. How to make the correspondence will be variable with the application.

Operational Information Exchange Matrix (OV-3): A matrix description of the information flows among nodes. This is normally done as an augmented form of data dictionary table. It can be derived from the block model above.

Command Relationships Model (OV-4): A modestly structured model of command relationships.

Activity Model (OV-5), Both a and b Parts: Essentially a data flow model for operational activities, which are seen as functions expressed at the operational level. In SysML terms, this will be an activity model organized around mission threads, as discussed earlier, with exchanger elements corresponding to the logical exchanges (and occasionally physical exchanges) in the operational environment.

Operational Rules Model (OV-6a): Defines the sequencing and timing of activities and information exchange through textual rules.

Operational State Transition Model (OV-6b): Defines the sequencing and timing of activities and information exchange through a state transition model, which is usually quite formal. In SysML, we could use a state machine model for this, driven by events generated from the model of operational activities.

Operational Event/Trace Description (OV-6a): Defines the sequencing and timing of activities and information exchange through scenarios or use-cases. This is behavioral specification by example, as discussed in Chapter 8.

Logical Data Model (DIV-2): Usually a class–object model or other type of relational data model. No specific notation is required, but most of the popular notations used are fairly formal. This is a natural match for SysML blocks representing operational data objects using the normal elements of SysML value properties and types. The intent of the DIV-2 is to define the data requirements and relationships from a logical rather than implementation perspective.

As a guideline, it is suggested that OV-1, OV-2, OV-3, OV-5, and DIV-2 should always be provided. It is not mandated that they be, but it is typically done.

DODAF defines the system view as:

System View: Description, including graphics, of a system and interconnections providing for, and supporting, warfighting functions.

There are many individually defined elements within the DODAF under the system view, although several are just small variations on each other. The most important are:

System Interface Description (SV-1): This model identifies the physical nodes of the systems and their interconnections. It is similar to an architecture interconnection diagram in the Hatley–Pirbhai sense, described in Chapters 8 and 10. A graphic representation method is called out but is not formally defined. In SysML, we would use block definition diagrams.

Systems Resource Flow Description (SV-2): An SV-2 specifies the System Resource Flows between systems and may also list the protocol stacks used in connections. In SysML, these would be internal block diagrams and the definitions of the ports used to type interfaces.

Systems Functionality Description (SV-4): The SV-4 is used to specify the functionality of resources in the architecture (in this case, functional resources, systems, performer, and capabilities). The SV-4 is the behavioral counterpart to the SV-1 Systems Interface Description (in the same way that OV-5b Operational Activity Model is the behavioral counterpart to OV-2 Operational Resource Flow Matrix).

Systems Traceability Matrices (SV-5 a and b): Matrices tracing from operational activities to systems or system functions.

Systems Measures Matrix (SV-7): The SV-7 specifies qualitative and quantitative measures (metrics) of resources; it specifies all of the measures. The measures are selected by the end user community and described by the architect.

There are many other defined SV products, and some supporting products in other views (such as standards, mainly for communications). In assessing the DODAF standard, and relating it to processes described here, we have to engage directly in the purpose of the ADD. Consider two distinct use-cases for an ADD. The first is focused on the ADD as an evolutionary step in the complete design. The acquirers write an ADD that defines the non-tradespace aspects of the system of interest (sometimes called a Government Reference Architecture or GRA in US DoD acquisitions) and the developers elaborate on that model. The GRA would be detailed in operational concerns and very light on the system view, normally only covering untradeable aspects of the system design. The elaborated models built by developers will be very heavy on the systems side with a primary goal to capture the design above the disciplinary level (meaning above the point where components are specifically mechanical, electrical, etc.). The second use-case is where the ADD is primarily intended to be consumed by stakeholders other than the system's developers. These stakeholders may be responsible for assessing the system of interest, or integrating with it, or planning how it fits into larger contexts. Those stakeholders will be concerned with what kinds of external interfaces the system provides or supports, what capabilities are provided and when, and what demands (e.g., logistic or hosting) is places on other systems.

The products listed are more aligned with the second use-case (assessment and integration) than the first (development). As discussed below, DoDAF has often been used as a requirement on the acquisition side, something for which it is not necessarily very well suited. Architectural decisions when acquiring a system virtually always include cost or the drivers of cost, but cost is not a DODAF view or product. Schedule likewise is essential in architectural decision-making for acquisition. Schedule is present in some DODAF-defined products but is not treated as a first-class view. On the other hand, viewed from an external perspective outward facing interfaces, capabilities delivered, and the like are value-driving and thus architectural. The two perspectives are not orthogonal, nor congruent, they overlap and link.

Careful use of the processes and modeling methods defined in Chapters 9 and 10 provide means for reconciling some of the conflicts. Consider Table 11.1 that summarizes APM-ASAM process elements and DODAF correspondences and also the concept of abstracting a model into more limited representation. External operational and physical interfaces are a focus of APM-ASAM and are represented in the recommended SysML notation view elements. These are obviously of high importance in a system development effort and are likely to be part of essential architecture decisions. They are also important to external stakeholder assessors and integrators. Both needs can be satisfied by either the same or an edited set of models. The value or performance model discussed in ASAM and the SV-7 measures in DODAF has a much clearer abstraction or simplification relationship. A full value model goes from acquirer-relevant, operationally based measures to technical performance measures on physical and software components of the design. In a description framework using SysML, this will be based primarily on parametric models and the associated diagrams. A simple listing of measures, scales, and ranges (what would be called for in an SV-7) is embedded in that parametric model. How easy it is to extract, rather than to require redundant description and maintenance, depends on the tool and care of the model architect. Conceptually, the simpler model is derivable from the more complex model.

Evaluation and Issues with the Use of the DODAF

The DODAF has been available and in wide use long enough for a body of experience to be generated. As noted above, an essential source of problems is mismatch between the use-case to be supported by the resulting DODAF-compliant ADD and the information in that ADD. When a system is to be developed using model-based approaches from the beginning and into detailed engineering, there are many issues that need to be thought and carefully specified. No single architecture framework known to the authors can provide an out-of-the-box solution, and careful planning is required in each case.

Likewise, the DODAF does not contain the elements necessary to cover all the architectural concerns in particular domains, for example software-intensive systems or space systems. Many domains have well-established best practices for architecture analysis and description insofar as the drivers of value, cost, and risk of often well-characterized within a domain. If 70% of the eventual development cost will be on software, then, logically, choices in how the software is structured and developed will be core contributors to values, cost, and risk. But neither the DODAF nor UAF contain any representations specifically directed at software and driving software decisions. The lack of domain-specificity cannot be considered a fault of the DODAF developers, as it was not part of their original purpose. It is a fault of those who specify use of the DODAF for purposes for which it is not intended. But accordingly it is a lesson for future directions. If an architecture framework is intended to be used for acquisition of major systems, and those systems are frequently software dominated, then an ADF must deal with those issues to be relevant to the intended use.

Just as an architecture must be fit for purpose, so must an ADF. If the DODAF is misused, the fault is much more in the misuser than in the framework. Nevertheless, it can be cited as a weakness of the DODAF that its parts are very loosely related. Very disparate concerns and models are lumped together into the views. Neither intra-view nor inter-view consistency is addressed at all. The individual models are so loosely defined, especially in some of the required elements, that ostensibly compliant descriptions can be produced that will not come close to guaranteeing interoperability. Because the DODAF adopts such a neutral stance to methodology, it cannot enforce stronger consistency and completeness checks. It is probably not possible to strengthen consistency and completeness properties without adopting much more formal modeling methods. The incorporation of ISO/IEC 42010 (or the earlier ANSI/IEEE 1471) definitions and concepts can be helpful in these areas.

Aside from this large issue, there are numerous practical and conceptual problems that must be resolved by individual user groups:

1. The notion of what constitutes an "operational node" in the operational view is unclear and inconsistently used. In the activity modeling sections of an operational view, an operational node is purely functional. In the higher-level diagrams, like OV-1s and OV-2s, common usage, and even examples in the defining documents, equates operational nodes to specific physical entities (e.g., an AWACS aircraft, a command center).
2. The high-level operational view diagrams are often, in practice, elaborately produced professional graphics with little technical content. Yet, they are held up in examples as centerpieces.
3. The hierarchy in the system view does not incorporate the widely accepted layering concepts from computer networks. At least five layers of the network stack (physical, data-link, network, transport, and application) are firmly established in theory and practice, but the concept is absent from the system view definitions. This is a barrier as systems are built, which incorporate the widely available off-the-shelf network components. It is unlikely that any single hierarchy model will solve this as the physical hierarchy from total system to configuration items is a very important representation (it is a representation of what we buy or build) even when there is a simultaneous layered structure).
4. The models in the system view do not make clear distinctions between node and connector types by layer. Within a particular layer (e.g., physical or network), the identity of the physical nodes and the nature of connection channel and data exchange is usually clear. But the DODAF models do not directly support that information.
5. The definitions of the elements within a view focus on diagrams rather than graphics independent models, which could have various visualizations. As a result, users fixate on the diagrams rather than the model content.

6. There is relatively little in the way of explicit consistency and completeness checks, especially between views. There has been some improvement on this point, but it is still immature.

One area of relevance with DODAF and moving to strongly MBSE-based acquisition structure is dealing with traditional documents, especially the CONOPS and requirements documents. A subset of DODAF products, mainly OV-5a and -5b, DIV-1 and -2 elements, OV-1 diagrams, SV-7 measures, and some abstracted SV-1 and other diagrams effectively, define most of what appear in a CONOPS document. Likewise, a requirements document specifies functional and nonfunctional requirements. Those requirements should correspond to OV and SV-7 elements. One way to look at the situation is that if there is a complete set of relevant models, the CONOPS and requirements documents should be derivable from the models. This observation is even stronger while following the APM-ASAM with MBSE methods discussed in Chapters 9 and 10.

There are two ways one might look at the situation. First, traditional CONOPS and requirements documents could be seen as primary (the "Authoritative Sources of Truth" or ASOT in current terminology). In that case, the problem is traceability from documents to models. The alternative formulation is that the model base should be the ASOT and the documents should be derived from the models. This latter approach has much more to be said for it. The model-based description is built on a notation (SysML or otherwise) with well-defined rules of syntax and semantics. The models will, in all practical situations, already be contained in an information technology environment that can be queried and managed. The challenge is developing adequate tools to parse the models and extract the relevant information into document form. This is not new, and it has been accomplished for many years (Rader 1991), albeit in a somewhat ad hoc way that has not become a standard template or method. This is an important area for further work. The way to treat this is important from a standards perspective. Do we want to standardize on traditional requirements representations, and then drive traceability into models, or do we want to standardize on model-based representations and then use that as point to build alternative representations? If we know we want to "project" from the model-based representation into a traditional CONOPS or requirements document, it will constrain the to-be standardized form of the model-based representation.

4+1 ORGANIZATION IN SOFTWARE AND INFORMATION TECHNOLOGY

The importance of software, and software architecture, within the practice of systems architecting is clear. Within a domain, there will usually be recognizable architecture and architecture description best practices, or else there would be nothing worth standardizing. Given the importance of software architecture in the systems architecting picture, what standards should we be aware of and incorporate into systems architecting practice? A powerful concept in the software domain is the "4+1 Model" for architecture (Kruchten 1995). The 4+1 model was developed for software but has applicability to more general information technology architecture. We touched on 4+1 ideas in Case Study 5 and Chapter 6 with the discussion of re-architecting a hypothetical information technology architecture. Here we will take up the 4+1 ideas outlined there in greater detail as an example of both architecture and architecture description standardization (a mixture of the type two and type three reference architecture discussed previously in this chapter under Reference Architectures).

The 4+1 approach requires that the architecture of a software system be described in five views. The information content of each view is defined, but there is no required notation. In most cases (but not all), one can fairly easily select a suitable notation from the MBSE notations already discussed, like SysML or UML. The 4+1 views are:

1. **Logical View:** This view defines the information comprising the software system, especially the persistent information, and the functions acting on that information. This is usually modeled with UML class–object diagrams or the SysML equivalent block-role-instance diagrams.
2. **Execution View:** This view defines the software–hardware relationship. Where does the software constituting the system of interest execute? This is effectively modeled with SysML block diagrams or resource diagrams in UAF and allocation relationships.
3. **Process or Concurrency View:** This defines the number and relationships between concurrent executing threads in the software. Any software system consists of one or more current threads. A simple, single application might be a single thread, and a very complex application might have many. There is no broadly standardized notation for this view, though several have been proposed and used in limited ways (Cochran and Gomaa 1991, Buhr 1993, Gomaa 1994).
4. **Development View:** This defines the software as seen by the developers. Its elements are code modules, files, and build structures. Notation methods for this are necessarily tied to the development environment chosen for the system development and will be a programming language and environment specific. More general models may be possible at a higher level, as where the team wants to capture how development is split over multiple applications with multiple developer teams. Component diagrams from UML and related diagram types can be used for this purpose.
5. **Use-Case View:** This is listed last and the "+1" since it ties the others together, but it is more logically first since we usually develop use-cases first. This view is a collection of use-cases to be executed by the software of interest. Use-cases are extensively supported in SysML, as already discussed.

From a system architecture perspective, we are interested in software architecture primarily to the extent that value–cost–risk drivers in software are or will become value–cost–risk drivers at the system level. Is that true, and will a 4+1 description help identify and capture those value–cost–risk issues that flow up to the system level? The logical, execution, and use-case views in 4+1 have obvious and direct connection to system-level MBSE models. Information and data models at the system level usually allocate directly to software since any complex persistent data is usually implemented in software. The execution picture is very much the physical block picture, with some possible adaptation for "information domain view" below. Use-cases in software are use-case at the system level but factored to the software boundary of the system. These parts link quite directly.

The development view in 4+1, on the other hand, is typically a concern only of software and unlikely to be a value–cost–risk driver. An exception may be where the approach to how software is developed somehow creates a value–cost–risk issue. Some examples are:

- The approach to development involves breaking the software across multiple development organizations. This can easily create risks because of distributed responsibility or mitigate risks by involving very specialized knowledge or intellectual property.
- The development view captures large-scale software re-use decisions, with various possible benefits, costs, and risks.
- The development view captures decisions on usage of very specialized implementation technology required to achieve safety or surety certifications and thus associated costs or risks.
- The software development view captures deliberate adversarial relationships, as would be the case if the system interfaces to other systems noncooperatively.

The process or concurrency view is typically poorly understood outside the software engineering circles but can at times create system-level issues. The concurrency design of software is an essential aspect of software architecture because it is important how a software application manages its real-time requirements. For example, consider a flight control system whose software must ingest sensor information and drive the control effects and do so with strict time behavior to maintain system stability. There may be a complex interplay between system-level choices and how the software is architected to do this. Some sensor inputs and effect outputs might need to be serviced at very regular times with narrow windows. Others might operate asynchronously but have little tolerance for delay in service when they are ready. If the software architecture can very flexibly accommodate a variety of synchronous and asynchronous timing demands, it may simplify the choices at the hardware level, and vice versa, a software architecture whose timing performance is brittle and will drive the hardware choices. Ultimately this may float up to the whole system level and if there are few or no compatible combinations of hardware and software choices, then system-level capabilities may be precluded or may be possible only with extensive constraints on both the detailed hardware and software side.

The process view is also important for understanding architecture alternatives at the information system level. Recall in Case Study 5 that horizontal integration of software across platforms could take multiple forms. There could be horizontal commonality in source or object code, which was development view commonality but not run-time integration. There could be shared use of concurrent services, which is a form of run-time commonality as well as development view commonality. The use of cloud-computing architectures further shares computing resources in a run-time view commonality arrangement. Each of these leads to different concurrent abstractions, from services to virtual machines. Each also has different sets of trades involving desired and undesired attributes.

The concept of "use-case factoring" from system to software level has already been addressed in Chapter 10 through behavior model decomposition from the core model to the enhanced model. The process we went through was to define the use-case at the border of the system of interest, decompose the system interest to lower-level components, and then elaborate a behavior model for a use-case until it was allocable to the components (refining the data to the physical interfaces of the blocks). At some point, the components become software components and the behavior allocated to the software component became a use-case "factored" and allocated to the software components. This is a process of behavioral refinement as outlined in Chapter 9.

While 4+1 is not a standard in the sense of an official ISO or ANSI standard, it is a de-facto standard for representing software architectures. It effectively captures aspects that are bridge between systems architecting, information system architecting, and software domains. It can be a useful standard on the edge of the system architecture world.

An Information Domain View

One technique we've found useful, inspired by the Execution View in 4+1, is the Information Domain View. In this view, the objects of interest are collections of computing hardware, networks, and software components that share some common policies, typically security or access policies. Each information domain is expected to have controlled, but only controlled, interfaces with other information domains. In a business environment, there might be separate domains for human resources, finance, manufacturing control (on factory floor operations), research and development, sales, public facing, and others. In military environments, domains may correspond to classification levels, fire-control interfaces, or command and control versus administrative domains.

In SysML terms, an information domain can be a block type and a diagram showing all the information domains in an enterprise and the exchanges among them is of high architectural interest. In UAF, a basic element is the "enclave" in the security view, see OMG (2022, pp. 211). A particular use of such a model is to identify all the inter-domain exchanges (and the mechanisms for those exchanges), not just the obvious exchanges involved in day-to-day operations. The more extended set of exchanges will include things like configuration (how is a domain configured and reconfigured?) and update (how is the software in a domain evolved and updated?). Inter-domain exchange interface models will obviously be of high interest for analysis of security vulnerability and should be played against important anti-use-cases.

CONCLUSIONS

Starting with ISO/IEC/IEEE 42010, there is a workable set of standards for describing architectures. The 42010 standard sets the overall framework, and SysML and UAF are effective MBSE notations. They can be used as-is or within the DODAF framework. The other 420xy standards are useful guides, but do not provide directive content. The 4+1 model for software architecture is an important addition to the overall picture.

REFERENCES

Al-Fedaghi, S. and M. Alsulaimi (2018). Reconceptualization of IT services in banking industry architecture network. *2018 7th International Conference on Industrial Technology and Management (ICITM),* New York: IEEE.

Buhr, R. J. A. (1993). “Pictures that play: Design notations for real-time and distributed systems.” *Software: Practice and Experience* **23**(8): 895–931.

Cochran, M. and H. Gomaa (1991). Validating the ADARTS software design method for real-time systems. *Proceedings of the Conference on TRI-Ada’91: Today’s Accomplishments; Tomorrow’s Expectations*, New York.

DoDAF (2010). *Department of Defense Architecture Framework (DoDAF) Version 2.02*, New York: DoD Deputy Chief Information Officer.

EC, I. (2002). *International ISO/IEC Standard 15288*, New York: International Standards Organization.

Gomaa, H. (1994). “Software design methods for the design of large-scale real-time systems.” *Journal of Systems and Software* **25**(2): 127–146.

ISO (2018). *ISO/IEC/IEEE FDIS 42020 Enterprise, Systems, and Software - Architecture Processes*, New York: International Standards Organization.

ISO (2022). *ISO/IEC 24039:2022 Information Technology - Smart City Digital Platform reference architecture - Data and Service*, New York: International Standards Organization.

ISO (2022). *ISO/IEC/IEEE 42010 Software Systems and Enterprise - Architecture Description: 69*, New York: International Standards Organization.

ISO (Committee Draft). *ISO/IEC/IEEE CD 42042 Enterprise, Systems, and Software - Reference Architectures*, New York: International Standards Organization.

ISO (To appear). *ISO/IEC 42042 Reference Architectures*, New York: International Standards Organization.

Krob, D. and A. Roques (2024). “On reference architectures.” *Systems Engineering* **27**(6): 1027–1042.

Kruchten, P. B. (1995). “The 4+1 view model of architecture.” *IEEE Software* **12**(6): 42–50.

Maier, M. and E. B. Wendoloski (2020). Weather Satellite Constellation As-Is and To-Be Architecture Description: An ISO/IEC/IEEE 42010 Example. Aerospace Technical Report. El Segundo, CA: The Aerospace Corporation.

Maier, M. W., D. Emery, and R. Hilliard (2001). “Software architecture: Introducing IEEE standard 1471.” *Computer* **34**(4): 107–109.

Maier, M. W., D. Emery, and R. Hilliard (2004). “ANSI/IEEE 1471 and systems engineering.” *Systems Engineering* **7**(3): 257–270.

OMG (2022). *Unified Architecture Framework (UAF) Domain Metamodel*, Needham, MA: Object Management Group.

OMG (2024). “Unified Architecture Framework.” Retrieved 15-September, 2025, from https://www.omg.org/uaf/.

Pavlovski, C. (2013). “A multi-channel system architecture for banking.” *International Journal of Computer Science, Engineering and Applications (IJCSEA)* **3**(3): 5.

Rader, J. A. (1991). Automatic document generation with CASE ON A DOD avionics project. *IEEE/AIAA 10th Digital Avionics Systems Conference,* New York: IEEE.

Roper, W. (2021). *Bending the Spoon - Guidebook for Digital Engineering and e-Series*, Washington, DC: United States Air Force.

Part IV

Introduction

The Systems Architecting Profession

The first three parts of this book have been about systems architecting as an activity or as a role in systems development. This fourth part is about systems architecting as a profession—that is, as a recognized vocation in a specialized field. Three factors are addressed here. The first is the embedding of architecting in the context of commercial or government systems developments, with primary attention to how architecting and organizational strategy overlap and interrelate. This is vital because architecting can only happen in a supportive organizational environment, whether in business or government. The second, the political process, the way government operates, especially in making value judgments and allocating resources, is important because it interacts strongly with the architecting process. Politics directly affects the missions and designs of large-scale complex systems. The third, the professionalization of systems architecting, addresses how architecting, a niche within the far larger engineering enterprise, is developing.

Chapter 12 covers the positioning of architecting in business and government in general, but its major focus is on how strategy relates to architecture. Organizations have strategies in the sense of objectives, selected means for achieving those objectives, and patterns for changing as their environment changes. The dominant means of executing strategy is the conduct of operations by an organization's personnel. However, organizations also build systems, create programs to build systems, and structure themselves as organizations. Building is, at least in part, an architectural activity. Because the architecting of systems is already the subject of most of this book, Chapter 12 focuses primarily on the architecting of programs, and how the architecture of systems, programs, and organizational strategy relate.

Chapter 13 is based on a course originated and taught at the University of Southern California by Brenda Forman of the University of Southern California and the Lockheed Martin Corporation. The chapter describes the political process of the American government and the heuristics that characterize it. The federal political process, instead of company politics or executive decision-making, was chosen for the course and for this book on architecting for three reasons.

First, federal governments are major sponsors and clients of complex systems and their development. Federal sponsorship plays a larger role in determining which complex systems are developed than sponsorship by any other single organization. Second, the American federal political process is a well-documented, readily available, open source for case studies to support the heuristics given here. Third, the process is assumed by far too many technologists to be uninformed, unprofessional, and self-serving. Nothing could be worse, less true, or more damaging to a publicly supported system and its creators than acting under such assumptions. In actuality, the political process is the properly constituted and legal mechanism by which the general public expresses its judgments on the value of the goods and services it needs. The fact that the process is time-consuming, messy, litigious, not always fair to all, and certainly not always logical in a technical sense, is far more a consequence of inherent conflicts of interests and values of the general public than of base motives or intrigue of its representatives in government.

The point that has been made many times in this book is that value judgments must be made by the client—the individual or authority that pays the bills—and not by the architect. For public services in representative democratic countries, that client is represented by the legislative, and occasionally the judicial, branch of the government. In principle, the executive branch carries out the value judgments expressed by the legislative, though as any observer of politics knows that authority is often delegated or just not exercised and so value judgments diffuse to many places. Chapter 13 states a number of heuristics, the "facts of life," if you will, describing how that client operates. In the political domain, these rules are as strong as any in the engineering world. The architect should treat them with at least as much respect as any engineering principle or heuristic. For example, one of the facts of life states:

The best engineering solutions are not necessarily the best political solutions.

