O'REILLY®



Domain-Driven Transformation

Modernize Legacy Systems and Mitigate Risk

Carola Lilienthal &
Henning Schwentner
Foreword by Michael Feathers

Praise for *Domain-Driven Transformation*

No one ever said reworking complex legacy systems is easy. But where should you start? These authors have answers. This book is packed full of practical advice on how to assess the current state of your legacy systems and options for transforming tangled legacy systems (and the organizations that support them) into more sustainable systems using the values, principles, and practices of modern Domain-Driven Design.

—Rebecca Wirfs-Brock, object-design pioneer, author, and inventor of Responsibility-Driven Design (RDD) who by accident started the xDD meme

I really enjoyed this book! It is the best DDD book on the market.

—Sergio Morazán, senior technical product owner, OBI Group Holding

This book goes way beyond the surface of DDD and software architecture. It presents a holistic and well-structured approach toward modernizing legacy systems in a domain-driven manner.

—Michael Plöd, independent tech consultant

Domain-Driven Transformation provides a step-by-step guide to help you regain control of the complexity of your legacy system. It fills a missing piece in our Domain-Driven Design literature by focusing on how to apply DDD principles to transform legacy systems.

—Gien Verschatse, consultant and trainer, coauthor of Collaborative Software Design

This book is a practical guide to modernizing legacy systems with clarity and purpose. Carola and Henning guide readers from foundational principles to strategic decision-making, showing how Domain-Driven Design (DDD) can illuminate the path to sustainable modernization. Their emphasis on domain modeling as a lens for risk mitigation and architectural alignment makes this an essential read for those transforming and modernizing legacy systems.

—Joseph (Joe) Yoder, author of Cloud Application Architecture Patterns, CEO of The Refactory, coined the term "Big Ball of Mud"

Domain-Driven Transformation delivers a powerful and practical approach to incremental legacy modernization. It brilliantly bridges the gap between current and target architectures by connecting domain discovery, bounded contexts, and the Modularity Maturity Index into actionable transformation paths.

—Susanne Kaiser, tech consultant and author of Architecture for Flow

Modernization of a system is hard, whether you are working from a big ball of mud or a technology that's going out of support. Where other books tell you to rewrite or replace, Domain-Driven Transformation shows you how to reshape—incrementally, safely, and in alignment with your business domain. The Domain-Driven Transformation method recognizes that software transformation is as much about people and context as it is about code.

—Jacqui Read, author of Communication Patterns, Chief Architect at The Ministry of Software Design, co-organizer of DDD London

If you are starting a transformation project, I recommend you read this book!

—Michael Feathers, author of Working Effectively with Legacy Code

The real test of a software engineer isn't building greenfield systems, but untangling the legacy ones. This book arms you with the strategy and tactics needed to ace the ultimate test of software engineering.

—Vlad Khononov, author of Learning Domain-Driven Design and Balancing Coupling in Software Design

Despite all the recent advances, most organizations will continue to deal with legacy systems. The common challenge to many of us is figuring out how to deliver business value while making the legacy system more loosely coupled and maintainable. Carola and Henning describe actionable advice that will help you achieve this!

—Mufrid Krilic, Domain-Driven Design coach, CoWork

Domain-Driven Transformation is a rare blend of deep insight and real-world guidance that makes the daunting task of modernizing legacy systems feel achievable. Carola Lilienthal and Henning Schwentner draw from years of hands-on experience to show not just how to refactor code, but how to rebuild understanding, collaboration, and trust around complex software. It's an honest, inspiring guide for anyone determined to bring life and clarity back to aging systems.

—Philipp Kostyra, Developer Experience, SAP

Transforming your legacy systems is more than a technology upgrade. Baked into your legacy system is an out-of-date view of how the business works—words, concepts, and relationships that no longer match how domain experts think about your business. To get anywhere near the full benefits of a legacy transformation, you need to reverse engineer what's in the legacy system and facilitate the design of a new domain model. This will then lead you to the optimal architectural boundaries, design of your code, and team organization which you need to implement. Doing all of this well requires a whole range of skills, and you can find them all in this fantastic book.

—Nick Tune, Principal Consultant at Empathy Software and coauthor of Architecture Modernization and Patterns, Principles, and Practices of Domain-Driven Design

Domain-Driven Transformation

Modernize Legacy Systems and Mitigate Risk

Carola Lilienthal and Henning Schwentner
Foreword by Michael Feathers



Domain-Driven Transformation

by Carola Lilienthal and Henning Schwentner

Copyright © 2026 Carola Lilienthal and Henning Schwentner. All rights reserved.

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Louise Corrigan

Development Editor: Michele Cronin

Production Editor: Beth Kelly

Copyeditor: Sonia Saruba

Proofreader: Krsta Technology Solutions

Indexer: Potomac Indexing, LLC

Cover Designer: Susan Brown

Cover Illustrator: Monica Kamsvaag

Interior Designer: David Futato

Interior Illustrator: Kate Dullea

November 2025: First Edition

Revision History for the First Edition

- 2025-11-10: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9798341640122 for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Domain-Driven Transformation*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher’s views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

979-8-341-64012-2

[LSI]

Foreword

Before I wrote *Working Effectively with Legacy Code* [Feathers 2004] 20 years ago, I spent a lot of time looking for books on legacy code and legacy transformation. The few books that I found were centered around mainframe software. At that time, moving code from "the mainframe" to something else was seen as an urgent task. It still is.

Today, we move code from "the monolith" to microservices. Time will tell whether our major industry initiatives will continue to start with an "M." Regardless, we will always need to modernize. Software goes stale when it falls out of sync with its environment—the business, the system's users, and its technology. We will always need to move forward.

I was glad to see Carola and Henning start this book because I've long felt that Domain-Driven Design is one of the most valuable distillations of software design knowledge available to us. It gave us a set of organizing principles and a nomenclature that makes design conversation easier. Carola and Henning use it as a base, and they've added considerable knowledge of social-technical design to give us a guidebook that covers the breadth of transformation and the considerations that you must keep in mind when undertaking that task.

If you are starting a transformation project, I recommend that you read this book.

Michael Feathers
***Author of* Working Effectively with Legacy Code**

Preface

"All happy families are alike; each unhappy family is unhappy in its own way." This *Anna Karenina* principle applies not only to families, but also to legacy software development teams. Different software systems suffer from different "diseases," and from this comes the need for a variety of remedies to cure them all. We have observed the following diseases in various combinations in legacy systems—whether monoliths or microservices—and their organizations in recent years:

- The legacy system has become a *big ball of mud*[1] because no one controlled the dependencies and now everything is connected to everything else (Chapter 5).
- The domain knowledge is entangled into one huge domain model whose parts fit together only partially or even contradict each other (unbounded models in Chapter 12).
- Business source code and technical source code are mixed, and thus the replacement of obsolete technology or a domain extension becomes a Herculean task (Chapter 8).
- People are organized in a team structure that is unsuitable for making fast progress (Chapter 9).

On the cover of this book you can see an Atlas moth. Just like the legacy systems in this book, it has begun as something small (an egg), then developed into something bulky and clumsy (a caterpillar) and further into a big ball (a silken cocoon), and finally transforms into something beautiful and spreads its wings to fly into a promising future. Another similarity between insect and legacy system is the size; the Atlas moth is one of the largest moth species in the world with a wingspan of 9–10 inches.

In this book, we present the treatments for these diseases that we have learned over the past 20 years working with *Domain-Driven Design (DDD)* and *legacy software*. These include: *refactorings*, *Domain Storytelling*, *EventStorming*, *Team Topologies*, and the *Modularity Maturity Index (MMI)*.

When we approach the healing in our own projects and with our customers, the respective development team looks significantly more positively into the future and motivation increases. The further the healing progresses, the happier users, project leaders, and managers will also be because the stodgy and expensive legacy software becomes more stable, can be adapted more quickly, and ultimately even allows innovative, forward-looking enhancements (again).

> **NOTE**
>
> Our goal with this book is to provide you with the means to help you divide your software systems so that changes have as local an impact as possible. In this way, the individual software modules again become manageable and understandable for the development teams and can be processed by correspondingly small and powerful teams.

That is why you will find a number of source code examples in this book. The legacy systems we encounter are built in very different programming languages:[2] Java, C#, PHP, C++, and many others. We have chosen to write the examples in the language that we encounter most frequently: Java. The examples are short enough that they should be understandable to programmers of other languages, and the procedure we describe is independent of the programming language.

Some readers may have read our other books, *Sustainable Software Architecture* [Lilienthal 2019] and *Domain Storytelling* [Hofer and Schwentner 2022]. The book in front of you is for us the union and logical consequence of the other two books and our daily work in our projects. Readers of these two books will recognize the example domains of cinema and car leasing and our view on architecture. Beyond that, there is a lot of

new stuff to discover :-) We sincerely hope you enjoy this synthesis and look forward to your feedback!

Structure of This Book

The book consists of an introduction, four parts, and a catalog of refactorings.

In the "Introduction: Domain-Driven Transformation in a Nutshell" (Chapter 1) we apply the method to our first example system and give a brief overview of the method.

Part I, "Fundamentals of Domain-Driven Transformation," begins with an introductory chapter on complexity in software development (Chapter 2). In the three chapters that follow, we summarize fundamental knowledge that is important for understanding this book: Domain-Driven Design (Chapter 3), Collaborative Modeling (Chapter 4), and architecture concepts (Chapter 5). From these three chapters, you should read all the sections in which you do not feel confident. In the final chapter of Part I, we discuss the question of the correct approach (Chapter 6).

Part II, "Technical, Tactical and Team-Organizational Domain-Driven Transformation," shows how to improve the legacy system's basis, source code, and developer organization. Generally, it pays to stabilize the legacy system technically (Chapter 7) and to strengthen the source code with domain knowledge through tactical transformation (Chapter 8). And finally, the development team should have clarity about how the team structures should change when embarking on the transformation of the legacy system (Chapter 9).

Part III, "Strategic Domain-Driven Transformation," presents the decomposition approach we have developed. The approach consists of four steps. It begins with rediscovering the domain (Chapter 10), continues with modeling the domain-driven target architecture (Chapter 11) and comparing the actual and target architecture (Chapter 12), and ends with prioritizing and executing the transformation measures (Chapter 13).

Part IV, “Conclusion,” contains two chapters. In Chapter 14, we present our ideas on what domain patterns exist and how each domain pattern affects Domain-Driven Transformation. In Chapter 15, we wrap up both the method and the book.

The appendices gather the *domain-driven refactorings* that are referenced in this book. This is an excerpt of the catalog at *https://hschwentner.io/domain-driven-refactorings*, where the refactorings that are needed to perform a transformation are collected online. We give specific references to the particular refactoring to be used at various points in the book. The refactorings we are referencing in the book are included in the appendices, so you don’t have to switch media if you want to look them up. The *catalog* is introduced in an overview (Appendix A) and split into four categories: “Strategic Refactorings” (Appendix B) help with splitting a monolith architecture into bounded contexts; “Tactical Refactorings that Support Strategic Refactorings” (Appendix C) are the smaller steps that the strategic refactorings are built from; “Sociotechnical Refactorings” (Appendix D) help reorganize the teams; and “Tactical Refactorings that Strengthen Domain Knowledge” (Appendix E) change the inner implementation of a bounded context.

AI-Aided Transformation

While we were writing this book, what might be called the AI revolution happened. Large language models (LLMs) became available that change many areas of life, and programming is certainly one of them.

An LLM obviously works with language, and legacy software is written in language, so it is natural to use the former to help with transforming the latter. Currently, what has been done is experimental but very promising. In this book we show at different points where AI can support the journey through legacy software modernization.

The development strides on rapidly, and by the time you’re reading this, you might smile and think, “ah, those were the days,” when we tell you that the developer tools of the time were GitHub Copilot and JetBrains AI

Assistant, and the then current models were OpenAI GPT-4o, Google Gemini 2.0 Flash, and Anthropic Claude 3.7 Sonnet. By the time you're reading this, the current LLMs have probably read this book already and will support you even better.

Conventions Used in This Book

The following typographical conventions are used in this book:

italic

Indicates new terms, URLs, refactorings, and patterns that we define. The latter two are additionally capitalized, like *Extract Bounded Context* or *Pipeline*. Also used to emphasize that we're referring to the special software development meaning of a term, e.g., *subdomain*.

"surrounded by quotation marks"

Indicates words from domain language used in case studies, like "ship-to-shore crane."

`constant width`

Used for program listings, as well as within paragraphs to refer to program elements like the class `Contract`.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

We provide four case studies and several field stories. Whenever we refer to the case studies, we do this in sidebars that start with a black bar that contains the name of the case study to remind you of the context. The field stories are told in sidebars with a gray background.

O’Reilly Online Learning

> **NOTE**
>
> For more than 40 years, *O’Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O’Reilly’s online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.

141 Stony Circle, Suite 195

Santa Rosa, CA 95401

800-889-8969 (in the United States or Canada)

707-827-7019 (international or local)

707-829-0104 (fax)

support@oreilly.com

https://oreilly.com/about/contact.html

We have a web page for this book, where we list errata and any additional information. You can access this page at *https://oreil.ly/domain-driven-transformation*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Email Carola Lilienthal: cl@wps.de.

Email Henning Schwentner: hs@wps.de.

Visit Henning Schwentner's website: *https://hschwentner.io*.

Visit the book's companion website: *https://domain-driven-transformation.com*.

Acknowledgments

We would like to take this opportunity to thank all those who have supported us over the past few years, given us ideas for this book, and discussed them with us. The very active and committed DDD community and our company, WPS – Workplace Solutions, should be mentioned here

first. Without a good environment, we would not be able to muster the energy it takes to write a book. Many of the ideas presented here originated with Christiane Floyd and Heinz Züllighoven. Thank you for letting us learn from you!

We are eternally grateful to our reviewers Alberto Brandolini, James Higginbotham, Stefan Hofer, Vlad Khononov, Mufrid Krilic, Sönke Magnussen, Michael Plöd, Nick Tune, Vaughn Vernon, Rebecca Wirfs-Brock, and Eoin Woods. They helped us to achieve a much better result with their critical, constructive comments.

Our guest author Jörn Koch contributed an awesome section on his Scenario Casting method. Alberto Brandolini passionately reviewed and extended the section on EventStorming. Thank you, Jörn and Alberto!

Our families have accompanied our focus on another book sympathetically and have given us space and support. Henning thanks his wife Julia and his children Lennart, Joris, Friedo, Tara, Aurelia, and Viola for their great support. Carola thanks her husband, her child, and her friends for just everything. Many thanks to all of you!

---

1 This term was coined by Brian Foote and Joe Yoder (see [Foote and Yoder 1997]).

2 Programmers love to talk shop about which programming language is the best. When it comes to building monolithic legacy software, there's a clear tie: you can do it with all of them.

Chapter 1. Introduction: Domain-Driven Transformation in a Nutshell

When you start a software project on a greenfield site, everything is fun and easy. Developers respond to new requirements at a rapid pace. Users are enthusiastic. Development moves forward in noticeable strides.

Over the lifetime of the system, this will change. Inevitably, the complexity of the software implementation increases. This increasing complexity leads to higher proneness to errors, slower and slower progress, and poorer maintainability. Eventually, even the smallest change takes half a year to reach production. The blooming greenfield has turned into a muddy, stinking brownfield. “Legacy system,” “ancient software,” “big ball of mud,” “monolith,” and “rubber boot project” are the unflattering names given to these kinds of systems.

But there is hope! Even these aging systems can regain flexibility, robustness, and development speed. This chapter will first give a concrete example of a legacy system that is modernized with Domain-Driven Transformation and then describe the method in an overview. This serves both to give you a general idea of the approach and to find out which of the many topics in the book are of deeper interest for you, and which chapters to read.

Example: A Legacy System in a Container Port

Henning’s great-grandfather was a cocoa trader. From his office at the port, he watched the giant ocean steamers rolling in and out. When he opened his

window, he could smell the big wide world: ripening bananas, roasting coffee, sometimes even wild animals. At that time, around 1930, ships were unloaded by dockworkers—men who carried the cargo in bags or boxes on their backs from the ships into large warehouses.

With the invention of the container in the 1960s, work in the port changed drastically. Warehouses were abandoned and replaced by container terminals. The dockworkers were reeducated so that they could drive the vehicles that were now used to transport the containers and thus the cargo on the terminal.

In Figure 1-1 you can see a sketch of the individual parts of a container terminal. The containers are unloaded from the container ship using a ship-to-shore crane, transported from the quay to the yard, and then moved to a truck or a freight train using a straddle carrier. Containers delivered to the container terminal by truck or freight train for transport by container ship take the opposite route.

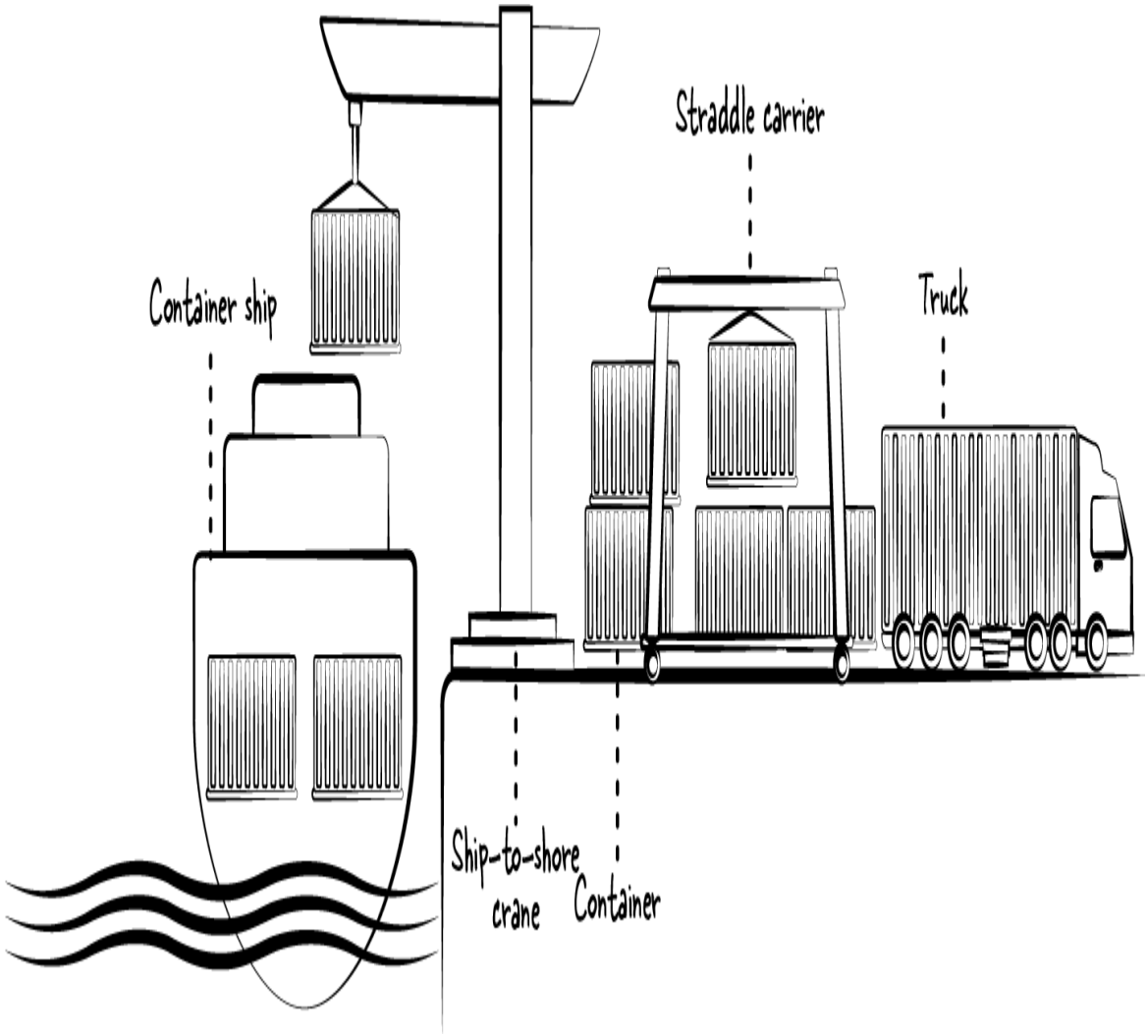


Figure 1-1. The container terminal—schematic.

In the 1970s/80s, container terminals began to use software to keep the resulting data under control and to automate the processes step-by-step. Today, the case study container terminal has an IT landscape with a variety of software systems: systems to plan the loading and unloading of container ships, systems to organize the arrival and delivery of containers at the truck gate and, of course, systems to manage the yard and organize transports in the yard.

Often, several of these functions are implemented in one system to which other peripheral systems are connected. This central system is called a terminal operating system (TOS). For the sake of simplicity,[1] we will only

refer to the TOS in the following example and omit the other possible systems (see Figure 1-2).

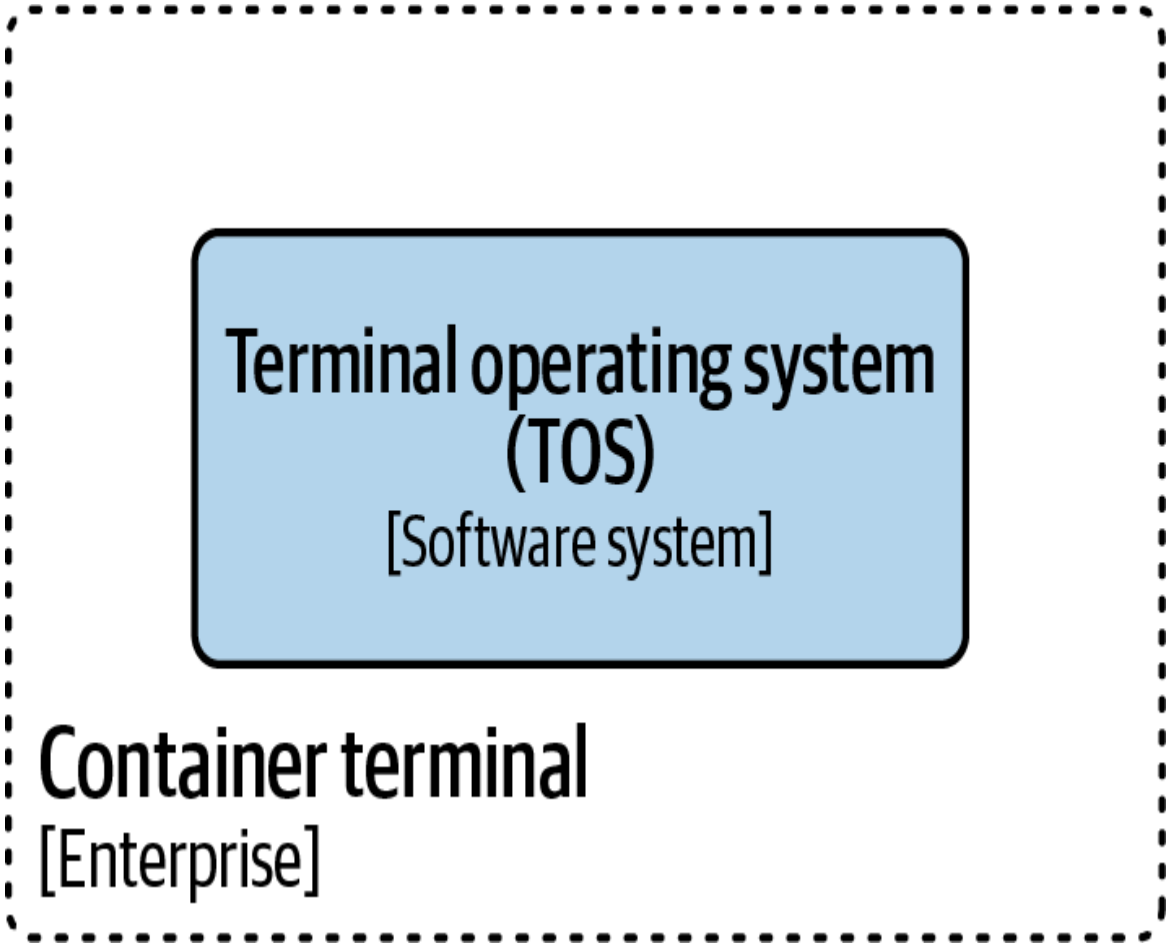


Figure 1-2. The terminal operating system.

So there is a TOS at the container terminal that has been dutifully fulfilling its task for 30 years. It has served the container terminal well in the past and has been repeatedly expanded and renewed.

However, these expansions and renovations were often patchwork rather than well-planned adjustments, with the result that the TOS has grown uncontrolled in all directions. At this point in time, the TOS has become an error-prone, difficult to extend, and old-fashioned-looking monolithic legacy software.

That is why the director of the container terminal has hired us, Carola and Henning, as consultants, to modernize the TOS. When we arrive, with the

help of the drawing in Figure 1-1 and the diagram in Figure 1-3, we are told that the TOS is used by gate checkers, straddle carrier operators, crane operators, controllers, and stowage planners.[2] Each role has its own masks in the TOS with specific functionalities for the respective work steps. The crane operators and straddle carrier operators also have their own TOS app on a mobile device in the operator's cab.

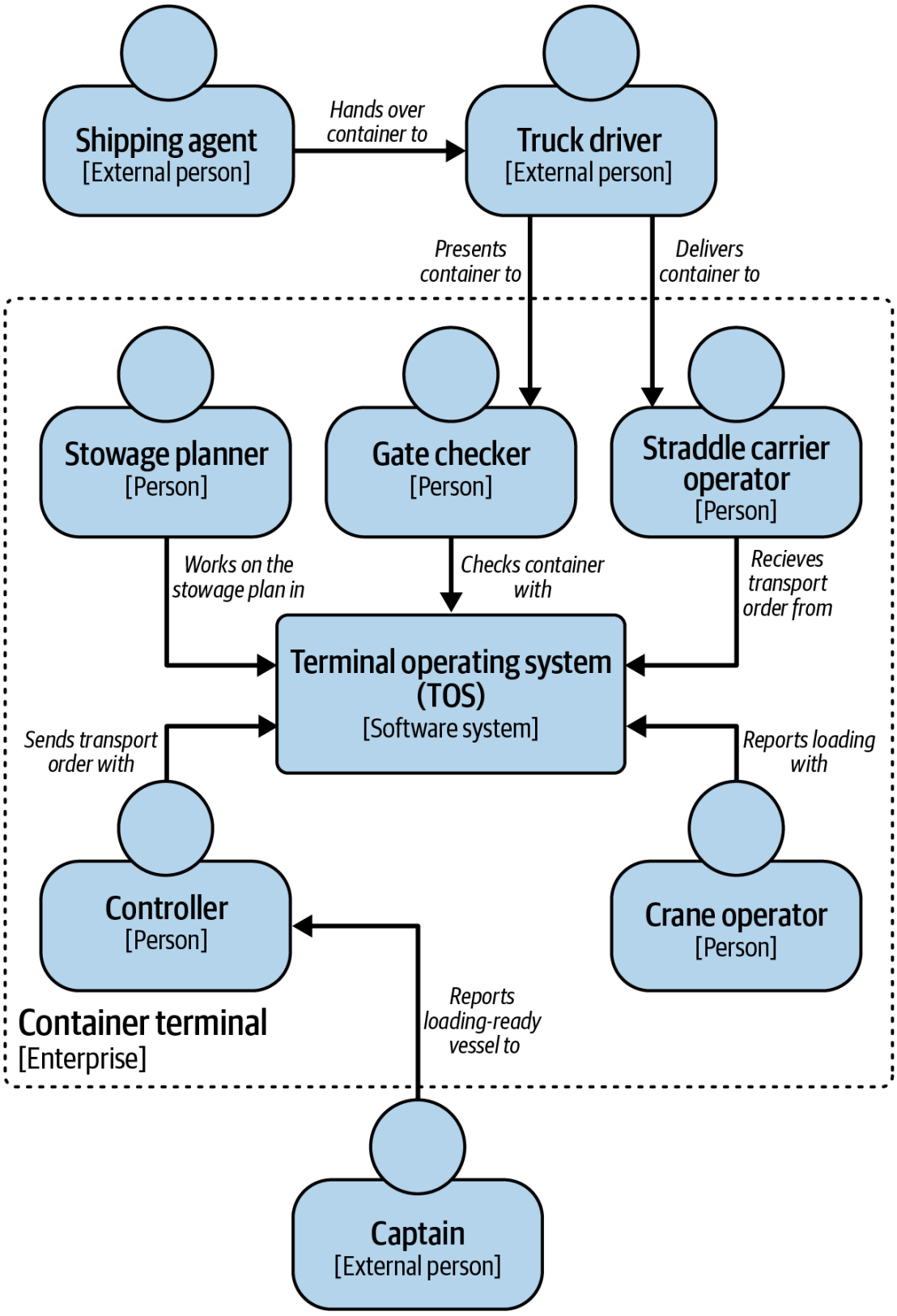


Figure 1-3. The TOS in context.