Ignoring such a fact is as great a risk as ignoring a principle of mathematics or physics—one can make the wrong moves and get the wrong answers.

Chapter 14 addresses the challenge in the Preface to this book to professionalize the field. This means to establish it as a profession recognized by its peers and its clients. In university terms, this means at least a graduate-level, specialized education, successful graduates, peer-reviewed publications, and university-level research. In industry terms, it means the existence of acknowledged experts and specialized techniques. This chapter, originally written by the late Elliott Axelband, reports on progress toward such professionalization, reviewing standards, university programs, professional societies, and the increasing base of publications.

12 Architecting in Business and Government

Architecture is the technical embodiment of strategy.

Most engineering disciplines are only loosely coupled to the context of their application. You do not have to know that someone is working for a builder or a government department to judge the application of aerodynamics or circuit design. Aerodynamics and circuit design (and most other methods from the established engineering disciplines) are application neutral. The equations, design methods, and models are the same no matter who applies them. Of course, some elements are tied to specific application areas. A weather radar, an air traffic control radar, and the homing radar in the nose of an air-to-air missile share the same principles and equations, but each application area has very important application-specific knowledge. But even in that case it doesn't matter if the analysis is being done by a contractor designing a system to sell or by a government agency analyzing what to buy.

Architecting is much more deeply embedded in the context of its practice. Although many techniques will remain constant from one context to another, the architect's practice is heavily influenced by where it is carried out. Moreover, architecting is not just about the technical nature of the system of interest. It is about the structure of the program that builds and operates the system and the organization that either buys or conducts architecting. Builders and buyers have different concerns and will work to different models.

This chapter explores that linkage between architecting and the business or governmental organization in which it is embedded. The focus, as suggested by the opening quote, is on strategy. There are many occasions when the architect may feel more like a strategic consultant than an engineer. Sometimes this is a sign of healthy practice, and sometimes it is a sign of looming trouble. This chapter will examine how we might tell the conditions apart.

PROBLEM–SYSTEM–PROGRAM–ORGANIZATION

We can identify many different scopes of interest, but architecture and strategy are most clearly understood with four scopes: problem, system, program, and organization. The problem is what we are trying to solve or achieve by way of building a system. A system is a technical object we build or buy to solve a problem. The program is how the system is developed, produced, and deployed. The organization, really the organizations in plural, is the human construct that carries out the program. This is schematically illustrated in Figure 12.1 (refer also to Figure 1.6).

First-level systems architecting is about the relationship between problem and system. The architect seeks a consistent and harmonious connection between a problem

DOI: 10.1201/9781003483595-23

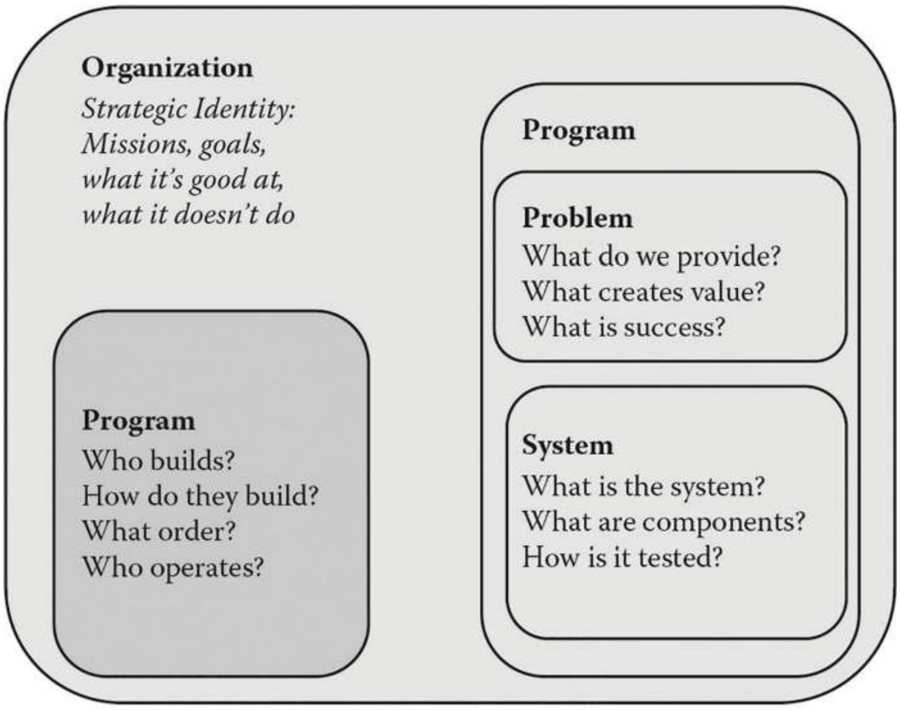


FIGURE 12.1 Organizations have programs that build systems in response to problems. Each exists in its own context, with issues unique to each context.

to be solved and a system to do it. But architecting is a problem-seeking activity not solely a problem-solving activity. Good architecting examines the problem scope in parallel with solutions. Good architecting challenges the problems presented and seeks better problems, better in the sense of more value for less risk and cost. The best architectural solutions usually involve reformulating the problem and may involve finding problems ahead of clearly expressed need.

DC-3 Example: The DC-3 was a success because it allowed the restructuring of the airline business. The Boeing 247, using nearly identical technology, was optimized for the operational environment of the time, where profits came from carrying airmail. The DC-3 had the size, capacity, range, and safety margin to allow profitable operation without subsidized airmail. To some extent, this was a happy accident, as the predecessors of the DC-3 (the DC-1 and DC-2) failed to pass the revolutionary threshold, although they were excellent airplanes. But the architects of the DC-3 had a customer who was looking at reformulating the immediate airline problem and were deliberately looking beyond the immediately stated version.

GPS Example: The revolutionary success of the GPS is one that continually looked ahead of the immediately presented problem statement. The original formulation by the U.S. Air Force program, with the slogan "Five bombs in

> the same hole," was not driven by an official operational need. The longer-term revolutionary aspects have come from exploitation into new problems. Some of the most effective exploitations have been in placing guidance on weapons, in surveying, in network synchronization, and in other civilian applications. Those applications represent not only the application of GPS technology but the reformulation of concepts of operation for both military and civilian activities.

In the scope of problem system, we talk about the fundamental structure of the system, its architecture. But the system has to be brought into being. Beyond the architecture design, it has to be fully developed, produced (in quantities from one to millions), deployed to users, and supported over a life cycle. We refer to these activities as the "program." A program also has a fundamental or organizing structure or an architecture. We identified a number of basic forms or architectural styles, which we discuss in detail in a subsequent section:

- Single object, waterfall construction, as in buildings and occasional one-of-a-kind systems.
- Single object, prototype development followed by incremental testing and improvement of the prototype unit until it is delivered for operational use. Most competitors in the actual Defense Advanced Research Projects Agency (DARPA) Grand Challenge race used this approach.
- Prototype development followed by serial production, with parallel manufacturing system development (discussed in Chapter 4).
- Breadboard-Brassboard-Flight incremental development, a typically hardware-centric process where functionality remains constant while environmental fitness is improved.
- Risk-spiral incremental development, where increments represent case-specific steepest-descent reduction of risk.
- Incremental delivery, where multiple systems are delivered with increasing functionality.
- Agile development, a further expression of incremental delivery.

Program structure may play an equal role with the architecture of the system in realizing stakeholder value. A fine system may be crippled by poor execution or doomed by a program structure that is inappropriate to the surrounding circumstances. Conversely, a well-chosen program structure may allow successful adaptation to errors in execution and surprises in technology or operational conditions.

Layered software is a response to rapidly changing technology and uncertainties in user demands. Well-chosen and implemented layers isolate areas of rapid change from each other and allow change in those isolated areas to proceed as quickly as technology or market changes demand. The Internet Protocol (IP) layer in the Internet protocol stack effectively separated the very high rates of change in physical communication technologies and network applications from each other, and allowed both to repeatedly abandon existing legacies independently. In negative contrast, layered architectures can introduce broad dependencies that may damage an organization's ability to deliver. A change to a deeper layer may have rippling effects in the higher-layer applications that use the shared, deep layer. If the deeper layers are pushed to change in response to user demands on applications, and the surrounding organization and technological infrastructure is unable to make changes without risk of affecting all applications, the layered structure may lead to development paralysis.

A program is carried out by an organization, which may be a single company or government division or a consortium of many. By organization we simply mean an organized grouping of humans whose purpose here is to carry out programs to build systems. Programmatic structures should be chosen to best fit the programmatically related objectives of a given development. In practice, the structures will also be influenced by the standing concerns of the organization. If the overall identity of the organization is well aligned with the programmatic and system mission, things are likely to go well. If the strategic identity of the organization conflicts with stakeholder concerns, programmatic imperatives, and system objectives, things are likely to go badly. This rising scope, and changing nature of concerns, is illustrated in Figure 12.2.

The strategic identity of an organization is the basic representation of what it does. The strategic identity should be a shared understanding among the organization's members. A strategic identity specifies what an organization's mission is, how that mission relates to other organization's missions, and how the organization's members can act in service of that mission. Organizations can be viewed productively as systems, and so have their own architectures. The architecture of an organization is its basic structure, not just in organizational chart terms, but in the fuller terms of expertise, experiences, internal and external working relationships, shared objectives among its members, and resources available to it. The architecture of an organization

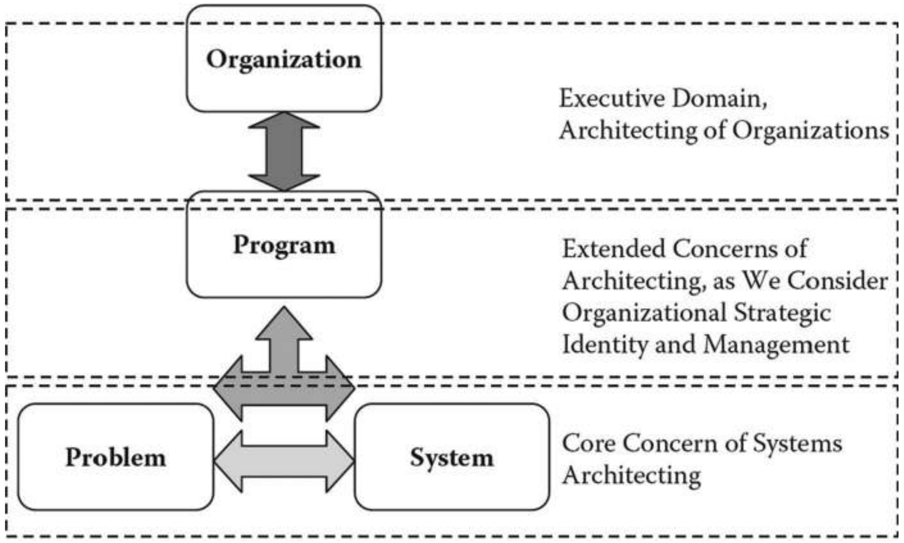

Organization
Executive Domain, Architecting of Organizations
Program
Extended Concerns of Architecting, as We Consider Organizational Strategic Identity and Management
Problem
System
Core Concern of Systems Architecting

FIGURE 12.2 Systems architecting is primarily concerned with the relationships between elements, whether at the level of system-problem, program and management, or (occasionally) at the level of organizations.

is not a principal concern of this book (Rechtin 2017), but we cannot understand the architecting of systems without considering how the hierarchy of contexts from problem to organization relate through the system and program.

STRATEGY AND ARCHITECTURE IN BUSINESS AND GOVERNMENT

In the classic model of architecting, the paradigm derived directly from classic civil architecting, the architect is an executor of the client's strategy. The client has a strategy. Perhaps it is to build a house well suited to his or her family's life, to build a profit-making facility in a given business, to combine business return with brand identity, or to build a long-term educational institution. Whatever that strategy is, the job of the architect is to understand it well enough to be able to produce a fit physical, technical embodiment of that strategy.

Architects do not create the strategy, although they may need to elicit the client well beyond just asking "what is your strategy?" Architecting accepts that the problem is unlikely to be presented in well-structured form and is probably fundamentally ill structured. With ill-structured problems, the process of forming a solution influences the client's understanding of his or her problem, and not just the architect's. Thus, to some extent, the process of working with an architect may help a client formulate his or her own strategy, even though it is not the architect's role to formulate the client's strategy.

As we talk about unprecedented systems, the sharp border between the architect and client in strategy becomes unclear. Who formulated the strategy of moving to operations and capabilities for global positioning beyond the concept in the original program? It was not entirely the client or the architect. As the GPS case study recounted, the client for the Air Force 621B program did not have the ambitious vision for a global, all-service program. The ambitious vision came from the program manager convincing higher DoD authority, and eventually encompassing other programs into that vision. The long-term GPS revolution was driven by organizations beyond the client and that in many cases did not even exist when the program was formed. These lateral exploitation applications, which are growing to the point they now drive global positioning well beyond one program, were partially inside the original architect's visions, but were not the original architect's responsibility. Many others had to become participants, typically independent participants in a collaborative system, for the revolution to happen.

When the border between architecting and strategy formation becomes fuzzy, the architect may find himself or herself acting more like a strategy consultant than an architect. Even though it is not impossible for this to be effective, it is fraught with problems. Architecting requires technical depth, and good architects have that technical depth. Effective strategy formulation requires much more knowledge and insight into the operational situation faced by an organization (whether business, military, or diplomatic) than is necessary for architecting, and requires much less technical depth. Both architects and strategists are bridging the engineering to operations gap, but they approach the gap from opposite sides. As such, they can be extremely effective partners but are less likely to be effective substitutes.

A basic embodiment of strategy produced by architecture is in the structure of the program. Architecture may also embody strategy through other technical choices, but program structure as a strategy is almost always present. To explore this, we need to consider different forms of strategy as related to technical system development, specifically static and dynamic strategies.

Static Strategies

A static strategy is unchanging, or slowly changing. Static strategies seek to understand the world and to determine a set of objectives for systems that will yield a superior position. Typical static strategies include the following:

- Be the low-cost producer or supplier. This involves leaning down design, production, and delivery systems relative to other competitors.
- Be feature superior to the competition. Deliver systems with superior quality, cost, and delivery, measured on the same scale as the competition.
- Bring superior firepower and concentration to the battlefield.
- Use concepts of operation similar to one's opponent, but with longer range, greater accuracy, and larger effect. Do what your competitor does, but do it better.

Architecting in a static strategy environment is relatively similar to classic systems engineering. The problem may still be ill structured, because we do not know where in the feature space we will find suitable problem–solution combinations, but the feature space is assumed to be knowable. The objective of architecting is to elaborate on the problem and solution spaces and find excellent combinations.

Dynamic Strategies

A dynamic strategic approach assumes that the playing field is continuously changing, or even better, that our actions can force the playing field to change. Instead of trying to beat competitors at an established game, we seek to create new games. We try to avoid head-to-head competition on features, cost, or delivery and instead choose actions that are unexpected by the competition and that the competition is incapable of imitating. A truly dynamic approach to strategy is to further assume that any dominant move we make is temporary and must be followed by a succession of game-changing moves.

This discussion of static and dynamic strategy is heavily influenced by the strategy concepts of Colonel John Boyd. A generally available reference to Boyd's ideas is provided by Richards (2004). His primary presentation of his ideas is the briefing patterns of conflict (Boyd 2007). In Boyd's terms, one can consider a static strategy as "attrition warfare" and a dynamic strategy as "maneuver warfare." Boyd was strongly on the side of the superiority of maneuver warfare, but it must be admitted that attrition warfare armies win wars too, albeit often at terrible cost.

Architecting in a dynamic strategy is a twofold process. First, the architect is driven to come up with unprecedented system concepts. A dynamic strategy has a continuous appetite for the new and unexpected. Second, the organizational processes and supporting system must be architected to support continuous and rapid change. An organization very good at executing a static strategy is unlikely to be good at executing a dynamic strategy, and vice versa.

The real world is not clearly divided between exponents of static and dynamic strategy. Even if one believes that a dynamic strategy is inherently superior, in considerable measure much of economic and military life is dominated by mature operational concepts where fierce competition in static strategy prevails. Just because one wants to "change the game" doesn't mean conditions are ripe to do it. Where the investment in legacy capabilities is very large, it is very uncommon for dynamic shifts to upset the entire operational picture quickly. Even when the operational picture can be changed quickly, it may settle down to maturity with time. A useful metric for understanding where static or dynamic strategies are likely to play a larger role is system obsolescence time or depreciation rate. How quickly after introduction does a system lose most of its value? In how much time will the original system owner be willing to throw away the system and find it is not worth of use?

- From the 1920s to the early 1960s, aircraft depreciated in 5 years, lengthening to 10 years. Military aircraft built at the end of World War II were of low value within 5 years and scrap before 10 years elapsed. This time lengthened considerably from the 1960s to the present, when both military and civilian aircraft are still flying usefully 25 years or longer.
- At the beginning of the space era, a given satellite design was useful for a few years at most. By the 1970s, satellite architectures settled down. Lifetimes of 5–10 years are now not unusual, and designs can be valuable longer.
- A 5-year-old computer is, with few exceptions, something to sell at a flea market. But innovation in things like operating systems has slowed way down, and many applications are in a phase of much slower evolution.
- Cell phones and related personal electronics are disposable on a 2-year timeline. However, the basic architecture of the cell phone has now been stable for over a decade, even as the internals continue their rapid Moore's law-driven evolution.

An organization should have an explicit strategic position about which it pursues its mission. Some organizations, those supporting stable mission areas or markets, logically pursue strategies that are mostly static. This is not a bad thing. If a static strategy that carefully focuses on stable sources of value and stable means of delivering value can achieve a competitive advantage against static strategy measures, which is very hard to match or overcome. As an example, various automakers, both United States and Japanese, established long-term competitive advantages that endured for decades (in different eras). But static strategies can be overcome by frontal competition and by "end-runs" when strategic conditions change.

Organizations that pursue pure dynamic strategies can likewise be very successful and can fail abjectly. An organization that solely pursues the unprecedented is vulnerable every few years. Even for the most capable, the business of producing unprecedented capabilities is very uncertain. Luck is required and runs of luck always come to an end. If the organization cannot weather a string of failures, it will disappear.

ARCHITECTURE OF PROGRAMS

As discussed above, the next step of context above Problem-System is Program. At the program level, we are concerned with the structure of the effort to develop, produce, deploy, and maintain the system of interest. Obviously, as with systems, there are countless possible such structures. We cannot enumerate them all. However, we can identify several program styles or repeating patterns of program organization, and the heuristics for their application. These program styles relate directly to an organization's pursuit of a dominantly static or dynamic strategy.

Single Pass, Waterfall Construction

The paradigm for this case is constructing a house or other building. The process normally proceeds very linearly: A design is developed and approved, contractors are hired, the building is constructed on the site, and it is approved for occupancy and delivery after completion. There are few or no intermediates in this approach. Modeling is conducted during design and may involve the construction of scale models, but we do not build trial buildings as part of the process. On occasion, some subsystem elements might be built early for testing. An example is building a unit of windows for a major skyscraper to test their weather integrity if they use an innovative method of holding the window glass.

In practice, there may be some level of incrementalism. For example, in a building complex, we may build all of the infrastructure but only some of the buildings in an initial phase, with the remaining buildings deferred for a later phase after the first set is occupied and in use. Sometimes a building is designed with options for remodeling or extension in mind, but the intention is minimal or no rework of what is built. The basic pattern is simple; we directly design and build the final system we intend to deliver in one pass, and we get it right the first time.

In INCOSE Handbook terms (INCOSE 2023), this is the sequential development template. We determine the requirements at the beginning and maintain them to delivery. We deliver a system a single time (or a single block).

This programmatic pattern is most applicable where:

- Only a single system is to be built and delivered.
- Risks are low. There is high certainty that a satisfactory system can be built and delivered at predictable cost from a design.
- The strategy is static. We can build a system in response to the strategy and believe it will be fit for the natural lifetime of the system.

One System, Prototype Iteration (Protoflight)

In this pattern, we build one copy of the system-of-interest, but understand it is a prototype, probably not ready for operational use. We iterate its design and construction, based on test results, but do not build additional copies. The ultimate production size is one (or a handful) that have been developed through testing and fixing. Unlike the classic waterfall, there is no pretension that we can get it entirely right on the first pass, we accept the need for learning and rework. The emphasis should be on learning; this strategy typically presumes that we are iterating the prototype because we have much to learn in the process of testing. We cannot just test to verify our original requirements were met, but we have to test to find out how the original requirements were probably insufficient for the task.

In INCOSE handbook terms (INCOSE 2023), this could be considered either the sequential case or the incremental case. To the extent that our sequential refinement of the object to be delivered results in a system of some utility prior to getting to the planned endpoint, the process is more incremental than sequential. If, on the other hand, we build essentially the whole system and then incrementally debug it, we are more in the sequential model.

The programmatic pattern is most applicable when:

- Only a single system is to be built and delivered.
- We cannot afford multiple copies of the system. Usually this means the hardware is expensive (relatively) and we can iterate based on software change.
- There is an architecture-level design that can accommodate learning during test and incorporation of that learning into the version of the system we have.

Serial Production

The basic pattern here is that we build one or more prototype systems, probably using the one-shot waterfall pattern, freeze a final design from analysis of the prototype, and then produce many copies of that prototype design. Alternatively, we may use one of the other patterns for prototype developments before freezing the design and proceeding to production.

This pattern is most applicable when:

- Many copies of the system are required.
- The cost of production is high relative to the cost of design and development. The overall cost is dominated by the costs incurred in production (typical for hardware-centric systems).
- Risks can be resolved by a prototype. Once we have the prototype, and have worked with it, we can have confidence that the produced system will be fully acceptable.

Breadboard-Brassboard-Flight

This pattern is an incremental pattern, in that we build a series of systems that are less capable than the final system and that lead to the final system. In the breadboard-brassboard-flight pattern, the series of systems that we build should all be functionally equivalent to the final delivered system but are not all environmentally suitable for operation in the delivered systems environment. In the classic version of this pattern, the breadboard system is spread out over laboratory benches. In electronics, it consists of large boards with many parts and no effort of design shrink. In optics, the components spread over an optical bench. In chemical engineering, the early versions are physically much smaller than the target version, with far lower capacity (the direction of improvement is increasing scale instead of decreasing size). The brassboard version has classically been shrunk to an operationally suitable size and form factor but is not yet fully hardened or reliable enough for operational use. The "Flight" version is the final version to be delivered to operational use.

Breadboard-brassboard-flight is an incremental development in the INCOSE Handbook sense (INCOSE 2023). Our requirements were determined at the beginning, they are not planned to change. But we build in increments; in this case the direction of incrementing is suitability for operation in the planned environment. All versions are intended to be fully functional, but not fully usable.

This pattern is particularly applicable when:

- Functional risks are low. We have high confidence that we can identify all of the desired functional characteristics early in the design process.
- Technology and implementation risks are (relatively) high. We have low confidence that we can build and package the desired functional characteristics in an environmentally suitable unit. If these risks were not high, the prototype and production strategy would be sufficient.
- Production numbers can either be very low (a single flight system), or be combined with the serial production pattern.
- The strategy allows for a static functional aim point.

Because of the second bullet, the pattern is mostly seen in hardware-centric systems. In most software-centric systems, the technology and implementation risks are relatively low. We know that if we can write functionally acceptable software, we can probably package it in an environmentally acceptable way. Obviously, many exceptions exist, but the point is that in software-centric systems we are typically driven by functional risks rather than technology and packaging risks. In contrast, in many sensor, aircraft, and spacecraft systems, we have mature knowledge of how to build a functionally acceptable system but not how to make it operate in the environmentally constrained environment of the operational target.

Consider the problem of sending remote sensing instruments to Jupiter. In most cases, the instruments we wish to send are well understood and widely used in terrestrial or even earth-orbiting environments. But packaging the instrument in a size, weight, power, reliability, radiation-resistant form factor that is survivable and reliable in the Jovian environment is a great challenge. This is the kind of thing well matched to this development approach.

Functional Incremental Delivery

The functional incremental delivery pattern can be thought of as the converse of the breadboard-brassboard-flight pattern. In the incremental delivery pattern, we again build a series of systems, each different from the previous, but the sequence grows in functional capability and not in environmental suitability. Each member of the sequence is fully usable in the target environment. In the classic version of this, each member of the sequence is not just fully usable, but each is intended to be delivered and used operationally. In commercial market terms, this is a series of incrementally developed products.

While we use the term "incremental" here in INCOSE Handbook terms (INCOSE 2023), this looks like the evolutionary development template described there. This depends primarily on what we intend to learn from the functional increments. If each increment helps us learn what users and other stakeholders really want, and we adjust our requirements (that is adjust out "endpoint"), then this is an evolutionary development in terms of handbook. The point of learning is to adjust what we build in a user-facing way. On the other hand, if what we learn by adding functional capabilities is about how we will best arrive at a fixed endpoint, then it would better match the incremental pattern presented in the handbook.

This pattern is particularly applicable where:

- The risks and uncertainty about what is functionally acceptable are high and can only be resolved by operational experience. No amount of requirements elicitation in the absence of a real system can be expected to resolve the questions about what functional capabilities the users really want.
- Risks in developing an environmentally suitable system are low. It is not difficult to meet user expectations of size, weight, power, reliability, or other physical quality characteristics.
- The cost, price, and revenue issues are such that multiple replacements of a delivered system are acceptable (or even desired).
- The strategy is dynamic, and we realize value substantially by adapting to new knowledge with different system configurations.

The third bullet is characteristic of software-centric systems, because software production deployment costs can be very low. The third bullet may also apply to systems with significant hardware content where market forces lead to rapid turnover. As an example, consider many consumer electronic segments where people rarely keep a device for more than 2–3 years and are willing to pay for replacements (as long as they offer new features).

The Agile Variation

Agile development is a variation on the functional incremental delivery approach. Agile emphasizes very short cycles of development, each cycle producing an incrementally improved (functionally larger) product. A generic functional incremental delivery approach does not force cycles to be of any particular length, they could be months long or longer. Agile also emphasizes continuous user feedback. This is a specific functional incremental feature that is not necessarily present in all instances. This goes to a conception of how a spiral or incremental process is organized in the large. Is the point of the spiral process to approach a pre-designated end point, but do it in a step-wise manner to control risk? Or is the point is to leave the end point open, to be determined through interaction with users during the incremental delivery process? In the former case, we do not emphasize user feedback, we emphasize working off functional increments in a way that controls risk. In the latter case, we emphasize getting continuous user feedback and use that to adjust the end point. Really, in the latter case there is no end point of development, we keep moving out in functional space addressing the most desired needs identified from user feedback.

The agile variation is most applicable when development cycles can be made very low overhead (as is possible in software projects but rarely otherwise) and we accept that the desired endpoint is not pre-determined but should be adjusted based on user feedback. This generally means that determining user needs is the highest risk element of the system, or that user needs are expected to change substantially on exposure to incremental versions of the system and so trying to pre-determine them is fruitless.

Risk Spiral

The risk spiral is an integrated combination of breadboard-brassboard incrementalism and incremental delivery. In the risk spiral (the concept original to Boehm), each cycle through development yields a system, though not necessarily one intended for operational use. The objectives of each cycle are driven by an overall assessment of risk. If the assessment is that currently the risks and uncertainties about what functions have value to users, then the next spiral cycle will emphasize a user-delivered system that can assess the value of functions. If the assessment is that the highest risk is engineering and packaging, then the next spiral cycle will emphasize the breadboard-brassboard-type of development.

The risk spiral is a close match to the INCOSE Handbook incremental pattern, in that its purpose is not to adjust the requirements for the end system, but its purpose is to resolve risks in getting there.

Each spiral cycle consists of all of the conventional activities of the waterfall: requirements analysis, design, build, integration, and test, as illustrated in Figures 1.5 and 6.1. Architecting in all spiral or incremental situations differs in two basic ways from one-shot or waterfall architecting. First, architecting becomes episodic. We do architecting every time we go around the spiral. Each cycle around the spiral requires decisions about the problem and solution content of that cycle around the spiral. Each cycle is a complete pass-through development and requires a set of architectural decisions on the concept to be developed (at the beginning of the cycle), and decisions on acceptance for use (at the end of the cycle). Second, we architect the invariants, or the things that do not change as we spiral.

The choice of an incremental development approach, versus a one-shot development or some other pattern, is an architectural choice. It is the choice of program architecture. That choice, of program architecture, is rooted in an understanding of the overall strategy and how architectural decisions embody that strategy.

This risk spiral pattern is particularly applicable where:

- There are high risks to developing a successful system, but those risks are understood and discrete. It is especially applicable to systems with well-defined discrete technology risks.

- The sponsors can be convinced that a risk is retired with a highly targeted demonstration. It is not necessary to build a near-full system or a partial system with user-facing features to convince the sponsors that a specific, identified risk has been retired.

The second bullet is particular to the risk spiral. Sometimes the best way to retire a risk is with something that looks very little like the final system. Imagine an active sensor system dependent on a laser achieving a threshold combination of power, pulse duration, wavelength, etc. Imagine that the required combination is a big risk because it has never been achieved in a package usable in the mission concept, but that the other elements of the system concept (processing, user interaction) are low risk. The risk spiral approach would have a cycle targeted on packaging the laser and nothing else. The resulting demo would look like a laboratory laser demonstration with obscure instrumentation for power, wavelength, beam quality, and so forth. It would not look like a prototype sensor in any way. Would this be convincing to the sponsors? If not, then the risk spiral with its discrete risk focus would probably not be suitable.

Collaborative Formation

In Chapter 7, we examined the concept of a collaborative system, a system formed by the partially or whole voluntary interaction of autonomous systems. We can undertake the formation of a collaborative system as a deliberate effort, although the fact that we cannot control all aspects means that there is an element of uncertainty. In creating a program whose goal is the collaboration formation of a system, we are deliberately choosing to orchestrate a process whose end point we cannot precisely predict. We must accept the uncertainty of the collaborative assemblage process in return for the benefits that it brings.

A collaborative formation approach is especially appropriate when:

- The strategy is dynamic, and we believe our power to shape is greater than our power to actually implement.
- The environment inherently contains multiple autonomous players, and it is neither sensible nor feasible to replace them.
- The risks associated with the strong preexisting players are more significant than the risks of a particular configuration being achieved or not.

STRATEGIC ARCHITECTING OF PROGRAMS

Given that programs have architectures, and that the architecture of the program needs to be considered in parallel with the architecture of the system, how does the architect go about it? Architectural thinking in business and government should consist of all of the following:

- Understand the organizational context in which architecting takes place.
    - Who are the competitors?

  - Who are the opponents (not the same as competitors)?
  - Is the organization involved with architecting a constant or variable? Is a new organization logically an outcome of architecting?
- Understand the overall strategy of the organization involved with architecting. What are the static and dynamic aspects of the strategy? What is the strategic identity of the organization?
- What financial criteria are used to fund the system development? Is it return on investment (ROI) driven, budget allocation driven, or something else?
  - Budget allocation driven occurs when development funds are relatively fixed and each effort must compete for a portion of the funds
  - ROI-driven means the funding pool is flexible, there is no "budget" to allocate, but a development must justify its return to pull any funds from the flexible pool
- Use the context in Problem-System aspects of architecting. Begin exploration of the program architecture as system concepts emerge.
- Select system and program styles consistently with the strategic identity of the organization.

The third bullet, financial criteria, has not been introduced before and is an important consideration. In the author's experience a common discussion when architecting any system is "how will this fit into the budget?" Affordability is based on fitting into a budget, a point to be made strongly in Chapter 13 on the political process. The efficient frontier analysis of Chapter 10 implicitly asks what is the best value we can get for a given cost (budget). But not all environments work this way. One of the authors participated in an architecture review for a major initiative at a large commercial company. One of the reviewers raised an objection to one of the considered alternatives: "There is no way you can do this, it would require a huge delta for a two year period in your budget." The response from the presenter exactly captured the difference between the ROI perspective and the budget perspective. The response was: "Budget doesn't matter to us. Our CFO can raise a billion dollars in the bond market tomorrow. What matters is ROI. If I can show the right ROI, I can get any amount of money. If I can't show ROI I can't have a dollar." Even within Government organizations, where budget allocation usually rules, sometimes it is all about ROI no matter the budget impact, as those around after the September 11 attacks can remember.

Consider how the factors above interrelated in the DC-3 example. For the background, see Case Study 2 prior to Chapter 3 and Van der Linden and Seely (2011). The whole picture, considering both the Boeing 247 and the DC-3, involved several organizations with different positions relative to each other, different strategies, and different architecting responsibilities. Superficially, Boeing was responsible for architecting (and building) airplanes to satisfy a commercial mission (make money by being sold to airlines to be operated commercially by those airlines). Their strategy, at the level of the 247, was a static strategy — perform existing missions with better performance and cost. At the level of the whole company, the strategy was much more dynamic. While one group pursued incremental improvement in the 247 with deep business ties to one airline, and one business model other groups were pursuing advanced military prototype aircraft. The efforts funded by the U.S.

Government included development of much larger, four-engine aircraft. The strategic identity of the 247 group was the pursuit of the static strategy. The strategic identity of the corporation as a whole included a dynamic strategy of shaping the aircraft market (albeit the military market).

The architects of the 247 dutifully pursued the static strategy of performance and cost improvements and were successful within that context. The program style was prototyping followed by serial production. There was no incremental development for the 247. When the DC-3 overtook the 247, Boeing's response was to make another architectural jump (the corporate-level dynamic strategy), but that was cut off by the beginning of World War II.

Douglas and Boeing were competitors, and at the corporate level they were pursuing similar dynamic strategies. At the local level of the aircraft program, Douglas pursued a more dynamic strategy for meeting commercial market needs in the mid 1930s, in contrast to Boeing's static local strategy. Considering the DC-1, DC-2, and DC-3 as a series, we see an incremental development strategy. The architectural jump on the problem side was to move away from the known airmail market. As a result of the uncertainties, this was pursued with incremental development, with each subsequent aircraft a bigger step into the unknown in size and performance. The program style was incremental development because each of the models was a fully usable, customer deliverable system. Indeed, all three models were customer delivered, although the total production of the DC-3 swamps the other two.

Luck and randomness also played a role. The crashes of the Boeing bomber prototype and the early Boeing 307 was outside any plan. Had it not happened the response to the DC-3 in the form of the Boeing 307 might have been significantly earlier, and having been significantly earlier might have happened far enough in advance of the start of World War II to be a mature program when the acquisition decisions for U.S. Army Air Force transports were made.

The historical accounts are not clear on the role of budget-driven versus ROI-driven financial thinking. The commercial developments were self-funded in both cases. At least some level of ROI thinking was present, but the self-funding undoubtedly placed some budget constraints. Boeing was significantly constrained financially in the years immediately after the introduction of the DC-3 because of changes to the law that forced the break-up of their parent company and cancellation of existing airmail contracts. The parallel military program Boeing had was based on what the U.S. Army Air Force was willing to pay for advanced development.

Jump and Exploit

The DC-3 and its derivative models are an illustration of a larger heuristic applicable at the program and organization level. This is the pattern of "jump and exploit." Jump and exploit describes the strategic approach of seeking unprecedented systems (the jump) followed by extensive lateral exploitation of the unprecedented jump. This combines the notions of static and dynamic strategy. We make jumps in creating new systems and coupled new concepts of operation or markets. When the jump is successful, we vigorously pursue the static strategy of improving performance and cost for the newly revealed operational concept or market.

The interplay of architectural jumps and long-term steady improvement is a strategic challenge. Leaders must be able to evaluate when the time for a jump is ripe, invest when the time is ripe, and stop focusing on incremental improvement. Conversely, failing to run a strategy of focused incremental improvement can easily cede competitive advantage to another player who does focus on continuous improvement. While recognizing when each situation pertains is inherently hard, one heuristic has been found useful.

> An architectural leap can rarely be justified when the consequence of a successful leap is a drop in revenue. Markets must expand to make cost reductions justifiable.

This is a hard heuristic to swallow in many cases, but it is important to examine. The simplest case is where an essentially fixed number of systems will be produced. Consider the case of space launch vehicles. Imagine that the government buys an average of five of a particular type per year. If each launch vehicle costs $100 million, the government expends an average of $500 million a year with this particular supplier. Now suppose there is a proposal to develop a new launch vehicle with the same performance but an estimated per launch cost of $50 million. Is this likely to be a workable proposal? The heuristics suggest it would not be, and the heuristic is developed from past experience with space systems and other limited market systems where demand is inelastic with price. It is very hard to show a good ROI when the result of the investment is a drop in revenue.

Why might this be so, and when might it not be true? When the supplier base is relatively fixed, we can imagine that existing suppliers would be less than enthusiastic about a program that promises to cut their revenue in half. Even if the government was to pay for the development, the overall situation is unlikely to be favorable. Financial capital is not the only capital of importance. Human capital is attracted to growing markets and is an essential fuel. In most cases, the program to cut costs in half is likely to be successful only if price elasticity is such that volume will likewise increase by at least a factor of two. Our hypothetical launch vehicle cost reduction program might be successful if a price cut by a half would more than double the launch rate. We see this effect in play in the electronics industry. While the price per transistor drops steadily by repeated factors of two, total production and sales of transistors go up even more quickly. The total revenues of the electronics industry have risen rapidly. If that were not so, they would be unlikely to have been able to attract the capital (both financial and human) that was necessary to fuel the growth.

ENTERPRISE ARCHITECTURE

A natural conclusion to this discussion of architecting in organizations is enterprise architecture. Enterprise architecture is a big business. It exists as an established practice with numerous books, consultants, service providers, methods, and courses. The purpose of this book is not to replace any of the large body of work, or even to engage in a detailed critique. Nevertheless, the concepts of this book, and especially this chapter, can be used to usefully inform the practice of enterprise architecture and understand some of the most commonly encountered difficulties.

Given that the field is large and the companies are so diverse, it is perhaps not surprising that there is a good deal of disagreement on what enterprise architecture is. If we take an enterprise to be an organization with a defined mission (a company and a government department would both normally qualify), then a "natural" definition of the architecture of an enterprise would be the fundamental and unifying structure of the enterprise. Then the practice of enterprise architecture would concern itself largely with business strategy and business processes and how the enterprise might be best organized to carry out its mission. In the case of a company, this would mean long-term value creation in particular markets. In the case of a government department, it would depend on the case (human services versus environment versus research versus security). But in reality, enterprise architecture as actually practiced almost always is largely concerned with information technology, either substantially or solely.

A good definition of enterprise architecture comes from Peter Weill of the Massachusetts Institute of Technology (MIT):

> The organizing logic for key business processes and IT capabilities reflecting the integration and standardization requirements of the firm's operating model.
>
> *Weill (2007)*

This book is concerned primarily with the architecting of systems. The information technology of a firm is certainly a system. The structure of that system should support the overall mission of the firm. As stated at the beginning of this chapter, the architecture of the firm's information technology (IT) should embody the strategy for the firm. From a simple insight, we can glean some important lessons.

> The strategy an IT system embodies should be that of the organization it belongs to as a whole, not that of just the IT controlling organization.

Several times one of the present authors has encountered the situation of how a research and development (R&D) group's IT is arranged within a larger organization of which R&D is a small part. Consider a hypothetical large specialty chemical manufacturer. The firm will undoubtedly have a chief information officer (CIO) and a corporate-wide information system. That corporate-wide information system needs to support internal functions (such as time and attendance, human resources, corporate-wide e-mail, and so forth) as well as core business activities. The core business activities would include sales and marketing, customer interaction, production and transportation planning, finance and reporting, and many others. Because these core functions represent virtually all of the firm's revenue, the CIO is likely to be very concerned with how they are supported. The CIO's priorities are likely to be dominated by system and application stability, availability, security, regulatory compliance, and cost control. When the CIO's office conducts an enterprise architecture exercise, it is likely to focus on central control and standardization. The ideal will change slowly, be carefully controlled, and provide a well-chosen set of common services with high availability.

Within this large manufacturer, there is an R&D group. The R&D group has dominant responsibility for new products and production methods. At any given time, they are not a revenue source; they are likely a sink for money. However, the long-term future of the company (5 and 10 years out) depends almost totally on the success of the R&D group. In an environment where products age out in 5–10 years, failure to have a full pipeline of new products will spell the end of the company and its value. How do the information technology needs of the R&D group align with the priorities of the CIO? In many cases, they align very poorly.

On the one hand, the employees of the R&D group have many of the same needs for centralized information services as everybody else. They need access to corporate-wide e-mail, human resources applications, and other centralized tools just like other groups. But, today, the R&D group in a specialty chemical company is likely to be trying to rapidly exploit computational chemistry, cheminformatics, a myriad of tools for chemical engineering, biology-based products and production methods, and collaborations with groups around the world. This environment changes quickly with tools being updated monthly, tools coming from all over the world, and all being run on a diversity of platforms. There is often a serious collision of strategies between the R&D group and the CIO.

Good enterprise architecture recognizes the diversity of business strategies within a firm and tries to appropriately accommodate them all. It realizes that the strategy the whole firm's IT should embody is the strategy of the whole firm, not that of a narrow segment. It would be as inappropriate to let the needs of R&D drive the IT for the whole firm as it would be to entirely subordinate R&D IT to the needs of day-to-day core operations. The IT in a firm should be there to execute the purpose of the firm, over both the long and the short term. Good architecting goes back, again and again if necessary, to root purpose. The purpose of a firm's IT is not to cost less than it did last year (even if that makes somebody's metrics look good); it is to support the business strategy of the whole firm. This is the holistic view of architecting, a system that we embraced from Chapter 1.

> An architecture description is not an architecture, and neither is an architecture framework.

This is simply a reiteration of a point made early in this book to not confuse architecture with architecture description. It is embraced by standards (ISO 2022). It is, unfortunately, not uncommon for a group to point at a large binder and say "This is our enterprise architecture." It is not and cannot be. At best, the binder will contain a description of the decisions that define the architecture of the enterprise. At best, those decisions will be good ones and will have captured the firm's business strategy effectively. Unfortunately, the best may not be true. Quite possibly the key decisions contained in that binder are buried from view and unwise to boot. If the decisions are clear, the architect should be able to highlight and explain them without many pages of description in the binder. If the decisions are wise, the reasoning should be clear and explainable and not buried in an opaque trade study where an answer is touted as the best simply because it scored the best on an evaluation function, but nobody can clearly articulate why and with what sensitivity.

The frameworks commonly cited in the enterprise architecture literature are all architecture description frameworks. That is, they define how to write a document about a system or systems. They do not define the decisions, and in most cases they provide little guidance on how to go about the decision-making. By itself this is not a great problem. Standardization of description methods can be quite useful in promoting wide communication. Where it becomes a problem is when framework adherence and unthinking artifact production take the place of critical thinking about an organization's missions, the diversity of missions that make up the overall mission, and the sources of value from what is being architected. Rote application of frameworks typically yields large documents that are then either applied inflexibly or ignored (which may be better than inflexible application).

In thinking about the example of the large chemical company above, no amount of framework application will resolve the essential tensions. The essential tensions are how to balance the need for diversity, change, and local control within the R&D group with the need for stability and standardization in the firm as a whole. Technology or business process adaptation may provide ways to relieve that tension. The tension is unlikely to ever be fully resolved, and there will inevitably be problems between the R&D group and other groups over how IT is selected and used. But the absence of perfection is no excuse to avoid deep thinking about how best to resolve the tension. There is almost certainly a great deal of long-term business value to allowing R&D to fully utilize the rapidly growing technology in accelerating product and production development and a great threat in the possibility that competitors will resolve it better, sooner.

> Program structure may be as important, or more important, than product structure.

In a large firm, the way they select, procure, deploy, and operate their IT is likely to be more important than the precise components chosen In the terms of the chapter, the structure of the program is likely to be as important as or more important than the Problem-System pair. The structure of the program must be inside the scope of architectural consideration, not outside it.

> Good architecting thinks as much or more about the problem as about the solution. Architecting teams need the skills relevant to the problem scope they are engaged in.

It often seems natural that an IT-centric enterprise architecture job should be done by IT specialists. But, if the scope of consideration includes how we might change business processes in concert with IT deployments in order to better carry out the firm's mission, an all-IT-specialist team will be wholly inadequate. This is reflective of the basic nature of systems architecting. As discussed here, the lowest scope that is "architectural" is Problem-System. That is, the problem is inside the scope of investigation, not outside it. As discussed in the beginning chapters of the book, architecting addresses ill-structured problems where the statement of the problem is in play. If all the requirements can be readily determined, it is not architecting. If the nature of the problem is "Find the best solution to this precisely stated and well-structured problem," it might be a very worthy and difficult thing to do, but it is not architecting. If it is not architecting, it does not need the machinery of architecting, and we can rely instead on the established machinery of engineering science.

In the enterprise case, the situation is often more difficult because the problem includes basic strategic issues for the firm. A team constructed to do what is viewed as a technical architecting job rarely contains the expertise and authority to challenge enterprise-level strategic decisions. They may be unable to command sufficient attention from senior executives whose purview definitely includes strategy. Lacking that attention, they may default to extracting a well-structured problem they can solve, whether or not that is really relevant to the organization's greatest need.

Many darkly humorous tales can be told of the technologist supposedly empowered to make a revolution in a firm running after executives trying to have a strategic discussion (Kay 2000), or giving up, pursuing a technological solution, and failing because of an inability of the firm to connect technological success to business strategy. The DC-3 and GPS are stories of success, albeit with the fits and starts and blind alleys of the real world. One of the most famous stories of failure to connect technological architecting to business strategy execution is Xerox PARC in the 1970s. The story is lengthy and well told in the published literature, but certain points bear repeating (Hiltzik 1999).

Xerox executives had a clear strategic vision of the need to make a change from the copier business by the late 1960s. They took tremendous advantage of the availability of a whole cadre of the best computer scientists and engineers who became available as a result of U.S. Defense Advanced Research Projects Agency (DARPA) funding cuts. They set their recruits up in a new organization that produced an unprecedented series of technical innovations (laser printers, object-oriented programming languages, and window-mouse-graphics displays famously among them), with many of those innovations taken to the point of prototype products. But they were then unable to convert those innovations and products into revenue, at least revenue commensurate with how much value the innovations eventually realized for others. The failure was largely one of mismatched strategies and a lack of coupled change in business models (that is, operational concepts) to go with the innovative products. To turn the new products into value would have required new business concepts to go with them. As it turned out, other firms were much faster to find those altered operational concepts and implement them.

Ironically, it was the ability and willingness of Xerox to discover and use coupled technological and business model change in the 1950s, replacing a purchase model for copiers with a lease and per-page-charge model that made the company such an enormous success originally. But what was possible in the start-up days became impossible in the days of maturity. The lesson is only reinforced from the other case studies. Dealing with change at the right scope is critical. The biggest successes come from coupled change in technology, system, and operational concept.