The notation (called the C4 Model) used in Figures 1-2 and 1-3 is explained in Chapter 5. To understand enough for now, see "Transformation Helper: C4 Model".

> **TRANSFORMATION HELPER: C4 MODEL**
>
> The C4 Model is a hierarchical approach to software architecture that uses four diagram types to show a system from different levels of abstraction for better stakeholder understanding. Here we're using the highest-level view—a context diagram that shows how a system fits into its environment by depicting users, external systems, and their interactions.

Domain Rediscovery

Now that we know who works with the aging TOS, we invite a selection of these TOS users (i.e., employees from different areas of the container terminal) into a workshop. We interview them and ask how their work is done at a coarse-grained level.

This step surprises the architects and developers of the aging system. Their expectation was that we would first analyze the source code of the system to find the places where it should be modernized. Such an approach can certainly lead to quality improvements, but it does not address the core problem of legacy systems. The architecture of the system is not based on the different business areas that are recognizable in the domain, but is the result of unorganized growth. Business areas can only be found *if the business domain is rediscovered*—which is why we start by talking to the employees.

The employees tell us about the typical journey of a container through the terminal:

1. Before any transportation takes place, the delivery documents for the container are sent to the container terminal by the shipping agent—so the terminal can plan ahead.

2. Once the goods are packed into the container and all documents are ready with the shipping agent, a truck drives with the container to the gate of the container terminal.

3. There, the delivery documents and the cargo are checked by gate checkers and, if everything is in order, the truck with the container is allowed to drive to the transfer point at the terminal. From there, a straddle carrier picks up the container and places it in the yard.

4. A few days before the container can start its journey on the container vessel it is intended for, a stowage planner places the container into the stowage plan of the vessel that is to transport it. Once the vessel has arrived at the container terminal, the captain reports to the controller responsible for the vessel and loading can begin.

5. The controller sends the straddle carrier driver a transport order. The straddle carrier then picks up the container and takes it to the quayside at the vessel's berth. There, a crane loads the container to the appropriate bay on the vessel.

6. Finally, the vessel departs and eventually reaches the port of destination, where the container is unloaded.

As we listen to this narrative, we record it as a domain story (see Figure 1-4).[3] This domain story contains the various roles from Figure 1-3, and we can see how they work without the software system itself becoming visible.

By developing the domain story collaboratively, as both consultants and domain experts, we learn a lot. For example, the straddle carrier operator did not know how the delivery document check was handled. Now they understand why it sometimes takes such a long time until the container is handed over to them.

How to develop such domain stories, as seen in Figure 1-4, and which notation they follow, are explained in Chapter 4. To understand enough for now, see "Transformation Helper: Domain Storytelling".

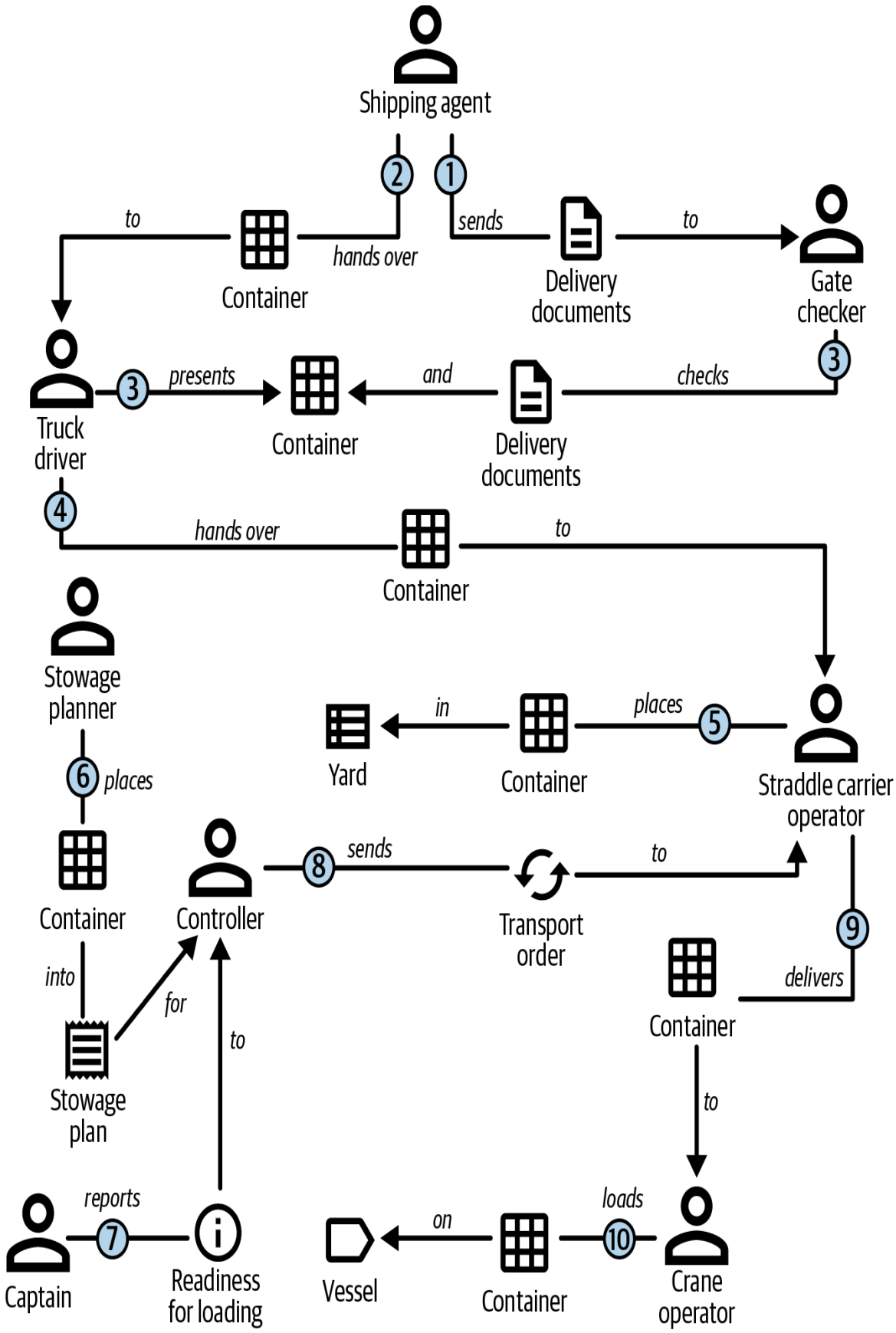

Shipping agent
2
1
to
Container
hands over
sends
Delivery documents
to
Gate checker
Truck driver
3
presents
Container
and
Delivery documents
checks
3
4
hands over
Container
to
Stowage planner
Yard
in
Container
places
5
Straddle carrier operator
6
places
Container
Controller
8
sends
Transport order
to
9
into
for
Stowage plan
to
Container
delivers
to
reports
7
Captain
Readiness for loading
Vessel
on
Container
loads
10
Crane operator

Figure 1-4. The coarse-grained process of a container terminal. Follow the numbers and the arrows to read the story in the right order.

TRANSFORMATION HELPER: DOMAIN STORYTELLING

Domain Storytelling is a Collaborative Modeling method that brings together domain experts and developers to gain a common understanding of the business. Together they visualize what they have understood in a diagram that shows actors (the stick figures), work objects (the other icons), and activities (the arrows). To read the story in the right order, follow the numbers and the arrows.

Determining Target Architecture

Now that everybody has grasped the coarse-grained process and has certainly learned many more details about working at a container terminal, our next task is to identify business areas (called subdomains) in the coarse-grained process.

An intense discussion with the domain experts follows. Time and again, we ponder these questions:

- Which steps or activities in the domain story belong together?
- Where can we identify coherent business areas that form a subdomain?
- What are the right names for the subdomains?

We learn, for example, that the shipping agent, truck driver, and gate checker work together in a business area that has the goal of bringing the container into the port. The container terminal people call this business area “port entry.” One indicator that helps find this boundary is the one-way characteristic of step 5. After this step, there is no return to port entry or to the actors shipping agent, gate checker, and truck driver.

As an outcome of this discussion, we draw several boxes in the domain story to group the related steps into subdomains (see Figure 1-5).

How subdomains are found and which heuristics and indicators can help are explained in Chapters 3 and 10.

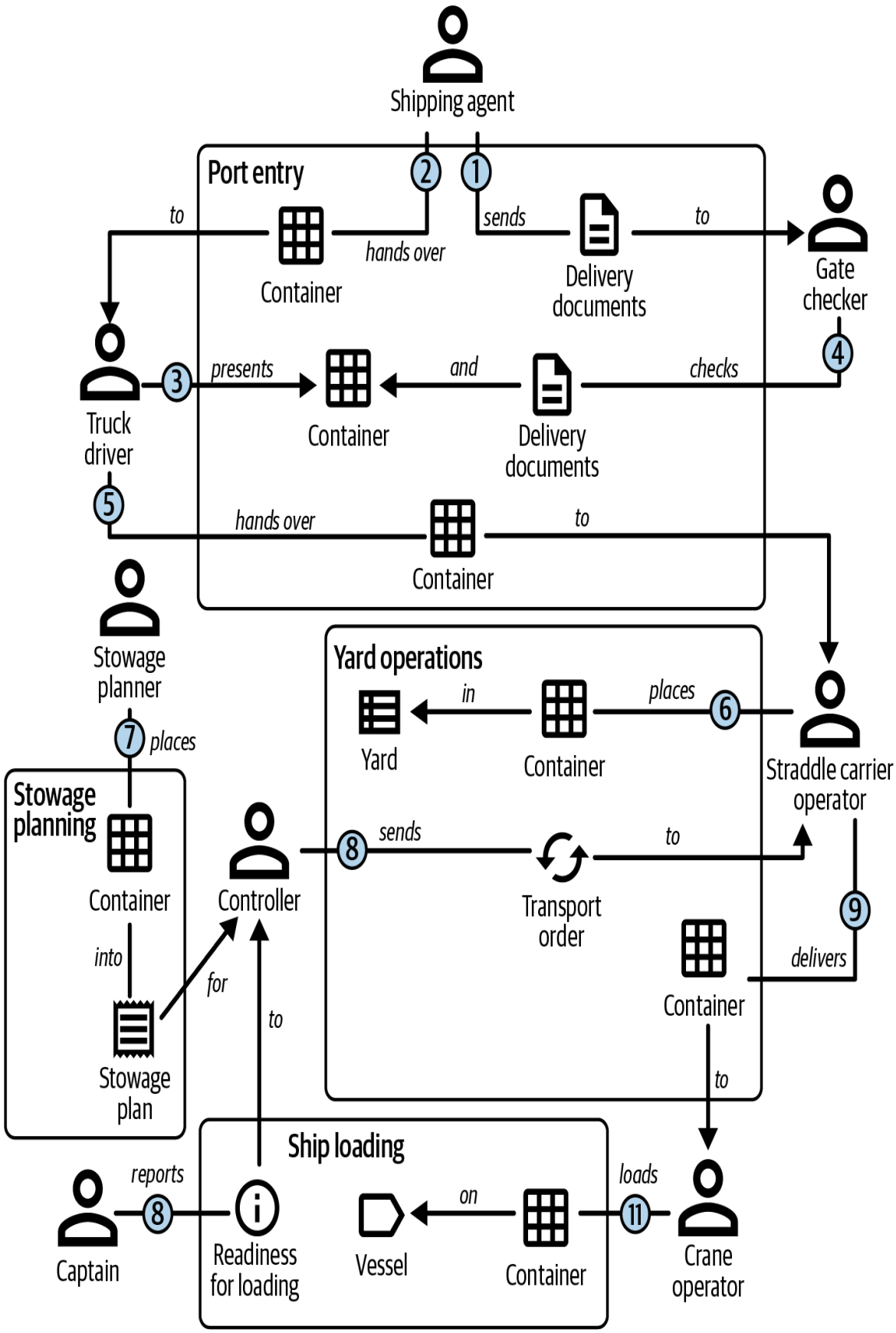

Shipping agent
Port entry
2
1
to
Container
hands over
sends
Delivery documents
to
Gate checker
Truck driver
3
presents
Container
and
Delivery documents
checks
4
5
hands over
Container
to
Stowage planner
Yard operations
in
Yard
Container
places
6
Straddle carrier operator
7
places
Stowage planning
Container
into
Stowage plan
for
Controller
8
sends
Transport order
to
9
delivers
Container
to
to
Ship loading
reports
8
Captain
Readiness for loading
on
Vessel
Container
loads
11
Crane operator

Figure 1-5. The coarse-grained process of a container terminal—with boundaries.

Next, we discuss which subdomains are supported in the system. We copy the names of these subdomains to a whiteboard and add the dependencies between these parts of the process. The result is what is called a context map with domain-driven bounded contexts (see Figure 1-6). This context map is the template for our target architecture.

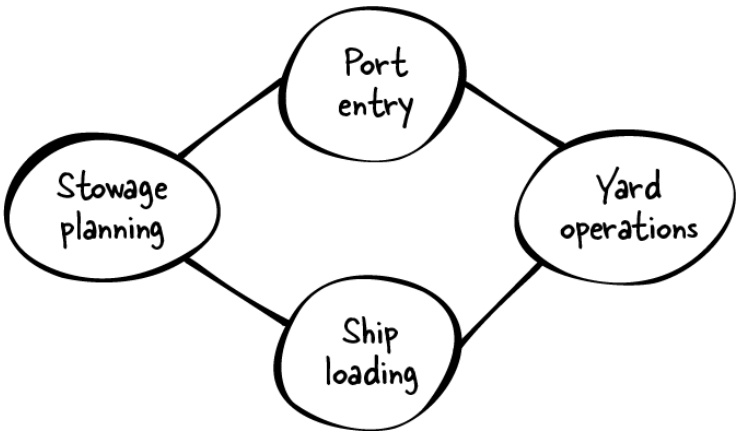


Figure 1-6. Target context map of the container terminal.

For each bounded context from the context map, there should be a corresponding domain-driven module in our target architecture that exchanges information with the other modules via narrow interfaces, as in the context map.

How bounded contexts are derived from subdomains, and how context maps are drawn and used, is explained in Chapters 3 and 11.

Comparing Actual Architecture with Target Architecture

Now that we know what our system *should* look like, we compare the target architecture with the actual architecture. At this point, the architects and developers are happy because we are finally turning to their source code. The TOS was written in Java. To compare the target and actual architecture, we read the source code of the TOS into an architecture review tool and model the desired structure on the source code. You can find out which tools are suitable for this step in *Sustainable Software Architecture* [Lilienthal 2019].

In Figure 1-7, you can see the sad result of this first comparison. In this diagram, we see the four domain-driven bounded contexts from the context map as squares. We will call these *modules*. We have created these modules in the architecture review tool and assigned to them the parts of the package tree with the classes they contain.

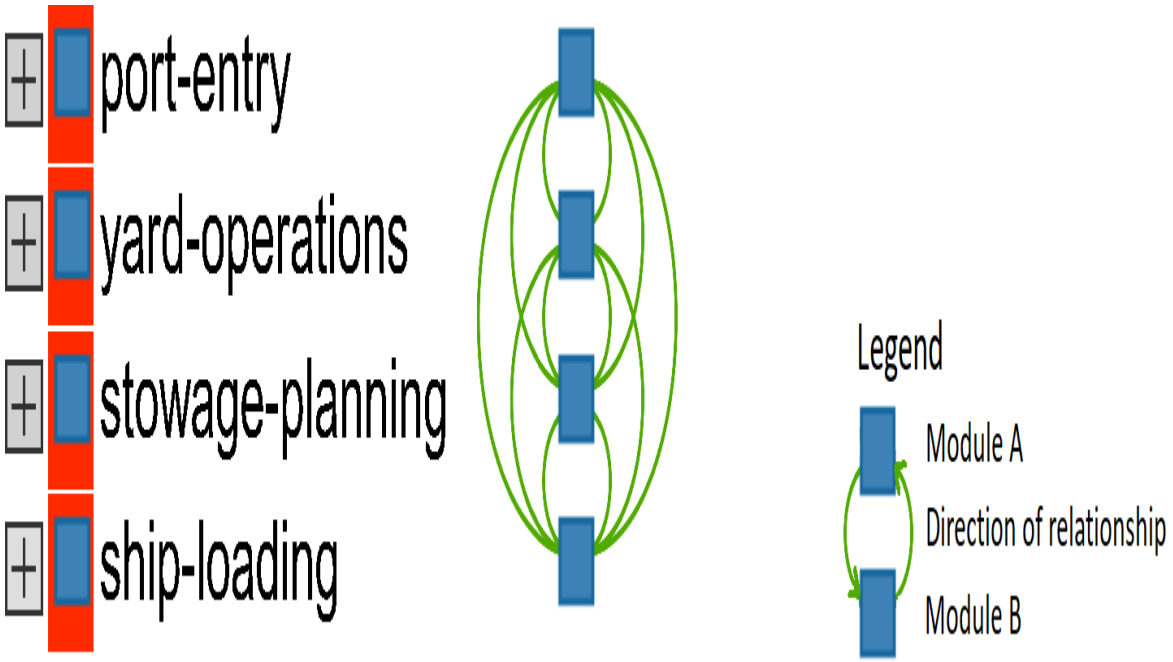


Figure 1-7. Screenshot of the architecture review tool: comparison of actual and target architecture. If you are unfamiliar with this kind of tool, see "Transformation Helper: Architecture Review Tools".

TRANSFORMATION HELPER: ARCHITECTURE REVIEW TOOLS

An architecture review tool can help visualize and get a common understanding of the current architecture. It is also helpful to compare this with the target architecture.

In this book we show screenshots of a tool called Sonargraph. If you click on the plus sign in front of the squares (the modules), the package tree and the classes will expand. On the righthand side of the diagram, you can see the real relationships from the source code, represented by arcs. There are arcs on the lefthand side. These arcs represent the relationships between the classes that are sorted into the modules. The arcs on the lefthand side go from top to bottom, and on the righthand side, from bottom to top.

We draw a corresponding component diagram on a whiteboard (again in C4 Model notation; see Figure 1-8) so that all developers and architects can understand the result of the analysis with the tool from Figure 1-7. The result indicates that there should be four domain-driven modules in the system on the one hand, and a technical structure consisting of layers for the UI, application for the domain processes, the domain model, and the database on the other. This is what a good structured modular architecture would look like, *but currently the TOS does not reflect this architectural concept in any way!* We are looking at a highly interwoven structure with many dependency cycles.

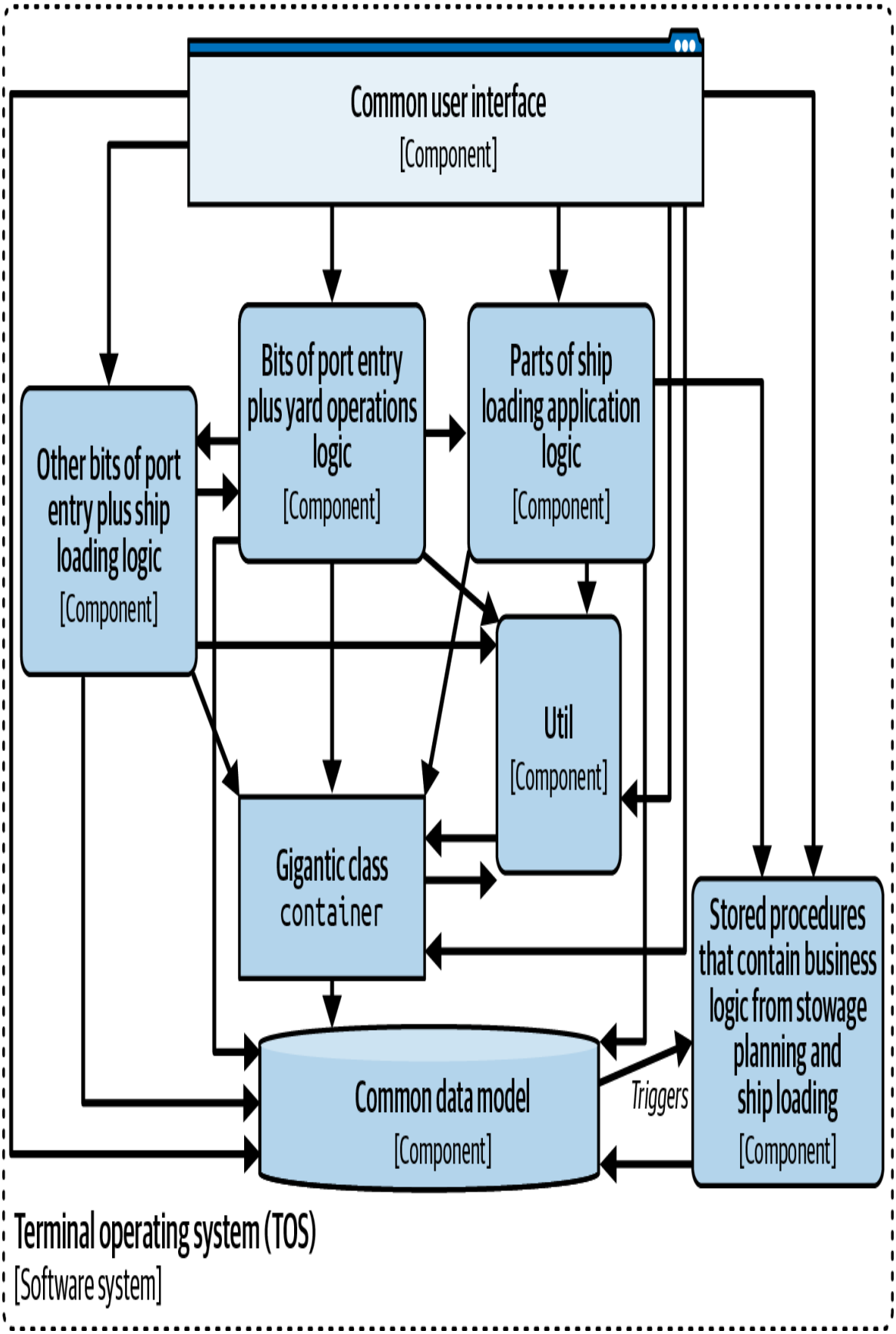

Common user interface
[Component]
Bits of port entry plus yard operations logic
[Component]
Parts of ship loading application logic
[Component]
Other bits of port entry plus ship loading logic
[Component]
Util
[Component]
Gigantic class container
Stored procedures that contain business logic from stowage planning and ship loading
[Component]
Common data model
[Component]
Triggers
Terminal operating system (TOS)
[Software system]

Figure 1-8. The TOS—currently a highly interwoven structure with many dependency cycles but also some technical layering.

The arrows between the boxes in Figure 1-8 indicate that there are many relationships between the components of the system—and some of these are even cyclic. The unappealing result: strongly interwoven components that somehow provide functionality for the four subdomains. At least there seems to be some *technical layering*: the common user interface forms the top layer, the common database forms the bottom layer, and everything else (the "business logic") forms the middle layer.

We have often seen systems in such a state and are no longer surprised that the TOS is error-prone and difficult to expand. To understand the true extent of the problem, we determine the Modularity Maturity Index (MMI) for the system.

You can find out exactly which aspects the MMI evaluates and how it is determined and calculated in Chapter 5. To learn enough to understand the MMI at this point, see "Transformation Helper: Modularity Maturity Index".

TRANSFORMATION HELPER: MODULARITY MATURITY INDEX

The MMI is calculated from a series of metrics and quality criteria and thus maps the quality of a system's modularity on a scale of 0 to 10 (see Figure 1-9).

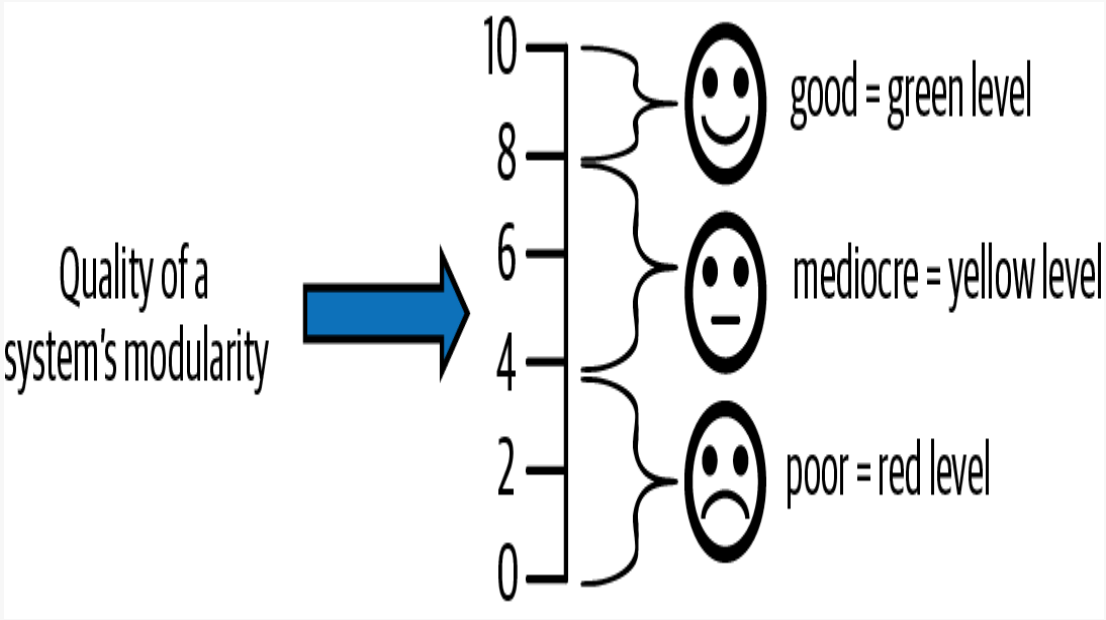


Figure 1-9. The Modularity Maturity Index.

The calculated number is in turn summarized in a traffic light system. Well-structured systems have an MMI of 8 to 10, mediocre systems are between 4 and 8, and systems with a poor structure have an MMI below 4.

We calculate an MMI of 4.9 for the TOS—the system is apparently in a mediocre state at all levels.

The MMI of 4.9 for the TOS is partly due to the fact that the domain concept "container," with all of its functions for port entry, yard operations, stowage planning, and ship loading, is modeled in a single huge class, with a few other classes to which subtasks are delegated. The data from the class `Container` and its appendages are in turn stored in a single gigantic database table.

At the beginning of the development of the TOS, the class `Container` and the associated database table were still small, and the right fit for the first part that was programmed as the first small TOS to support the first processes at the container terminal. But with each extension of the TOS with new functionality, the container class and its database table became larger and larger, so that we must now speak of a *god class* or an *unbounded domain model*.