Of course, this story about Xerox is only loosely about a firm's IT architecture. But that is exactly the point. Good architecting knows its scope. Focusing on just a firm's IT architecture is a narrow focus, one unlikely to lead to changes in strategy or overall approach, but likely to lead to efficiency and improvements within a mission area. It leads to small solutions within preexisting mission areas instead of large solutions in new mission areas. Focusing on efficiencies can create strong competitive advantage when missions or markets are stable. When missions or markets are unstable and revolution is in the air, focusing on efficiencies is a distraction, and possibly a fatal distraction. Of course, the opposite is also possible when missions and markets are stable, and one can be distracted by a futile search for revolutions where none can exist and fail to develop the efficiencies that others will use to win the competition. An organization must be wise enough to know the difference.

An Enterprise Thought Experiment

Let us return briefly to Chapter 6 on software and the case discussion on layered systems. Many exercises in enterprise architecture turn, one way or another, on how to provide common services across a large enterprise. A popular buzz-word is "service-oriented architecture" (SOA). As a thought experiment, what are issues in the program or strategic sense for providing a common infrastructure of software services? Two programmatic alternatives (of course, there are others) are to license a commercial enterprise service bus (ESB) and have new software written on top of it, and to license an open-source ESB equivalent and write new software on top of it using a continuing open-source rule (all components written become enterprise shared property).

Some of the issues are as follows:

- What is the impact on which developers can be used? Will some developers refuse to contract to write code that is shared with other enterprise developers? With a commercial ESB, will the cost of developer licenses inhibit how many developers can be used and when (for example, cannot afford experimental programming)?
- In either case, will making a transition away from current practice devalue current developers? Can the organization afford the costs involved in building a new developer community? As an aside, that cost might run from very large to negative depending on the nature of the market and the skills of the current developers.
- Does business with other enterprises on the part of the ESB vendor bring economies of scale? Versus, is there a significant open-source development community beyond the local enterprise for an open-source alternative?
- How are third-party developers affected? If you want a particular tool of high importance brought in to the enterprise service environment, how will that tool vendor integrate with the ESB (commercial from another party versus open source)?
- Does the choice of implementation strategy affect the achievement of the larger business strategy?

DIGITAL ENGINEERING AND ARCHITECTURE ENABLEMENT

Embracing MBSE methods, as discussed in Chapter 10, is itself part of an organizational strategy. The MBSE vision is that design information from system initiation through manufacturing is captured in a unified set of digital models. This would start with architecture, but continue far deeper in design to include the full system-level

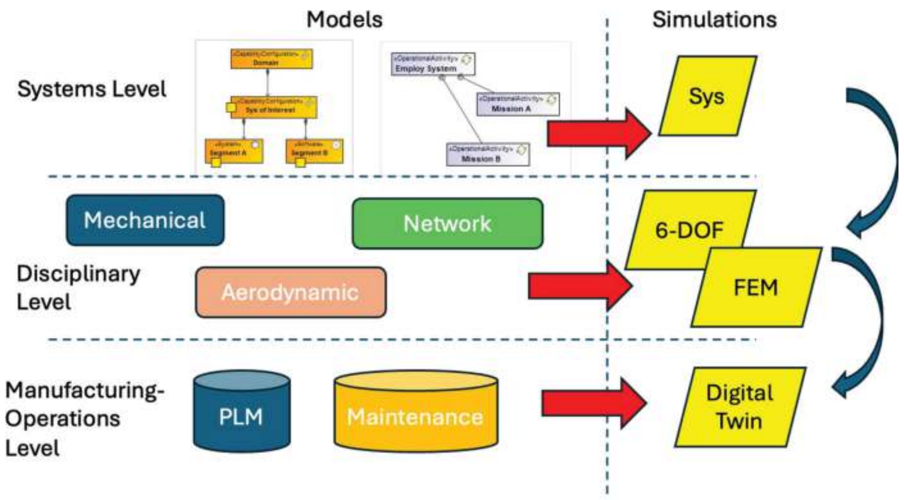


FIGURE 12.3 Conceptual tree of digital models from architecture to manufacturing to as-built digital twin.

design, disciplinary designs, manufacturing data packages, and digital models representing the as-built system (a "digital twin"). Schematically this is represented by Figure 12.3.

Taking this comprehensive approach to digital engineering implies several important and challenging transition points between models at different levels of abstraction and disciplinary specificity.

- Architecture models cover function, physical components, and performance (among other views). If multiple detailed design approaches are to be supported (as in a competitive acquisition), then the architecture models must be constructed to allow linkage to multiple more detailed design models.
- SysML methods are appropriately suited to capturing a detailed physical decomposition (and parallel functional and performance decompositions) to the level of configuration items. The organization will need standards and representation discipline to carry this out in a complete and consistent way.
- At the configuration item level, there is typically a transition to more traditional engineering disciplines (e.g., mechanical, electrical, software). In a digital environment, this means the organization will need an approach for moving from the system-level, trans-disciplinary tool set to discipline-specific tools.
- Sub-contracting packages (e.g., bills of materials, drawings, manufacturing specifications) are typically organized at the configuration item level. Again, this will be at the system-level tool and disciplinary tool interface. While this is not architectural, if the whole model arrangement started with architecture models in the system-level tool, it needs to be structured to support the eventual detailed breakdown.

CONCLUSION

Architecting must be situated in its business or government operational context. When viewed in its context, the relationship to strategy becomes evident. Where strategies are clear, good architecting can follow. Where strategies are unclear, good architecting will be very difficult. As we consider not just the classic architecting relationship of problem-system but expand the focus to the structure of development programs, there becomes a synergy of architectural and strategic thinking.

The most successful, unprecedented systems involve changes to business or operational models in parallel with new systems and technology. Just introducing a new system is not enough; when the system is revolutionary, the context has to change as well for the greatest success to be realized. Coupled business or operational change must be enabled by the organization's strategy.

The leading guidelines in this chapter were as follows:

- Remember problem–system–program–organization.
- Understand static and dynamic strategies, and how they map into program styles.
- Have a catalog of development program styles and understand where each is best suited.
- Know when to jump, and know when to settle for continuous improvement.
- Architect holistically for strategy, not just for local stakeholders.
- Do not confuse architectures, architecture descriptions, and architecture description frameworks.
- Think of architecture as the technical embodiment of strategy.

REFERENCES

Boyd, J. (2007). *Patterns of Conflict*, Georgia: Defense and the National Interest Atlanta.
Hiltzik, M. (1999). *Dealers of Lightning: Xerox PARC and the Dawning of the Computer Age*, New York: HarperCollins.
INCOSE (2023). *INCOSE Systems Engineering Handbook*, Hoboken, NJ: John Wiley & Sons.
ISO (2022). *ISO/IEC/IEEE 42010 Software Systems and Enterprise - Architecture Description: 69*, New York: International Standards Organization.
Kay, J. (2000). *My Year at a Big High Tech Company*, Weston, FL: Forbes ASAP.
Rechtin, E. (2017). *Systems Architecting of Organizations: Why Eagles Can't Swim*, London: Routledge.
Richards, C. (2004). *Certain to Win: The Strategy of John Boyd, Applied to Business*, Bloomington, IN: Xlibris Corporation.
Van der Linden, F. R. and V. J. Seely (2011). *The Boeing 247: The First Modern Airliner*, Washington, DC: University of Washington Press.
Weill, P. (2007). Innovating with Information Systems: What do the most agile firms in the world do. *6th e-Business Conference*, Barcelona.

13 The Political Process and Systems Architecting

Brenda Forman

INTRODUCTION: THE POLITICAL CHALLENGE

The decision process of systems architecting requires two things above all others—value judgments by the client and technical choices by the architect. Decisions are not implemented unless there are resources to implement them. The political process is the way that the general public, when it is the end client, expresses its value judgments and allocates its resources. High-tech, high-budget, high-visibility, publicly supported programs are therefore far more than engineering challenges; they are political challenges of the first magnitude. Governments build what the political process deems valuable and are willing to allocate the resources to. A program may have the technological potential of producing the most revolutionary weapon system since gunpowder or health intervention since the smallpox vaccine, elegantly engineered and technologically superb, but if it is to have any real life expectancy or even birth, its managers must take its political element as seriously as any other element. It is not only possible but likely that the political process will not only drive such design factors as safety, security, producibility, quantity, and reliability, but may even influence the choice of technologies to be employed. The bottom line is:

If the politics don't fly, the system never will.

This chapter is based on a course originated and taught at the University of Southern California by Dr. Forman. As indicated in the Introduction to Part IV, the political process of the American Federal Government was chosen for four reasons: The U.S. Federal Government plays an enormous role in what high complex systems get built and how they get built, it is a leading example of the process by which the general public expresses its value judgments as a customer, it is well documented and publicized, and it is seriously misunderstood by the engineering community, to the detriment of effective architecting. Outside of the government sphere, it may seem that politics disappears, but it does not. Large corporations are also political entities, and the heuristics of politics operate in organizations on many levels. Of course, as an organization becomes commercially focused, its objectives are different and the motivations of its leaders are likewise different, but many of the same rules will apply. Thus, this chapter can be seen as a guide to the political demands on architects, even outside of the purely political sphere. Some observers see "politics" as decision-making for "inappropriate" reasons, usually meaning reasons that lie outside of quality, effectiveness, cost, and similar factors. See Clausing and Katsikopoulos (2008) for a clear

DOI: 10.1201/9781003483595-24

presentation of this perspective, specifically on the corporate side. Clausing makes a good case that a commercial firm should make decisions based on quality, cost, and delivery, and decisions based on internal "politics" are inherently flawed. We take the view that public choice expresses the views of the public, factored through some system of representative government, and as such expresses whatever preferences it has, regardless of how rational or appropriate those views may be.

POLITICS AS A DESIGN FACTOR

Politics is a determining design factor in today's high-tech engineering. Its rules and variables must be understood as clearly as those of stress analysis, electronics, or support requirements. However, these rules differ profoundly from those of Aristotelian logic. Its many variables can be bewildering in their complexity and often downright orneriness.

In addition to the formal political institutions of Congress and the White House, a program must deal with a political process that includes interagency rivalries, intra-agency tensions, dozens of lobbying groups, influential external technical review groups, powerful individuals both within and outside government, and always and everywhere, the media.

These groups, organizations, institutions, and individuals interact in a process of great complexity. This confusing and at times chaotic activity, however, determines the budgetary funding levels that either enable the engineering design process to go forward or threaten outright cancelation. More often of late, it directly affects the design in the form of detailed budget allocations, assignments of work, environmental impact statements, and the reporting of risks or threats.

Understanding the political process and dealing successfully with it are therefore crucial to program success.

Example: Perhaps no major program has seen as many cuts, stretch outs, reviews, mandated designs, and risk of cancelation as the planetary exploration program of the 1970s and 1980s. Much of the cause was the need to fund the much larger Shuttle program. For more than a decade, there were no planetary launches and virtually no new starts. From year-to-year, changes were mandated in spacecraft design, the launch vehicles to be used, and even the planets and asteroids to be explored. The collateral damage to the planetary program of the Shuttle Challenger loss was enormous in delayed opportunities, redesigns, and wasted energy. Yet the program was so engineered that it still produced a long series of dramatic successes, widely publicized and applauded, using spacecraft designed and launched many years before. Notably, in the 1990s, JPL made an aggressive move toward shorter missions with more rapid turnover. This can be seen as emphasizing the longer-term importance of learning and adapting in scientific exploration over immediate cost efficiency. A consistent, year-to-year stream of results also helps build constituency. Ironically, in the mid-2000s, planetary exploration was once again under pressure, this time from the expense of terminating the Shuttle program.

Begin by understanding that power is very widely distributed in Washington. There is no single, clear-cut locus of authority to which to turn for support for long-term, expensive programs. Instead, support must be continuously and repeatedly generated from widely varying groups, each of which may perceive the program's expected benefits in quite different ways and many of whose interests may diverge rather sharply when the pressure is on.

Example: The nation's space program is confronted with extraordinary tensions, none of which are resolvable by any single authority, agency, branch, or individual. There are tensions between civilian and military, science and application, manned and unmanned flight, complete openness and the tightest security, the military services, NASA centers, and the commercial and government sectors, to name a few. Typical of the contested issues are launch vehicle development, acquisition and use; allocations of work to different sections of the country and the rest of the world; and the future direction of every program. No one, anywhere, has sufficient authority to resolve any of these tensions and issues, much less to resolve them all simultaneously.

This broad dispersion of power repeatedly confuses anyone expecting that somebody will really be in charge there. Rather the opposite is true: anything that happens in Washington is the result of dozens of political vectors, all pulling in different directions. Everything is the product of maneuver and compromise. When those fail, the result is policy paralysis—and, all too possibly, program cancelation by default or failure to act.

There are no clear-cut chains of command in the government. It is nothing like the military or even like a corporation. The process gets even more complicated because power does not remain fixed in Washington. Power relationships are constantly changing, sometimes quietly and gradually, at other times suddenly, under the impact of a major election or a domestic or international crisis. These shifts can alter the policy agenda—and therefore funding priorities—abruptly and with little advance warning. A prime example is the ever-changing contest over future defense spending levels in the wake of the welcomed end of the Cold War, then the war on global terror, and later the return of great power competition.

The entire process is far better understood in dynamic than static terms. There is a continuous ebb and flow of power and influence between Congress and the White House, among and within the rival agencies, and among ambitious individuals. And through it all, everyone is playing to the media, particularly television, in efforts to change public perceptions, value judgments, and support.

THE FIRST SKILL TO MASTER

To deal effectively with this process, the first skill to master is the ability to think in political terms. And that requires understanding that the political process functions in terms of an entirely different logic system than the one in which scientists, engineers, and military officers are trained. Washington functions in terms of the logic of politics. It is a system every bit as rigorous in its way as any other, but its premises and rules are profoundly different. It will therefore repeatedly arrive at conclusions quite different from those of engineering logic, based on the same data.

Scientists and engineers are trained to marshal their facts and proceed from them to proof: for them, proof is a matter of firm assumptions, accurate data, and logical deduction. Political thinking is structured entirely differently. It depends not on logical proof but on past experiences, negotiation, compromise, and perceptions. Proof is a matter of “having the votes.” If a majority of votes can be mustered in Congress to pass a program budget, then—by definition—the program has been judged to be worthy, useful, and beneficial to the nation. If the program cannot, then no matter what its technological merits, it will lose out to other programs that can.

Mustering the votes depends only in part on engineering or technological merit. From an architectural perspective, it really should not, those charged with making value judgments should make them, not engage in technical assessments. Members of Congress do make value judgments, but the basis for those judgments is diverse. To a major extent, it is what logically it should be, on utility for missions of national security, weather forecasting, scientific development, or other missions. These are always important—but getting the votes frequently depends as much or even more on a quite different value judgment, the program’s benefits in terms of jobs and revenues among the Congressional districts.

Example: After the Lockheed Corporation won NASA’s Advanced Solid Rocket Motor (ASRM) program, the program found strong support in Congress because Lockheed located its plant in the Mississippi district of the then Chairman of the House Appropriations Committee. Lockheed’s factory was only partially built when the chairman suffered a crippling stroke and was forced to retire from his Congressional duties. Shortly thereafter, Congress, no longer obliged to the chairman, reevaluated and then canceled the program.

In addition to the highest engineering skills, therefore, the successful architect-engineer must have at least a basic understanding of this political process. The alternative is to be repeatedly blindsided by political events—and worse yet, not even to comprehend why.

HEURISTICS IN THE POLITICAL PROCESS: “THE FACTS OF LIFE”

Following are some basic concepts for navigating these rocky rapids—“The Facts of Life.” They are often unpleasant for the dedicated engineer, but they are perilous to ignore. Understanding them, on the other hand, will go far to help anticipate problems and cope more effectively with them. They are as follows and will be discussed in turn.

- Politics, not technology, sets the limits of what technology is allowed to achieve.
- Cost rules.
- A strong, coherent constituency is essential.

- Technical problems become political problems.
- The best engineering solutions are not necessarily the best political solutions.

Fact of Life # 1: Politics, not technology, sets the limits of what technology is allowed to achieve

If funding is unavailable, any program will die, and getting the funding—not to mention keeping it over time—is a political undertaking. Furthermore, funding—or rather, the lack of it—sets limits that are considerably narrower than what our technological and engineering capabilities could accomplish in a world without budgetary constraints. Our technological reach increasingly exceeds our budgetary grasp. This can be intensely frustrating to the creative engineer working on a good and promising program. Anybody involved in developing programs in response to U.S. Federal Government missions knows there are far more ideas with good expectations of delivering value than will ever be funded. Any senior Federal executive in a technology area will know how double his or her budget could be spent without dipping into ideas that transparently are low value for the mission area. That does not mean poor ideas never get funded, only that obtaining resource is the constraint far more than technological availability.

Example: The space station program can trace its origins back to the mid-1950s. By the early 1960s, it was a preferred way station for traveling to and from the moon. But when, due to the launch vehicle size and schedule, the Apollo program chose a flight profile that bypassed any space station and elected instead a direct flight to lunar orbit, the space station concept went into limbo until the Apollo had successfully accomplished its mission. The question then was, what next in manned spaceflight? A favored concept was a manned space station as a waypoint to the moon and planets, built and supported by a shuttle vehicle to and from orbit. Technologically, the concept was feasible; some argued that it was easier than the lunar mission. Congress balked. The President was otherwise occupied. Finally, in 1972, the Shuttle was born as an overpromised, underbudgeted fleet, without a space station to serve. Architecturally speaking, major commitments and decisions were made before feasibility and desirability had been brought together in a consistent whole.

A variation on this fact of life is where regulation has fundamental impact on what can and cannot be built. Here, politics is not enabling or disabling a system by providing or withholding support; the impact is through policy and regulations constraining (or not) those outside government who would build the system of interest. There are many cases of this in energy, transportation, and medical systems. An emerging illustration is autonomous systems. The 2020s are seeing very rapid evolution in technology for autonomy. It is increasingly possible to build systems that outperform humans much of the time, while also underperforming in important circumstances, circumstances that are often not easily predicted or well-understood. The policy question is who is responsible for the actions taken by an autonomous system? Is it the owner, the builder, or someone else? Is the autonomous system held to the same standard of responsibility (and liability) as a human operator, or a different standard? These are political and legal questions, and all legal questions are ultimately political questions in that the law is a product of politics. Political action, or inaction, will determine how responsibility is allocated and this in turn will determine what kinds of autonomy can be effectively deployed. Politics is more determinant than technology of what can be built and used in this sector, even though the mechanism is not direct sponsorship of systems.

Fact of Life #2: Cost rules

High technology gets more expensive by the year. As a result, the only pockets deep enough to afford it are historically those of the government. The fundamental equation to remember is Money = Politics. Reviews and hearings will spend much time on presumably technical issues, but the fundamental and absolutely determining consideration is always affordability—and affordability is decided by whichever side has the most votes.

We do have to say "often" rather than "always" for Government funding of major technical initiatives, because private research and development spending in some areas at some times may exceed the U.S. Federal Government. The largest launch vehicle, the SpaceX Starship and its associated booster, has been developed with private funding, as are significant competitors. Artificial intelligence breakthroughs, dependent on very large computations and data gathering, have been pioneered by private companies. Much biotechnology investment is private.

Funding won in one year, moreover, does not stay won, at least not in government. Private money can be more stable, but has its own fickle nature in being controlled by one person or a small group who may have other competing interests and demands. Government funding must be fought for afresh every year. With exceedingly few exceptions, no program in the entire federal budget is funded for more than one year at a time. Every year is therefore a new struggle to head off attackers who want the program's money spent somewhere else, to rally constituents, to persuade the waverers, and, if possible, to add new supporters.

This is an intense, continuous, and demanding process requiring huge amounts of time and energy. And after one year's budget is finally passed, the process starts all over again. There is always next year. Keeping a program "sold," in short, is a continuous political exercise, and like the heroine in the old movie serial, "The Perils of Pauline," some programs at the ragged edge will have to be rescued from sudden death on a regular basis. Rescue, if and when, may be only partial—not every feature can or will be sustained. If one of the lost features is a system function, the end may be near.

Example: After the Shuttle had become operational, the question again was, what next in manned spaceflight? Although a modestly capable space station had been successfully launched by a Saturn launch vehicle, the space station program had otherwise been shelved once the Shuttle began its resource-consuming development. With developmental skill again available, the space station concept was again brought forward. However, order-of-magnitude life-cycle cost estimates of the proposed program placed the cost at approximately that of the Apollo, which in 1990-decade dollars would have been about $100 billion—clearly too much for the size of constituency it could command. The result has been an almost interminable series of designs and redesigns, all unaffordable as judged by Congressional votes. Even more serious, the cost requirement has resulted in a spiraling loss of system functions, users, and supporters. Microgravity experiments, drug testing, on-board repair of on-orbit satellites, zero-g manufacturing, optical telescopes, animal experiments, military research and development—one after another had to be reduced to the point of lack of interest by potential users. A clearly implied initial purpose of the space station, to build one because the Soviet Union had one, was finally put to rest with the U.S. government's decision to bring Russia into a joint program with the United States, Japan, Canada, and the European Space Agency. One apparent certainty: the U.S. Congress made the value judgment that a yearly cap of $2.1 billion is all that a space station program is worth. The design must comply or risk cancelation. Cost rules.

Example: Once the decision to terminate the Shuttle was made, there was consensus a new program was required to maintain U.S. human access to space. Ideally, the old program (Space Shuttle) should not be terminated until the new program has proven its capability. However, that could only be accomplished by an expensive overlap of the program of several years that requires a large (several billion U.S. dollars) increase in the NASA top-line budget. NASA instead structured the programs with a multiyear gap in space access, to avoid the funding hiccup (using savings from Shuttle termination to ramp up production funding for the replacement). Although there was widespread unhappiness, in Congress and NASA, about the resulting gap, there was no willingness to raise budgets enough to close it. Again, cost rules. Cost continued to rule in the years to come as the replacement program was re-structured, canceled, and re-started in different form. In the meantime, money was found to fly U.S. astronauts on Russian flights. Somewhat unexpectedly, commercial providers appeared and are now providing U.S. flights to the space station and the NASA program (Space Launch System) no longer prioritizes space station access. Some argue that the NASA manned launch system is superfluous, and NASA should reorient to use commercial for anything commercial is willing to provide and only do itself what is not provided. But this runs into its own constituency issues, and so to the next fact of life.

Fact of Life #3: A strong, coherent constituency is essential

No program ever gets funded solely—or even primarily—on the basis of its technological merit or its engineering elegance. By and large, the Congress is not concerned with its technological or engineering content (unless, of course, those run into problems—see Fact of Life #4). Instead, program funding depends directly on the strength and staying power of its supporters—that is, its constituency.

Constituents support programs for any number of reasons, from the concrete to the idealistic. At times, the reasons given by different supporters will even seem contradictory. From the direct experience of one of the authors, some advocates may support defense research programs because they are building capability; others support them because research in promising better systems in the future permits reduction, if not cancelation of present production programs.

Example: The astonishing success of the V-22 tilt-rotor Osprey aircraft program in surviving 4 years of hostility during the 1988–1992 period, and several fatal accidents, is directly attributable to the strength of its constituency, one that embraced not merely its original Marine Corps constituency but other Armed Services as well—plus groups that see it as benefiting the environment (by diminishing airport congestion), as improving the balance of trade (by tapping a large export market), and as maintaining U.S. technological leadership in the aerospace arena.