What we want to achieve is a structure like that in Figure 1-10, in which the individual bounded contexts from Figure 1-6 each have their own container class reduced to the respective context, and communicate with each other via narrow interfaces.

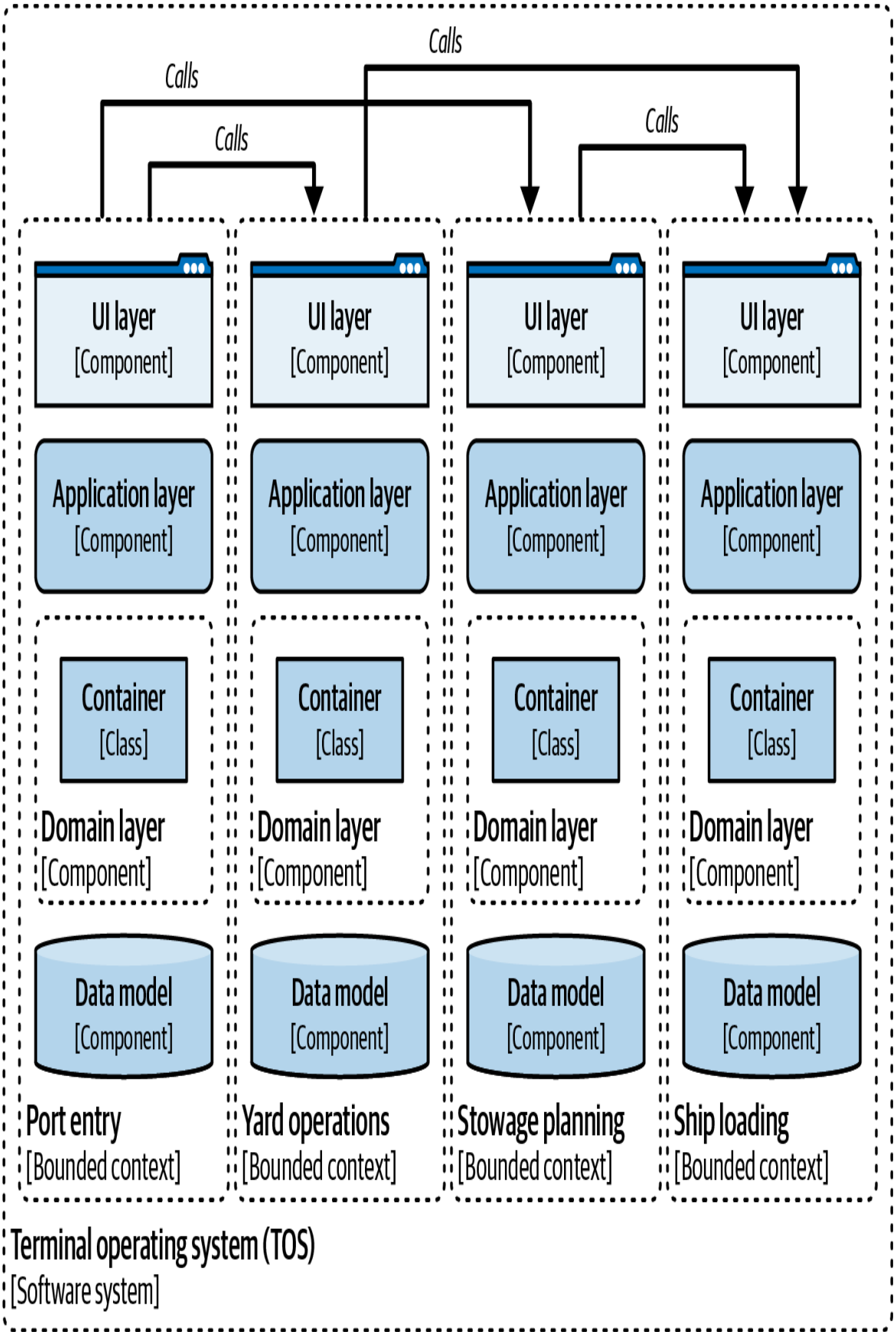

Calls
Calls
Calls
Calls
UI layer
[Component]
UI layer
[Component]
UI layer
[Component]
UI layer
[Component]
Application layer
[Component]
Application layer
[Component]
Application layer
[Component]
Application layer
[Component]
Container
[Class]
Container
[Class]
Container
[Class]
Container
[Class]
Domain layer
[Component]
Domain layer
[Component]
Domain layer
[Component]
Domain layer
[Component]
Data model
[Component]
Data model
[Component]
Data model
[Component]
Data model
[Component]
Port entry
[Bounded context]
Yard operations
[Bounded context]
Stowage planning
[Bounded context]
Ship loading
[Bounded context]
Terminal operating system (TOS)
[Software system]

Figure 1-10. Target architecture: bounded contexts with different implementations of the class `Container`.

In the architecture review tool, this structure would also appear much tidier (see Figure 1-11).

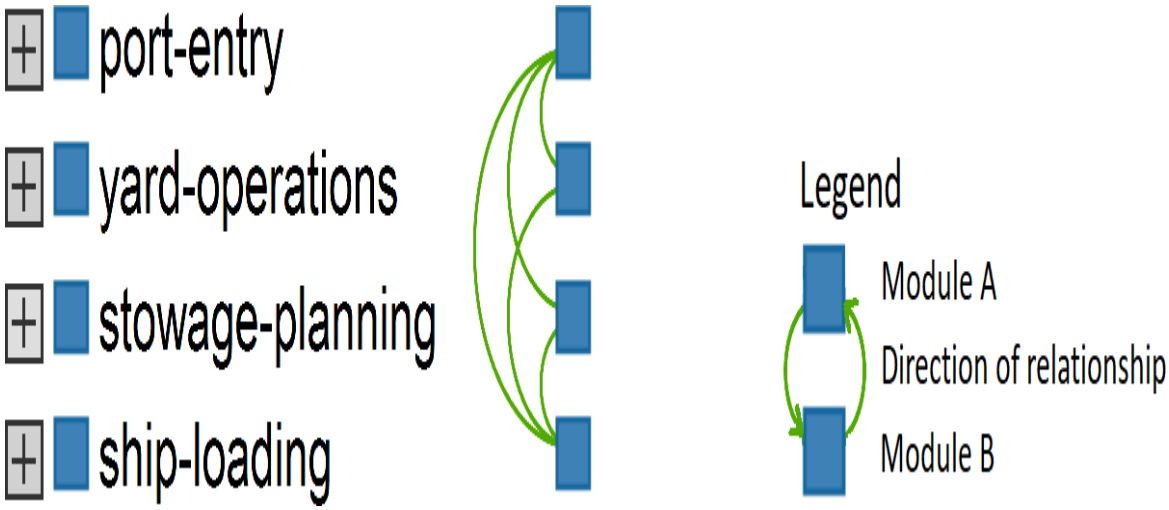


Figure 1-11. Screenshot of the architecture review tool: desired result of the alignment. There are no more upward relationships (which would appear on the right side of the boxes).

To bring order to the structures and relationships in such a case, you need a well-thought-out plan that divides the path from the actual architecture to the target architecture into doable chunks. In Part III, you will learn techniques that will enable you to develop such a plan for restructuring your legacy system yourself (see Chapter 12).

We identified a starting point for our TOS when determining the MMI (see above), which we discuss with the developers and architects of the TOS and thus work out part of the future conversion plan. Our starting point is the design patterns that the developers of the TOS used to bring some order into their source code. In the code, we find a series of service classes that all follow the common design pattern *Service* and offer with their methods stateless functionality. One example is the `PlanningService`, whose interface is shown in the class diagram in Figure 1-12.

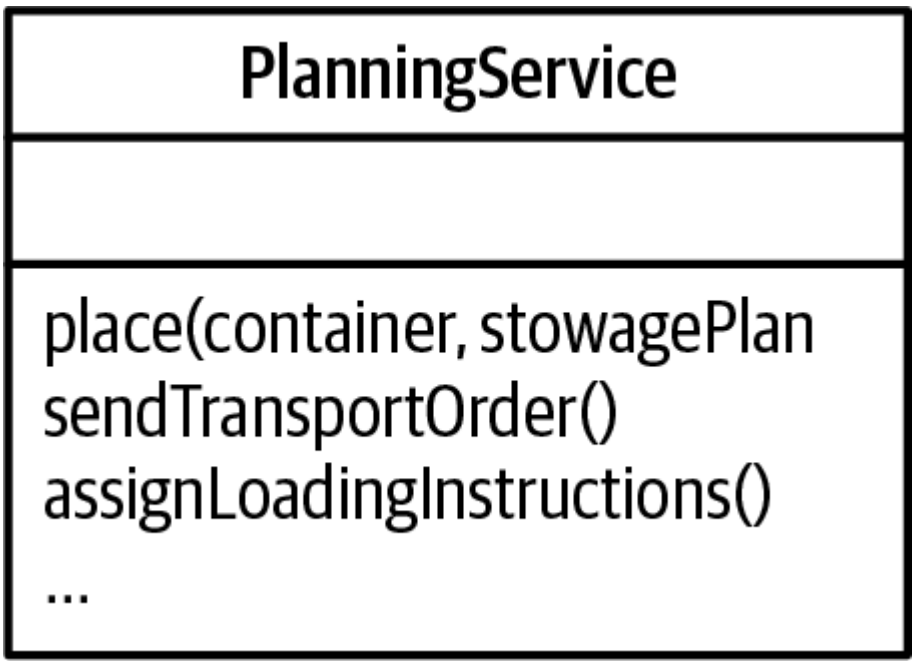


Figure 1-12. Actual architecture (extract). Problem: the `PlanningService` combines unbounded business logic in a huge service class.

On closer inspection of the interface, together with the architects and developers, we realize that this class offers functionality for two bounded contexts from the context map (shown back in Figure 1-6): for stowage planning and for yard operations. In Figure 1-13 we have overlaid the two contexts on the class diagram of `PlanningService` to make the problem more visible.

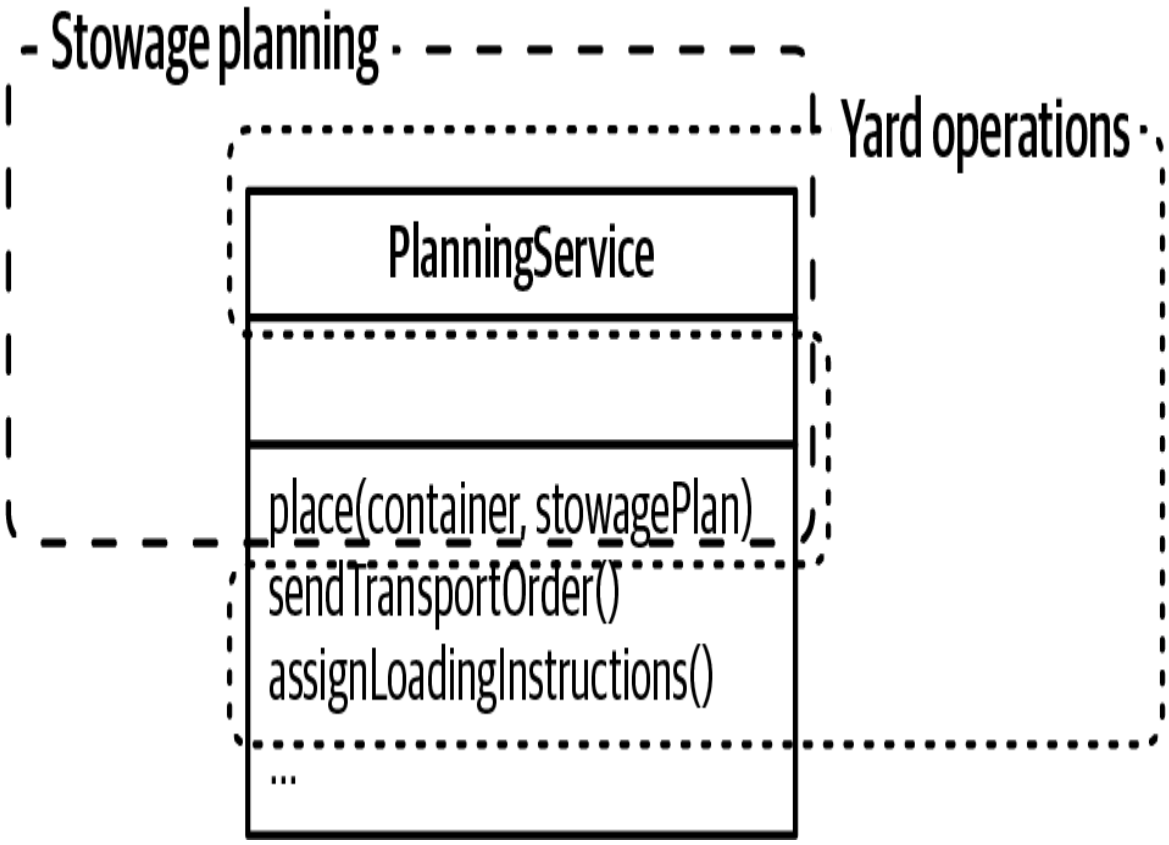


Figure 1-13. Aligning actual architecture with target architecture for the `PlanningService`. Problem: there are no boundaries expressed in the software yet; "stowage planning" (dashed line) and "yard operations" (dotted line) overlap (and make the diagram hard to read).

The architects and developers of the TOS had had a Domain-Driven Design (DDD) training with us earlier. Now they are reminded of DDD's strategic design and start to understand what we are aiming for (if DDD is new to you, see "Transformation Helper: Domain-Driven Design").

TRANSFORMATION HELPER: DOMAIN-DRIVEN DESIGN

Domain-Driven Design (DDD) is a software development philosophy in which an understanding of the business and its processes drive the architecture of the system. Processes, structure, work items, and language of the domain are all modeled in code.

DDD provides several tools; one of them—the strategic design—helps find the boundaries in the domain to derive the right modularization for the software.

The architects and developers begin discussing how they can split up the current `PlanningService` class. As a goal there should be two separate `PlanningServices`, each of which must exist in the domain-driven module intended for them. The difficulty and discussion arise when considering the implementation of the individual methods, but the developers and architects also develop ideas here as to how a separation of the `PlanningService` class could succeed. The target architecture with two separate classes can be seen in Figure 1-14.

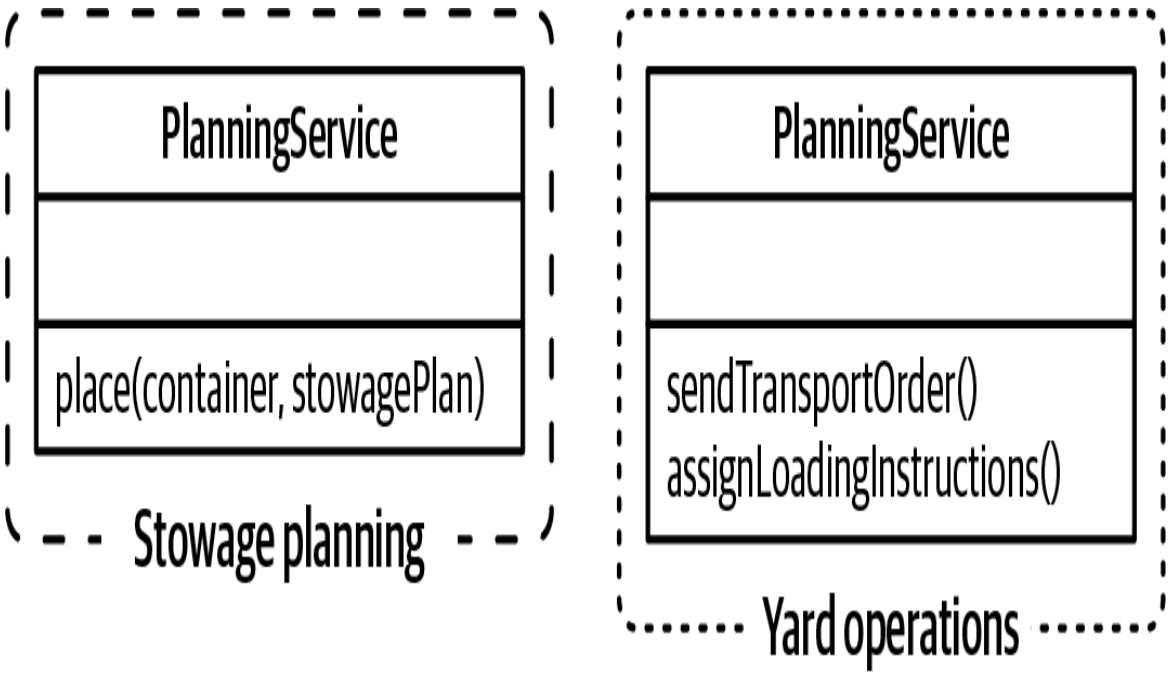


Figure 1-14. Target architecture on the class level (extract). Improvement: there is no more overlap; "stowage planning" (dashed line) and "yard operations" (dotted line) are now separated bounded contexts.

As seen earlier with the `Container` classes, it can be appropriate to split one big class into several classes with the same name in different contexts. The team isn't particularly happy with the name `PlanningService`. Here it might be more appropriate to find different names for the classes in the different contexts.

Also, both `PlanningService` classes combine application and domain logic, so it would be better to further split up the application and domain model logic. Since not everything can be done at once, the team decides to postpone attacking these problems.

In discussions with the developers and architects, step-by-step we identify several refactorings that have to be carried out on the system. We first collect them on sticky notes on the whiteboard (see Table 1-1). Some refactorings, such as 1.0 and 2.0, are too big to be implemented in one sprint. They are split up into smaller refactorings, such as 1.1, 1.2, and 2.1. It is important that the refactorings are split up repeatedly until they are small enough, so that they can each be implemented in a sprint during the

transformation. A big refactoring is only then done once all the small refactorings that it has been split into are done.

Table 1-1. Improvement backlog with refactorings for the container terminal (excerpt)

No.	Refactoring	Small enough for one sprint
1.0	Extract bounded context “stowage planning”	no
1.1	Separate parts of `PlanningService` from the monolith into specialized version in bounded context “stowage planning”	yes
1.2	Separate parts of `Container` from the monolith into specialized version in bounded context “stowage planning”	yes
2.0	Extract bounded context “yard operations”	no
2.1	Separate parts of `PlanningService` from the monolith into specialized version in bounded context “yard operations”	yes
2.2	Move bounded context “yard operations” into its own microservice	yes

Table 1-1 numbers the refactorings in a way that shows what belongs together. In the example, the refactoring 1.0 is split up into refactoring 1.1 and 1.2 (and probably 1.x and 1.y, which are left out for brevity).

The team decides that the extracted bounded contexts will keep the Java implementation language. Also, the bounded contexts will first be implemented as modules in the same codebase. Refactoring 2.2 shows that a move toward a microservices architecture is considered later on.

In the appendices you can find a catalog of the different domain-driven refactorings that help with the transformation.

Prioritization and Implementation of the Transformation

With a well-filled improvement backlog, the development team plans the first sprint, in which it dedicates 30% of the effort for the transformation. The first refactoring planned for the sprint is 2.1 ("Separate parts of `PlanningService` from the monolith into specialized version in bounded context *yard operations*"). The pair of programmers that starts with 2.1 creates an empty module `yardPlanning` and in it an empty class `yardPlanning.PlanningService`. Next, the team moves the method `sendTransportOrder()` from the monolith to the new module. For the time being, the new module is kept as part of the same codebase as the monolith. Later the decision might be made to move the new module into a program in its own right, thus turning it into a microservice.

Such small steps are used to redesign the old system. We visit the team again and again over weeks and months. The developers regularly inspect what they have achieved and the impact their modifications have had on the structure of the system. They must continuously adjust the plan for the transformation, add new refactorings, and adapt existing refactorings. It's impossible to understand all the necessary work up front. As changes take place, new necessary work is identified.

After a few weeks, the first positive changes are noticeable—there are areas in the system where it is easier to add or change functionality. After months,

further development of the entire system becomes simpler—the team has now been able to resolve many dependencies, thus significantly reducing the high degree of coupling in this former big ball of mud. Finally, after years, the system becomes completely modular—what a moment! The individual bounded contexts are clearly recognizable in the source code. Narrow interfaces connect the domain-driven modules, but only where necessary.

What Has Just Happened?

In this introductory section, we have taken you through the various steps of a *Domain-Driven Transformation* in a quick run-through. We have given you an initial overview, applying an example of a container terminal. To realize such a quick run-through, we have simplified the example considerably.

> **NOTE**
>
> When transforming a real software system, many more difficulties arise than are shown in this simplified example. The domain is much larger and more complex, so that many more processes have to be analyzed with domain stories, and a decomposition into bounded contexts is only possible after much discussion. In practice, the first decomposition is often only a starting point for a step-by-step transformation in which the transformation has to be run through several times—Domain-Driven Transformation is an iterative agile process. The software in the real world is also much larger and more complex than in our small example. It usually consists of significantly more parts, possibly several systems. These parts or systems are often very tightly knotted, and disassembling them in small steps is in no way obvious.

To provide further information on how Domain-Driven Transformation can be carried out with different degrees of chaos in the architecture, in the next section we will move away from the example and describe Domain-Driven Transformation as a method in combination with different typical architectures that we find in practice.

The Method

The Domain-Driven Transformation method consists of four strategic steps, which are visualized in Figure 1-15.

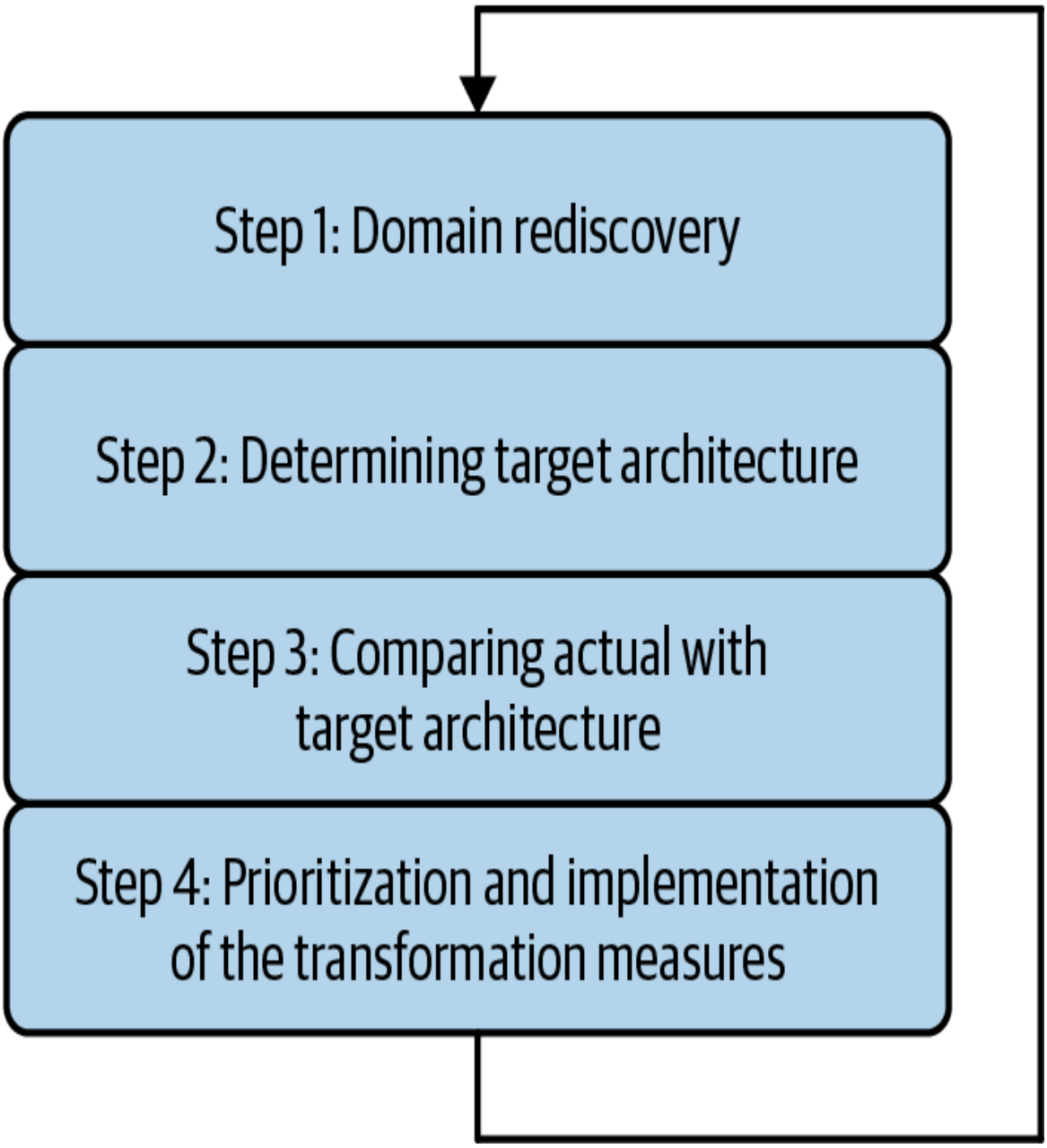


Figure 1-15. The steps of a Domain-Driven Transformation.

You have already seen these steps in the container terminal example, because we have divided the description of our approach to modernize the TOS into corresponding subsections:

Step 1: Domain rediscovery (see Chapter 10)

> Put the existing solution aside and focus on the problem with today's knowledge. Use Collaborative Modeling methods to reunderstand that problem, its domain, and its processes. For the TOS, we used the Collaborative Modeling method known as Domain Storytelling (see "Domain Storytelling").

Step 2: Determining target architecture (see Chapter 11)

> Derive the ideal target architecture from the newly gained understanding of the domain. Draw boundaries in the collaboratively modeled processes and extract a target context map.

Step 3: Comparing actual with target architecture (see Chapter 12)

> Analyze the actual architecture of the system and identify the differences from the target architecture. Collect the refactorings to be done to transform the system from actual to target.

Step 4: Prioritization and implementation of the transformation measures (see Chapter 13)

> Plan and carry out the refactorings. Breakdown the big refactorings into small enough subrefactorings. Identify the refactorings for the next iteration. Refactor the code and thus the system in small steps toward a better state.

These four steps are at the heart of Domain-Driven Transformation, but our method has more in store for you, namely guidance on how to proceed depending on the state of your architecture.

When we are asked by customers to carry out Domain-Driven Transformation with them, we repeatedly find in step 3, when comparing the actual architecture with the target architecture, that we cannot immediately begin step 4 with the transformation of the source code toward the target architecture, modeled in step 2. This is because the state of these

systems is very poor and requires some preparatory work before the system can be decomposed. What is this preparatory work and how do we select it?

Now, when comparing the actual architecture with the target architecture in step 3, we determine the MMI for all systems on a scale of 0 to 10. In the introductory example, we briefly introduced the MMI as a value that makes statements about the quality of a system's modularity. The MMI of the TOS was quite low at 4.9. In fact, the MMI gives us an indication of the path we need to take with the preparatory work and the transformation.

Paths Through Domain-Driven Transformation

The MMI divides the scale from 0 to 10 into three areas in which we find typical architectures:

MMI 0–4 (red level): Big ball of mud

> A system without structure, where everything knows and uses everything else.

MMI 4–8 (yellow level): Technical layering

> A system with technical layers in which there is a direction of use from the top from the front end to the bottom via the application layer and the domain layer to the database.

MMI 8–10 (green level): Domain-driven modularization

> A system that is divided into independent modules according to the domain and its subdomains, with narrow interfaces between the modules.

In the next three sections, we explain the three levels and give our recommendations for the appropriate further measures in each case. Figure 1-16 gives you an overview of all the means and paths that will be explained in the next sections.