Assembling the right constituency can be a delicate challenge, because a constituency broad enough to win the necessary votes in Congress can also easily fall prey to internal divisions and conflicts. Such was the case for the Shuttle and is the case for the Space Station. The scientific community proved to be a poor constituency for major programs; the more fields that were brought in, the less the ability to agree on mission priorities. On the other hand, a tight homogeneous constituency is probably too small to win the necessary votes. The superconducting supercollider proved to be such. The art of politics is to knit these diverse motivations together firmly enough to survive successive budget battles and keep the selected program funded. Generally speaking, satellites for national security purposes have succeeded in this political art. It can require the patience of a saint coupled with the wiliness of a Metternich, but such are the survival skills of politics.

Fact of Life #4: Technical problems become political problems

In a high-budget, high-technology, high-visibility program, there is no such thing as a purely technical problem. Program opponents will be constantly on the lookout for ammunition with which to attack the program, and technical problems are tailor-made to that end. The problems will normally be reported in a timely fashion. As many programs have learned, mistakes are understandable; failing to report them is inexcusable. In any case, reviews are mandated by Congress as a natural part of the program's funding legislation. Any program that is stretching the technological envelope will inevitably encounter technical difficulties at one stage or another. The political result is that "technical" reports inevitably become political documents as opponents berate and advocates defend the program for its real or perceived shortcomings.

Judicious damage prevention and control, therefore, are constantly required. Reports from prestigious scientific groups such as the Nuclear Regulatory Commission (NRC) or Defense Science Board (DSB) will routinely precipitate Congressional hearings in which hostile and friendly Congressmen will pit their respective expert witnesses against one another and the program's fate may then depend not only on the expertise but on the political agility and articulateness of the supporting witnesses. Furthermore, although such hearings will spend much time on ostensibly technical issues, the fundamental and absolutely determining consideration is always affordability—and affordability is decided by whichever side has the most votes.

Examples: Decades-long developments are particularly prone to have their technical problems become political. Large investments must be made every year before any useful systems appear. The widely reported technical difficulties of the B-1 and B-2 bombers, the C-17 cargo carrier, the Hubble telescope, and the Galileo Jupiter spacecraft became matters of public and legislative concern. The futures of long-distance air cargo transport, of space exploration, and even of NASA are all brought up for debate and reconsideration every year. Architects, engineers, and program managers have good reason to be concerned.

Fact of Life #5: The best engineering solutions are not necessarily the best political solutions

Remember that we are dealing with two radically different logic systems here. The requirements of political logic repeatedly run counter to those of engineering logic. Take construction schedules: in engineering terms, an optimum construction schedule is one that makes the best and most economical use of resources and time and yields the lowest unit cost. In political terms, the optimum construction schedule is the one that the political process decides is affordable in the current fiscal year. These two definitions routinely collide; the political definition always wins.

Example: NASA and other agencies often refer to what is called the program cost curve. It plots the total cost of development and manufactures as a function of its duration (Figure 13.1). There is an optimal duration for a program of a given complexity. If the duration is too short, costs rise due to inefficient parallelization and the need to get maximum labor on the effort no matter the suitability. Too long, resources stay around and cost money, even if they are not needed.

The foregoing example leads to another provisional heuristic:

With few exceptions, schedule delays and life-cycle cost increases are accepted grudgingly; annual cost overruns are not, and for good reason.

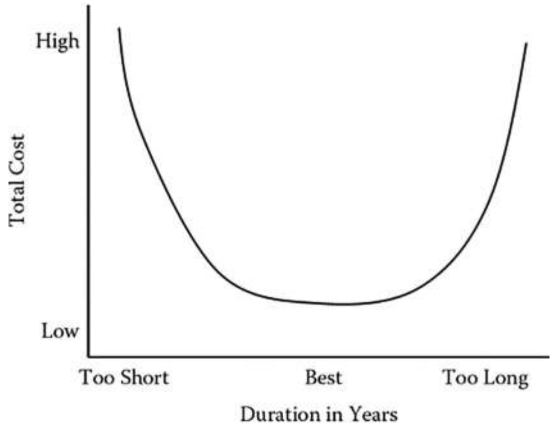


FIGURE 13.1 Qualitative relationship between total program cost and program duration.

The reason is basic. A cost overrun—that is, an increase over budget in a given year—will force the client to take the excess from some other program, and that is not only difficult to do, it is hard to explain to the blameless loser and to that program's supporters. Schedule delays mean postponing benefits at some future cost—neither of which affects anyone today. At the worst end of this heuristic, it leads to managers "kicking problems down the road" knowing they will have moved on before the problem comes due.

Example: Shuttle cost overruns cost the unmanned space program and its scientific constituency two decades of unpostponable opportunities, timely mission analyses, and individual careers based on presidentially supported, wide-consensus planning.

The curve in 13.1 is logical and almost always true. But it is irrelevant because the government functions on a cash-flow basis. Long-term savings will almost always be foregone in favor of minimizing immediate outlays. Overall life-cycle economies of scale will repeatedly be sacrificed in favor of slower appropriations, even if they cause higher unit costs and greater overall program expenses. There is also the contradictory perception that if a given program is held to a tight, short schedule will cost less, facts notwithstanding. (See Chapter 5, "Social Systems," Facts versus Perceptions: An Added Tension.)

By the same token, a well-run program that sticks to a budget can encounter very difficult technical problems and survive.

As an aside, this heuristic leads us directly back to observations on program and organization architecture and the role of feedback loops at the enterprise level. Consider what happens when an organization caps the duration of programs within its sphere of responsibility to, for example, 5 years. If they do so honestly they will, of course, push some long-term, ambitious endeavors out of feasibility. But they will create a situation where program managers and architects can be expected to be assigned to the same program for its full lifetime. Those managers and architects can expect to be personally accountable for the end-point consequences of their start-point actions. After a few program cycles, when selecting program managers and architects for the next program, executives can use the actual results of past programs in evaluations, not the results reported before the responsible parties have moved onto other projects. In the language of economists, the "moral hazard" issue of people undertaking risks that others will have to suffer from will be largely eliminated. The feedback loop at the enterprise level now synchronizes programmatic and human responses.

The political process can be bewildering and intimidating to the uninitiated. But it need not be so. In addition to being confusing and chaotic, this is a profoundly interesting and engrossing process, every bit as challenging as the knottiest engineering problem. Indeed, it is an engineering challenge because it molds the context in which systems architecting and engineering must function. This is one area where commercial and government politics are often dramatically different. Commercial enterprises tend to be return-on-investment driven rather than cash-flow driven. When return on investment is driven, and with the ability to move money between time periods via financial markets, there is the possibility of making different choices. However, firms that are driven by quarterly results will tend to resemble the cash-flow-driven government budgeting process.

The reader may well find the craziness of the political process distasteful—but it will not go away. The politically naive architect may experience more than a fair share of disillusion, bitterness, and failure. The politically astute program manager, on the other hand, will understand the political process, will have a strategy to accommodate it, and will counsel the architect accordingly. Some suggestions for the architect: It helps to document accurately when and why less-than-ideal technical decisions were made—and how to mitigate them later, if necessary. It helps to budget for contingencies and reasonably foreseeable risks. It helps to have stable and operationally useful interim configurations and fallback positions. It helps to acknowledge the client's right to have a change of mind or to make difficult choices without complaint from the supplier. Above all, it helps to acknowledge that living in the client's world can be painful, too. Finally, select a kit of prescriptions for the pain such as the following from Appendix A.

- The Triage, when there is only so much that can be done: Let the dying die. Ignore those who will recover on their own. And treat only those who would die without help.
- The most important single element of success is to listen closely to what the customer [in this case, the Congress] perceives as his requirements and to have the will and ability to be responsive. (J. E. Steiner, The Boeing Company, 1978)
- Look out for hidden agendas.

That does not mean that architects and engineers have to become expert lobbyists—but it does mean having an understanding of the political context within which programs must function, the budget battle's rules of engagement, and those factors that are conducive to success or failure. The political process is not outside; it is an essential element of the process of creating and building systems.

A FEW MORE SKILLS TO MASTER

Following are a few more basic coping skills for the successful systems architect. Foremost, understand that Congress and the political process are the owners of your project. They are the ultimate clients. It is absolutely essential to deal with them accordingly by making sure they understand what you are trying to do, why it is important, and why it makes political sense for them to support you.

Be informed. This is your life, so be active. Learn the political process for yourself and keep track of what is going on. Figure out what information the political system needs in order to understand what the program needs—and arrange to supply it to them. A chief engineer has utterly different information requirements from a Congressional oversight committee. Learn what sort of information furthers your program's fortunes in Washington and then get it to your program managers, so they can get it to the political decision makers who determine your program's funding. Perhaps your program has a great job-multiplier effect in some crucial lawmaker's district. Or maybe its technology has some great potential commercial applications in areas where the United States is losing a competitive battle with another country.

The point is that the political process bases its decisions on very different information than does the engineering process. Learn to satisfy both those sets of requirements by plan.

CONCLUSION

The political process is a necessary element of the process of creating and building systems. It is not incomprehensible; it is different. Only when the political Facts of Life are not understood do they instill cynicism or a sense of powerlessness. Once understood, they become tools like any others in the hands of an astute architect. It is a compliment to the client to use them well.

REFERENCE

Clausing, D. P. and K. V. Katsikopoulos (2008). "Rationality in systems engineering: Beyond calculation or political action." *Systems Engineering* **11**(4): 309–328.

14 The Professionalization of Systems Architecting

Elliott Axelband

Profession: Any occupation or vocation requiring training in the liberal arts or the sciences and advanced study in a specialized field.[1]

INTRODUCTION

To readers who have progressed this far, the existence of systems architecting as a process, regardless of who performs it, can be taken for granted. Functions and forms have to be matched, system integrity has to be maintained throughout development, and systems have to be certified for use. Visions have to be created, realized, and demonstrated. Somebody will be making the cost, value, and risk-driving decisions at the beginning of system developments.

This chapter, in contrast, covers the evolution of the systems architecting profession. An appropriate place to begin is with the history of the closely related profession of systems engineering, the field from which systems architecting evolved.

THE PROFESSION OF SYSTEMS ENGINEERING

Systems engineering as a process began in the early 1900s in the communication and aircraft industries. It evolved rapidly during and after World War II, as an important contributor to the development of large, innovative, and increasingly complex systems. Systems engineering is famously associated with the development of Intercontinental Ballistic Missiles (particularly Atlas and Minuteman) and the work of Simon Ramo and General Bernard Schriever (Booton and Ramo 1984, Johnson 2002). But by the early 1950s, systems engineering had reached the status of a recognized, valued profession. Communication networks, ballistic missiles, radars, computers, and satellites were all recognized as systems. The "systems approach" entered into everyday language in many fields, social as well as technical. Government regulations and standards explicitly addressed systems issues and techniques. Thousands of engineers called systems engineering as their vocation. Professional societies formed sections with journals devoted to systems and their development.[2] Universities established systems engineering departments or systems-oriented programs. Books addressing the process, or aspects of it, started to appear (Machol 1965, Chestnut 1967, Fabrycky and Blanchard 1981). Most recently, the profession became formally represented with the establishment of the International Council on

DOI: 10.1201/9781003483595-25

Systems Engineering (INCOSE). The first INCOSE Symposium was 1990 and the inaugural issue of the journal *Systems Engineering* in 1994.

The core of the systems approach, from its beginnings, has been the definition and management of system interfaces and trade-offs, of which there can be hundreds in any one system. Systems analysis, systems integration, systems test, and computer-aided system design were progressively developed as powerful and successful problem-solving techniques. Some have become self-standing professions of their own under the rubric of systems engineering. Their academic, industrial, and governmental credentials are now well established.

All are science based—that is, based on measurables and a set of assumptions. The assumptions are that requirements and risks can be quantified, solutions can be optimized, and compliance specified. We encountered this at the beginning of this book, characterizing the "footprint" of complexity as systems engineering or systems architecting related. These same assumptions are also constraints on the kinds of problems that can be solved. In particular, science-based systems engineering does not do well in problems that are abstract, data-deficient, perceptual, or for which the criteria are immeasurable.

For example, the meanings of words such as safe, survivable, affordable, reliable, acceptable, good, and bad, are either outside the scope of systems engineering—"ask the client for some numbers"—or are force-fitted to it by subjective estimates. Yet these words are the language of the clients. Quantifying them can distort their inherent vagueness into an unintended constraint.

There is neither a group of professionals that better understands these difficulties than systems engineers and executives—nor who wish more to convert immeasurable factors to quantitatively statable problems by whatever techniques can help. The first step they made was to recognize the nature of the problems. The second was to realize that almost all of them occur at the front (and back) ends of the engineering cycle. Consider the following descriptive heuristics, developed long ago from systems engineering experience:

- All the most serious mistakes are made on the first day.
- Regardless of what has gone before, the acceptance criteria determine what is actually built.
- Reliability has to be designed in, not tested in.

It is no coincidence that many systems engineers, logically enough, now consider systems architecting to be "the front end of systems engineering" and that architectures are "established structures." More precisely, systems architecting can be seen as setting up the necessary conditions for systems engineering and certifying its results. In short, systems architecting provides concepts for analysis and criteria for success. In evolving systems, the functions of systems architecting, systems engineering, and disciplinary engineering are all more episodic. Concepts for analysis and criteria for success are established in early phases, but are revised with each new spiral through the development process. Systems engineers must control interfaces through many cycles of design, development, and integration, not just through one. In addition to conducting classical architecting episodically, the systems architect must also consider the issue of stable forms. The evolving system should not change everything on each cycle; it needs to retain stable substructures to evolve effectively. The definition of these substructures is part of the architect's role.

The immediate incentive for making architecting an explicit process, the necessary precursor to establishing it as a self-standing profession complementary to systems engineering, was the recognition in the late 1980s by systems executives that "something was missing" in systems development and acquisition. And the omission was causing serious trouble: system rejection by users, loss of competitive bids to innovators, products stuck in unprofitable niches, military failures in joint operations, system overruns in cost and schedule, and so on—all traceable to their beginnings. Yet, there was a strong clue to the answer. Retrospective examinations of earlier, highly successful systems showed the existence in each case of an overarching vision, created and maintained by a very small group that characterized the program from start to finish (Rechtin 1991).

Software engineers and their clients were among the first to recognize that the source of many of their software system problems was structural—that is, architectural. Brooks' famous book (Brooks Jr 1995) cited the presence or absence of an architect as a key determinant of success. Research in software architecture developed in universities such as Carnegie Mellon, the University of North Carolina Chapel Hill, the University of California at Irvine, and the University of Southern California. Practitioners began identifying themselves as software architects and forming architectural teams. Communication, electronics, and aerospace systems architects followed shortly thereafter.

Societies established architecture working groups, notably the INCOSE Systems Architecting Working Group (noted in the inaugural issue of the journal) and the IEEE Software Engineering Standards Committee's Architecture Working Group, that led to the standard ANSI/IEEE 1471 and then to the ISO/IEC standards later. These activities are essential to the development both of a common internal language for systems architecting and for the integration of software architecture models and overall systems architectures in complex, software-intensive systems.

At the scale of the profession of engineering, the recognition that something was missing led to identifying it, by direct analogy with the processes of the classical architectural profession, as systems architecting. Not surprisingly, the evolution of systems architecting tools was found to be already underway in model building, discussed in Part III, and systems standards, discussed in the next section.

SYSTEMS ARCHITECTING AND SYSTEMS STANDARDS

Earlier chapters have pointed out that the abstract problems of the conceptual and certification phases require different tools from the analytic ones of system development, production, and test. One of the most important sets of tools is that of systems standards. Chapter 11 discussed the most established standards for architecture descriptions, as well as their extension to process and evaluation standards. We examine process standards here, taking a larger perspective than the current architecting tailored standards. For historical reasons, architectural process standards were not developed as a separate set until very recently. Instead, general systems process standards were developed that included systems architecting elements and principles understood at the time, most of them induced from lessons learned in individual programs. Some key elements appeared as early as in the 1950s.

Driven by much the same needs, the recognition of systems architecting in the late 1980s was paralleled, independently, by a recognition that existing systems standards needed to be modified or supplemented to respond to long-standing systems-level structuring problems. Bringing the two tracks, architecting and standards, together should soon help both. Architecting can improve systems standards, while systems standards can provide valuable tools for the systems architecting profession.

Some of the earliest systems standards in which elements of systems architecting appeared were those of system specification, interface description, and interface management. They proliferated rapidly. A system specification can beget 10 subsystem specifications, each of which is supported by 10 lower-level subsystem specifications, and so on. All of these had to be knitted together by a system of 100 or so interface specifications.

Even though modern computer tools (computer-assisted system engineering [CASE] tools) have been developed to help keep track of the systems engineering process, extraordinarily disciplined efforts are still required to maintain the systems integrity. Systems engineering tools have come and gone, the field has not been one with long-term stable providers.

As systems complexity increased, systems engineers were faced with increasingly difficult tasks of assuring that the evolving form of the system met client needs, guaranteeing that trade-offs maintained system intent in the face of complications arising during development, and finally assuring that the system was properly tested and certified for use. In due course, the proliferation of detailed specifications led to a need for overarching guidelines, an overview mechanism for "structuring" the complexity that had begun to obscure system intent and integrity.

Before continuing, it should be pointed out that overarching guidelines are not, and cannot be, a replacement for quantitative system standards and specifications. The latter represent decades of corporate memory, measurable acceptance criteria, and certified practices. Guidelines—performance specifications, tailorable limits, heuristics, and the like—have a fundamental limitation. They cannot be certified by measurables. They are too "soft" and too prone to subjective perceptions to determine to the nearest few percentage points whether a system performs, or costs, what it should. At some point, the system has to be measured if it is to be judged objectively.

From the standpoint of an architecting profession, the most important fact about system standards is that they are changing. To understand the trend, their development will be reviewed in some detail, recognizing that some of them are continuing to be updated and revised.

THE ORIGINS OF SYSTEMS STANDARDS

In Chapter 11, we reviewed standards directly applicable to the practice of systems architecting. These are standards that speak directly to architecture description, processes, evaluation, or any of the former in specific domains of application ("Reference Architectures"). A broader question is to what extent standards for systems development have recognized, or now recognize the architecting activity?

The Ballistic Missile Program of the 1950s

Urgent needs induce change and, eventually, improvement. The US/Soviet ballistic missile race, which begun in the mid-to-late 1950s, brought about significant change, leading to the development and fielding of innovative and complex systems in an environment where national survival was threatened. To its credit, the US Air Force recognized the urgent need to develop and manage the process of complex system evolution, and did so. Much of what we now regard as mainline systems engineering, at least as applied to aerospace and military systems, originated in that era. Some references include Booton and Ramo (1984) and Johnson (2002), though significant dissent also exists (Hall 1984). The response in the area of standards was the "375" System Standard, subsequently applied to the development of all new complex Air Force equipment and systems.

"375" required several things that are now commonplace in systems architecting and engineering. Timelines depicting the time-sequenced flow of system operation were to be used as a first step in system analysis. From these, system functional block diagrams and functional requirements were to be derived as a basis for subsequent functional analysis and decomposition. The functional decomposition process, in turn, generated the subsystems that, along with their connections and constraints, comprised the system and allowed the generation of subsystem requirements via trade-off processes.

"375" was displaced in 1969 by a MILSTD 499 (Military Standard—499), which was applied throughout the Department of Defense. MILSTD 499A, an upgrade, was released in 1974 and was in effect for 20 years. MILSTD 499B, a later upgrade, was unofficially released in 1994, and was almost immediately replaced by EIA/IS 632 Interim Systems Engineering Standard.

The Beginning of a New Era of Standards

The era of MILSTDs 499/499A/499B was an era in which military standards became increasingly detailed. It was not only these documents that governed system architecting and engineering, but they in turn referenced numerous other DoD (Department of Defense) MILSTDs that addressed aspects of system engineering, and which were imposed on the military system engineering process as a consequence. To cite a few: MILSTD 490, Specification Practices, 1972; MILSTD 481A, Configuration Control—Engineering Changes, Deviations and Waivers (Short Form), 1972; MILSTD 1519, Test Requirements Document, 1977; and MILSTD 1543, Reliability Program Requirements for Space and Missile Systems, 1977. See Eisner (1988) for additional examples.

This mindset changed with the end of the Cold War in the late 1980s. Cost became an increasingly important decisive factor in competitions for military programs, supplanting performance, which had been the dominant factor in the prior era. Lowest cost, it was argued, could only be achieved if the restrictions of the military standards were muted. The detailed process ("how to") standards of the past, which specified how to conduct systems engineering and other program operations, needed to be replaced by standards that only provided guidelines, leaving the engineering specifics to the proposing companies that would select these so as to be able to offer a low-cost product. Further supporting this reasoning was the reality that the most sophisticated components and systems in fields such as electronic computer chips and computers were now available at low cost from commercial sources, whereas in the past, the state of the art was available only from MILSTD-qualified sources. It was in this environment that EIA/IS 632 was born.

EIA/IS 632, An Architectural Perspective

EIA/IS 632 is short by comparison with other military standards. Its main body is 36 pages. Including appendices, its total length is 60 pages, and these include several which have been left intentionally blank. And most significantly, no other standards are referenced.

The scope and intent of the document is best conveyed by the following quotes from its contents:

- "The scope … of systems engineering (activities) are defined in terms of what should be done, not how to do … (them)." (p. i)
- "(EIA/IS-632) identifies and defines the systems engineering tasks that are generally applicable throughout the system life cycle for any program." (p. 7)

This standard does, at least generally, recognize the systems architecting role. One of the major activities of the systems architect, which of giving form to function, is addressed in pages 9 through 11. These pages summarize, in their own words and style, the client/architect relationship, the establishment of the defining system functions, the development of the system's architecture, and the process of allocating system functions to architectural elements via trade-offs. By implication, the trade-offs continue, with varying degrees of concentration, throughout the life cycle. It does not recognize the notion of architectures being smaller subsets of the larger set of decisions, the idea that an architecture is the set of fundamental or unifying structure that is recognized in the later, directly architecture relevant standards in the ISO/IEC/IEEE 420xy series.

Further Developments

Even though EIA/IS 632 applies only to military systems engineering, that was not its original intent. The objective was to develop a universal standard for systems engineering that would apply to both the military and commercial worlds and be ratified by all of industry. However, there was an urgency to publish a new military standard and in the 4-month schedule that was assigned, only it could be developed. This led to two consequences. First, IEEE 1220, a commercial systems engineering standard, was separately published. Second, the merging of EIA/IS 632 with IEEE 1220 to create the first universal standard for system engineering was planned and eventually realized in 12207. At the international level, ISO (The International Standards Organization) issued an overall standard covering the development and engineering of systems (15288). This standard is not a systems engineering standard per se, but it lays out a set of activities, including systems engineering activities, associated with developing a system. 15288 is intended to be a standard at a level above that of the other cited standards. The IEEE joined the 15288 effort, which merged some other efforts into it, and that resulted in a series of issuances of ISO/IEC/IEEE 15288, most recently 2023.

Architecture Definition in 15288

15288 (ISO 2023) has several important things related to architecting, some very explicit and some more buried in the clauses. 15288 defines technical processes in development, clustered into groups where there is expected to be high iteration in a group, with some additional iteration between groups.

Concept Definition consists of two technical processes: 6.4.1 Business or Mission Analysis and 6.4.2 Stakeholder Needs and Requirements Definition. **System Definition** consists of 6.4.3 System Requirements Definition, 6.4.4 System Architecture Definition, and 6.4.5 Design Definition. These are followed by System Realization and System Deployment and Use. Overall, this follows a fairly conventional vision of sequential or waterfall development, something we elaborated on and critiqued elsewhere in this book. The breakdown clearly identifies Systems Architecture as a technical process. No longer is it implied, it is explicit in the 15288 series. 15288 also calls out the ISO/IEC/IEEE 420x0 standards (specifically 42010, 42020, and 42030) as elaborating the System Architecture Definition technical process, so the definitions, concepts, and elements of the 420x0 series are incorporated. As we have seen in Chapter 11, those standards fit well both with the overall message of this book (integrated consideration of problem and solution, identification of the high leverage decisions) as well as APM-ASAM process heuristics and MBSE implementation.