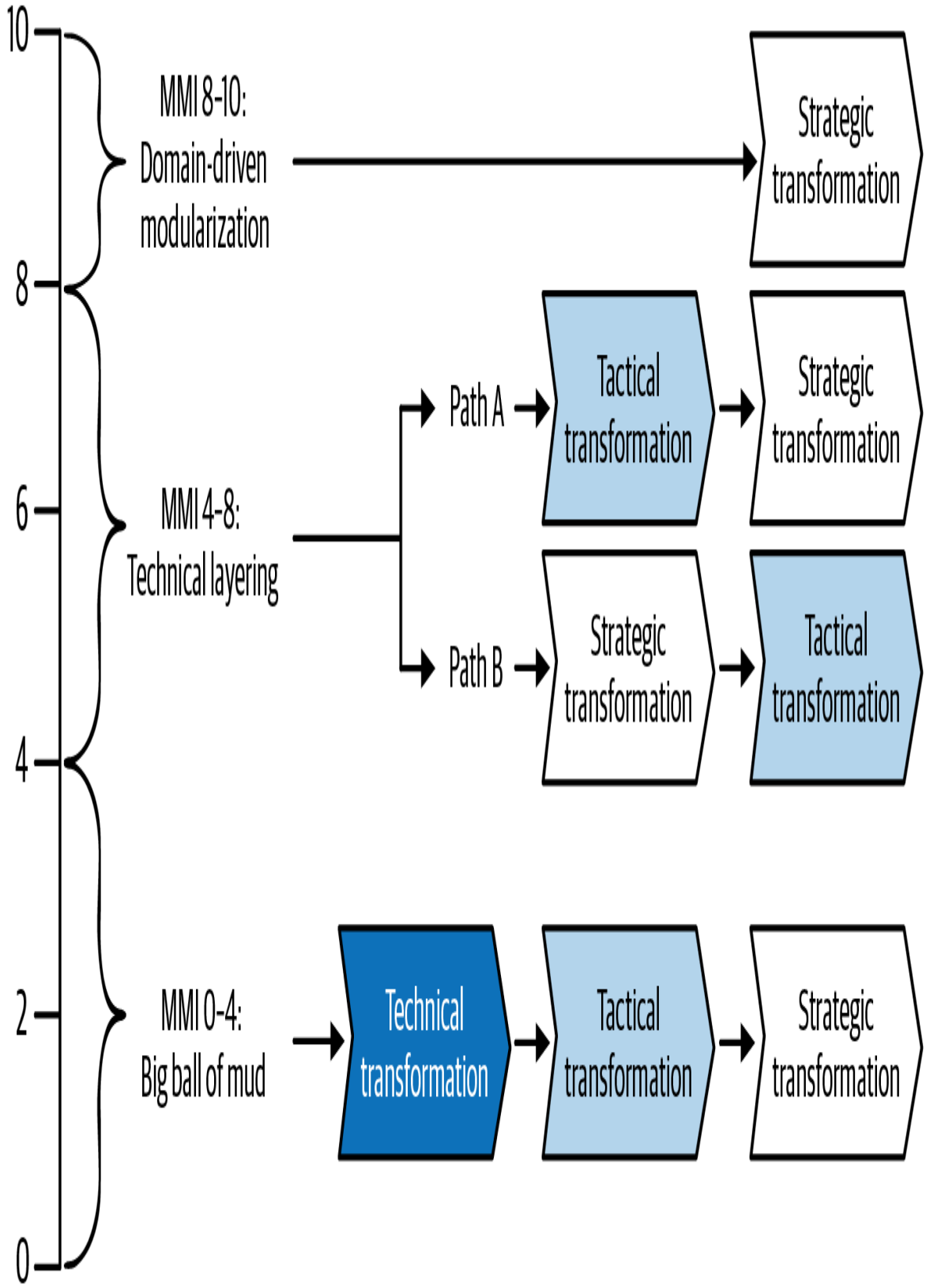


Figure 1-16. Paths through Domain-Driven Transformation.

MMI 0–4: Big ball of mud

If a system has an MMI between 0 and 4, it has all the bad properties of a big ball of mud (see Chapter 5). In Figure 1-17 you can see an abstract representation of this big ball of mud "architecture." Neither a hierarchical structure that follows a technical layering nor the structure of the domain can be found in the source code and its elements. There is no top or bottom, no left or right. Instead, everything is connected to everything else.

Because in systems below an MMI of 4 there are hardly any recognizable structures and everything is connected to everything else, their design at the class level is poor and the structures at the architecture level are not modular. The technical code and the business logic are mixed up, and you can hardly identify the classes that could be converted to a domain model. In addition, these systems often have a lot of bugs and major problems with security and infrastructure. There is a lot of work to do with such a system before even thinking about breaking it down into domain modules using the four steps of strategic Domain-Driven Transformation.

In our experience, a system with an MMI between 0 and 4 should first of all be technically stabilized (see Chapter 7) and then strengthened with domain knowledge (see Chapter 8). In our method the technical stabilization is known as the *technical transformation*, and the strengthening with domain knowledge is known as the *tactical transformation*.

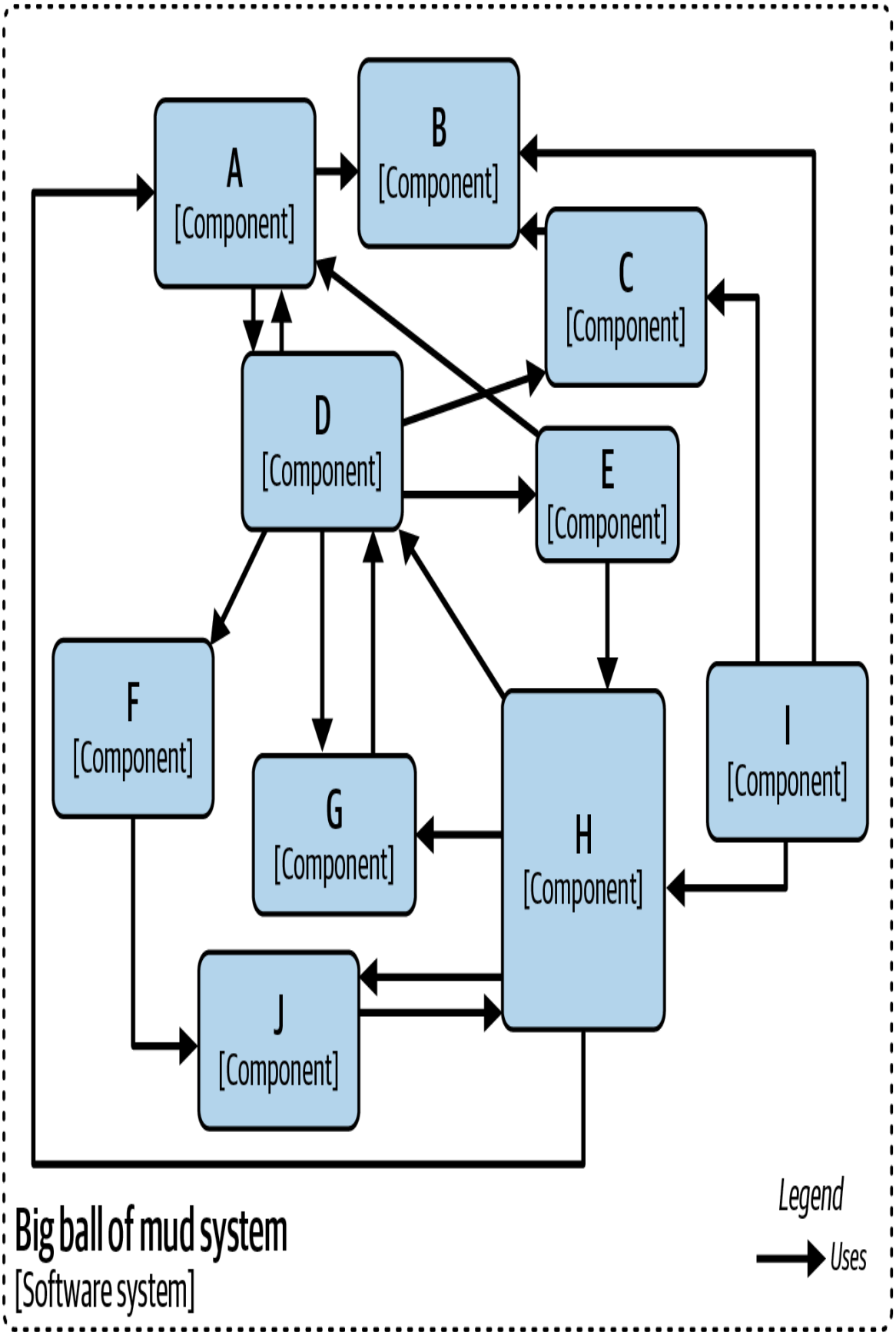

A
[Component]
B
[Component]
C
[Component]
D
[Component]
E
[Component]
F
[Component]
G
[Component]
H
[Component]
I
[Component]
J
[Component]
Big ball of mud system
[Software system]
Legend
Uses

Figure 1-17. Big ball of mud: neither horizontal nor vertical structure.

Technical transformation consists of measures such as automating build and deployment, updating outdated frameworks and libraries, increasing test coverage for bug fixing, and improving error resilience (see Chapter 7). The tactical transformation consists of measures such as enriching domain-specific modeling, introducing value objects and Design by Contract, and reducing cycles, dependencies, and inheritance (see Chapter 8). We explain these measures in detail in Part II. After the technical and tactical transformations, the four steps of the strategic transformation can be completed, as we describe in Part III.

Figure 1-18 shows the path for a system with an MMI between 0 and 4 through the three areas of transformation: technical, tactical, and strategic.

In the next two subsections, we add the paths that should be taken with systems that have a higher MMI.

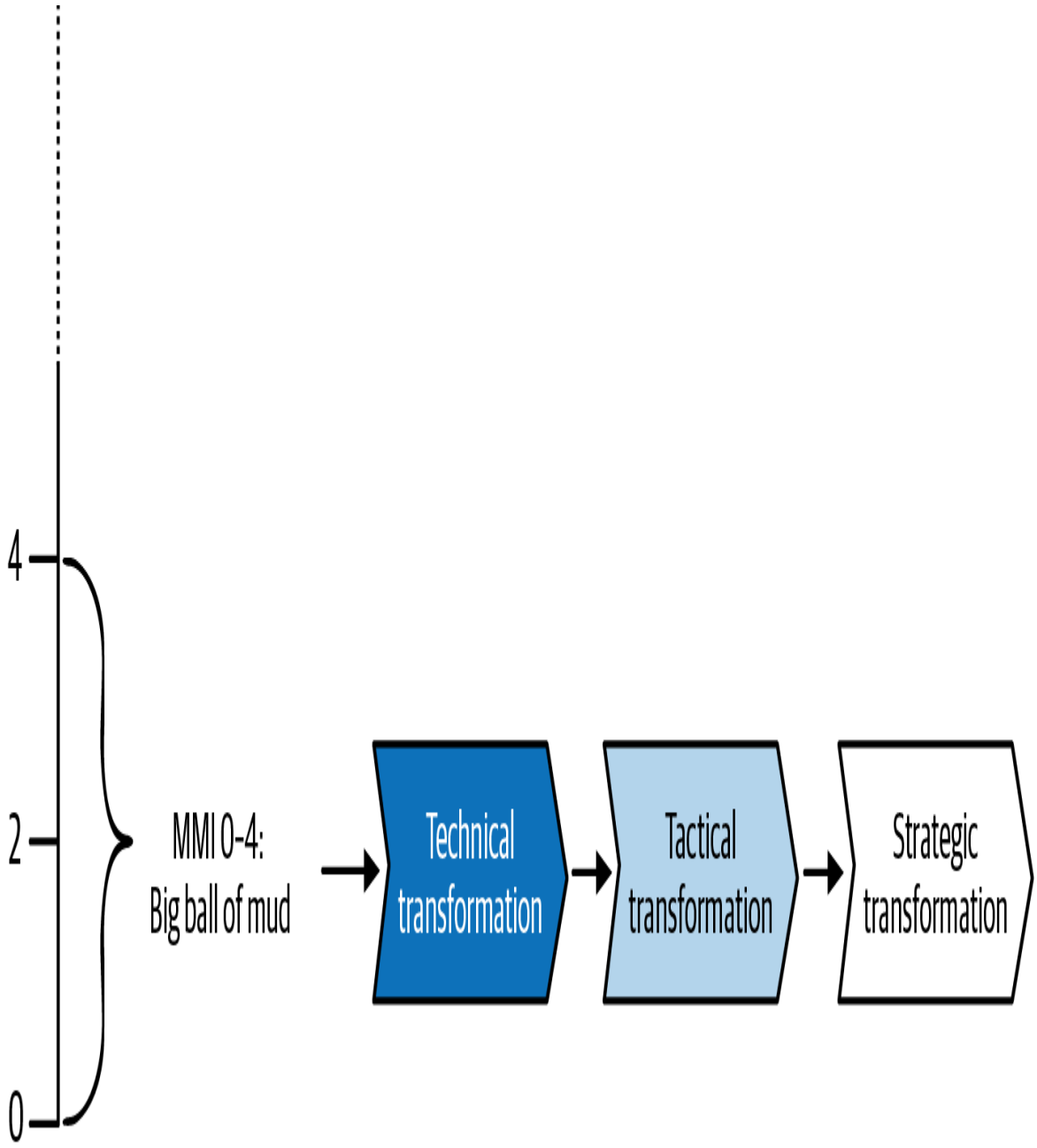


Figure 1-18. The MMI and the first path through transformation.

MMI 4–8: Technical layered architecture

In systems with an MMI between 4 and 8, the technical basis is usually quite OK, and you will find a typical technical layered architecture in which the layers are calling each other from top to bottom; i.e., the layers are acyclic. Further explanations of the layered architecture and other architecture concepts can be found in Chapter 5.

As you can see in Figure 1-19, systems with an MMI between 4 and 8 may have layers that follow a technical decomposition, but vertical slices according to the domain (see Chapter 3) are largely absent.

If you find such a system, there are two ways to get out of this situation:

Path A

Tactical transformation first

Path B

Strategical transformation first

The choice of the first or second path depends on the structure of the domain layer that is found in the respective system.

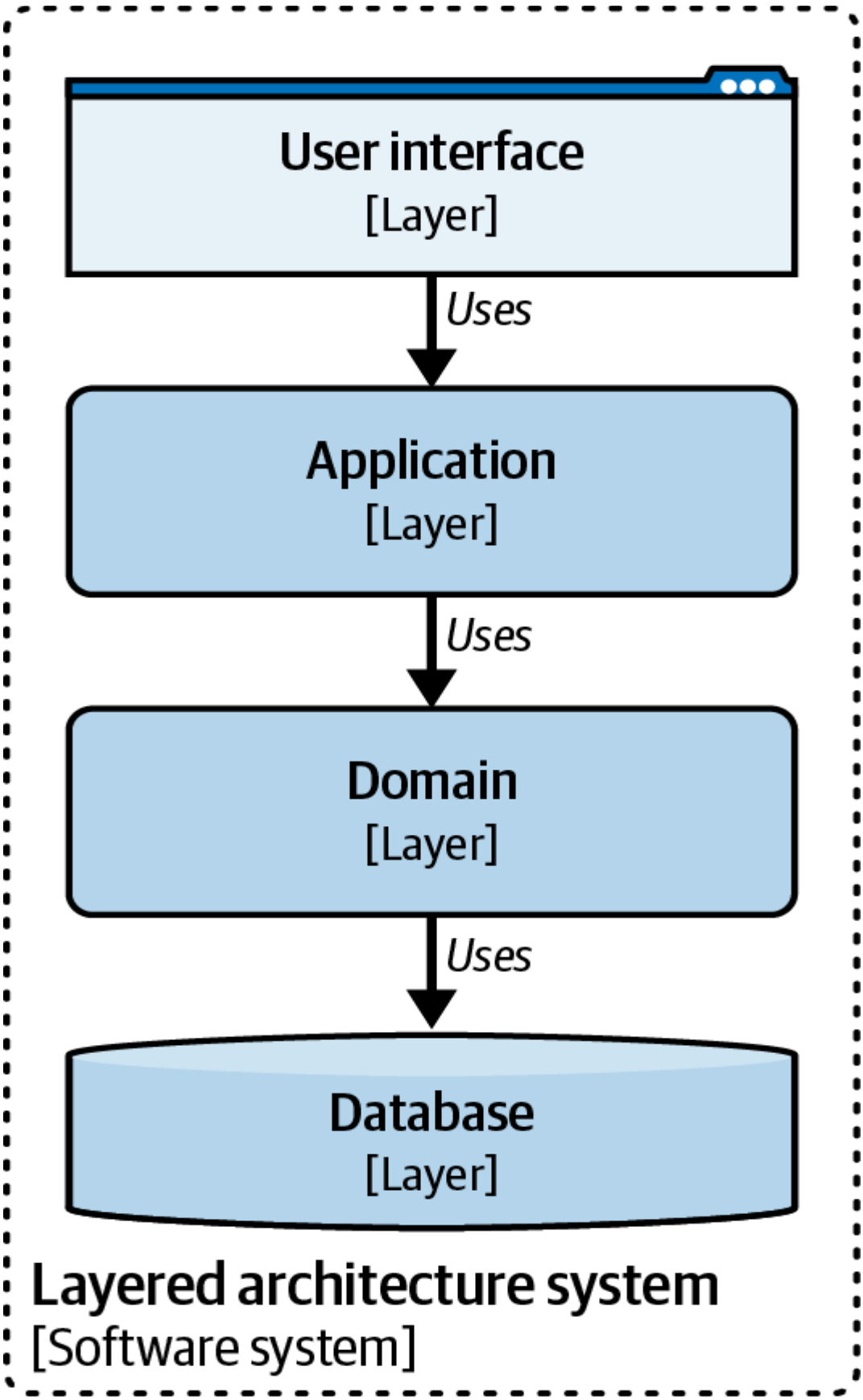

User interface
[Layer]
Uses
Application
[Layer]
Uses
Domain
[Layer]
Uses
Database
[Layer]
Layered architecture system
[Software system]

Figure 1-19. Technical layered architecture: horizontal (i.e., technical hierarchical) structure, but no vertical (i.e., domain) structure.

Path A: Tactical transformation first

You should take path A if you have a system with a layered architecture and an MMI between 4 and 8 in which the domain layer is a big ball of mud. This usually occurs when the database structure has been transferred from tables and their relationships directly into the source code of the domain layer. The result is a large interwoven domain layer without any hint of domain-related structure.

The idea of this path is to first strengthen the domain model with tactical Domain-Driven Transformation (see Chapter 8) and only then decompose it into bounded contexts with strategic Domain-Driven Transformation (see Part III). In Figure 1-20 you can see path A for an MMI of 4–8 of Domain-Driven Transformation.

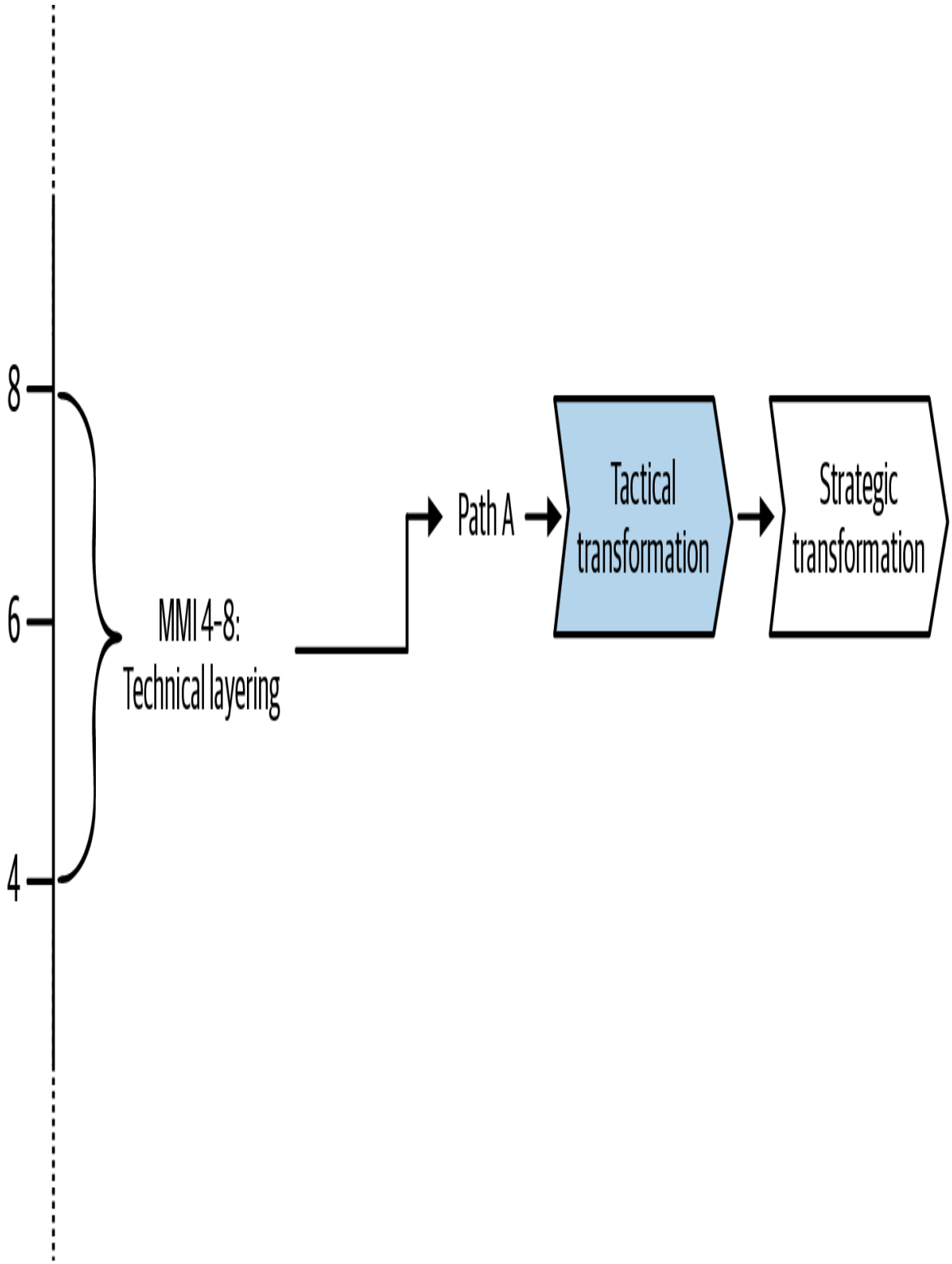


Figure 1-20. The MMI and the second path through transformation.

For the core domain (see Chapter 3), the most important bounded context in your system, we recommend that you pay particular attention to

strengthening the design with domain knowledge (see Chapter 8). The core domain provides the company's key business differentiators in software and must be in good condition so that it can be expanded and further developed quickly and securely. For the supporting and generic subdomains (again, see Chapter 3), especially those that change infrequently, you can often skip this effort.

Path B: Strategical transformation first

If you have a system with an MMI of 4–8 in which the domain layer is not a big ball of mud, but instead has a recognizable structure apart from a few smaller dependency cycles, then you can start with the strategic Domain-Driven Transformation despite the poor design before improving it.

Here, the idea is to first decompose the system into bounded contexts with strategic transformation (see Part III), even if the structure of the business logic is poor.

Once the bounded contexts have been established in the architecture, the second step can be tackled and that poorly structured business logic can be strengthened with domain knowledge through tactical transformation (see Chapter 8). In Figure 1-21 we extend the graphic for an MMI of 4–8 with path B.

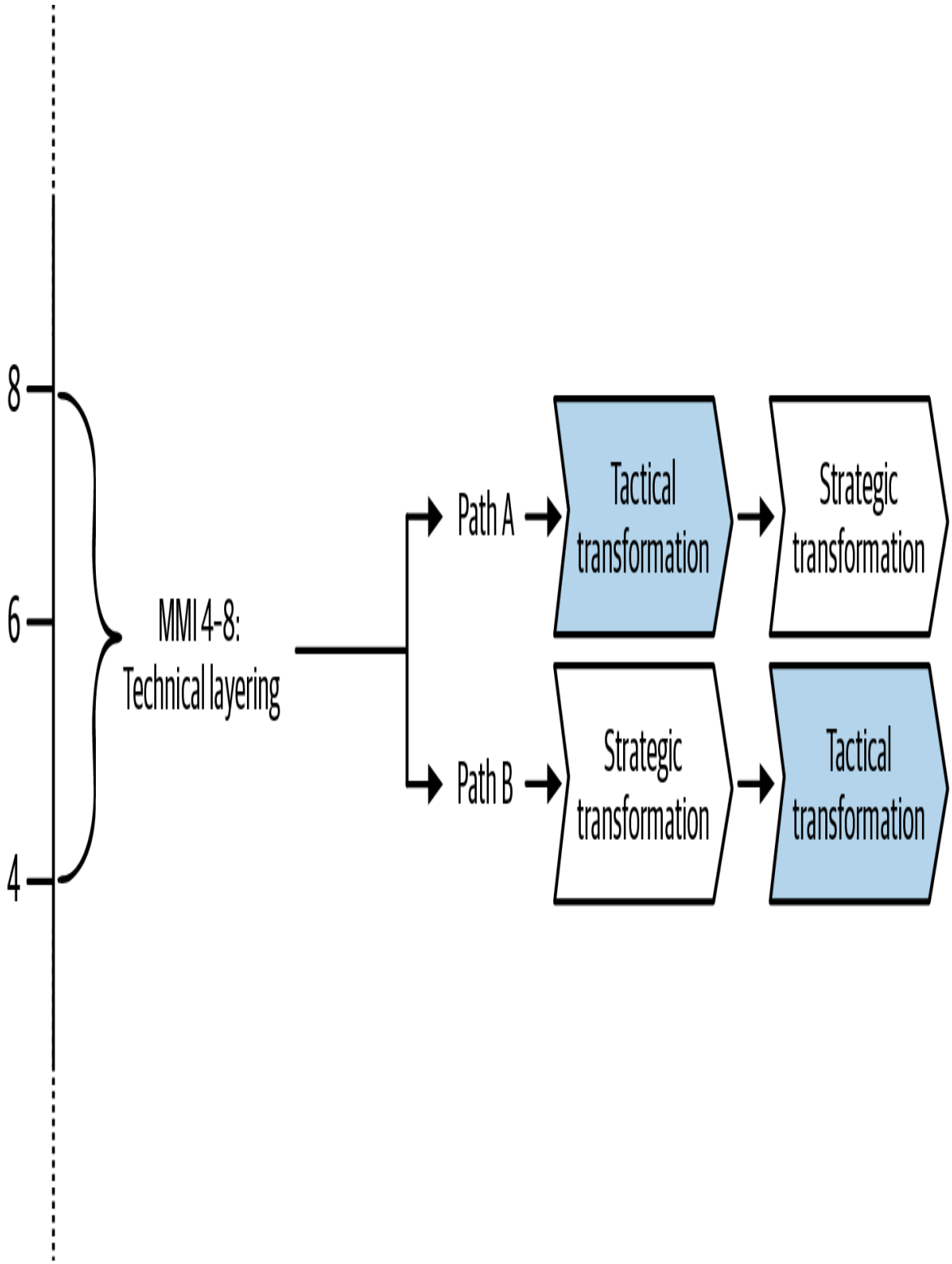


Figure 1-21. The MMI and the third path through transformation.

Path A or path B?

The transitions from a large, tangled domain layer to a domain model in which the beginnings of a domain-driven structure can be recognized are usually fluid, so that in some cases, it is not possible to clearly decide whether path A must be followed or path B can be taken.