The 15288 standard further goes onto note in Section 5.8.2 that iteration among mission analysis, stakeholder needs definition, system requirements definition, system architecture definition, and design definition is common and valuable. It is needed to reach a common understanding of both problem to be solved and a satisfactory solution (ISO 2023, pp. 27). This is strongly the message of systems architecting here, and helps identify the profession of systems architecting appropriately, as working the problem–solution interaction and not just deriving a top-level solution from system requirements, as might otherwise be assumed.

Clause 5.9 further elaborates on the relationships of concept and system definition. Page 29 of ISO (2023) notes "The system architecture definition process focuses on defining an architecture that addressed the stakeholder concerns and is applied iteratively and concurrently with business or mission analysis, stakeholder needs and requirements definition, and system requirements definition to determine the best solution to address stakeholder concerns." Here we have a clear basis for focusing architecting on a holistic picture of needs and solutions.

As noted earlier in the book, ISO/IEC/IEEE 42020 (ISO 2018), the standard on architecture processes, defines six processes. The 15288 technical process 6.4.4, System Architecture Definition Process, directly maps to three of the six: architecture conceptualization, architecture evaluation, and architecture elaboration (really description, based on the definition). The other portions defining 6.4.4 are planning and management related, not the core activities.

SYSTEMS ARCHITECTING GRADUATE EDUCATION

Systems Engineering Universities and Systems Architecting

Graduate education, advanced study, and research give a profession its character. They distinguish it from routine work by making it a vocation, a calling of the particularly qualified.

The first university to offer master's and doctorate degrees in systems engineering was the University of Arizona, beginning in 1961. The program began as a graduate program; an undergraduate program and the addition of Industrial Engineering to the department title came later. Still in existence, the graduate department has well over 1,000 alumni.

The next to offer advanced degrees was the Virginia Tech in 1971, but not until after 1984 did additional universities join the systems engineering ranks. They included Boston University, George Mason University, the Massachusetts Institute of Technology (MIT), the University of Maryland, the University of Southern California (USC), the University of Tel Aviv, and the University of Washington. Other universities have since added degrees as well, usually as options within Industrial and Systems Engineering or Mechanical Engineering departments.

To the best knowledge of the authors of this book, the University of Southern California was the first to offer a graduate degree in Systems Architecting and Engineering with the focus on systems architecting. However, of the universities offering graduate degrees in systems engineering, some half dozen now include systems architecting within their curricula. Notable among them is the MIT Systems Design and Management (SDM) program. This program, which is intended as a new kind of graduate education program for technical professionals, is built on three core subjects: Systems Engineering, Systems Architecture, and Project Management. Although the degree is not focused on systems architecting, that subject forms a major part of the curriculum's core. The MIT SDM curriculum is becoming more of a national model as it is spread through the Product Development in the 21st century (PD21) program. PD21 is creating programs that are similar to MIT's SDM program in universities across the country. The current universities involved are the Rochester Institute of Technology, the University of Detroit–Mercy, and the Naval Postgraduate School.

Architecting is also becoming a strong interest in universities offering advanced degrees in computer science with specializations in software and computer architectures; notably, Carnegie Mellon University, the Universities of California at Berkeley and Irvine, and USC. At USC, the systems architecture and engineering degree began with an experimental course in 1989, and formally became a Master of Science degree program in 1993 following its strong acceptance by students and industry.

In the last 10 years, there has been growing recognition of the value of interdisciplinary programs, which of itself would favor systems architecting and engineering. These have been soul-searching years for industry, and the value of systems architecting and engineering has become appreciated as a factor in achieving a competitive advantage. Also, the restructuring of industry has caused a rethinking of the university as a place to provide industry-specific education. These trends, augmented by the success of the systems architecting and engineering education programs, have caused university architect-engineering programs to prosper.

The success of these programs can be measured in several ways. First, the direction is one of growth. New programs are being formed and several award Ph.D. degrees in systems engineering. MIT formed a cross-cutting Engineering Systems Division (ESD) that drew from all of the traditional departments, and has awarded a number of Ph.D. degrees in Engineering Systems. Second, systems architecting and systems architecting education are making a positive difference in industry, as supported by industry surveys. In point of fact, company-sponsored systems architecting enrollments have increased even during the part of this period where there was industrial contraction.

Curriculum Design

It is not enough in establishing a profession to show that universities are interested in the subject, there has to be something to teach. To have a distinctive program, there has to be distinctive material. Architecting is design centric. To be a good architect means being able to do very early phase, original design. As discussed in this book, a good architect understands the problem domain deeply, and can translate that into solution concepts. Logically, an educational program needs to address the same components, problem domain understanding, solution expertise, and the softer skills of a good designer.

The USC program is an example of addressing these needs in an academic setting. The USC master's program admits students satisfying the School of Engineering's academic requirements and having a minimum of 3 years applicable industrial experience. Students propose a 10-course curriculum that is reviewed, modified if required, and accepted as part of their admission. The curriculum requires graduate-level courses as follows:

- An anchor systems architecting course.
- An advanced engineering economics course.
- One of several specified engineering design courses.
- Two elective courses in technical management from a list of 11 that are offered.
- One of eight general technical area elective courses.
- Four courses from one of 11 identified technical specialization areas, each of which has six or more courses offered.

The structure of this M.S. in Systems Architecture and Engineering curriculum has been designed based on both industrial and academic advice. Systems architecture is better taught in context. It is too much to generally expect a student to appreciate the subtleties of the subject without some experience. And the material is best understood through a familiar specialty area in which the student already practices. The 3-year minimum experience requirement and the requirement of four courses in a technical specialty area derive from this reasoning.

The need for an anchor course is self-evident. Systems architecture derives from inductive and heuristic reasoning, unlike the deductive reasoning used in most other engineering courses. To fully appreciate this difference, the anchor course is taken early, if not first, in the sequence. The course contains no examinations as such but requires two professional-quality reports so that the student can best experience the challenges of systems architecting and architecture by applying his or her knowledge in a dedicated and concentrated way.

Experience has shown that a design experience course, the advanced economics course, and courses in technical management are valuable to the systems architect, and therefore they are curriculum requirements. The additional course in a general technical area allows the student to select a course that most rounds out the student's academic experience. Possibilities include a system architecting seminar, a course on decision support systems, and a course on the political process in systems architecture design.

When teaching systems architecting in a professional/industrial setting, even greater emphasis can be placed on the design experience. In some programs taught by one of the authors, students were invited to bring in their ideas for new systems in their home offices, recruit other students into a team, and pursue the idea as a class project. In some cases, groups of students were sent by their home office as a team with an assigned problem area. In ways, this mirrors some of the methods of traditional architecture education where studios and design exercises, with critical feedback from faculty, is an essential element of the program.

Advanced Study in Systems Architecting

A major component of advanced study in any profession is graduate-level research and refereed publications at major universities. In systems architecting, advanced study can be divided into two relatively distinct parts: that of its science, closely related to that of systems engineering, and of its art. Advanced study in its art, though often illustrated by engineering examples, has many facets, and has historically been part of university research (if a far smaller portion than engineering science). Some examples include the following:

- Complexity, by Flood and Carson at City University London, England (Flood and Carson 2013).
- Problem solving, by Klir at the State University of New York at Binghamton (Klir 1985) and by Rubinstein at the University of California at Los Angeles (Rubinstein and Firstenberg 1994).
- Systems and their modeling, by Churchman at the University of California at Berkeley (Churchman 1972) and Nadler (Nadler 1981, 1982, 1985) and Rechtin (Rechtin 1991) at the University of Southern California (USC).
- The behavioral theory of architecting, by Lang at the University of Pennsylvania (Lang 1987), Rowe at Harvard (Rowe 1991).
- Machine (artificial) intelligence and computer science, by Genesereth and Nilsson (Genesereth and Nilsson 2012) at Stanford, Newell (Newell 1994) and Simon (Simon 2019) at Carnegie Mellon University.

- Software architecting, by Garlan and Shaw.(Garlan and Shaw 1993, Shaw and Garlan 1996) at Carnegie Mellon University and Barry Boehm at USC.

All have contributed basic architectural ideas to the field. Many are standard references for an increasing number of professional articles by a growing number of authors. Most deal explicitly with systems, architectures, and architects, although the practical art of systems architecting was seldom the primary motivation for the work. That situation predictably will change rapidly as both industry and government face international competition in a new era.

Professional Societies and Publications

Existing journals and societies were the initial professional media for the new fields of systems architecting and engineering. Because much early work was done in aerospace and defense, it is understandable that the IEEE Society on Systems Man and Cybernetics, the IEEE Aerospace Electronics Society, and the American Institute of Aeronautics and Astronautics, and their journals, along with others, became the professional outlets for these fields. One excellent sample paper from this period (Booton and Ramo 1984) explained the contributions that systems engineering had made to the US ballistic missile program.

The situation changed in 1990 when the first International Council on Systems Engineering conference was held and attracted 100 engineers. INCOSE became the first professional society dedicated to systems engineering and soon established a Systems Architecture Working Group. INCOSE now has more than 20,000 members, publishes a quarterly newsletter and a journal, and has held annual symposia for more than 30 years. The journal first appeared in 1994.

In 2005, the IEEE formed a Systems Council and soon after established the *IEEE Systems Journal*. The *IEEE Systems Journal* and the *Journal of INCOSE*, *Systems Engineering*, have carried many architecting related papers.

CONCLUSION: AN ASSESSMENT OF THE PROFESSION

The profession of systems architecting has come a long way. We are well past the beginning, but a long way from the maturity of many other specialities. The present body of professionals in industry and academia, began their careers in disciplinary specialities, most often electronics, control, and software systems, then broadened into systems engineering, formed the core of small design teams, and now consider themselves as architects. The profession has been nurtured within the framework of systems engineering, and no doubt will maintain a tight relationship with it. A masters-level university curriculum now exists, and the material and ideas are suffusing into many other systems-oriented programs. Applicable research is underway in universities. Applicable standards and tools now exist at the national and international levels. It has an acknowledged home within INCOSE as well as other professional societies, which, together with their publications, provide a medium for professional expression and development.

It is interesting to speculate on where the profession might be going and how it might get there. The cornerstone thought is that the future of a profession of systems architecting will be largely determined by the perceptions of its utility by its clients. If a profession is useful, it will be sponsored by them and prosper. To date, all indicators are positive.

Judging by the events that have led to its status today, and by comparable developments in the history of classical architecture, systems architecting could evolve well as a separate business entity. The future could hold more systems architecting firms that bid for the business of acting as the technical representative or agent of clients with their builders. There are related precedents today in Federally Funded Research and Development Centers (FFRDCs) and Systems Engineering and Test Assistance Contractors (SETACs), independent entities selected by the Department of Defense to represent it with defense contractors that build end products. Similar precedents exist in NASA and the Department of Energy.

The role of graduate education is likely to grow and spread. Today's products and processes are more netted and interrelated than those of 10 years ago, and tomorrow's will be even more so. System thinking is proving to be fundamental to commercial success, and systems architecting will increasingly become a crucial part of new product development. It is incumbent upon universities to capture the intellectual content of this phenomenon and embody it in their curricula. This will require a tight coupling with industry to be aware of important real-world problems, a dedication to research to provide some of the solutions, and an education program that trains students in relevant architectural thinking.

In summary, all the indicators point to a future of high promise and value to all stakeholders.

NOTES

1 Webster's II, New Riverside University Dictionary, Boston, MA: Riverside, 1984, p. 939.

2 Early examples include the *IEEE Transactions on Systems, Man, and Cybernetics*, the *IEEE Transactions on Aerospace and Electronic Systems*, and the *Journal of the American Institute of Aeronautics and Astronautics*.

REFERENCES

Booton, R. C. and S. Ramo (1984). "The development of systems engineering."*IEEE Transactions on Aerospace and Electronic Systems* **4**: 306–310.

Brooks Jr, F. P. (1995). *The Mythical Man-Month(Anniversary ed.)*, Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Chestnut, H. (1967). *Systems Enginering Methods*, New York: John Wiley.

Churchman,C. W. (1972). *The Design of Inquiring Systems*, New York: Basic Books.

Eisner, H. (1988). *Computer Aided Systems Engineering*, Englewood Cliffs, NJ: Prentice Hall.

Fabrycky, W. J. and B. S. Blanchard (1981). *Systems Engineering and Analysis*, Englewood Cliffs, NJ: Prentice-Hall.

Flood, R. L. and E. R. Carson (2013). *Dealing with Complexity: An Introduction to the Theory and Application of Systems Science*, Berlin: Springer Science & Business Media.

Garlan, D. and M. Shaw (1993). *In Introduction to Software Architecture*, Pittsburgh, PA: Carnegie Mellon University.

Genesereth, M. R. and N. J.Nilsson (2012). *Logical Foundations of Artificial Intelligence*, Burlington, MA: Morgan Kaufmann.

Hall, E. N. (1984). *The Art of Destructive Management: What Hath Man Wrought?*, New York: Vantage Press.

ISO (2018). *ISO/IEC/IEEE FDIS 42020 Enterprise, Systems, and Software - Architecture Processes*, New York: International Standards Organization.

ISO (2023). *ISO/IEC/IEEE 15288: 2023 Systems and Software Engineering - System Life Cycle Processes*, New York: IEEE.

Johnson, S. B. (2002). "Bernard Schriever and the scientific vision." *Air Power History* **49**(1): 30–45.

Klir, G. J. (1985). *Architecture of Systems Problem Solving*,New York: Plenum Press.

Lang, J. (1987). *Creating Architectural Theory*, New York: van Nostrand Rheinhold.

Machol, R. E. (1965). *System Engineering Handbook*, New York: McGraw-Hill.

Nadler, G. (1981). *Planning and Design Approach*, New York: John Wiley & Sons, Inc.

Nadler, G. (1982). "The planning and design professions." *Human Systems Management* **3**(4):289–300.

Nadler, G. (1985). "Systems methodology and design." *IEEE Transactions on Systems, Man, and Cybernetics* **6**: 685–697.

Newell, A. (1994). *Unified Theories of Cognition*,Cambridge, MA: Harvard University Press.

Rechtin, E. (1991). *Systems Architecting: Creating and Building Complex Systems*, Englewood Cliffs, NJ: Prentice Hall.

Rowe, P. G. (1991). *Design Thinking*, Cambridge, MA: MIT Press.

Rubinstein, M. F. andI. Firstenberg (1994). *Patterns of Problem Solving*, Englewood Cliffs, NJ: Prentice Hall.

Shaw, M. and D. Garlan (1996). *Software Architecture: Perspectives on An Emerging Discipline*, Englewood Cliffs, NJ: Prentice-Hall, Inc.

Simon, H. A. (2019). *The Sciences of the Artificial, Reissue of the Third Edition with a New Introduction by John Laird*, Cambridge, MA: MIT Press.

Appendix A
Heuristics for System-Level Architecting

Experience is the hardest kind of teacher.
It gives you the test first and the lesson afterward.

Susan Ruth (1993)

INTRODUCTION: ORGANIZING THE LIST

The heuristics to follow were selected from Rechtin (1991), the *Collection of Student Heuristics in* Systems Architecting, 1988*–1993* (Rechtin 1994), and from subsequent studies in accordance with the selection criteria of Chapter 2. The list is intended as a tool store for top-level systems architecting. Heuristics continue to be developed and refined not only for this level but also for domain-specific applications, often migrating from domain-specific to system level and vice versa.[1]

For ease of search and use, the heuristics are grouped by architectural task and categorized by being either descriptive or prescriptive—that is, whether they describe an encountered situation or prescribe an architectural approach to it.

There are over 180 heuristics in the listing to follow, far too many to study at any one time. Nor were they intended to be. The listing is intended to be scanned as one would scan software tools on software store shelves, looking for ones that can be useful immediately but remembering that others are also there. Although some are variations of other heuristics, the vast majority stand on their own, related primarily to others in the near vicinity on the list. Odds are that the reader will find the most interesting heuristics in clusters, the location of which will depend on the reader's interests at the time. The section headings are by architecting task. A "D" signifies a descriptive heuristic; a "P" signifies a prescriptive one. Clusters are identified by insetting more specific versions of the heuristic, usually prescriptive under a descriptive, but sometimes more specific versions that are still descriptive. In the interests of brevity, an individual heuristic is listed in the task where it is most likely to be used most often. As noted in Chapter 2, some 20% can be tied to related ones in other tasks.

A major difference between a heuristic and an unsupported assertion is the credibility of the source. To the extent possible, the heuristics are credited to the individuals who, to the authors' knowledge, first suggested them. To further aid the reader in judging credibility or in finding the sources, the heuristics to follow are given source-indicating symbols. These symbols indicate the following:

[ ] *An informal discussion* with the individual indicated, unpublished.

( ) A formal, dated source, with examples, located in the University of Southern California (USC) Master of Science in Systems Architecture and Engineering (MS-SAE) program archive, especially in the Collection of Student Heuristics in Systems Architecting, 1988–1993. For further information, contact the Master of Science Program in Systems Architecture and Engineering, USC School of Engineering, University Park, Los Angeles, California 90089-1450.

* Rechtin 1991, where it is sourced more formally. By permission of Prentice Hall Inc., Englewood Cliffs, New Jersey 07632.

Bold Key words useful for quick search. Otherwise, heuristics to follow are in plain type to make page reading easier. Real-world examples of each can be found in the references indicated.

The authors apologize in advance for any miscrediting of sources. Corrections are welcome. Readers are reminded that not all heuristics apply to all circumstances, just most to most.

HEURISTIC TOOL LIST

Multitask Heuristics

D **Performance, cost, and schedule** cannot be specified independently. At least one of the three must depend on the others.*

D With few exceptions, schedule **delays** will be **accepted** grudgingly; cost **overruns** will **not**, and for good reason.

D The **time to completion** is proportional to the ratio of the time spent to the time planned to date. The greater the ratio is, the longer the time to go.

D **Relationships** among the elements are what give systems their added value.[2]

D **Efficiency** is inversely proportional to **universality** (Douglas R. King 1992).

D **Murphy's Law**, "If anything can go wrong, it will."*

P **Simplify**. Simplify. Simplify.*

P The first line of defense against complexity is simplicity of design.

P Simplify, combine and eliminate. (Suzaki 1987)

P Simplify with smarter elements. (N. P. Geiss 1991)

P The most **reliable part** on an airplane is the one that isn't there—because it isn't needed. [DC-9 Chief Engineer 1989]

D One person's **architecture** is another person's **detail**. One person's system is another's component. [Robert Spinrad 1989]*

P In order to **understand anything**, you must not try to understand everything. (Aristotle, 4th century B.C.)

P Don't confuse the functioning of the parts for the functioning of the system. (Jerry Olivieri 1992)

D In general, each **system level** provides a context for the level(s) below. (G. G. Lendaris 1986)

- **P** Leave the **specialties** to the specialist. The level of detail required by the architect is only to the depth of an element or component critical to the system as a whole. (Robert Spinrad 1990) But the architect must have **access** to that level and know, or be informed, about its criticality and status. (Rechtin 1990)
- P Complex systems will develop and **evolve** within an overall architecture much more rapidly if there are **stable intermediate** forms than if there are not. (Simon 1969)*

D Particularly for social systems, it's the **perceptions**, not the facts, that count.

D In introducing technological and **social** changes, **how** you do it is often more important than **what** you do.*

- P If social cooperation is required, the **way** in which a system is **implemented** and introduced must be an **integral part** of its architecture.*

D If the **politics** don't fly, the hardware never will. (Brenda Forman 1990)

- D Politics, not technology, sets the limits of what technology is allowed to achieve.
- D Cost rules.
- D A strong, coherent constituency is essential.
- D Technical problems become political problems.
- D There is no such thing as a purely technical problem.
- D The best engineering solutions are not necessarily the best political solutions.

D **Good products** are not enough. Implementations matter. (Morris and Ferguson 1993)

- P To remain competitive, determine and **control the keys** to the architecture from the very beginning.

Scoping and Planning

The beginning is the most important part of the work.

Plato, 4th century B.C.

Scope! Scope! Scope!

William C. Burkett, 1992

D **Success** is defined by the **beholder**, not by the architect.*

P The most important single element of success is to **listen** closely to what the customer perceives as his requirements and to have the will and ability to be responsive. (J. E. Steiner 1978)*

P **Ask early** about how you will **evaluate** the success of your efforts. (F. Hayes-Roth et al., 1983)

P For a system to meet its **acceptance criteria** to the satisfaction of all parties, it must be architected, designed, and built to do so—no more and no less.*

P Define how an **acceptance** criterion is to be certified at the same time the criterion is established.*

D Given a **successful** organization or system with valid criteria for success, there are some things it **cannot do**—or at least not do well. Don't force it!

P The **strengths** of an organization or system in one context can be its **weaknesses** in another. Know when and where!*

D There is nothing like being the **first success**.*

P If at first you don't succeed, but the architecture is sound, try, try again. Success sometimes is where you find it. Sometimes it finds you.*

D A system is successful when the **natural intersection** of technology, politics, and economics is found. (A. D. Wheelon 1986)*

D Four questions, **the Four Who's**, need to be answered as a self-consistent set if a system is to succeed economically, namely, who benefits? who pays? and, as appropriate, who loses?

D **Risk** is (also) defined by the **beholder**, not the architect.

P If being absolute is impossible in estimating system risks, then be relative.*

D **No** complex system can be **optimum** for all parties concerned, nor can all functions be optimized.*

P Look out for hidden agendas.*

P It is sometimes more important to know **who** the customer is than to know what the customer **wants**. (Whankuk Je 1993)

D The phrase, **"I hate it,"** is direction. (Lori I. Gradous 1993)

P Sometimes, but not always, the best way to solve a difficult problem is to **expand** the problem itself.*

P Moving to a **larger purpose** widens the range of solutions. (Gerald Nadler 1990)

P Sometimes it is necessary to **expand the *concept*** in order to simplify the problem. (Michael Forte 1993)

P [If in difficulty,] **reformulate** the problem and re-allocate the system functions. (Norman P. Geis 1991)

P Use **open** architectures. You will need them once the market starts to respond.

P Plan to **throw one away**. You will anyway. (F. P. Brooks, Jr. 1982)

P You can't avoid **redesign**. It's a natural part of design.*

P Don't make an architecture **too smart** for its own good.*

D Amid a **wash of paper**, a small number of documents become critical pivots around which every project's management revolves. (F. P. Brooks, Jr. 1982)*

P Just because it's written, doesn't make it so. (Susan Ruth 1993)

D In architecting a new [software] program, all the **serious mistakes** are made in the **first day**. [Spinrad 1988]

P The most **dangerous** assumptions are the **unstated** ones. (Douglas R. King 1991)

D Some of the **worst** failures are **systems** failures.

D In architecting a new [aerospace] system, by the time of the first **design review**, performance, cost, and schedule have been **predetermined**. One might not know what they are yet, but to first order all the critical assumptions and choices have been made which will determine those key parameters.*

P **Don't assume** that the original statement of the problem is necessarily the best, or even the right, one.*

P **Extreme** requirements, expectations, and predictions should remain under challenge throughout system design, implementation, and operation.