If we find ourselves in such a situation, we generally recommend path B. We therefore start with the four steps of the strategic Domain-Driven Transformation (see Part III) and only start to strengthen the code with domain knowledge after splitting and decomposing the system into bounded contexts. Otherwise, although domain-driven, very large (i.e., monolithic) domain classes are created. As we said earlier, this doesn't have to be black and white—for example, after a first bounded context is extracted, it might make sense to work on this particular bounded context with tactical transformation before extracting the next bounded context.

A similar mix of the two paths can be useful if we start with path A: you start with some preparatory tactical steps (e.g., reducing coupling by splitting an active record into aggregate and repository), then carve out a part with strategic steps (e.g., extract a bounded context), and then apply tactical steps again (e.g., fight the model anemia in the newly extracted bounded context).

We will get to know case studies in the following chapters to make *both paths visible*:

- In Chapter 4 we will introduce a case study from the *banking* domain, which we will take up again in Chapter 8, because we have chosen *path A—tactical transformation first* for this case study. In the bank case study, we first transform tactically and then strategically.
- In Part III, a case study from the *car leasing* domain will make an appearance. For the leasing case study, we have chosen *path B—strategic transformation first*. It is first transformed strategically and then improved tactically.

MMI 8–10: Domain-driven modularization

If a system is rated between 8 and 10 by the MMI, it already has a domain-driven modular structure. The architecture of a system that is in such a good state should look schematically as shown in Figure 1-22.

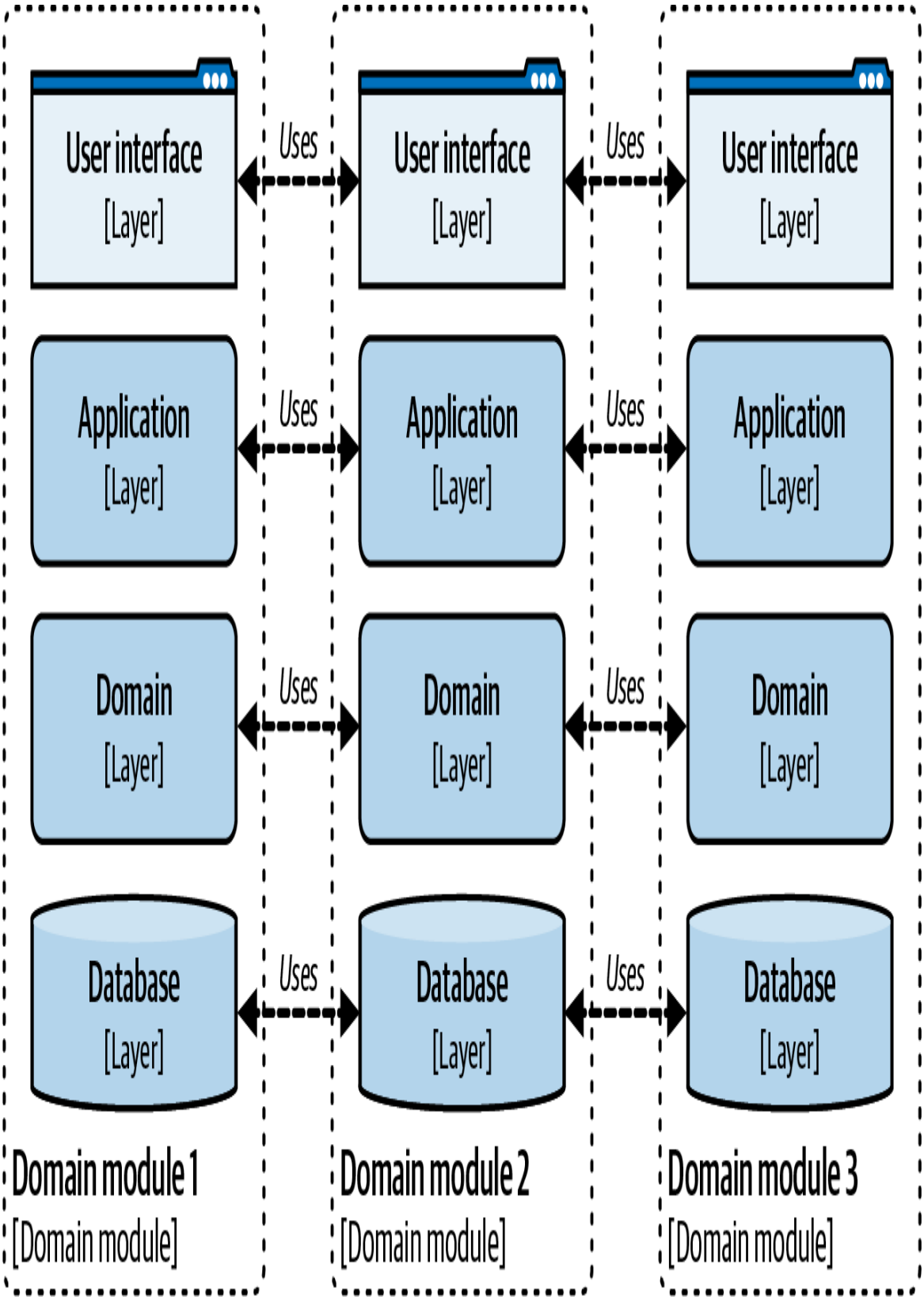


Figure 1-22. Domain-driven modularization: technical layers inside domain modules.

The dashed double-headed arrows between the modules in Figure 1-22 indicate that even with these well-formed systems, there are some relationships that need to be straightened out. It may also be necessary to move some classes so that the domain-driven decomposition is really successful. Often, the domain modules that are implemented in the system aren't real bounded contexts yet.

A run-through of the four steps of the strategic Domain-Driven Transformation is worthwhile for these systems, because their structure is already almost at its goal and a sustainable decomposition can be achieved with just a few improvements.

Finally, Figure 1-23 shows the section for an MMI of 8–10. The complete diagram for all three levels of the MMI, 0–4 (big ball of mud), 4–8 (technical layering), and 8–10 (domain-driven modularization) can be found at the beginning of this section.

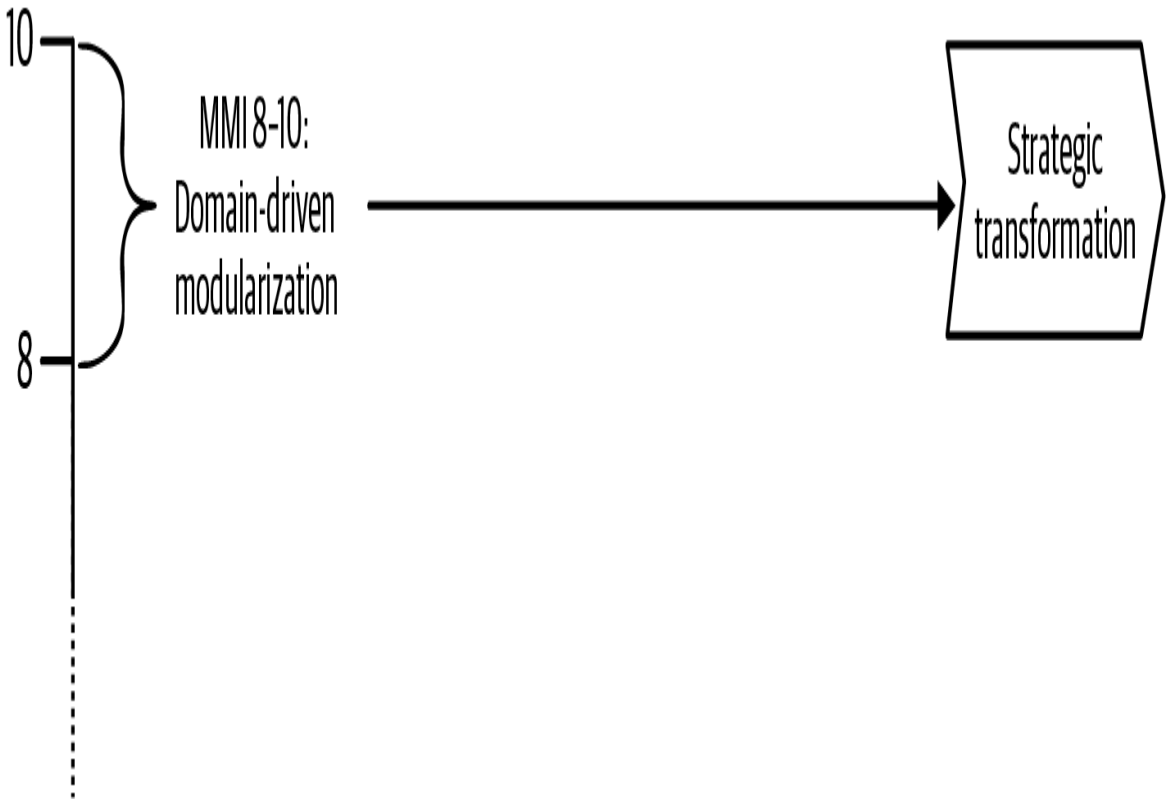


Figure 1-23. The MMI and the fourth path through transformation.

We describe the four steps of the strategic transformation in detail in Part III, with many more examples and detailed explanations.

Summary

In this chapter, we provided an overview of our Domain-Driven Transformation method and described the various procedures, depending on the state of the system and its current architecture. With this basic, albeit still abstract, overview, you can now move on to the other individual sections:

- If you lack the fundamentals or parts of the fundamentals, take a look at the individual chapters from Part I.
- If you suspect that you have a software system with an MMI below 4, then read through the measures in Part II and use them for your system.
- If you want to tackle the strategic transformation, then start with Part III.
- For a strategic transformation to be successful, usually not only does the structure of the system need to be changed, but also the structure of the teams that develop this system. How to achieve a *team-organizational transformation* is the topic of Chapter 9.

Have fun!

---

1 Disclaimer: transformation makes sense only to complex systems. To still give an overview of the method, this introductory example is oversimplified in many different ways: the domain is less complex, the architecture is less entangled, and the steps to be taken are easier than they are in real life. We nonetheless hope it provides a general idea of the method and invites you to dig deeper. The points that are rushed over here are explained more carefully, more thoughtfully, and more completely in the following parts of the book.

2 At a real container terminal there are other roles that we leave out here for simplicity.

3 To simplify things, we left out some of the process' details in this domain story.

Part I. Fundamentals

For a successful Domain-Driven Transformation, several different tools are necessary. Among these are Domain-Driven Design (DDD), Collaborative Modeling, and software architecture concepts, which we look into in this first part of the book. The following introduction reflects our understanding of how DDD came into being and how complementary techniques have been added to DDD over time.

When the book *Domain-Driven Design*—later known as the "blue book"—was published by Eric Evans in 2004 [Evans 2004], not all of its concepts were generally well understood outside of initiated circles. It wasn't until Vaughn Vernon's book *Implementing Domain-Driven Design*, called the "red book" [Vernon 2013], came out in 2013 that the ideas were understood by a broader audience. The advent of microservices, which emerged around 2012 as a new architectural style for scaling large software systems, additionally fueled the interest in DDD. This is because DDD not only has useful design patterns to offer with what is known as tactical design. DDD also offers a helpful tool for decomposing domains into subdomains and transferring the resulting domain-oriented structure to the modularization of software systems—whether monolithic or built from microservices: strategic design.

From the sheer number of publications that have appeared on Domain-Driven Design in the years following, you can see how important these concepts have become for software development. The "red book" in 2013 not only gave many good practical tips on how to implement according to DDD, but also started to add more terms and concepts. Then in 2015, *Patterns, Principles, and Practices of Domain-Driven Design* by Scott Millet and Nick Tune [Millet and Tune 2015] was published. Vernon has written other books (see [Vernon 2016]—which we translated to German

[Vernon 2017]—and [Vernon and Jaskuła 2022]), whose enhancements and insights we include in the chapter on DDD (see Chapter 3). Recently published books on the topic are *Hands On Domain-Driven Design* [Plöd 2020], *Learning Domain-Driven Design* [Khononov 2022], *Mastering Domain-Driven Design* [Junker 2025], and *Architecture for Flow* [Kaiser 2026]. Nick Tune's book *Architecture Modernization* (see [Tune and Perrin 2024]) was published at the same time as the German edition of this book. We were allowed to read it early, and Nick also gave us helpful comments on this text. From our point of view, our books complement each other very well, because Nick writes on a similar topic, but more on the enterprise level, while this book is more detailed at the individual services and integration level; that is, focused on the architecture of subsystems and the overall system.

Eric Evans has also written further texts on Domain-Driven Design since his first "blue book": in 2013 he published a 20-page article titled "Getting Started with DDD When Surrounded by Legacy Systems" (see [Evans 2013]), in which he introduces bubble contexts and four strategies on how to use them when modernizing legacy systems. Finally, in 2015, he wrote a 50-page abstract of his book titled *Domain-Driven Design Reference: Definitions and Pattern Summaries* (see [Evans 2015]). These works have also inspired and accompanied us on our way to this book on DDD and are referenced in the relevant places in the chapter on DDD (see Chapter 3). In the preface of the DDD reference, Evans thanks everyone who has already contributed to the topic of DDD and hopes for many new articles, blogs, and books to help the DDD approach continue to grow and thrive. We hope to contribute another building block to the DDD approach with our perspective on Domain-Driven Design and legacy systems.

EventStorming was introduced at conferences by Alberto Brandolini around the same time as Vaughn Vernon's first book in 2013 and became known, among other things, as a support for decomposing software into independent modules [Brandolini 2021]. A little later, Stefan Hofer, together with Henning, started to publicize *Domain Storytelling* as another method for decomposition, which eventually resulted in their book of the

same name (see [Hofer and Schwentner 2022]). EventStorming and Domain Storytelling belong to a family of lightweight methods known as *Collaborative Modeling*. We dedicate Chapter 4 to these methods in the fundamentals part of the book.

In this first part of the book, we describe the basics of *software architecture* that are important for legacy systems and their decomposition with DDD. Chapter 5 will show how we use architectural styles such as monoliths; microservices; Hexagonal, Clean, and Onion Architectures; service-oriented architectures; and Self-contained Systems. The evaluation of software architectures using the *Modularity Maturity Index (MMI)* will also be presented here. Our view of the architecture styles, repeatedly discussed in connection with DDD, and their evaluations, can be found in Chapter 5.

To give the chapters on DDD, Collaborative Modeling, and software architecture a common fundamental theme, the first part of the book begins with a chapter on *complexity* in software development. And ultimately, this first part concludes with a section on the advantages and disadvantages of different *approaches* to consider when rebuilding a system.

Chapter 2. Mastering Complexity

> *The most fundamental problem in software development is complexity. There is only one basic way of dealing with complexity: divide and conquer.*
>
> —Bjarne Stroustrup [Stroustrup 2024]

This chapter will discuss where complexity comes from in general and in a legacy system in particular, how to deal with necessary complexity, and how to avoid unnecessary complexity.

To classify how complexity can be dealt with in software development, the subtitle of Eric Evans' book *Domain-Driven Design* is a good guide: *Tackling Complexity in the Heart of Software*. Evans promises the reader that the approach he describes reduces complexity in software (development) or at least makes it manageable. Domain-Driven Design helps on different levels. To distinguish between these levels, we use Figure 2-1, which summarizes the various aspects of complexity that you'll encounter in software development based on [Lilienthal 2008] and [Lilienthal 2019].

The next two sections explain the two dimensions from Figure 2-1 and their interaction. As the book proceeds, you will be see that the basics that are introduced in the following chapters offer methods and heuristics for some of the complexity sources listed in Figure 2-1.

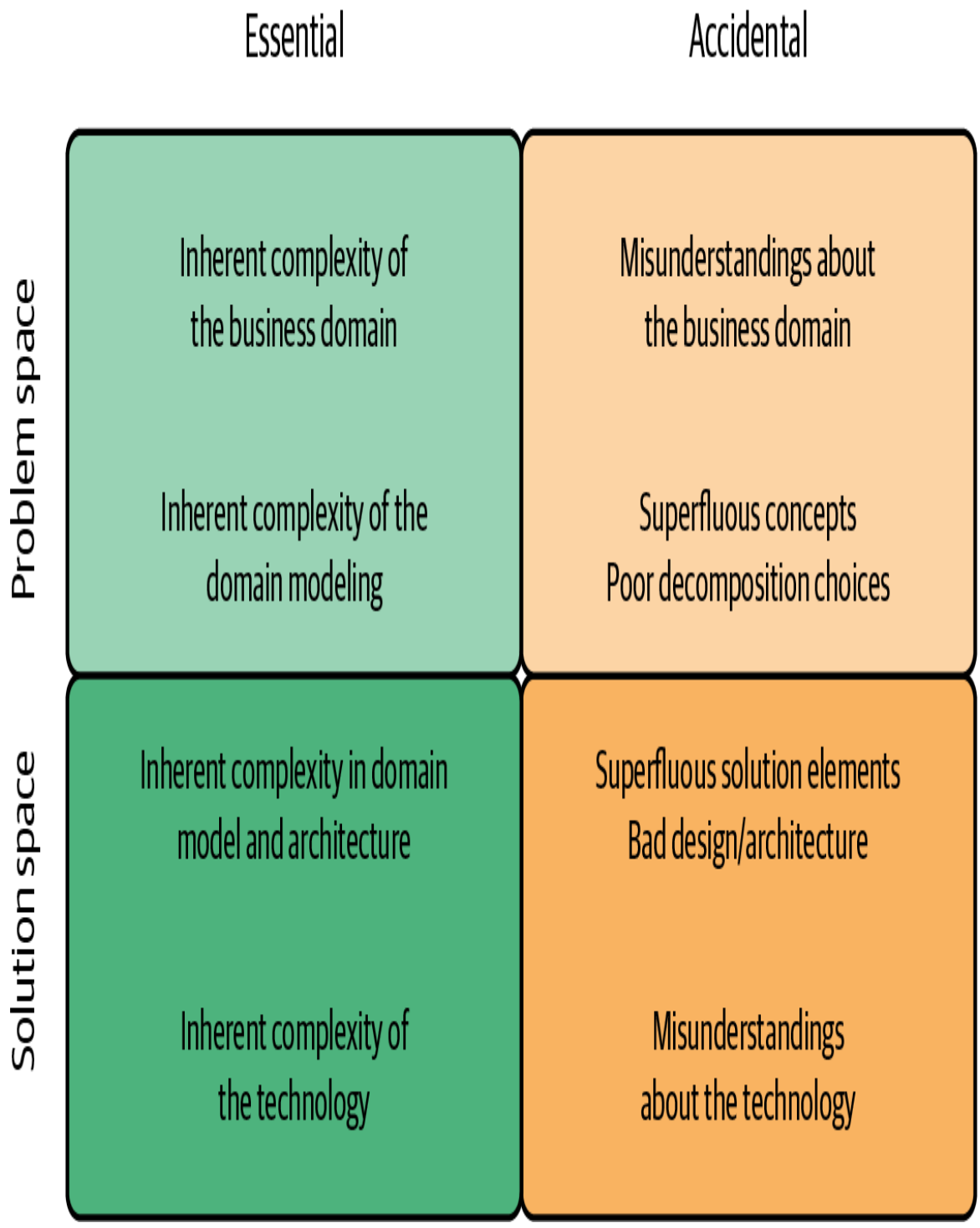


Figure 2-1. Complexity and its dimensions.

Origin of Complexity: Problem and Solution Space

The origin dimension of complexity in software development is shown in Figure 2-1 on the vertical axis. This complexity arises from the desire to build software for a domain; that is, to move from the domain—a part of the real world (the *problem space*)—to software with appropriate business processes (in the *solution space*). Thus there are two sources:

Problem space

> The complexity of the domain for which the software system was built —what we refer to as *problem-inherent complexity*

Solution space

> The complexity that arises in the implementation during modeling, in the architecture and design, and in the use of the chosen technology— what we refer to as *solution-dependent complexity*

AT THE CONTAINER TERMINAL: COMPLEXITY IN PROBLEM AND SOLUTION SPACE

Recall the container terminal example from the introduction. One source of complexity in the problem space here is the large number of roles that must work together (and some of the roles weren't given). Each role has its own specific tasks when working with the containers —again, with very little of their actual diversity in that simplified example. To appreciate the full complexity of the problem space at the container terminal, it would be necessary to go into more detail than we did.

One source of complexity in the solution space is easy to identify, despite the many simplifications we have made: the large amount of automation and the extensive use of technical equipment that is achieved at a container terminal today. All technical devices, such as ship-to-shore cranes, storage cranes, straddle carriers, etc., must interact with the software system via interfaces, and this interaction is largely automated.

In past decades, the software development profession has repeatedly found that it is very difficult, if not impossible, to estimate the complexity in the problem space at the beginning of a project and write it down in a requirements specification. This is why projects often take much longer than planned, why the waterfall model works so poorly, and why Agile software development (unfortunately often misunderstood) became so popular.

When understood correctly, Extreme Programming, Scrum, Kanban, and the like can help, because, on the one hand, they try to capture the complexity in the problem space in smaller increments, step-by-step, intertwined with the implementation. On the other hand, Agile software development helps teams react to *changing requirements* by building the software in evolutionary steps.

In Domain-Driven Transformation, the team also moves in small steps. The steps of the transformation are intertwined with the steps to build new features, thus enabling us to deal with changing requirements that are a moving target.

Type of Complexity: Essential Versus Accidental

In Figure 2-1, the type of complexity is found on the horizontal dimension with the two terms *essential* and *accidental*.[1] Which parts of complexity are described with these two terms?

Having a skilled and experienced team in place and a product owner well-versed in their domain, there is a good chance that the team will deliver a good software solution. A good software solution includes the appropriate essential complexity and avoids accidental complexity. If the solution chosen by the development team is more complex than the actual problem, then the development team has not been able to properly grasp the essential complexity, and the solution is too complex, thus not good. This difference between better and worse solutions is referred to as *essential complexity* and *accidental*[2] *complexity*.

AT THE CONTAINER TERMINAL: ESSENTIAL AND ACCIDENTAL COMPLEXITY

For example, let’s imagine that the container terminal is located in a free port, which means that goods that arrive at the container terminal by ship and leave by ship do not have to be cleared through customs. Only goods that are delivered or transported by truck or train have to be cleared through customs.

If third-party software was introduced at the container terminal in which the customs process is *always* carried out for all goods, then this third-party software would increase complexity for its users at the container terminal in the free port. Users would face customs processes every time containers arrive and leave the terminal, even if the containers do not cross a border, and the customs processes would actually not be necessary. The users cannot concentrate on the essence of their work, but have to carry out superfluous work steps (filling out customs forms for containers that do not cross a border) in the software.

Essential complexity is the kind of complexity that lies in the very nature of a thing; that is, it is part of its essence. This complexity can never be resolved or avoided by a particularly good design. If you want to build software for a complex domain—for a domain in which a high level of complexity is part of its essence—then the complexity in the software in the solution space must be correspondingly high. The software should be as simple as possible, but no simpler.

AT THE CONTAINER TERMINAL: AS SIMPLE AS POSSIBLE, BUT NO SIMPLER

For example, the software solution for the container terminal seen in Chapter 1 has greater essential complexity than the software for managing club members in a rowing club that has only the requirement to store the names, dates of joining, and contact information.

Sources of Accidental Complexity

Accidental complexity, unlike essential, is not necessary. It is a result of the following:

- From *misunderstandings in the analysis of the domain*, so that the domain appears more complex than it actually is[3]
- From *changes in the domain and the developer's understanding of the domain*, so that a solution that used to be appropriate for yesterday's problem is not appropriate for the problems faced today
- Because developers believe that users need support for certain work steps, although they work quite differently in reality, the developers *build things that nobody needs*
- Because developers make decisions that lead to *ill-fitting design and architecture*, or because design and architecture erode over time through maintenance, change, and ignorance
- Because developers use unsuitable or obsolete *technology* that must be self-managed, needs to be replaced, or because they don't use the technology as it was intended

If a simple solution is not found during development due to lack of knowledge, experience, and/or careful consideration of the problem space,

the software system will be unnecessarily complex. Examples of this are multiple implementations, addition of unneeded functionality, and the ignorance of software technical design principles. Accidental complexity can also be deliberately introduced by developers. For example, this is the case when software developers use a new technology only because they want to gain experience with it, not because it is needed.

Decision Areas of Software Architecture

To make the different aspects of complexity manageable from an architectural point of view in the book, we divide the diagram in Figure 2-1 into the following four areas: requirements analysis, domain modeling, domain architecture, and technical architecture. The methods and techniques available in these various areas can be applied to deal with the various sources of complexity (see Figure 2-2).

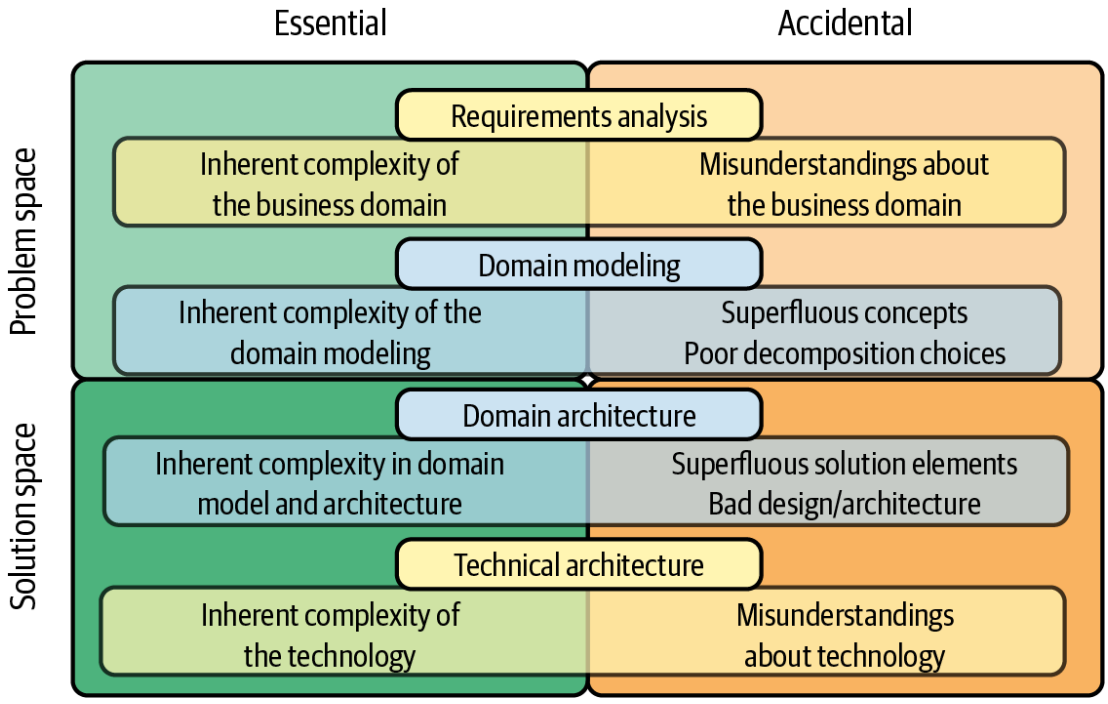


Figure 2-2. Areas of software architecture that can be applied to deal with the sources of complexity from Figure 2-1.

In Figure 2-2 you can see that requirements analysis and domain modeling in the problem space act on both the essential and the accidental side (upper

half of the figure). Domain and technical architecture impact essential and accidental complexity in the solution space (lower half of the figure).

DDD offers approaches to requirements analysis that are well suited in an Agile context. More on that in Chapter 4.

Dealing with Essential and Accidental Complexity

The methods and techniques of requirements analysis, domain modeling, domain architecture, and technical architecture are used both in new development and in the transformation of legacy systems to distill the essential complexity and eliminate as much accidental complexity as possible in these systems (see Figure 2-3).

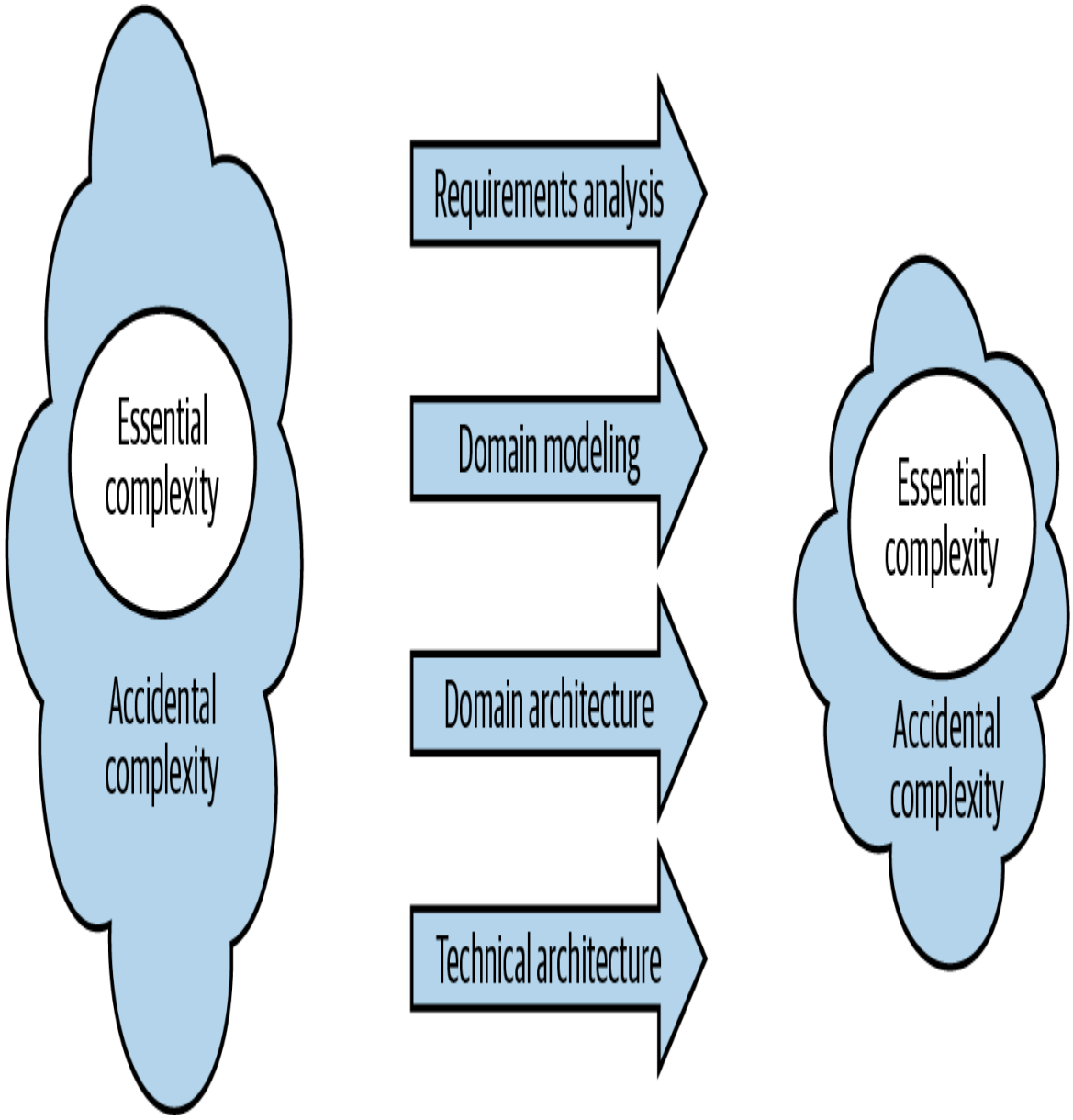


Figure 2-3. The right decisions in the areas of software architecture from Figure 2-2 reduce the accidental complexity.

It is impossible to completely resolve the accidental complexity even with the best methods, because software development and software maintenance happen in a continuous learning process. This learning process includes the fact that developers repeatedly misunderstand parts of the domain and the technology, and they design or implement superfluous or poor solutions for subareas.

Another reason why accidental complexity increases is that the domain changes over time. On the one hand, this change can come from the business alone, because business areas evolve. On the other hand, it often happens that the software itself triggers changes because users discover new possibilities for their business and rethink their work and processes. This means that previously optimal (or good) designs lose their sharpness, and accidental complexity creeps in as domain concepts and processes become refined over time.

> **WARNING**
>
> A typical mistake that a team can make is to consider a domain as easy to implement because they gravitate toward requirements that they already understand well. It's likely that the team will start with an overly simplistic (typically CRUD) solution. By the time the team realizes the solution approach is not appropriate, it is already hard to change and full of accidental, solution-dependent complexity. Vaughn Vernon calls this "easy becomes complex and can't be solved with simple."[4]

Therefore, even with perfect implementation at all levels in Figure 2-3, a certain amount of accidental complexity will remain.

Complexity in Legacy Systems

Legacy systems that have a number of years under their belt and that have been maintained and modified by various generations of developers and architects are likely to have a lot of accidental complexity. Reasons for this are:

- Over the years, the users' work, and with it the domain, have changed. New functionality has been added to the software to help users with their new or somewhat changed tasks as well. But the *domain modeling* and the *domain-driven modularization* of the software system in the architecture have not been adapted to this shift, or not sufficiently.

- Over the years, the IT strategy of the company and the importance of IT systems to the company has changed. Many design decisions have been made at a time when economies of scale (a.k.a. reusability) were super important and when IT was seen as a cost center. These designs may no longer be up to the challenges of economies of speed and the perspective of IT as a strategic asset (see [Hohpe 2020, chap. 35]).
- Over the years, the developers and architects of the system, due to time pressure, limited understanding, or disinterest, watered down the *original design and architecture* and increasingly entangled the individual parts, making them interdependent.
- Most legacy systems were built around *large domain concepts* rather than streamlining them into bounded-context-based domain-specific concepts, because the architects and developers did not know (or ignored) the ideas of strategic design.
- Every *technology becomes obsolete* in the software development discipline and needs to be replaced by something new every 10 to 15 years. With legacy systems, the right time is often missed.

When Evans' book was published in 2004, he was concerned with new software development. So the question was, in essence, "What do I need to do to make my *new* software system domain-driven in design?" In this book, the ideas of Domain-Driven Design are applied to *legacy* systems. Over the past few years, we could observe that DDD provides concepts and guidance that are ideally suited for the necessary redesign and sometimes extensive rebuilding of legacy systems.

Key to this is gradual and careful modernization using a combination of DDD and other tools, which we introduce over the course of this book.

The Fundamental Methods

The following three chapters on the fundamentals present our understanding of Domain-Driven Design (see Chapter 3), Collaborative Modeling (see Chapter 4), and architectural concepts (see Chapter 5). These three topics cover the complexity space that was introduced at the beginning of this overview chapter with Figure 2-1 and related to architecture with Figure 2-2. In Figure 2-4 you can see where each comes into play.

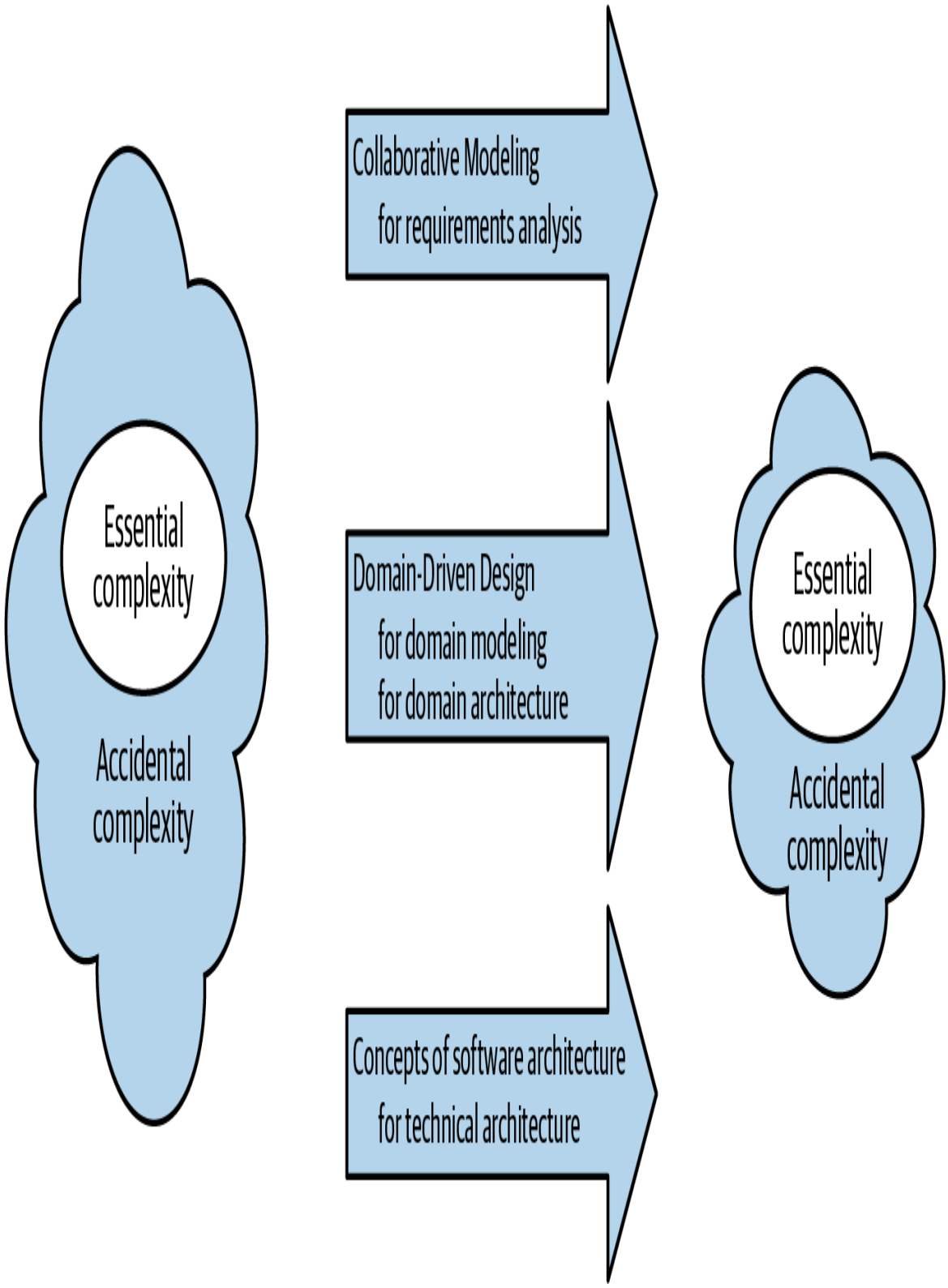


Figure 2-4. Methods to reduce complexity.

Collaborative Modeling covers the area of requirements analysis that gives access to business processes, roles, and work items in the problem space.

Domain-Driven Design is concerned with the areas of complexity in the middle of Figure 2-2. DDD has concepts to offer in the problem space as well as in the solution space. DDD deals exclusively with the domain-oriented parts of software development, which include domain modeling in the problem space, and design and architecture in the solution space.

Domain-Driven Design generally avoids introducing issues related to technology, because it doesn't recommend specific implementation details. Further, it avoids introducing misunderstandings about the problem space domain because it doesn't attempt to think for you. It's simply a set of tools accompanied by a philosophy and approach to addressing the activities with respect to gaining a deep understanding of any sphere of domain-specific knowledge. This is what makes DDD so broadly applicable and long-lasting.

To ensure that the technology still works well, architects and software developers have described *architectural concepts* over the past few decades that help us reduce the accidental complexity in the solution space.

What's Next?

So in the next chapters, let's take a look at the Domain-Driven Design, Collaborative Modeling, and architectural concepts that are needed as a foundation for Domain-Driven Transformation.

---

1 Of course, we cannot use the terms and their connection to complexity without quoting [Brooks 1986] and mentioning that Domain-Driven Transformation is no silver bullet either.

2 Accidental complexity does not necessarily mean that it was added by accident. Unfortunately, many developers build intentional complexity that is not essential to solve the problem (and is thus accidental).

3 For the readers who are familiar with the Cynefin framework, this happens when the problem is classified in the wrong Cynefin domain and therefore unsuitable practices for the solution

are chosen; that is in fact being in the confusion (a.k.a. disorder) domain. For example, the problem is conceived as complex, although it is in the complicated domain. (Warning: the terms *domain* and *complex* have different meanings in Cynefin than in DDD and this chapter. See [Kurtz and Snowden 2003] and [Snowden and Boone 2007].)

4 See Vernon's X post at *https://x.com/VaughnVernon/status/1815477838885339597*.

Chapter 3. Domain-Driven Design

> **TIP**
>
> This chapter presents knowledge that is foundational for Domain-Driven Transformation. If you are already familiar with Domain-Driven Design, you can skip this chapter and come back to it later for reference.

Domain-Driven Design (DDD) is a philosophy of software development that places the business domain at the center—not only during requirements analysis, but especially during modeling (see "Modeling in Code") and programming. This chapter is a distillation of our view of the DDD knowledge that was gathered over the last decades—from both literature and contributions of the community—with a special focus on legacy modernization. So as to not only present abstract theory, we describe the concepts with another concrete case study.

MODELING IN CODE

Note that "modeling" as expressed here is not considered a separate step from programming, such as when drawing UML diagrams or by using some other graphical notation.

Of course, programming is where code is written. In fact, there might be no diagrams (pictures) drawn at all during modeling. It could be that modeling happens with only the use of programmed tests that are first red and then green as the domain model is progressively implemented. The resulting programmed model *is the model itself* and the modeling took place as a pure programming exercise.

There is nothing wrong with doing part of the modeling on a whiteboard or using drawing tools, and even Collaborative Modeling tools as presented in the next chapter. Yet the point is not to consider "modeling" as a separate step that uses drawings to represent the entirety of the creative aspects and relegates programming to a mechanical step that requires relatively little thought.

Consider modeling and programming as a combined creative discovery and implementation step. This is referred to as *modeling in code*. After all, the goal is valuable working software, not drawings.

Language, events, actions, means of work, and structures of the domain are deliberately encoded into the software so that the users get the best possible support for their work.

DDD Concepts Against Complexity

Domain-Driven Design can not only help teams avoid accidental complexity when creating new systems, but also—and this is the topic of this book—reduce the complexity of legacy systems.

Figure 3-1 lists the different DDD concepts that help reduce the complexity in the problem space and in the solution space to the essential part. DDD

thus helps teams move their systems from the left (a state with a lot of accidental complexity), to the right (a state with little accidental complexity and a focus on the essential complexity).

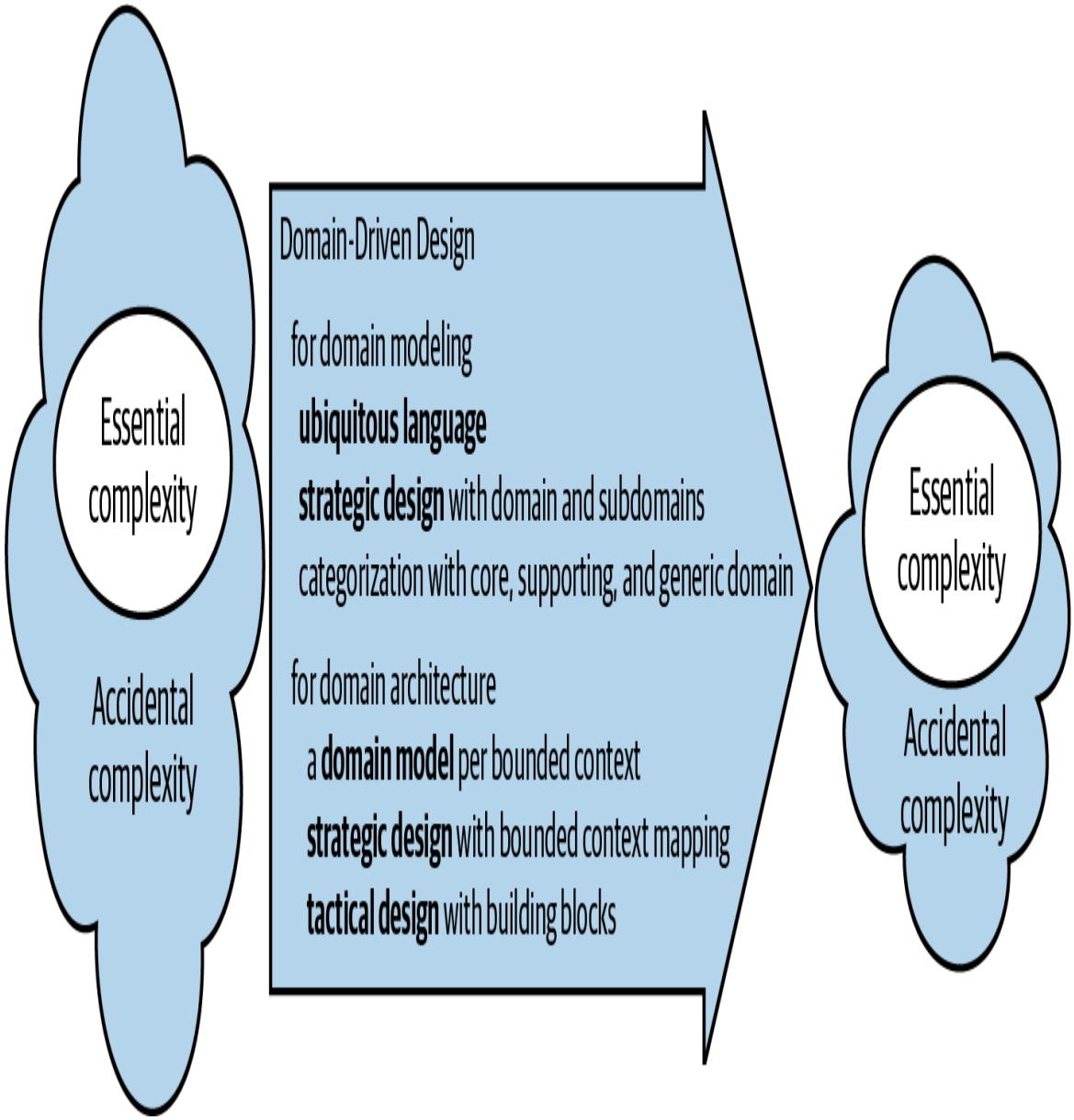


Figure 3-1. Complexity and DDD.

TIP

This chapter explains the individual concepts and methods of DDD, with a particular eye on legacy software. If you are already familiar with these concepts and methods, we recommend that you jump to Chapter 4 after looking at the following case study example, where will discuss the expansion of DDD with methods for requirements analysis.

CASE STUDY “CINEMA”

ANOTHER EXAMPLE: LEGACY SOFTWARE IN THE CINEMA

The *Clintwood Cinemas Corp.* is a large chain that owns cinemas in many big cities. Clintwood is very traditional, and both its processes and its software are a bit outdated. The company operates with the legacy software *Cine2001*, which is used at the movie theaters of Clintwood for scheduling shows and selling tickets (see Figure 3-2).

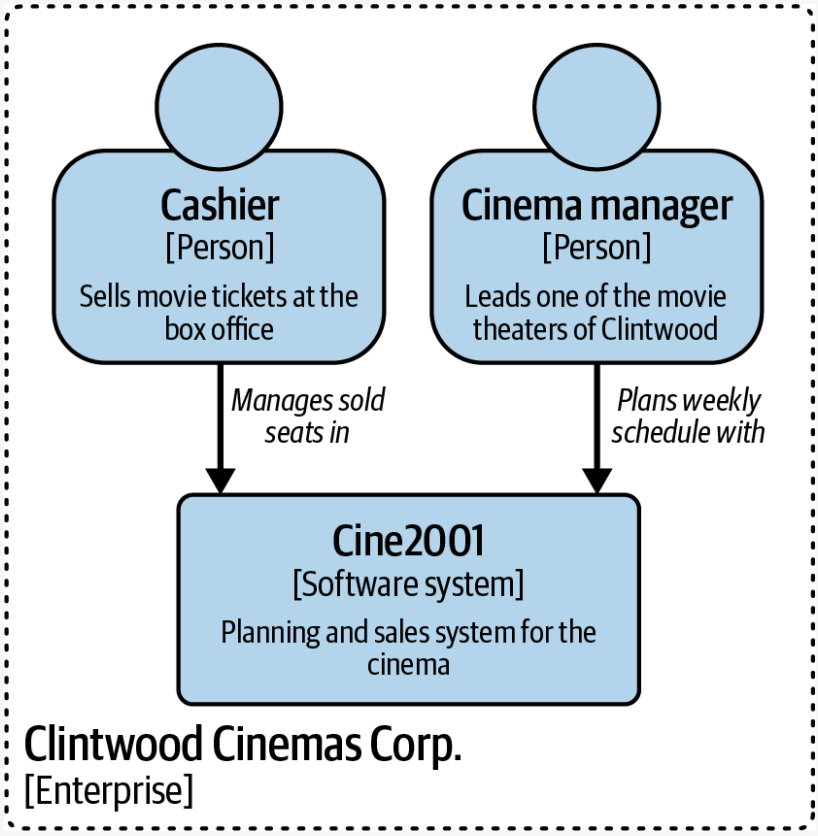


Figure 3-2. C4 context diagram of Cine2001 (for more information about the C4 Model notation, see Chapter 5).

So far, each Clintwood movie theater operates as follows. Every week, the cinema manager works on creating a new weekly schedule with all the films that will be shown during the next week. This weekly schedule is always released on Thursday, and after the release in Cine2001, tickets can be sold for the shows in the new cinema week. The sale of tickets is carried out by the cashier in the box office directly in the cinema. For this purpose, Cine2001 offers the cashier seating

plans for all shows, where they can reserve seats for, and sell tickets to, the moviegoers.

The founding CEO has retired, and the new management wants to open new markets for the Clintwood Cinemas Corp. They want to expand and be able to open new sites relatively quickly.

Cine2001 cannot cope with these adaptations in its current architecture. The technical consultants who are brought in recommend to the management that Cine2001 be converted into a microservices architecture (see Chapter 5) so that both scaling requirements and additional functionalities for the accelerated opening of new movie theaters become possible. To put Cine2001's architecture on a new foundation, the consultants propose an analysis with Domain-Driven Design.

Let's look at the recommendations of the consultants in detail.

DDD and the Business Domain

For developers to be able to modify a piece of software, they must understand the domain—because only what the developers understand can be implemented correctly in the software (see [Brandolini 2021]). This learning process about the domain happens most quickly if the developers can learn directly from the domain experts.

To learn comprehensively, the developers need a group of domain experts who perform different tasks with the software system in the problem space and also use other systems. Only when all or most of the tasks are understood and their interrelationships are known does the problem space for which the software was built become visible in its entirety.

Other useful discussion partners can be the product manager, product owner, and similar roles. With the product manager, you can usually clarify the question of what the software is good for; in other words, how it contributes to the success of the company. The product owner (who is, in

the authors' experience, often also one of the domain experts) "is accountable for maximizing the value of the product" and "effective product backlog management" [Schwaber and Sutherland 2020], and often has a lot of domain knowledge to be able to do that.

CASE STUDY “CINEMA”

IN THE CINEMA: WHO IS INVOLVED?

At Clintwood, with its legacy system Cine2001, one of the sites is chosen as a reference movie theater—the *Metropolis*. At this theater, the cinema manager and two box office employees (also called cashiers) are available as expert users.

The cinema's staff also interacts with people outside the cinema (see Figure 3-3).

In Figure 3-3, in addition to the cinema manager and the cashiers as internal employees, the moviegoer and employees of the film distributor and the advertising agency can also be seen.

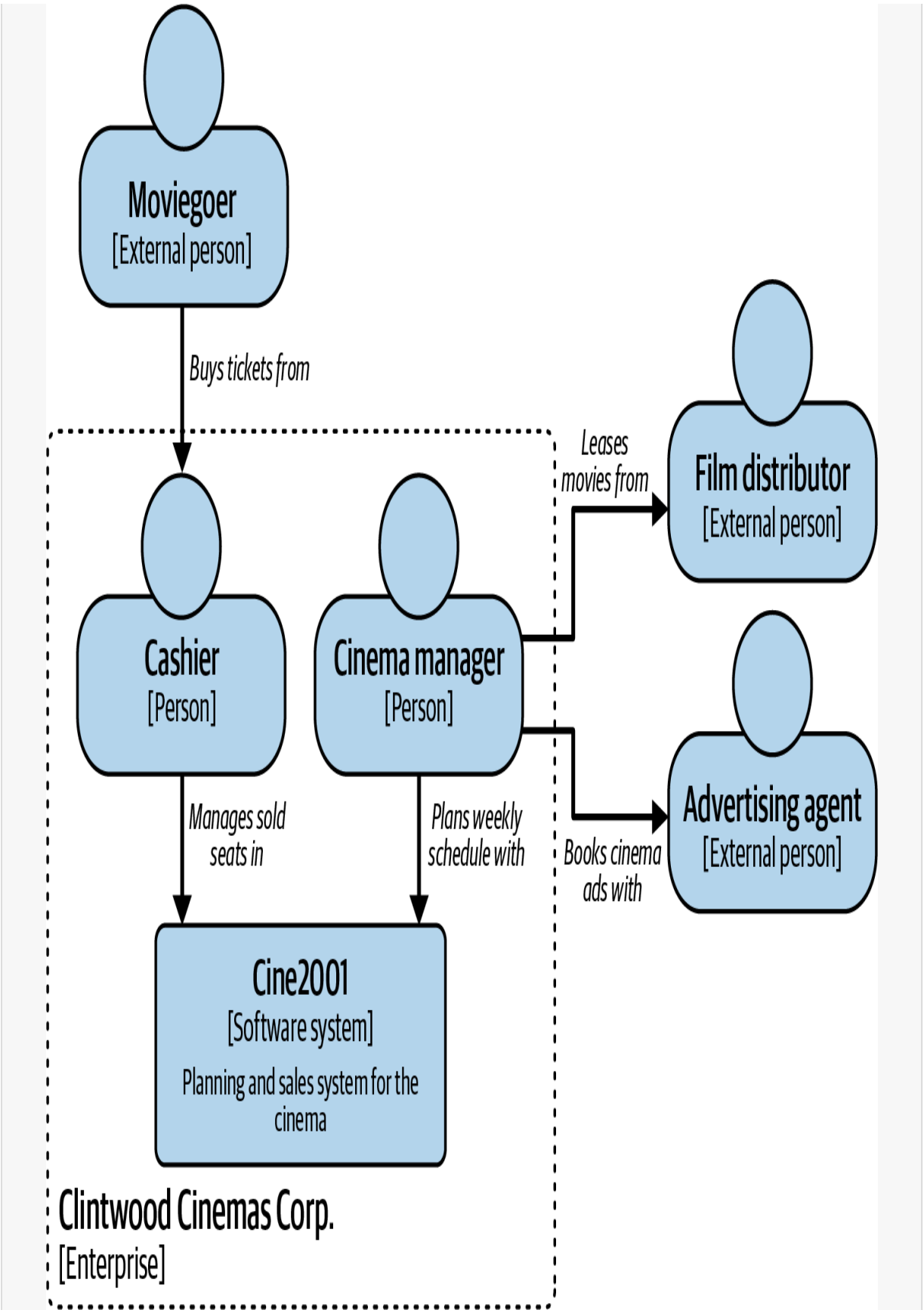

Moviegoer
[External person]
Buys tickets from
Leases movies from
Film distributor
[External person]
Cashier
[Person]
Cinema manager
[Person]
Manages sold seats in
Plans weekly schedule with
Books cinema ads with
Advertising agent
[External person]
Cine2001
[Software system]
Planning and sales system for the cinema
Clintwood Cinemas Corp.
[Enterprise]

Figure 3-3. C4 context diagram of Cine2001 with external people.