P Any extreme requirement must be intrinsic to the system's design philosophy and must validate its selection. "Everything must pay its way on to the airplane." [Harry Hillaker 1993]

P **Don't assume** that previous **studies** are necessarily complete, current, or even correct. (James Kaplan 1992)

P Challenge the process and solution, for surely someone else will do so. (Kenneth L. Cureton 1991)

P Just because it worked in the past there's no guarantee that it will work now or in the future. (Kenneth L. Cureton 1991)

P Explore the situation from more than one point of view. A seemingly impossible situation might suddenly become transparently simple. (Christopher Abts 1988)

P **Work forward and backward**. (A set of heuristics from Rubinstein 1975)*

Generalize or specialize.
Explore multiple directions based on partial evidence.
Form stable substructures.
Use analogies and metaphors.
Follow your emotions.

P Try to hit a solution that, at worst, won't put you **out of business**. (Bill Butterworth as reported by Laura Noel 1991)

P The **order** in which decisions are made can change the architecture as much as the decisions themselves. (Rechtin 1975, IEEE SPECTRUM)

P Build in and maintain **options** as long as possible in the design and build of complex systems. You will need them. OR … Hang on to the agony of decision as long as possible. [Robert Spinrad 1988]*

P Successful architectures are **proprietary, but open**. [Morrison and Ferguson 1993]

D Once the architecture begins to take shape, the sooner contextual constraints and **sanity checks** are made on assumptions and requirements, the better.*

D Concept **formulation** is complete when the **builder** thinks the system can be built to the **client's** satisfaction.*

D The **realities** at the end of the conceptual phase are not the models but the **acceptance criteria**.*

P Do the **hard parts** first.

P Firm **commitments** are best made after the **prototype works**.

Modeling[3]

P If you can't analyze it, don't build it.

D Modeling is a **craft** and at times an art. (William C. Burkett 1994)

D A **vision** is an **imaginary architecture …** no better, no worse than the rest of the models. (M. B. Renton Spring 1995)

D From psychology: If the concepts in the mind of one person are very different from those in the mind of the other, there is no **common model** of the topic and no communication. [Taylor 1975] OR … From telecommunications: The **best receiver** is one that contains an internal model of the transmitter and the channel. [Robert Parks & Frank Lehan 1954]*

D A model is not **reality**.*

D The **map** is not the territory. (Douglas R. King 1991)*

P Build **reality checks** into model-driven development. [Larry Dumas 1989]*

P Don't believe **nth order consequences** of a first order [cost] model. [R. W. Jensen circa 1989]

D Constants aren't and variables don't. (William C. Burkett 1992)

D One **insight** is worth a thousand **analyses**. (Charles W. Sooter 1993)

P Any war game, systems analysis, or study whose results can't easily be explained on the back of an envelope is not just worthless, it is probably dangerous. [Brookner-Fowler circa 1988]

D Users develop **mental models** of systems based [primarily] upon the user-to-system interface. (Jeffrey H. Schmidt)

D If you can't explain it in **five minutes**, either you don't understand it or it doesn't work. (Darcy McGinn 1992 from David Jones)

P The **eye** is a fine **architect**. Believe it. [Wernher von Braun 1950]

D A good solution somehow **looks nice**. (Robert Spinrad 1991)

P **Taste:** an aesthetic feeling that will accept a solution as right only when no more direct or simple approach can be envisaged. [Robert Spinrad 1994]

P Regarding **intuition**, trust but **verify**. (Jonathan Losk 1989)

Prioritizing (Trades, Options, and Choices)

D In any resource-limited situation, the **true value** of a given service or product is determined by what one is willing to **give up** to obtain it.

P When choices must be made with unavoidably inadequate information, **choose** the best available and then **watch** to see whether future solutions appear faster than future problems. If so, the choice was at least adequate. If not, go back and **choose** again.*

P When a decision makes sense through several different **frames**, it's probably a good decision. (J. E. Russo 1989)

D The **choice** between architectures may well depend upon which set of **drawbacks** the client can handle best.*

P If trade results are inconclusive, then the wrong selection criteria were used. Find out [again] what the customer wants and why they want it, then repeat the trade using those factors as the [new] selection criteria. (Kenneth Cureton 1991)

P The triage: Let the dying die. Ignore those who will recover on their own. And treat only those who would die without help.*

P Every once in a while you have to go back and see what the real world is telling you. [Harry Hillaker 1993]

Aggregating ("Chunking")

P **Group** elements that are strongly **related** to each other, **separate** elements that are unrelated.

D Many of the **requirements** can be **brought together** to complement each other in the total design solution. Obviously the more the design is put together in this manner, the more probable the overall success. (J. E. Steiner 1978)

P Subsystem interfaces should be drawn so that each subsystem can be implemented independently of the specific implementation of the subsystems to which it interfaces. (Mark Maier 1988)

P Choose a configuration with **minimal communications** between the subsystems. (computer networks)*

P Choose the elements so that they are as independent as possible; that is, elements with low external complexity (low coupling) and high internal complexity (high cohesion). (Christopher Alexander 1964 modified by Jeff Gold 1991)*

P Choose a configuration in which local activity is high speed and global activity is slow change. (P. J. Courtois 1985)*

P Poor aggregation results in **gray** boundaries and **red** performance. (M. B. Renton Spring 1995)

P **Never aggregate** systems that have a **conflict** of interest; partition them to ensure checks and balances. (Aubrey Bout 1993)

P **Aggregate** around **"testable"** subunits of the product; **partition** around logical **subassemblies**. (Ray Cavola 1993)

P Iterate the partition/aggregation procedure until a model consisting of **7 ± 2 chunks** emerge. (Moshe F. Rubinstein 1975)

P The **optimum number** of architectural elements is the amount that leads to distinct **action**, not general planning. (M. B. Renton Spring 1995)

P System structure should resemble functional structure.*

P Except for good and sufficient reasons, **functional and physical** structuring should match.*

P The architecture of a **support** element must **fit** that of the system it supports. It is easier to match a support system to the human it supports than the reverse.*

P **Unbounded limits** on element behavior may be a **trap** in unexpected scenarios. [Bernard Kuchta 1989]*

Partitioning (Decompositioning)

P Do not **slice** through regions where **high rates** of information exchange are required. (computer design)*

D The greatest **leverage** in architecting is at the **interfaces.***

P **Guidelines** for a good quality interface specification: They must be simple, unambiguous, complete, concise, and focus on substance. Working documents should be the same as customer deliverables; that is, always use the customer's language, not engineering jargon. [Harry Hillaker 1993]

P The **efficient architect**, using contextual sense, continually looks for likely **misfits** and redesigns the architecture so as to eliminate or minimize them. (Christopher Alexander 1964)* It is inadequate to architect up to the boundaries or **interfaces** of a system; one must architect **across** them. (Robert Spinrad as reported by Susan Ruth 1993)

P Since boundaries are inherently limiting, look for solutions outside the boundaries. (Steven Wolf 1992)

P Be prepared for **reality** to add a few interfaces of its own.*

P Design the structure with **good "bones."***

P Organize personnel tasks to **minimize** the **time** individuals spend interfacing. (R. C. Tausworthe 1988)*

Integrating

D **Relationships** among the elements are what give systems their **added value**.*

P The greatest leverage in system architecting is at the interfaces.*
P The greatest **dangers** are also at the interfaces. [Raymond 1988]
P Be sure to ask the question, "What is the worst thing that other elements could do to you across the interface?" [Kuchta 1989]

D Just as a piece and its template must match, so must a system and the resources that make, test, and operate it. Or, more briefly, the **product and process** must match. Or, by extension, a system architecture cannot be considered complete lacking a suitable match with the process architecture.*

P When confronted with a particularly difficult interface, try changing its **characterization**.*

P Contain **excess energy** as close to the source as possible.*

P Place barriers in the paths between energy sources and the elements the energy can damage. (Kjos 1988)*

Certifying (System Integrity, Quality, and Vision)

D As **time to delivery** decreases, the **threat** to functionality increases. (Steven Wolf 1992)

P If it is a good design, **insure** that it stays **sold**. (Dianna Sammons 1991)

D Regardless of what has gone before, the **acceptance criteria** determine what is actually built.*

D The number of **defects remaining** in a (software) system after a given level of test or review (design review, unit test, system test, etc.) is proportional to the **number found** during that test or review.
P **Tally** the defects, analyze them, **trace** them to the source, make corrections, keep a **record** of what happens afterwards and keep **repeating** it. [Deming]
P **Discipline.** Discipline. Discipline. (Douglas R. King 1991)
P The principles of minimum communications and proper partitioning are key to system testability and **fault isolation**. (Daniel Ley 1991)*
P **The five whys** of Toyota's lean manufacturing. (To find the basic cause of a defect, keep asking "why" from effect to cause to cause five times.)

D The test **setup** for a system is itself a system.*

P The test system should always allow a part to pass or fail on its own merit. [James Liston 1991]*
P To be tested, a system must be designed to be tested.*

D An element **"good enough"** in a small system is unlikely to be good enough in a more complex one.*
D Within the same class of products and processes, the **failure rate** of a product is linearly proportional to its **cost**.*
D The **cost** to find and **fix** an inadequate or failed part increases by an **order of magnitude** as it is successively incorporated into higher levels in the system.

P The least expensive and most effective place to find and fix a problem is at its source.

D Knowing a **failure has occurred** is more important than the actual failure. (Kjos 1988)
D **Mistakes** are **understandable**, failing to report them is **inexcusable**.
D Recovery from failure or flaw is not complete until a specific mechanism, **and no other**, has been shown to be the cause.*
D Reducing **failure rate** by each **factor of two** takes as much effort as the original development.*
D **Quality** can't be tested in, it has to be **built in.***

D You can't achieve quality … unless you specify it. (Deutsch 1988)
P Verify the quality close to the source. (Jim Burruss 1993)
P The five why's of Japan's lean manufacturing (Hayes et al. 1988)
D High-**quality**, reliable systems are produced by high-quality architecting, engineering, design and manufacture, **not by inspection**, test, and rework.*
P Everyone in the development and production line is both a customer and a supplier.

D Next to interfaces, the greatest **leverage** in architecting is in aiding the recovery from, or exploitation of, **deviations** in system performance, cost, or schedule.*

Assessing Performance, Cost, Schedule, and Risk

D A good design has benefits in more than one area. (Trudy Benjamin 1993)
D System quality is defined in terms of customer satisfaction, not requirements satisfaction. (Jeffrey Schmidt 1993)
D If you think your **design** is perfect, it's only because you haven't shown it to **someone else**. [Harry Hillaker, 1993]

P Before proceeding too far, **pause and reflect!** Cool off periodically and seek an independent review. (Douglas R. King 1991)

D Qualification and **acceptance** tests must be both definitive and **passable.***

P High **confidence**, not test completion, is the **goal** of successful qualification. (Daniel Gaudet 1991)

P Before ordering a **test** decide what you will do if it is (1) **positive** or if (2) it is negative. If both answers are the same, **don't do** the test. (R. Matz, M.D. 1977)

D **"Proven"** and **"state of the art"** are mutually **exclusive** qualities. (Lori I. Gradous 1993)

D The **bitterness** of **poor performance** remains long after the sweetness of low prices and prompt delivery are forgotten. (Jerry Lim 1994)

D The **reverse of diagnostic** techniques are good architectures. (M. B. Renton 1995)

D Unless everyone who **needs to know** does know, somebody, somewhere will foul up.

P Because there's no such thing as immaculate communication, don't ever stop **talking** about the system. (Losk 1989)*

D Before it's tried, it's **opinion**. After it's tried, it's **obvious**. (Wm. C. Burkett 1992)

D Before the **war**, it's opinion. After the war, it's too late! (Anthony Cerveny 1991)

D The first **quick look** analyses are often **wrong**.*

D In correcting system deviations and failures, it is important that all the participants know not only **what** happened and how it happened, but **why** as well.*

P Failure reporting without a **close out** system is meaningless. (April Gillam 1989)

P Common, if undesirable, responses to **indeterminate outcomes** or failures:*

If it **ain't broke**, don't fix it.
Let's **wait and see** if it goes away or happens again.
It was just a **random** failure. One of those things.
Just treat the **symptom**. Worry about the cause later.
Fix everything that might have caused the problem.
Your **guess** is as good as mine.

D Chances for recovery from a **single failure** or flaw, even with complex consequences, are fairly good. Recovery from **two or more** independent failures is unlikely in real time and uncertain in any case.*

Rearchitecting, Evolving, Modifying, and Adapting

The test of a good architecture is that it will last.
The sound architecture is an enduring pattern.

[Robert Spinrad 1988]

P The team that created and built a presently successful product is often the best one for its evolution—but seldom for creating its replacement.

D If you **don't understand** the existing system, you can't be sure you're rearchitecting a **better** one. (Susan Ruth 1993)

P When implementing a change, keep some elements constant to provide an **anchor** point for people to cling to. (Jeffrey H. Schmidt 1993)

P In large, mature systems, **evolution** should be a process of **ingress** and **egress**. (IEEE 1992, Jeffrey Schmidt 1992)

P Before the change, it is your opinion. After the change it is your problem. (Jeffrey Schmidt 1992)

D Unless constrained, **rearchitecting** has a natural tendency to proceed unchecked until it results in a substantial transformation of the system. (Charles W. Sooter 1993)

D Given a change, if the anticipated actions don't occur, then there is probably an invisible barrier to be identified and overcome (Susan Ruth 1993).

EXERCISES

Exercise: What favorite heuristics, rules of thumb, facts of life, or just plain common sense do you apply to your own day-to-day living—at work, at home, at play? What heuristics have you heard on TV or the radio (for example, on talk radio, action TV, children's programs)? Which ones would you trust?

Exercise: Choose a system, product, or process with which you are familiar and assess it using the appropriate foregoing heuristics. What was the result? Which heuristics are or were particularly applicable? What further heuristics were suggested by the system chosen?

Were any of the heuristics clearly incorrect for this system? If so, why?

Exercise: Try to spot heuristics and insights in the technical literature. Some are easy; they are often listed as principles or rules. The more difficult ones are buried in the text but contain the essence of the article or state something of far broader application than the subject of the piece.

Exercise: Try to create a heuristic of your own—a guide to action, decision-making, or to instruction of others.

NOTES

1 The manufacturing, social, communication, software, management, business, and economics fields are particularly active in proposing and generating heuristics — though they usually are called principles, laws, rules, or axioms.

2 As indicated in the introduction to this appendix, an asterisk indicates that this heuristic is taken from Rechtin (1991). (With permission of Prentice Hall, Englewood Cliffs, New Jersey.)

3 See also Chapters 3 and 4.

REFERENCES

Hayes, R. H., S. C. Wheelwright, and K. B. Clark (1988). *Dynamic Manufacturing, Creating the Learning Organization*, New York: The Free Press.

Rechtin, E. (1991). *Systems Architecting, Creating and Building Complex Systems*, Englewood Cliffs, NJ: Prentice Hall. Note that throughout chapter, this reference will be referred to as Rechtin 1991.

Rechtin, E., Editor (1994). *Collection of Student Heuristics in Systems Architecting, 1988–1993*, Los Angeles, CA: University of Southern California (unpublished but available to students and researchers on request).

Appendix B
Reference Texts Suggested for Institutional Libraries

The following list of texts is offered as a brief guide to books that would be particularly appropriate to an architecting library.

SYSTEMS ARCHITECTING TEXTS

Rechtin, E. (1991). *Systems Architecting: Creating and Building Complex Systems*, Englewood Cliffs, NJ: Prentice Hall.

Crawley, E., B. Cameron, and D. Selva (2015). *System Architecture: Strategy and Product Development for Complex Systems*, Upper Saddle River, NJ: Prentice Hall Press.

Eisner, H. (2019). *Systems Architecting: Methods and Examples*, Boca Raton, FL: CRC Press.

ARCHITECTING BACKGROUND

Alexander, C. (1977). *A Pattern Language: Towns, Buildings, Construction*, New York: Oxford University Press.

Alexander, C. (1964). *Notes on the Synthesis of Form*, Cambridge, MA: Harvard University Press.

Alexander, C. (1979). *The Timeless Way of Building*, New York: Oxford University Press.

Kostoff, S. (1977). *The Architect*, New York: Oxford University Press, (paperback).

Lang, J. (1987). *Creating Architectural Theory*, New York: van Nostrand Rheinhold.

Rowe, P. G. (1987). *Design Theory*, Cambridge, MA: MIT Press.

Vitruvius (1960). *The Ten Books on Architecture*, Mineola, NY: Dover Publications (paperback). Translated by Morris Hicky Morgan.

MANAGEMENT

Augustine, N. R. (1982). Augustine's Laws, Reston, VA: AIAA, Inc.

Deal, T. E. and A. A. Kennedy (1988). *Corporate Cultures, The Rites and Rituals of Corporate Life*, Reading, MA: Addison-Wesley.

DeMarco, T. and T. Lister (1987). *Peopleware: Productive Projects and Teams*, New York: Dorset House.

Juran, J. M. (1988). *Juran on Planning for Quality*, New York: The Free Press.

Rechtin, E. (2017). *Systems Architecting of Organizations: Why Eagles Can't Swim*, Oxfordshire: Routledge.

MODELING

Eisner, H. (1988). *Computer Aided Systems Engineering*, Englewood Cliffs, NJ: Prentice Hall.

Friedenthal, S., A. Moore, and R. Steiner (2014). *A Practical Guide to SysML: The Systems Modeling Language*, Burlington, MA: Morgan Kaufmann.

Hatley, D., P. Hruschka, and I. Pirbhai (2013). *Process for System Architecture and Requirements Engineering*, Boston, MA: Addison-Wesley.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen (1991). *Object-Oriented Modeling and Design*, Upper Saddle River, NJ: Prentice Hall.

Ward, P. T. and S. J. Mellor (1985). *Structured Development for Real-Time Systems, Volume 1: Introduction and Tools*, New York: Yourdon Press (Prentice Hall).

SPECIALTY AREAS

Baudin, M. (1990). *Manufacturing Systems Analysis*, New York: Yourdon Press Computing Series.

Hammond, J. S., R. L. Keeney, and H. Raiffa (2002). *Smart Choices: A Practical Guide to Making Better Decisions*, New York: Broadway Books.

Hayes, R. H., S. C. Wheelwright, and K. B. Clark (1988). *Dynamic Manufacturing*, New York: The Free Press.

Keeney, R. L. (1992). *Value Focused Thinking*, Cambridge, MA: Harvard University Press.

Miller, J. G. (1978). *Living Systems*, New York: McGraw-Hill.

Simon, H. A. (1981). *Sciences of the Artificial*, Cambridge, MA: MIT Press.

Thome, B., Editor (1993*). Systems Engineering: Principles and Practice of Computer-Based Systems Engineering*, New York: John Wiley, Chichester, Wiley Series on Software Based Systems.

SOFTWARE

Boehm, B. (1981). *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice Hall.

Brooks, F. P. Jr. (1995). *The Mythical Man-Month, Essays on Software Engineering, 20th Anniversary Edition*, Reading MA: Addison-Wesley.

Deutsch, M. S. and R. R. Willis (1988). *Software Quality Engineering*, Upper Saddle River, NJ: Prentice Hall.

Gajski, D. D., V. M. Milutinovic, H. J. Siegel, and B. P. Furht (1987). *Computer Architecture*, Piscataway, NJ: The Computer Society of the IEEE (Tutorial).

Gamma, E. et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software Architecture*, Reading, MA: Addison-Wesley.

Shaw, M. and D. Garlan (1996). *Software Architecture: Perspectives on an Emerging Discipline*, Upper Saddle River, NJ: Prentice Hall.

Software Productivity Consortium (1991). ADARTS Guidebook, SPC-94040-CMC, Version 2.00.13, Vols. 1–2, September, 1991.

Yourdon, E. and L. L. Constantine (1979). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, New York: Yourdon Press.

SYSTEMS SCIENCES

Flood, R. L. and E. R. Carson (1988). *Dealing with Complexity, an Introduction to the Theory and Application of System Sciences*, New York: Plenum Press.

Genesereth, M. S. and N. J. Nilsson (1987). *Logical Foundations of Artificial Intelligence*, San Francisco, CA: Morgan Kaufmann.

Gerstein, D. R. et al., Editors (1988). *The Behavioral and Social Sciences, Achievements and Opportunities*, Washington, DC: National Academy Press.

Klir, G. J. (1985). *Architecture of Systems Problem Solving*, New York: Plenum Press.

SYSTEMS THINKING

Arbib, M. A. (1987). *Brains, Machines, and Mathematics*, 2nd edition, Heidelberg: Springer-Verlag.

Beam, W. R. (1990). *Systems Engineering, Architecture and Design*, New York: McGraw-Hill.

Boorstin, D. J. (1985). *The Discoverers*, New York: Vintage Books.

Boyes, J. L., Editor (1987). *Principles of Command and Control*, Fairfax, VA: AFCEA International Press.

Davis, S. M. (1987). *Future Perfect*, Reading, MA: Addison-Wesley.

Gause, D. C. and G. M. Weinberg (1989). *Exploring Requirements, Quality Before Design*, New York: Dorset House.

Hofstadter, D. R. (1980). *Gödel, Escher, Bach: An Eternal Golden Braid*, New York: Vintage Books.

Norman, D. A. (1988). *The Psychology of Everyday Things*, New York: Basic Books.

Pearl, J. (1984). *Heuristics*, Reading, MA: Addison-Wesley.

Rubinstein, M. F. (1975). *Patterns of Problem Solving*, Englewood Cliffs, NJ: Prentice Hall.

Weinberg, G. M. (1988). *Rethinking Systems Analysis and Design*, New York: Dorset House.

Appendix C
On Defining Architecture and Other Terms

This appendix is for those who need to come to a consensus in a group on a definition for architecture or other major terms used in this book rather than adopting definitions from current standards. Deciding on formal definitions is commonly part of setting up an official corporate training course or documenting a standard (public or corporate). In these situations, an inordinate amount of time can be spent arguing about fine details of definitions. The obvious approach is to use the definitions in widely accepted standards. In the past, there were not any, but as of this edition, the definitions in the ISO/IEC/IEEE 420xy standard series should probably be considered the default choices. Of course, the standards do not generally record the reasoning that brought them to a decision. This appendix records some of the definition-related discussions in which one of the authors (Maier) was involved in, partly for the standards and partly for other efforts. It is offered to help others who need to arrive at a group consensus on definitions, a ready-made set of choices and reasoning.

DEFINING "ARCHITECTURE"

One might think that, with 5,000 years of history, the notion of architecture in buildings would be clearly and crisply defined. Presumably, then, the definition could be extended to give a clear and crisp definition to architecture in other fields. However, this is not the case. A formal definition of architecture is elusive even in the case of buildings. And if the definition is elusive in its original domain, is it surprising that a wholly satisfactory definition is elusive in more general domains?

The communities involved in architecture in systems, software, hardware, and other domains have struggled to find a formal definition. Each group that has set out a formal definition has usually made a unique choice. The choices are often similar but reflect significantly different ideas. The sections to follow review some of the more distinctive choices. Of course, there are many small variations on each one.

To make sense of the different definitions, it is important to review them with some criteria in mind. In reviewing these definitions, try to answer the following questions with respect to each definition:

1. How does the definition establish what is the concern of the architect and what is not?
2. What is the purpose of the definition? Some purposes might be defining an element of design, education, organizational survival or politics, setting legal boundaries, or even humor.

3. Choose a building you are familiar with. What is its architecture, according to the definition? How well does the definition implied architecture match what you would expect to be the building architect's scope of work?
4. Choose a system you are familiar with. What is its architecture, according to the definition? What things are uniquely determined about the system from the application of that definition?
5. What is the architecture of the Internet, according to the definition?