This starts with communication among domain experts and software developers as domain modeling begins. Developers dig into the business domain, with domain experts learning and using accurate language that describes the business concepts that are specific to related areas of the problem space (see the following section).

Chapter 4 describes lightweight methods that help team members understand the full business problem space, and that can help make domain modeling even more efficient.

Ubiquitous Language

A foundational concept of DDD is that all participants in a software project develop a common domain-oriented language. Without collaboration between developers and domain experts, there is no DDD. And this very specific language, known as the *ubiquitous language*, is the key concept of this collaboration.

The ubiquitous language, common to domain experts and developers, is based on the business language of the domain experts, not on the technical language of the developers. The developers will likely still have some influence on the language because physical operations are moving into a new software realm, which will likely require some concepts that are not in the physical world. Yet developers must not interpret and translate business language to their own technical language within the software. The goal is to standardize the domain language in such a way that the domain terms from the problem space (domain) can find their equivalent in the solution space (software).

DDD recommends that this common language be used everywhere during the lifetime of a software development project. Everywhere here means in speech, in writing, in diagrams, and even in the source code. That is why this language is called *ubiquitous* language.

IN THE CINEMA: UBIQUITOUS LANGUAGE

Figure 3-4 shows the term "weekly schedule" as an example, which appears in the conversation as well as in the class diagrams and finally in the source code.

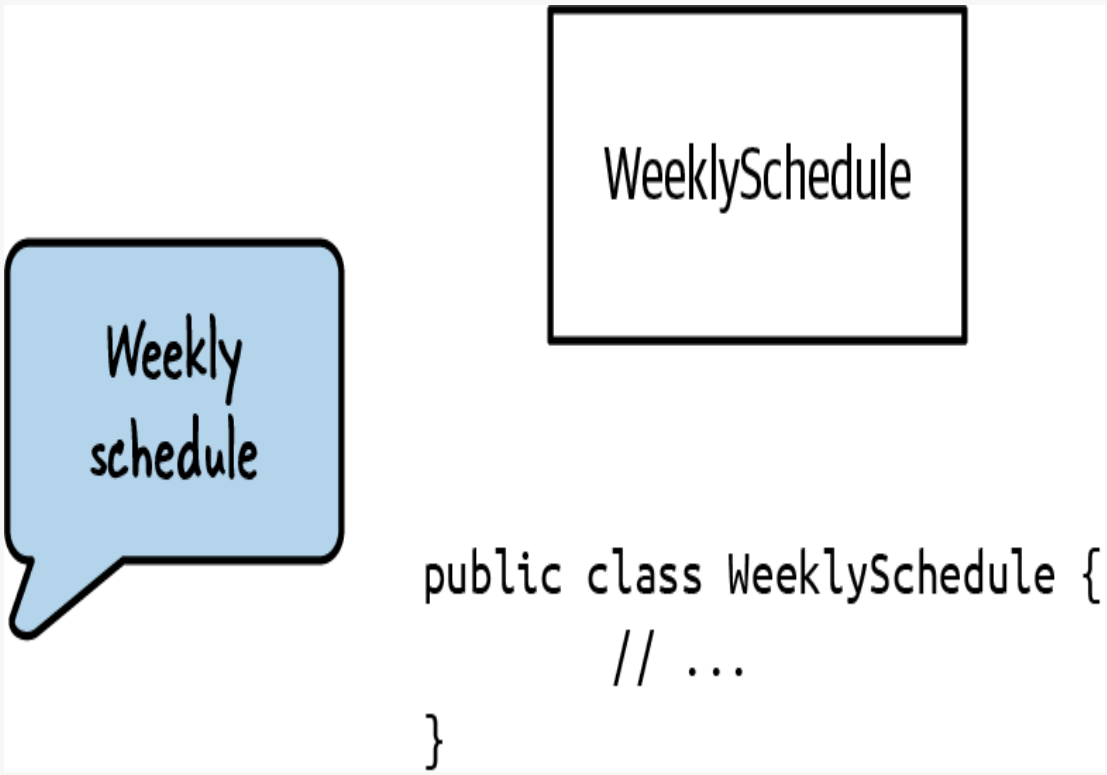


Figure 3-4. Ubiquitous language: the same words are used in conversation, in diagram, and in code.

NOTE

In a larger piece of software, there will be several ubiquitous languages—one ubiquitous language for each different area of the domain (called a bounded context; more on that in "Strategic Design in the Software"). So the word "ubiquitous" means that the same terms are used everywhere *inside a bounded context*, not that there may be only one canonical language for a domain that is used in all its subdomains.

Collecting domain terms

To learn the domain language (and thus help build the ubiquitous language), Eric Evans uses simple diagrams in his book and articles that collect the domain terms and relate them to each other. In a workshop setting, domain terms can be collected on the whiteboard or with sticky notes.

It is essential to keep in mind that one ubiquitous language is used within only one software domain model. If it is determined that multiple domain models are required by the solution, the terms should be separated by distinct models. This is the case even when terms overlap in models, because most often the terms mean something at least slightly, if not significantly, different in each model. It’s also likely that the same terms are implemented differently in each model. (More on this later.)

IN THE CINEMA: DOMAIN TERMS

In the first DDD workshop for Clintwood Cinemas, a number of domain terms emerge that are candidates for inclusion in the ubiquitous language: "weekly schedule," "show," "movie ticket," "seating plan," "seat," etc. The team is interested not only in the nouns, but also in the verbs: "to offer," "to pay," "to plan," etc. So, step-by-step a model of domain terms for a cinema emerges in the discussion with the domain experts. It can be seen in Figure 3-5.

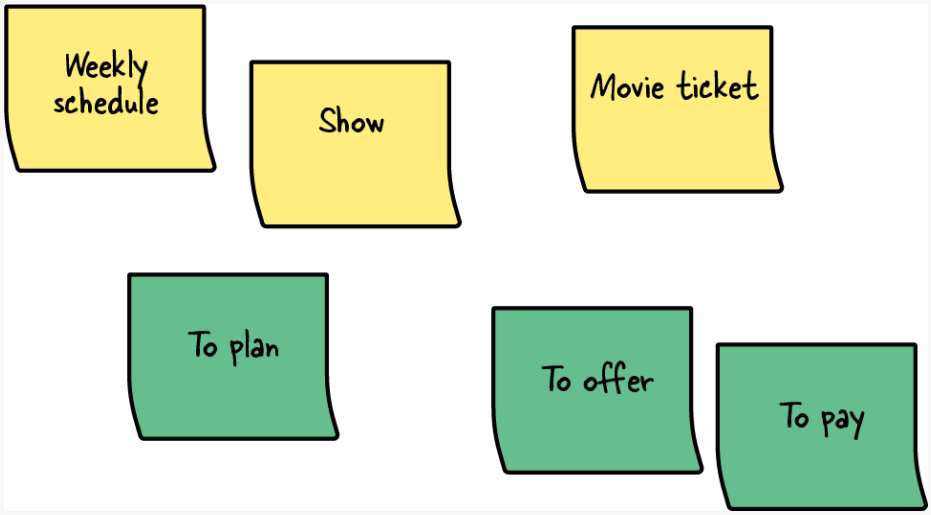


Figure 3-5. Ubiquitous language at the cinema: important terms.

When working on the ubiquitous language, the workshop participants have to be particularly careful that they actually collect the domain language of the domain experts and do not insert their own invented terms into the ubiquitous language. Time and again, the authors have observed that workshop participants used synonyms even though the experts never mentioned these terms.

IN THE CINEMA: SYNONYMS

At the Clintwood Cinemas workshop, the following questions, among others, are asked by the participating developers:

- Instead of "movie ticket," may we also use the term "cinema ticket" or just "ticket"? Does this mean the same thing?

  (Answer: More or less, but "movie ticket" is used at the ticket booth, and "cinema ticket" is used in accounting. The team has found a linguistic boundary. More on that in "Strategic Design on the Business Side".)

- Is "cinema program" the same as "weekly schedule"? Is it also the same as "showtimes"?

  (Answer: No, the "cinema program" is something very different from a "weekly schedule," but yes, "showtimes" is a synonym for "weekly schedule." For more on that see Figure 3-6.)

- Are "to pay" and "to buy" different activities?

  (Answer: Yes, buying a ticket involves choosing the seat and other cinema-specific activities, while payment is not something that is special for the cinema domain. More on that in "Generic and supporting subdomain".)

Consultants and developers cannot decide on their own which domain terms are the correct ones. This is possible only in discussion with the experts. Nonexperts carelessly deciding on their own can lead to misunderstandings in discussions with the domain experts, which can later lead to misinterpretations when the software is modified. The development of the ubiquitous language therefore requires a great deal of discipline from the workshop participants in order to avoid any kind of language confusion.

The domain experts do not always agree and also use synonyms or a fuzzy "domain vernacular." Working out a ubiquitous language therefore also means formalizing the language to such an extent that it becomes usable for software development.

Glossaries

To avoid language confusion, it can help to describe ubiquitous language terms worth explaining in glossary entries. These can also be initially noted on the whiteboard or on sticky notes.

CASE STUDY “CINEMA”

IN THE CINEMA: THE GLOSSARY

At Clintwood Cinemas, the glossary seen in Figure 3-6 is created.

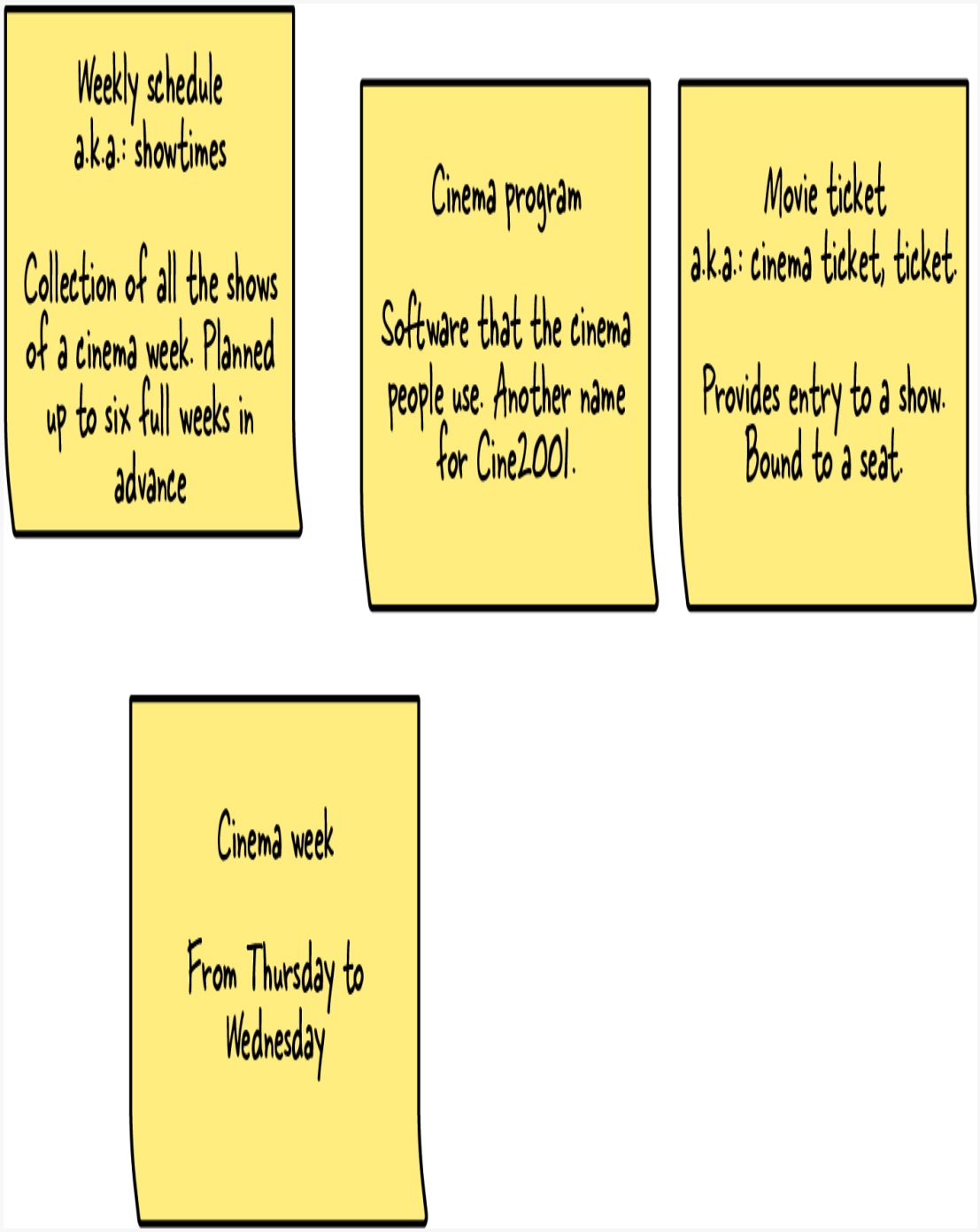


Figure 3-6. Glossary in the cinema.

Glossaries are an old method of recording terms and their meanings, as well as synonyms. The common criticism of glossaries is that they are not maintained in practice and quickly become outdated. However, if the glossary is viewed as a means to help learn about a domain and clarify terms, and may well be discarded later, then it makes a valuable contribution. We can therefore only warmly recommend the use of glossaries at the beginning of a transformation and later let them become outdated as the code becomes the anchor to the conversation rather than a separate document.

Pollution of domain language with technological terms

When analyzing legacy software, the situation is complicated by the fact that the domain language contained in the legacy software has, in most cases, been built without considering DDD principles and without domain experts.

In fact, most legacy systems contain vast amounts of supposed domain terms or even abbreviations that are foreign to the field, which have only found their way into the working world of the domain experts through the development of the legacy software or the use of standard software. Imagine that domain experts must look through glasses with the wrong prescription, so everything they see is hazy at best. That’s roughly what happens to domain experts when their familiar language is tossed aside by the developers who have never before used their language.[1]

IN THE CINEMA: POLLUTION WITH TECHNOLOGICAL TERMS

In discussing their work, the cinema manager shares two Excel files with the relatively cryptic names *BP_ADS.XLS* and *MOVIES-8.XLS*. After some discussion and the consultants' questions ("What is this in the cinema domain?") it becomes clear: these are actually domain concepts that also have domain-related names, "booking plan for advertising" and "guidelines for movies." So the workshop participants can give each of the two concepts a meaningful name that comes from the domain.

These two Excel files have no direct connection with Cine2001 itself, because the cinema manager has no way to import them into Cine2001 yet. The cinema manager just looks at them on their computer when creating the weekly schedule. But such terms also belong to at least one ubiquitous language.

While discussing the work of the cashier, the team also comes across a term that had moved away from the actual domain terms due to the use of Cine2001. When interviewed about finding and selling seats for a show, the cashier says, "Then I go to the auditorium list and open the auditorium for the show. That's where I look for the seats."

Fortunately, the cashier can still remember a time before the introduction of Cine2001, and in response to the question: "What did you do on paper in the time before Cine2001?" names the original domain term: "seating plan." For the cashier it became clear that they open the seating plan by clicking on the auditorium list in Cine2001. Since the introduction of Cine2001, they have had to use the wrong terms ("auditorium" and "auditorium list") because the developers had programmed it that way.

When trying to modernize and decompose a legacy system, it is important to rediscover the original domain language. To be able to make decisions that are well founded in domain terms during the decomposition process, the developers need to understand the real domain with its real terms. The corruptions caused by the legacy software should be resolved as best as possible. For this reason, the team includes the term “seating plan” in the ubiquitous language in the example.

English or the local language?

In non-English-speaking areas—like in our home country Germany—teams regularly encounter the question: should the developers use the local language (like German) or English for the domain terms in the code? Most teams discuss this topic very emotionally. It often comes down to these two points of view:

Everything in English

> Because the programming language keywords are English and being consistent is valuable.
>
> The disadvantage of this decision is that most terms can only be translated with a loss of information. Other terms cannot be translated at all, and then fantasy words are created because the domain terms in the local language are translated with the help of internet lexica and the like. The result is usually difficult to understand and ambiguous.

Everything in the local language

> Because the domain is in the local language, this way you stay closest to the meaning of the terms that comes from the domain.
>
> The downside of this decision is oddities, such as the German `gib` and `setze` instead of `get` and `set`[2] for accessor methods. Such translations do not help anyone and, taken to the extreme, lead to amusing stylistic blossoms: `Schlangenbaendiger` as a class name instead of `QueueHandler`.

These two positions strive for language purity and an attempt to employ a simple rule that always applies. In our opinion, this is the fundamental problem, because the world is not black-and-white or 0–1.

We recommend considering the following factors when deciding on the language:

- Is the domain *international* (such as is the case with aviation) or *local* (such as German tax law)?
- Do the developers all speak *the same language*?
- Everybody in the team will be affected, so they should find out together what works best for them. It's a team decision.

> **NOTE**
>
> If the idea of ubiquitous language is to be implemented correctly, then the answer to the question about words to be used for the domain terms in the code is not "local language words" or "English words" but "words that the domain experts themselves use and understand."

In practice, the domain experts typically speak a quirky mix of the local language with English (sometimes called Spanglish, Franglais, Denglisch, or similar), which then also characterizes the ubiquitous language. The terms are composed of words that are difficult for outsiders to understand, and which have a very special meaning in this domain. The task of the ubiquitous language is not to be a particularly pure or beautiful language, but one that both domain experts and developers working on a given bounded context can understand.

Strategic Design on the Business Side

To be able to discuss and identify the business capabilities of a software system and their relationships during a new development, DDD provides us with *strategic design*. Strategic design is about dividing a *domain* into

subdomains (described in this section), and distinguishing among *core*, *supporting*, and *generic subdomains* (see "Categorizing Subdomains—What Is Core?"). In addition, strategic design offers other concepts for the solution space (see "DDD and the Software Solution").

To divide the domain into subdomains, Vaughn Vernon uses diagrams with domain terms in his books (see [Vernon 2013], [Vernon 2016], and [Vernon 2017]). He represents each subdomain as a boundary containing related domain terms.

In the discussion with the domain experts and the development team, the consultants sort the different domain terms, draw subdomain boundaries around these terms, and give each subdomain a name. Usually, a language term belongs exclusively to one subdomain. Sometimes the same term is used in different subdomains. This happens when different actions are done with a central domain concept in different subdomains.

CASE STUDY “CINEMA”

IN THE CINEMA: SUBDOMAINS

Two subdomains can be identified at Clintwood Cinemas: “program planning” and “ticket sales” (see Figure 3-7).

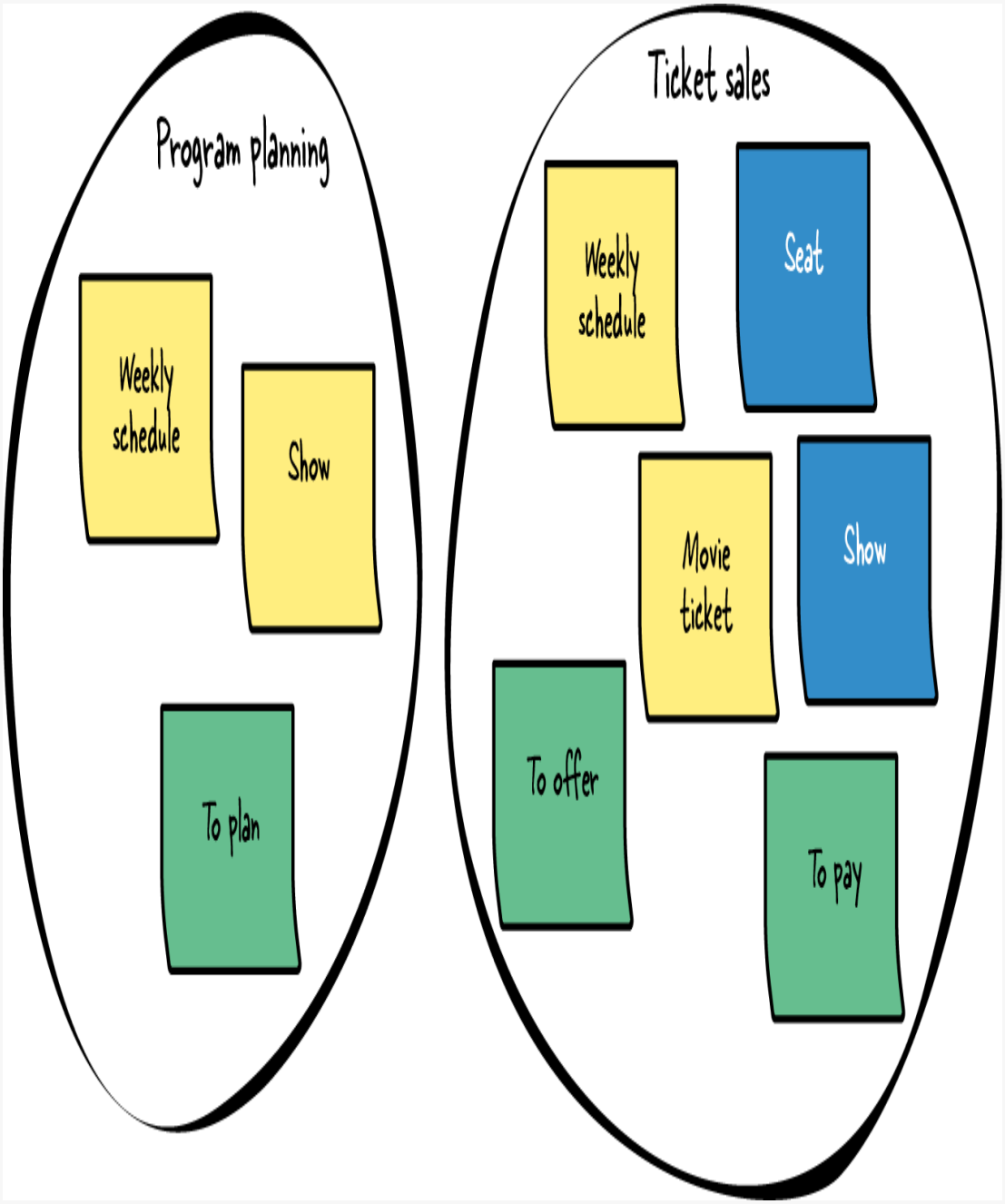


Figure 3-7. Terms in the domain landscape.

How does the team reach the conclusion that these two subdomains exist? Well, DDD provides the guidance to look for the domain concepts and terms and see how they are used in different ways.

Are there groups of words that are used together, but words from two different groups that are not used together? Then a language boundary exists between the groups of words, and they belong to different subdomains.

Are the domain experts using different names for the same basic concept? Are they using different definitions for the same term? In both cases it is likely that the terms belong to different subdomains (and thus ubiquitous languages).

IN THE CINEMA: HOW TO FIND THE SUBDOMAINS?

In the movie theater Metropolis, the concepts "to plan" (done by the cinema manager) and "to sell a movie ticket" (done by the cashier) are never used together. This is a strong indicator of the need for a language boundary. On the other hand, both the cinema manager and the cashier work on every weekly schedule. Do they both do the same with the weekly schedule? Or does each role have its own view and way of dealing with it?

The discussion with the domain experts reveals that the cinema manager works very extensively on the weekly schedule until they finally pass it on to the cashier and publish it for the moviegoer. The moviegoer looks at the weekly schedule to find out about the showtimes. The same term "weekly schedule" is used in both places, although when the cinema manager works on the weekly schedule, it contains considerably more information, and they are doing different actions with it than when the moviegoer uses the weekly schedule.

The cinema manager enters the advertising blocks and the advertising films scheduled there into their weekly schedule, so that they or the projectionists know what is to be shown and when. The version of the weekly schedule for the moviegoer does not contain the advertising blocks. If the moviegoer knew when ads were going to be shown, they would certainly not come to the theater until the actual movie started. The whole effect of advertising would thus be nullified.

So, the term "weekly schedule" is used in quite different expressions—this is a valuable indicator for splitting a domain into subdomains.

For more indicators that we have identified in recent years, see Chapter 10 in Part III, where we present the strategic part of our Domain-Driven Transformation methodology.

CASE STUDY “CINEMA”

IN THE CINEMA: SOME MORE SUBDOMAINS

Thinking a bit further, there will certainly be more subdomains at Clintwood that are not yet visible on the domain landscape in Figure 3-7:

- The cinema chain has "accounting" that relies on data from ticket sales to map the financial side of the business. While in ticket sales the term "movie ticket" is used, in accounting the term "cinema ticket" is more meaningful.
- There is also the subdomain "concessions," which takes care of the food and beverage requests of the moviegoers.
- The films are provided by an external film distributor. So "film distribution" is not done by Clintwood itself but is still an important external subdomain that other subdomains collaborate with.

The team collects the subdomains found so far on the whiteboard (see Figure 3-8).

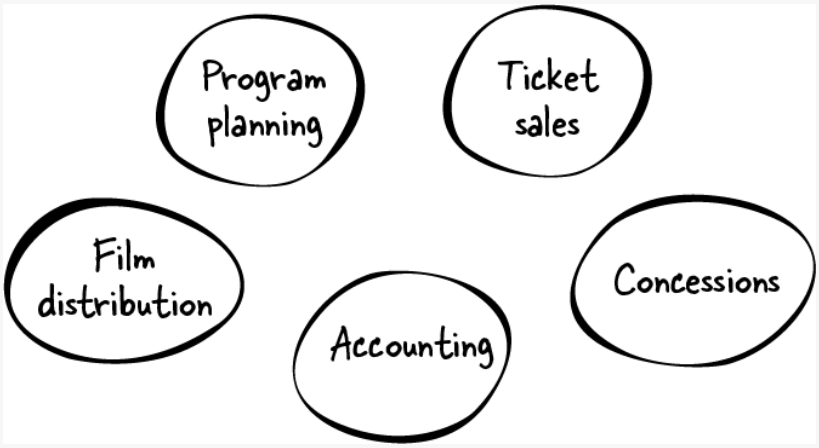


Figure 3-8. Subdomains of the cinema.

There are even more subdomains than these; more on that later. For the sake of simplicity, we will leave them out here—because the first step here is to explain the concepts of DDD, not to model a real cinema.

Categorizing Subdomains—What Is Core?

Having identified the various subdomains, DDD gives us a means to determine the importance of the subdomains by distinguishing them as *core domain*, *supporting subdomain*, and *generic subdomain*. The DDD literature calls this "distilling the core" (see [Evans 2004, chap. 15], [Vernon 2013], [Vernon 2016], and [Vernon 2017]).

Vlad Khononov notes in his book that the term *core domain* is actually misleading, and one should rather speak of *core subdomain* (see [Khononov 2022]). This is true, given that "domain" is an overloaded word within the DDD toolset. Yet the term core domain is used throughout the DDD community, and so we stick with it in this book.

Core domain

The *core domain* (or—if there is more than one—a core domain) is the subdomain that is of utmost importance to help the company achieve initial or continued success. With the innovative and distinguishing value within the exclusive subdomain, the business can differentiate itself from its competitors. A company that is better than others in its core domain is perceived as special by its customers. Such a company has a unique selling point due to its excellent service to customers.

To achieve and maintain this advantage over the competition, the company needs the best possible software support for the core domain. The goal is for the company to have such good software and such good processes in the core domain that it outperforms all its competitors. For this to happen, the projects in the core domain must have the highest priority. For the core domain, the company should have its own software development department or its own software development service provider. It does not make sense to buy standard software here since that could also be used by the competitors.