Webster's Dictionary Definition

We begin with the dictionary's definition (Merriam-Webster's, Inc. 1995)

> Architecture: 1. The art or science of building; specifically, the art or practice of designing and building structures and esp. habitable ones. 2a. Formation or construction as or as if the result of conscious act <the ~ of the garden> b. a unifying or coherent form or structure <the novel lacks ~> 3. Architectural product or work 4. A method or style of building 5. The manner in which the components of a computer or computer system are organized and integrated.

The interesting part of this definition, for our purposes, is part 2. The first definition uses architecture in the sense of the profession, not what we are looking for here. This definition says to speak of the architecture of a thing is to speak of its "unifying or coherent form." Unfortunately, it is not obvious what aspect of form is "unifying or coherent." It is something that can be judged but is hard to define crisply. The civil building example suggests several other ideas about architecture:

1. Architecture is tied to the structure of components, but if a novel can have an architecture, the notion of components is relatively abstract. Components may need to be interpreted broadly in some contexts. No one would confuse the structure of a novel with its organization into chapters—which is the "packaging" of that structure and is analogous to confusing the architecture of a system with its module structure.
2. The distinction between an architectural level of description and some other level of design description is not crisp. Architectural description is concerned with unifying characteristics or style, and an engineering description is concerned with construction or acquisition.
3. In common use, "architecture" can mean a conceptual thing, the work of architects, and architectural products. Other definitions make sharper distinctions.

This Book

The definition of architecture given in the glossary of this book is as follows:

> **Architecture**: The structure—in terms of components, connections, and constraints—of a product, process, or element.

This definition is specific; it refers to structure (although that term is open to some interpretation). Components, connections, and constraints are descriptive terms for things that represent architecture and arguable architecture itself. And, we can discuss the architecture of a wide variety of things using this definition and its concepts. This book is primarily about architecting, rather than architecture. The reason is that the most important constraints come from the process of doing the architect's role. The most important things come from working with clients to understand purpose and limitations. Architecture should, by the tenets of this book, proceed from the client's needs rather than from a presupposed notion of what constitutes an architectural-level definition of a system.

IEEE Architecture Working Group (AWG)

After extended discussion in 1995–1996 in association with developing ANSI/IEEE 1471 Recommended Practice for Architectural Description of Software-Intensive Systems, the Institute of Electrical and Electronics Engineers (IEEE) Working Group chose the following definition:

> An **Architecture** is the highest-level concept of a system in its environment.

"System" in this definition refers back to the official IEEE definition, "a collection of components organized to accomplish a specific function or set of functions." This definition of architecture was intended to capture several ideas:

1. An architecture is a property of a thing or a concept, not a structure. The term "structure" is avoided specifically to avoid any connotation that architecture was solely a matter of physical structure. Concept, which is much more generic, is used instead.
2. The term "highest-level" is used to indicate that architecture is an abstraction and that it is a fundamental abstraction. A major defect of this definition is that highest level carries a connotation of levels of hierarchy, particularly a single hierarchy, which is exactly one of the connotations to be avoided. Also, "highest-level concept" leaves a great deal of room for interpretation.
3. The definition says that architecture is not a property of the system alone, but that the system's environment must be included in a definition of the system's architecture. This has often been referred to as "architecture in context" as opposed to "architecture as structure." It was there to capture the idea that architecture has to encompass purpose and the relationship of the system to its stakeholders. The reader must judge whether or not that interpretation is clear.

This definition was used in several drafts of the 1471 standard but was replaced in the final balloted version. The definition in the final balloted version was as follows:

> **Architecture:** the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.

This definition is a refinement of definitions from the software engineering community, as discussed below. Those who do not like it might be more inclined to say it was a compromise between conflicting points of view that suffers from the usual problems of a committee decision. The definition starts with the software community's definitions (discussed shortly) and then adds back some of the ideas of the original 1471 definition. The primary refinement is the de-emphasis on physical structure and to say that architecture is "embodied" in components, relationships, and principles. Put another way, the definition tries to recognize that, for most systems, most of the time, the architecture is in the arrangement of physical components and their relationships. But, sometimes, the fundamental organization is on a more abstract level.

INCOSE SAWG

The International Council on Systems Engineering (INCOSE) Systems Architecture Working Group (SAWG) adopted a definition for systems architecture. It could as well be read as a definition for "Architecture, of a system." It is as follows:

> Systems Architecture: The fundamental and unifying system structure defined in terms of system elements, interfaces, processes, constraints, and behaviors.

This definition borrows the core of the dictionary definition that architecture represents a fundamental, unifying, or essential structure. Exactly what constitutes fundamental, unifying, or essential is not easily defined. It is presumed that recognizing it is partially art and up to the participants. In this definition, the role of multiple aspects making up the architecture is made explicit through the listing of elements, interfaces, processes, constraints, and behaviors. This definition makes, or facilitates making, a sharper separation between architecture as a conceptual object, an architecture description as a concrete object, and the process or act of creating architectures (architecting).

MIL-STD-498

MIL-STD-498, now canceled, had a definition of architecture that specifically pertained to a designated development task.

> Architecture: The organizational structure of a system or CSCI, identifying its components, their interfaces, and a concept of execution among them.

Here, architecture is described specifically in three parts: components, interfaces, and a concept of execution. In this sense, it supports the idea of architecture as inherently multiview, although it specifically defines the views where others leave them open. The meaning of "organizational structure" as opposed to some other structure (conceptual, implementation, detailed, and so forth) is not made clear, although the idea is congruent with the common usage of architecture. It also uses "concept" within the definition, but only in referring to execution. Like most definitions, it does not clearly make a distinction between architectural and design concerns.

This definition is also "structuralist" in the sense that it emphasizes the structure of the system rather than its purposes or other relationships. One could interpret the definition to mean that the architect was not concerned with the system's purpose and that architecture came after requirements were fully defined. In fact, that is exactly the interpretation it should be given, at least in the way the associated standards envisioned the systems engineering process executing.

The original IEEE definition (in IEEE 610.12-1990) is a shorter version of this:

The organizational structure of a system or component.

Perry-Garlan

A widely used definition in the software community is due to Perry and Garlan, although the exact place it first appeared is somewhat obscure.

The structure of the components of a system, their interrelationships, and principles and guidelines governing their design and evolution over time.

An almost identical definition is used as the definition of architecture in the U.S. DoD C4ISR Architecture Framework, where it is incorrectly credited to the IEEE 610.12 standard for terminology. This definition is another three-part specification: components, interrelationships, and principles–guidelines. As this definition is commonly used, components and interrelationships usually refer to physically identifiable elements. This definition is mostly used in the software architecture community, where it is common to see components identified as code units, classes, packages, tasks, and other code abstractions. The interrelationships would be calls or lines of inheritance.

The two basic objections to this definition are that it implies (if primarily through use rather than the words) that architecture is the same as physical structure and that it makes no distinction in level of abstraction. The common usage of architecture is in reference to abstract properties of things, not to the details. The Perry-Garlan definition can presumably apply to the structure of components at any level of abstraction. Although applicability to multiple levels is, in part, desirable, it is also desirable to distinguish between what constitutes an architectural-level description (whether of a whole system or a component) from descriptions at lower levels of abstraction.

Maier's No-Longer-Tongue-in-Cheek Rule-of-Thumb

A slightly flip but illustrative way of defining architecture is to go back to what architects are supposed to do.

An **Architecture** is the set of information that defines a systems value, cost, and risk sufficiently for the purposes of the systems sponsor.

Obviously, this definition reflects the issue back to architecting, where the definition of architecture reflects back to architecture. The point of this definition is that architecture is what architects produce and that what architects do is help clients make decisions about building systems. When the client makes acquisition decisions, architecture has been done (perhaps unconsciously, and perhaps very badly, but done). With some adjustment, the idea persists, including in this book, no-longer-tongue-in-cheek, but with a realization of the insight. This definition reflects the key insight early in the book that architecture is in decisions, specifically the relatively small set of decisions that drive value, cost, and risk. A core justification for the architecture concept is that design decisions have differential impact. Some have a large impact on value, cost, or risk; others have a low impact. Identifying which ones have large impact and putting the requisite attention on them is important to ultimate success.

Internet Architecture Discussion

One of the questions given at the beginning was, "What is the architecture of the Internet?" The point of the question is that no reasonable notion of unifying, organizing, or coherent form will produce a physical description of the Internet. The specific pattern of physical links is continuously changing and of little interest. However, there is a very clear unifying structure, but it is a structure in protocols. It is not even a structure in software components, as exactly what software components implement the protocols is not known even to the participating elements. The point about protocols being the organizing structure of the Internet, and in particular the Internet Protocol (IP), was made in Chapter 7 and Figure 7.1.

ISO/IEC/IEEE 420xy Standards

The ISO/IEC/IEEE 420xy series, beginning with 42010, grew out of ANSI/IEEE 1471. Unsurprisingly, it builds on the 1471 definitions. The 420xy series has been in active development since the early 2000s and several of the standards have been through revision cycles. Significant content has been added since the original releases. The current working definition of architecture in the series is:

> **Architecture:** Fundamental and enduring concepts and properties embodied in an entity and principles governing its life cycle.
>
> *ISO (2024)*

Both "entity" and "life cycle" are terms with their own definitions. The heritage to 1471 is obvious, as are the differences. "Entity" replaces systems because the 420xy series has evolved to address named things, such as systems-of-systems and enterprises, other than systems. The architecture of systems is viewed as somewhat of a specialty against the more general challenge of architecting other "entities." There is a strong constituency of concern on enterprises, in particular, so the terminology is generalized.

Summary

Those who must choose definitions have a lot to work with, probably more than they would want. The precise form of the definition is less important than the background of what architecture should be about. What architecting should be was discussed at length in Chapter 1. The specifics of what architects will produce—that is, what an architecture actually looks like—will differ from domain to domain. Ideally, the definition of a given organization should come from that knowledge—the knowledge of what is needed to successfully define a system concept and take it through development. If the organization has that knowledge, it should be able to choose a formal definition that encapsulates it. If the organization does not have that knowledge, then no formal definition will produce it.

Models, Viewpoints, and Views

The terms model, view, and viewpoint are important in setting architecture description standards or architecture frameworks using the community terminology of Chapter 11. The meaning of these terms changes from standard to standard. The discussion below is intended to capture an argument for a distinction between view and viewpoint. The distinction is mainly useful in writing standards and is important in understanding some of the current standards and the conventions used in some MBSE tools.

Why do we need some organizing abstraction beyond just models? Experience teaches that particular collections of models are logically related by the kinds of issues or concerns they address. The idea of a view comes from architectural drawings. In a drawing, we talk about the top view or the side view of an object in referring to its physical representation as seen from a point. A view is the representation of a system from a particular point or perspective. In this usage, a view is a representation of the whole system with respect to the perspective. A viewpoint is a perspective, an abstraction of many related views. It is the idea of viewing something from "the front," for example.

A view need not correspond to physical appearance. A functional view is a representation of a system abstracting away all nonfunctional or nonbehavioral details. A cutaway view shows some mixture of internal and external physical features in a mixture defined by the illustrator.

A view can be thought of both projectively and constructively. In the projective sense, a view is formed by taking the system and abstracting away all the details unnecessary to the view. It is analogous to taking a multidimensional object and projecting it onto a lower-dimensional space (like a viewing plane). So, for example, a behavioral view is the system pared down to only its behaviors—the set of input-to-output traces.

In the constructive sense, we build a complete model of the system by building a series of views. Each represents the system from one perspective, and with enough, the system should be "completely" defined. It is like sketching a front view, a side view, and a top view and then inferring the structure of the whole object from the union of those views. In more general systems, we might build a functional view, then a physical view, then a data view, then return to the functional view, and so forth, until a complete model is formed from the joint set of views.

In practice, it usually takes several models to represent that whole system relative to typical concerns, at least for high-technology systems. So, a view is usually a collection of models. For example, physical representation seems simple enough, but how many different models are needed to represent the components of an information-intensive system? A complete physical view might need conventional block diagrams of information flow, block diagrams of communication interconnection, facilities layouts, and software component diagrams.

Viewpoints are motivated by noticing that we build similar views, using similar methods, for many systems. By analogy, we will want to draw a top view of most systems we build. The civil architect always draws a set of elevations, and elevation drawings share common rules and structures. And, an information systems architect will build information models using standard methods for each system. This similarity is because related systems will typically have similar stakeholders, and these stakeholders find their concerns consistently addressed by particular types of models and analysis methods. Hence, a viewpoint can be thought of as a set of modeling or analysis methods together with the concerns those methods address and the stakeholders possessing those concerns.

Working Definitions

These are summarizing definitions, augmented with the notions of consistency and completeness.

Model: An approximation, representation, or idealization of selected aspects of the structure, behavior, operation, or other characteristics of a real-world process, concept, or system (IEEE 610.12-1990).
View: A representation of a system from the perspective of related concerns or issues (IEEE 1471-2000).
Viewpoint: A template, pattern, or specification for constructing a view (IEEE 1471-2000).
Consistency, of Views: Two or more views are consistent if at least one realized system can exist that possesses the given views.
Completeness, of View: A set of views is complete if they satisfy (or "cover") all of the concerns of all stakeholders of interest.

In the evolved standards the definitions have changed somewhat. The definition of "architecture view" shifted significantly in wording, if not so much in meaning, in ISO/IEC/IEEE 42010, at least as of the 2022 version. It shifted again in the 2024 committee draft of ISO/IEC/IEEE 42024, a compendium standard on "architecture fundamentals." The definition of viewpoint has been relatively stable. For the purposes of this book, we use the terms as above.

Consistency and Completeness

Given multiple views (like top, front, and side) of a physical object, the ideas of consistency and completeness are intuitively clear. A set of views is consistent if they are abstractions of the same object. A little more generally, they are consistent if at least one real object exists that has the given views. Consistency for physical objects and views can be checked through solid geometry. Figure C.1 illustrates the point. The views are consistent if the geometrical object produces them when projected onto the appropriate subspace. Even without the actual object, we can perform geometric checks on the different views.

We cannot (yet) treat consistency in the same rigorous manner if the views are functional and physical or something else more abstract and are of a complex system rather than a geometric object. As we employ more complex views, it is useful to return to the heuristic notion of consistency. Given a few models of a system being architected, we say they are consistent if at least one implementation exists that has the models as abstractions of itself.

Completeness can also be heuristically understood through the geometric analogy. Suppose we have a set of visual representations of a material object. What does it mean to claim that the set of representations (views) is "complete"? Logically, it means that the views completely define the object. However, any set of external visual representations can only define the external shape of the object; it cannot define the internal structure, if any. This trivial observation is actually extremely important for understanding architecture. No set of representations is ever truly complete. A set of

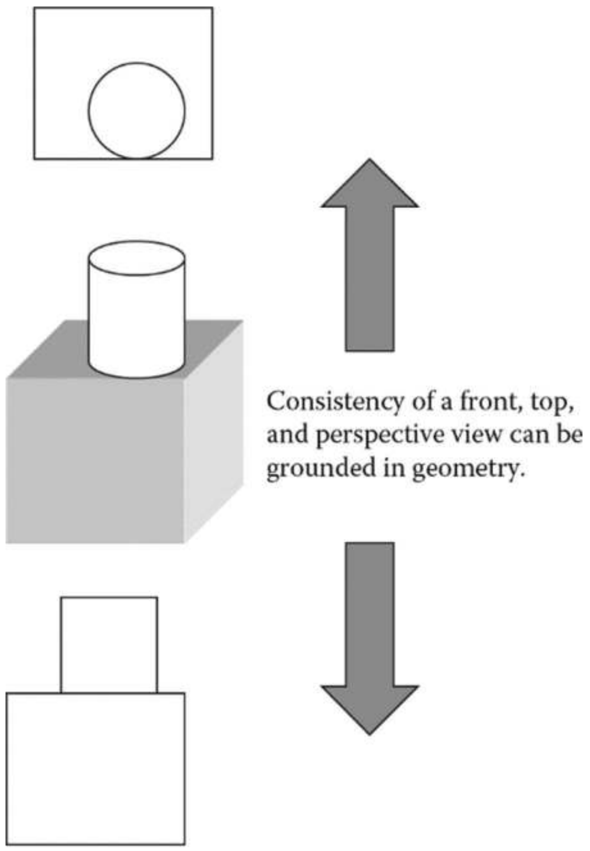

Consistency of a front, top, and perspective view can be grounded in geometry.

FIGURE C.1 A geometric illustration of the concept of consistency in views.

representations can be complete only with respect to something, say with respect to some set of concerns. If the concerns are external shape, then some set of external visual representations can be complete. If the concerns are extended to include internal structures, or strength properties, or weight, or any number of other things, then the set of views must likewise be extended to be "complete."

REFERENCES

ISO (2024). *ISO/IEC/IEEE FDIS 42024 Architecture Fundamentals*, New York: IEEE.

Merriam Webster's, Inc. (1995) *Merriam Webster's Collegiate Dictionary*, 10th edition, Springfield, MA: Merriam Webster's, Inc, p. 61.

Glossary

The fields of systems engineering and systems architecting are sufficiently new that many terms have not yet been standardized. Common usage is often different among different groups and in different contexts. However, for the purposes of this book, the meanings of the following terms are as follows.

Abstraction: A representation in terms of presumed essentials, with a corresponding suppression of the nonessential.

ADARTS: Ada-Based Design Approach for Real-Time Systems. A software development method (including models, processes, and heuristics) developed and promoted by the Software Productivity Consortium. Now of historical interest, the integration of models and heuristics is exemplary for MBSE-like methods applied to software.

Aggregation: The gathering together of closely related elements, purposes, or functions.

Architecting: The process of creating and building architectures. Depending on one's perspective, architecting may or may not be seen as a separable part of engineering. Seen as part of systems development, it comprises those aspects most concerned with conceptualization, objective definition, and certification for use.

Architectural style: A form or pattern of design with a shared vocabulary of design idioms and rules for using them (see Shaw and Garlan 1996, page 19).

Architecture: The structure—in terms of components, connections, and constraints—of a product, process, or element.

Architecture, open: An architecture designed to facilitate addition, extension, or adaptation for use.

Architecture (communication, software, or hardware): The architecture of the particular designated aspect of a large system.

ARPANET/INTERNET: The global computer internetwork, principally based on the TCP/IP packet communications protocol. The ARPANET was the original prototype of the current INTERNET.

Certification: A formal, but not necessarily mathematical, statement that defined system properties or requirements that have been met.

Client: The individual or organization that pays the bills. May or may not be the user. Used interchangeable with sponsor for architecture studies.

Complexity: A measure of the numbers, types, and strengths of interrelationships among system elements. Generally speaking, the more complex a system, the more difficult it is to design, build, and use.

Deductive reasoning: Proceeding from an established principle to its application.

Design: The detailed formulation of the plans or instructions for making a defined system element; a follow-on step to systems architecting and engineering.

Domain: A recognized field of activity and expertise or of specialized theory and application.

Engineering: Creating cost-effective solutions to practical problems by applying scientific knowledge to building things in the service of mankind (Shaw and Garlan 1996, page 6). May or may not include the art of architecting.

Engineering, concurrent: Narrowly defined (here) as the process by which product designers and manufacturing process engineers work together and simultaneously to create a manufacturable product.

Heuristic: A guideline for architecting, engineering, or design. Lessons learned are expressed as a guideline. A natural language abstraction of experience that passes the tests of Chapter 2.

Heuristic, descriptive: A heuristic that describes a situation.

Heuristic, prescriptive: A heuristic that prescribes a course of action.

ISO/IEC/IEEE 15288: A joint standard for how systems are engineered and developed.

ISO/IEEE/IEEE 42010, 42020, 42030: A series of standards defining architecture description, processes, and evaluation. Part of a growing family of standards on architecture activities.

Inductive reasoning: Extrapolating the results of examples to a more general principle.

Manufacturing, flexible: Creating different products on demand using the same manufacturing line. In practice, all products on that line come from the same family.

Manufacturing, lean: An efficient and cost-effective manufacturing or production system based on ultraquality and feedback. (see Womack et al. 1990.)

MBTI: Myers–Briggs Type Indicator. A psychological test for indicating the temperaments associated with selected classes of problem-solving. (see Meyers et al. 1989.)

Metaphor: A description of an object or system using the terminology and properties of another. For example, the desktop metaphor for computerized document processing.

MIL-STD: Standards for defense system acquisition and development.

Model: An abstracted representation of some aspect of a system.

Model, satisfaction: A model that predicts the performance of a system in language relevant to the client.

Modeling: Creating and using abstracted representations of actual systems, devices, attributes, processes, or software.

Models, integrated: A set of models, representing different views, forming a consistent representation of the whole system.

Normative method: A design or architectural method based on "what should be"—that is, on a predetermined definition of success.

OMT: Object Modeling Technique. An object-oriented software development method. (see Rumbaugh et al., 1991.) An important ancestor to many current object-oriented methods.

Objectives: Client needs and goals, with a minimal imposed structure of an object and a direction of preference. Often, but not always, have measures.

Paradigm: A scheme of things, a defining set of principles, a way of looking at an activity, for example, classical architecting.

Participative method: A design method based on the wide participation of interested parties. Designing through a group process.

Partitioning: The dividing up of a system into subsystems.

Progressive design: The concept of a continuing succession of design activities throughout product or process development. The succession progressively reduces the abstraction of the system through models until physical implementation is reached and the system used.

Purpose: A reason for building a system.

Rational method: A design method based on deduction from the principles of mathematics and science.

Requirement: An objective regarded by the client with a binary measure of satisfaction — that is, either passed or not.

Scoping: Sizing; defining the boundaries and defining the constraints of a process, product, or project.

Spiral: A model of system development that repeatedly cycles from function to form, build, test, and back to function. Originally proposed as a risk-driven process, particularly applicable to software development with multiple release cycles.

System: A collection of things or elements that, working together, produce a result not achievable by the things alone.

Systems, builder-architected: Systems architected by their builders, generally without a committed client.

Systems, feedback: Systems that are strongly affected by feedback of the output to the input.

Systems, form first: Systems that begin development with a defined form (or architecture) instead of a defined purpose. Typical of builder-architected systems.

Systems, politicotechnical: Technological systems, the development and use of which are strongly influenced by the political processes of government.

Systems, sociotechnical: Technological systems, the development and use of which are strongly affected by diverse social groups and social processes. Systems in which social considerations equal or exceed technical ones.

Systems architecting: The art and science of creating and building complex systems. That part of systems development is most concerned with scoping, structuring, and certification.

Systems architecting, the art of: That part of systems architecting based on qualitative heuristic principles and techniques—that is, on lessons learned, value judgments, and unmeasurables.

Systems architecting, the science of: That part of systems architecting based on quantitative analytic techniques — that is, on mathematics and science and measurables.

Systems engineering: A multidisciplinary or transdisciplinary engineering discipline in which decisions and designs are based on their effect on the system as a whole.

Technical decisions: Architectural decisions based on engineering considerations.

Ultraquality: Quality so high that measuring it directly in time and at a reasonable cost is impractical. (see Rechtin 1991, Chapter 8.) It is not possible to gather sufficient statistics to verify that the criteria are met.

Value judgments: Conclusions based on worth (to the client and other stakeholders).

View: A perspective on a system describing some related set of attributes, typically a related set of stakeholder concerns. A view is represented by one or more models.

Waterfall: A development model based on a single sequence of steps leading to a single delivery of the completed system. Typically applied to the making of major hardware elements.

Zero defects: A production technique based on the objective of making everything perfect. Related to the "everyone a supplier, everyone a customer" technique for eliminating defects at the source. Contrasts with acceptable quality limits in which defects are accepted, providing they do not exceed specified limits in number or performance.

Subject Index