Generic and supporting subdomain

In contrast, in the case of a *generic subdomain*, the development team should purchase a COTS (commercial off-the-shelf) product, subscribe to a SaaS (software as a service), or download an open source solution that does just enough to meet the system's general-purpose needs. A generic subdomain is a part of the domain where the company does not attempt to differentiate from its competitors because there is no value in doing so.

A typical example for a generic subdomain is accounting. Every company must perform accounting, but in-house development in this area will not provide an advantage over the competition. Here, a third-party solution (often with interfaces to the self-developed system of the core domain so that the core can create booking records for business transactions) should suffice.

Finally, there is the *supporting subdomain*. A supporting subdomain is a part of a domain for which a company needs a partially specialized solution, but which is less important than the core domain. The software for a supporting subdomain should be developed in-house, but third-party software could possibly be used (purchased, subscribed to, or downloaded open source)—depending on the availability of such software and what development resources are available. A third-party solution must be configurable to the extent necessary for the specialized supporting features to be enabled.

Core domain charts

A technique that can help with categorizing subdomains is *core domain charts* (see *https://github.com/ddd-crew/core-domain-charts* and [Tune and Perrin 2024, chap. 10.2]). It's a visualization to help the discussion that shows complexity and business differentiation as the two axes (see Figure 3-9).

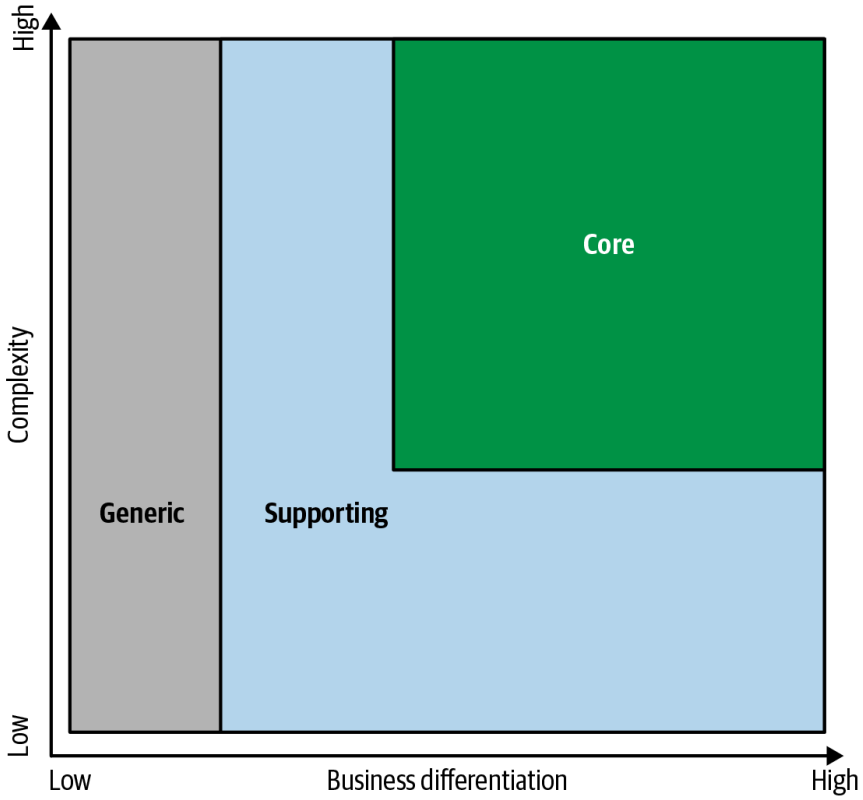


Figure 3-9. Empty core domain chart.

CASE STUDY “CINEMA”

IN THE CINEMA: WHERE IS THE CORE DOMAIN?

When Cine2001 was designed at the end of the last millennium, the moviegoers usually bought their tickets at the box office. Being able to purchase tickets on the internet was new at that time. Clintwood was one of the first to provide it, so it was a business differentiator—in DDD terms, "ticket sales" was a core domain. Accordingly, a lot of effort was spent in Cine2001 on the web sales.

Now, more than 20 years later, web sales has become a commodity, and all the competitors of Clintwood Cinemas provide it at more or less the same quality. Over time, "ticket sales" has become a generic subdomain.

The new management, with their expansion strategy, emphasized the ability to provide just the right weekly schedule for each movie theater. They have installed a new customer research department that has found out that the age structure, size of a city, neighborhood of the movie theater, and other demographics heavily influence which kinds of films sell well in that particular movie theater.

For the expansion to be successful, it is crucial that a new movie theater opens with the right weekly schedule. Thus, Clintwood will be able to provide the moviegoers with a better experience than competing cinemas. The customer research department has defined a sophisticated set of rules to help a cinema manager plan the schedule that will be implemented in Cine2001. "Program planning" has become a new core domain of the system. The evolution of the categorization of the subdomains is visualized in Figure 3-10.

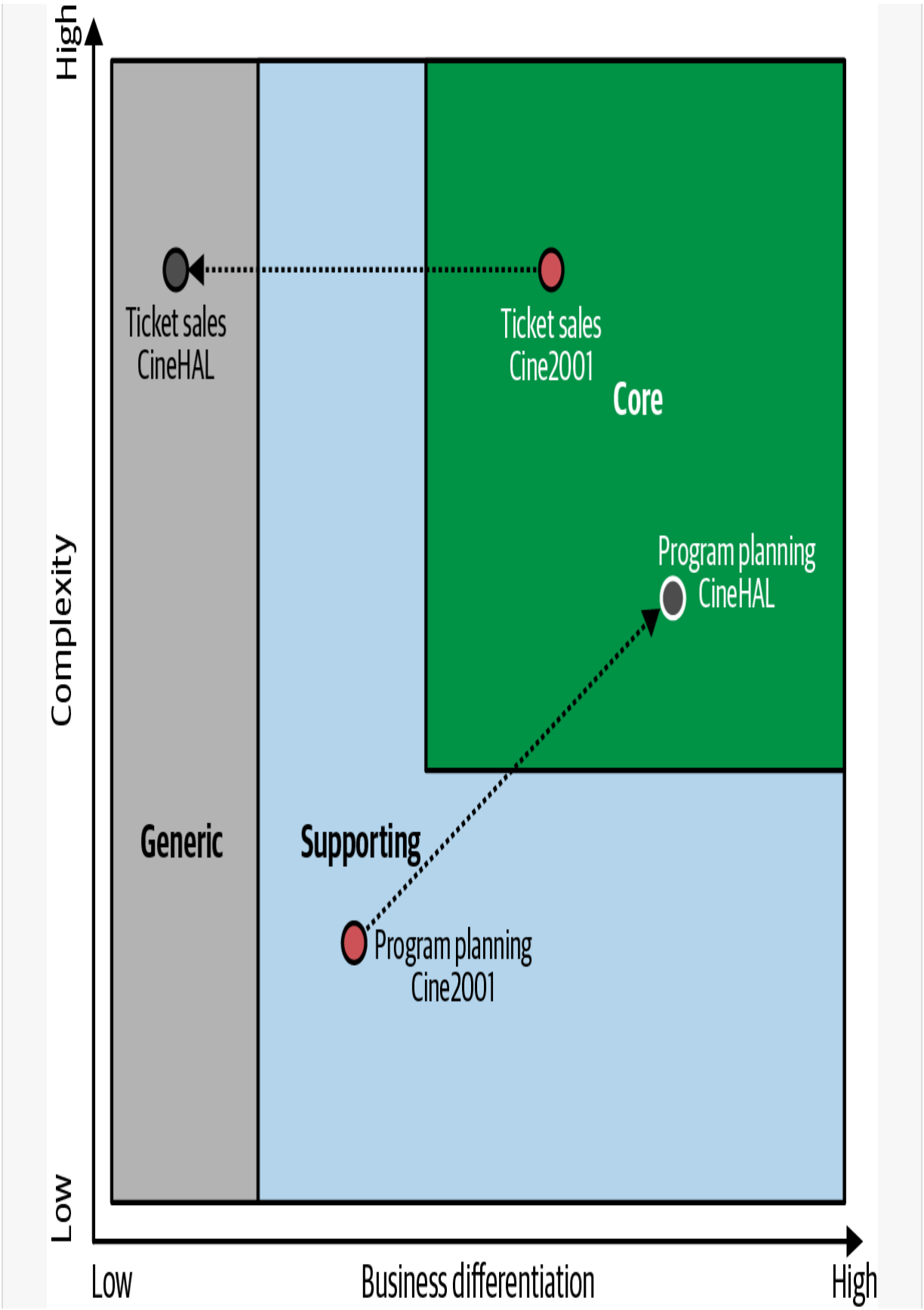

High
Complexity
Low
Ticket sales
CineHAL
Ticket sales
Cine2001
Core
Program planning
CineHAL
Generic
Supporting
Program planning
Cine2001
Low
Business differentiation
High

Figure 3-10. Core domain chart at Clintwood.

Following the AI hype that the management has heard about (the transformation begins in the year 2025), this new core domain will be supported by a large language model (LLM). To honor the new technology, the system will be renamed from Cine2001 to CineHAL.

As you can see, depending on the business model, the importance of subdomains is different and leads to different strategic priorities in each case.

CASE STUDY "CINEMA"

IN THE CINEMA: A NEW CORE DOMAIN

For the transformation of the legacy system Cine2001 to CineHAL, realizing that program planning will become the core domain, with the current business plans and ticket sales moved from core to supporting subdomain, will help. The first goal should be to separate program planning from Cine2001 and then extend it with the new rules from the customer research department.

With this knowledge of subdomains and their categorization into core domain, supporting subdomain, and generic subdomain, the focus can now turn to the software solution.

DDD and the Software Solution

Having delved into the methods and concepts developed in DDD for the problem space and domain modeling in the last section, the focus now shifts to what DDD offers for the solution space and domain-driven architecture (revisit Figure 3-1).

Strategic Design in the Software

For the overall architecture—the domain-driven modularization—the subdomains are mapped to *bounded contexts* in the *strategic design*, and their dependencies are modeled in a *context map*.

Bounded context

A *bounded context* defines the context in which a domain model applies. It is a boundary around a part of the application in terms of both implementation and ownership by a team. While subdomains and the boundaries between them are discovered (i.e., found in the business), bounded context and the boundaries between them are designed (i.e., constructed in the software system). In an ideal world, every bounded context should be an implementation of a subdomain.

> **NOTE**
>
> A bounded context comprises a part of the source code and has its own database schema. In this part of the system, one ubiquitous language exists because there is only one domain model in a single bounded context.

The contextual boundary should be expressed in source code, with no direct outside access granted to the model or database. Due to this boundary, the ownership and structure are clear to all developers on the team and those requiring integration with the bounded context.

> **NOTE**
>
> A bounded context is created and developed exclusively by one team and may not be modified by other teams.

In Figure 3-11 you can see the relationship between subdomain and bounded context as well as the responsible teams in each case.

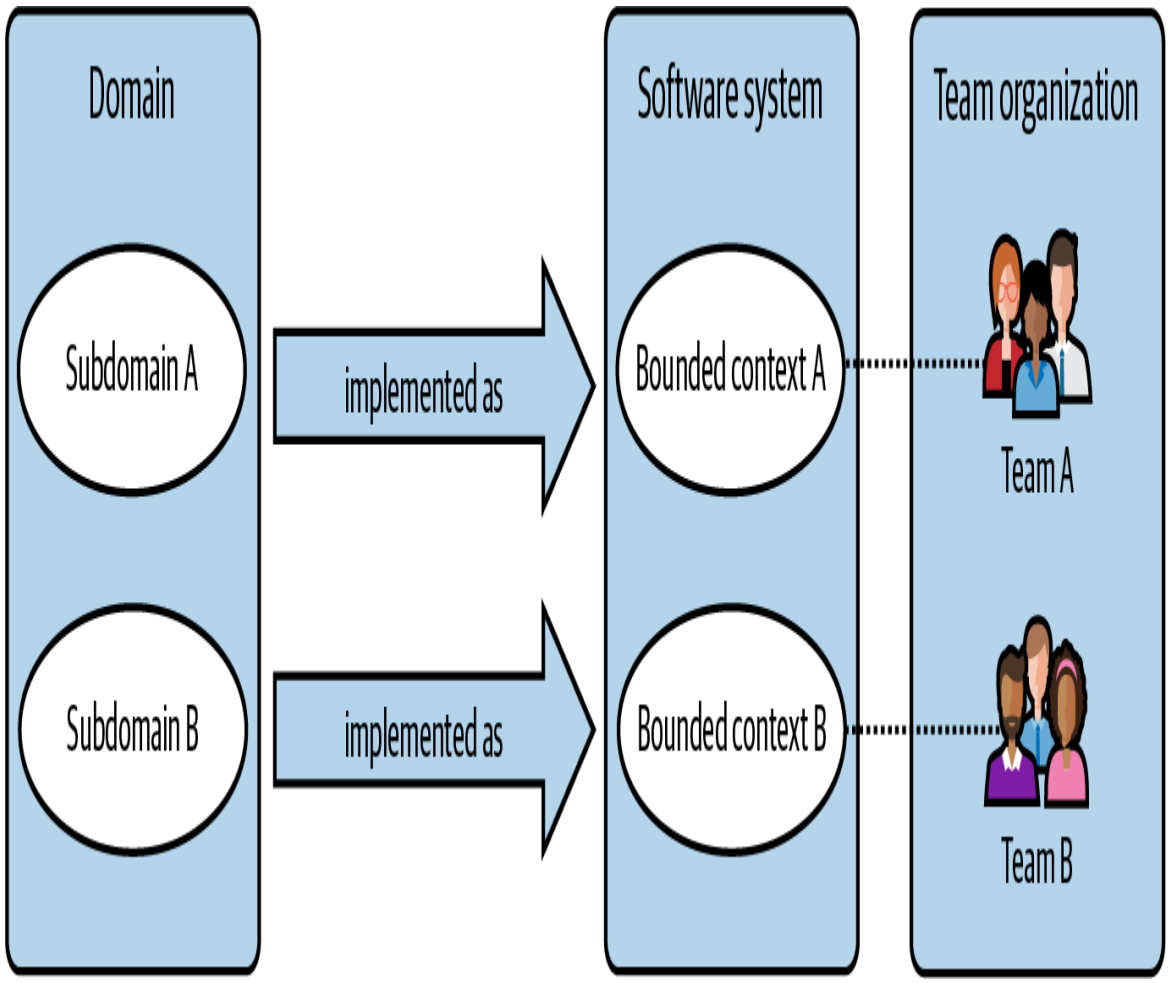


Figure 3-11. From subdomain to bounded context.

While a bounded context only belongs to one team, a team can create, develop, and own multiple bounded contexts.

Context map

The bigger the domain is and the bigger the software becomes, the more bounded contexts there will be. To maintain an overview here, DDD provides us with the context map. A *context map* shows the landscape of the bounded contexts. With the context map, discussions are started with the domain experts and developers in the workshop about which bounded contexts are dependent on each other and which are independent of each other.

CASE STUDY “CINEMA”

IN THE CINEMA: BOUNDED CONTEXTS

For Clintwood Cinemas, a context map could look like Figure 3-12.

In addition to the two bounded contexts “program planning” and “ticket sales” that the team wants to carve out of the legacy Cine2001 system, they are also planning other bounded contexts: “accounting,” “concessions,” and the external “film distribution.” In fact, in the course of transforming Cine2001 into CineHAL, they want to put into operation the interfaces that the standard software for accounting has and that the film distribution offers:

- The interface of accounting must be connected to ticket sales so that the revenue from ticket sales can be booked.
- The film distributor will share information about films with the program planning.
- And finally, an interface is needed between program planning and ticket sales, because the ticket sales department needs the weekly schedule every week in order to be able to sell tickets for the planned shows.

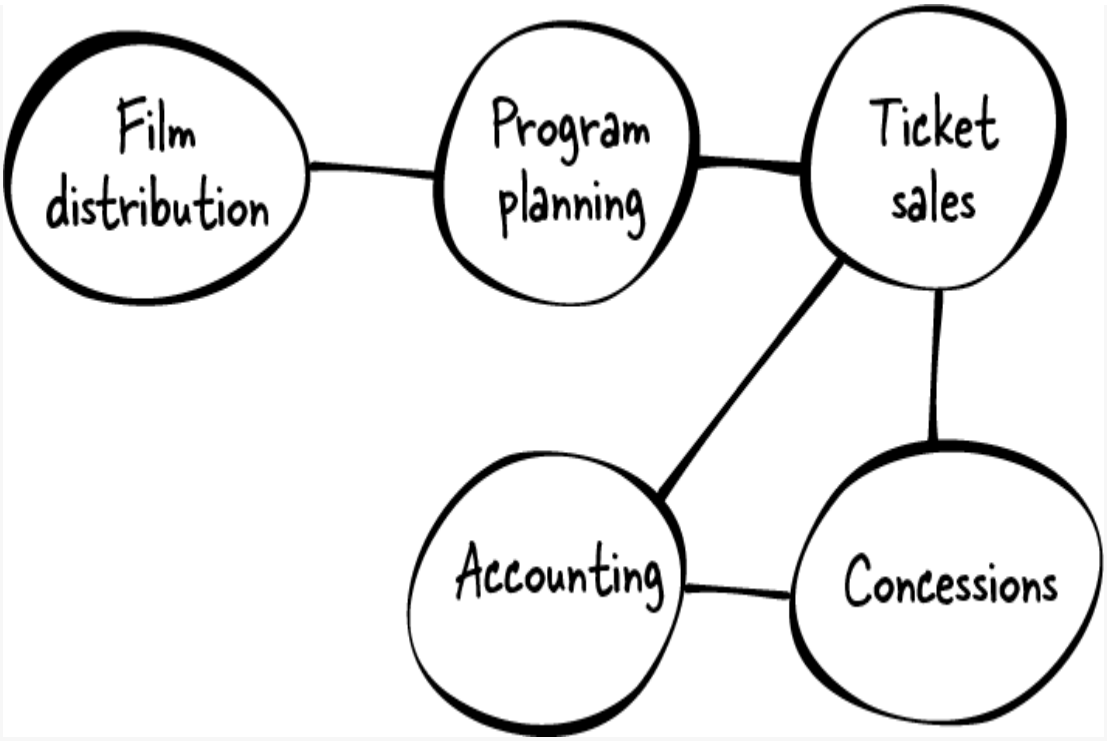


Figure 3-12. Context map for the cinema.

We already have a good picture of the bounded contexts and their interfaces. But DDD goes even further and helps to define the dependencies between the bounded contexts more clearly with context mapping.

Context Mapping

The DDD literature (e.g., [Evans 2004], [Evans 2015], [Vernon 2013], and [Plöd 2025]) describes different types of collaboration between bounded contexts as well as the teams working on these bounded contexts. Helpful terms are:

Mutually dependent

> This situation is often encountered in large organizations with in-house software development. Two teams with separate bounded contexts need to deliver their software together to be successful. These teams depend strongly on each other because of some close, often reciprocal, links in

the source code between their systems; e.g., one system needs data from the other, or there are function calls between the systems.

Free

A bounded context that can be successfully delivered independent of the success of other bounded contexts. There is no technical or organizational connection of any kind between the bounded context teams or their software.

Upstream-downstream

One team sits upstream and makes specifications that the second team seated downstream must accept. It may be possible for the downstream team to have some influence on the upstream team, but that is not guaranteed.

The technical implementation can vary. The upstream team can, for example, send domain objects as data via a message bus, which the downstream team then receives. Or the upstream team provides an interface that the downstream team can use to retrieve domain objects in a specific format.

It is important to realize that these patterns do not always describe the direction of the data flow, but rather the direction of who has the power to define the interface. In particular, if one team provides an interface to another team, the downstream team sends data to the upstream team via that interface.

The following context mapping patterns are defined:

Partnership

In a situation where two teams are mutually dependent, they might make the strategic decision to form an alliance and work as partners. The teams cooperate on the evolution of the interfaces, which leads to interfaces that satisfy the needs of both sides, and to a lot of coordination and collaboration.

Customer-supplier

A cooperative upstream-downstream relationship, where the upstream supplier team, which has veto power over the customer, may allow itself to be influenced by a downstream team, the customer team. In this case, the supplier asks the customer for its requirements for the shared interface. If possible, the supplier implements these requirements with appropriate prioritization, as promised by the customer-downstream team for its own planning toward its customers, i.e., the business users of its bounded context. Very cooperative, isn't it?

Conformist

Another upstream-downstream relationship, where the downstream team has no way to influence the upstream team. While customer-supplier development is cooperative, this is a dictatorial variant. Here, the downstream team conforms to the upstream team's model, taking what it gets. This decision requires the downstream team to accept the risk of not protecting itself from upstream changes.

Teams often decide for this variant when connecting to public interfaces or interfaces of standard software when that interface provides a fitting model, or they want to avoid the overhead of building an anticorruption layer. Yet, becoming a conformist could be a poor decision if the upstream interface remains in flux after the downstream team has embraced that relationship and the model still requires too many and frequent changes.

Anticorruption layer

If a team is in the downstream role and has little to no influence on the upstream team, then it is forced to take what the upstream team provides. In this case, which often includes integrating with a legacy system, it is best to create an anticorruption layer when possible.

The anticorruption layer is a software component at the interface to another bounded context or an adjacent system. It protects the

downstream domain model from being influenced by a foreign model, that is, corrupted by the undesirable language and shape of the upstream team. There are several design patterns that address this very problem: Wrapper, Mapper, Adapter, Façade, etc. (see [Gamma et al. 1995]).

Open host service

An open host service (OHS) defines a protocol or interface for a bounded context as a set of services. The resulting API is open to anyone who can use the protocol and should therefore be well-defined and well-documented. Such an interface would be expected in an upstream-downstream relationship so that the downstream team can use the interface documentation to learn what to expect from the upstream team at the interface. It can also be used in a partnership where the partners agree on the protocol of the OHS that can then also be used by other parties.

Published language

Often, an open host service is extended to include a published language, which reflects DDD's focus on creating good terminology in a ubiquitous language also at the interfaces. A published language is a well-documented, standardized language for exchanging information at the interface. It can be defined in different data formats, such as XML Schema, JSON Schema, Protobuf, etc.

Examples of some open standards capable of serving as a published language are: FHIR (Fast Healthcare Interoperability Resources) in healthcare, WMS (Web Map Service) for map data, OCPP (Open Charge Point Protocol) for charging electric cars, or SEPA XML for transfers and mandates in European banks. Even so, an organization can create its own standards or published languages.

Shared kernel

If two teams work together as partners or if one team works on two bounded contexts, then they might benefit from a small, common

software model that is developed and shared. Only one team owns the shared kernel, while both (or more than two) teams include this shared kernel in their bounded context and must react directly to changes. A shared kernel contains a model or part of a model that both bounded contexts agree on, and this model should be small. The rest of the bounded contexts (and the models inside) are separate from each other, as is always the case with bounded contexts. Be very careful not to overuse this pattern, as it could easily be misused as a canonical model.

Since the aim is for as little coupling as possible with modularization according to DDD, the shared kernel should be used with caution. Such a shared kernel can work well for common small types and maybe algorithms that must not repeat code. It helps if those parts of the software system have become relatively stable over time and are subject to few changes. It's also best when the model types are domain values and services rather than entities.

Separate ways

If integrating two bounded contexts is more pain than gain, a team may decide not to do it. Each of the two bounded contexts might develop their own specialized one-off solution within their own model.

Big ball of mud

A system (or part of a system) where everything is connected to everything else, boundaries don't exist or are inconsistent, and business logic and technical code are mixed up. It is the context mapping antipattern and is used in many context maps to separate the cleaner parts of the system from that chaotic mess. More on the big ball of mud can be found in the section of the same name in Chapter 5.

With these organizational and technical collaboration patterns, DDD helps to think about the interfaces between the bounded contexts in a context map.

CASE STUDY “CINEMA”

IN THE CINEMA: CONTEXT MAPPING

How should the bounded contexts at Clintwood work together? Let's take a look at the context map and the context mapping patterns used (see Figure 3-13).

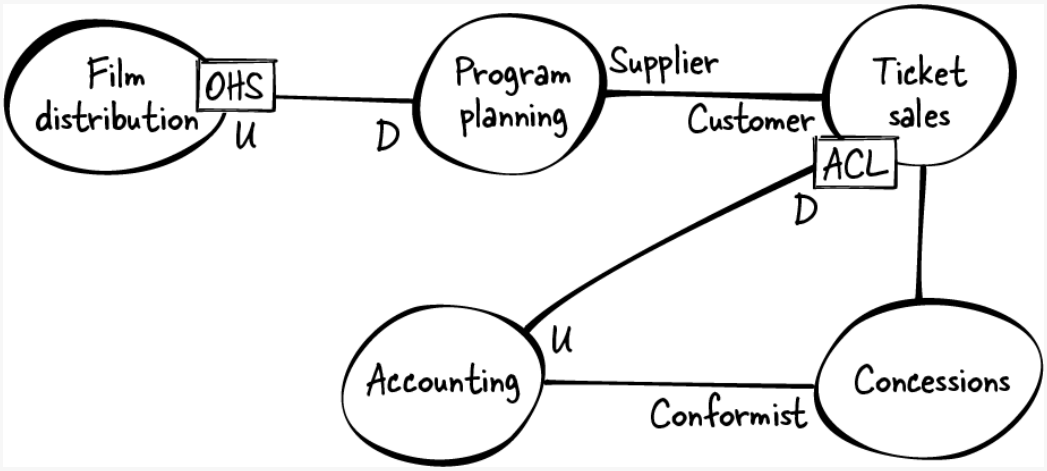


Figure 3-13. Context mapping in the cinema.

There will be customer-supplier collaboration between the two teams implementing program planning and ticket sales via an internal interface, where the supplier is responsive to most of the customer's needs. The program planning team as the upstream team will provide the weekly schedule, which ticket sales will need as the downstream team. Since the two teams are in-house and jointly want to deliver good software for the benefit of both theater employees and moviegoers, they will be able to work together as customer and supplier. That means the supplier team working on program planning will usually adapt the functionality and implementation of the interface according to the ticket sales team's wishes, if possible.

Since the cinema wants to buy standard software for accounting, the teams have to deal with the standard interface of that software, through which accounting records have to be delivered in a certain format. Accounting will be in the upstream role (U) toward ticket sales and concessions, and ticket sales and concessions will be in the downstream role (D). Since ticket sales is a core domain, we want it to be developed

in a domain-driven fashion without pollution from the outside. Therefore, the ticket sales team will protect its bounded context with an anticorruption layer (ACL) against the accounting interface. The concessions bounded context will also need to provide data to accounting. Since it is not part of the core, this interface will be implemented as a conformist.

Finally, program planning will receive data from film distribution via an interface. Since the film distributor has already built this interface for other cinema companies and tested it with them, this interface can be considered an open host service (OHS). It will probably even include a published language—but the team won't know until they get the description for it from the film distributor. Here, the program planning team is in the downstream role and needs to see if the interface from the film distribution fits their own model in the program planning.

One possibility here, as with the interface between ticket sales and accounting software, would be to use an anticorruption layer. However, it could be that the two situations are significantly different. The accounting software is a COTS software that does not consider the movie theater business in any way; when not protecting against its interface model, it could well pollute the domain model in ticket sales. The interface from film distribution has already been tried with several other movie theaters and has already been improved based on their feedback. If the domain model in the weekly schedule does not match the interface, then it could be that the domain model needs improvement.

The bottom line is that clarity about organizational collaboration is certainly more important than technical implementation for software development of large systems with multiple teams to succeed. So, the developers should focus their attention more on the points listed above than on the technical solutions.

Bubble Context

In his paper on DDD and legacy software [Evans 2013], Eric Evans introduces a special kind of bounded context: the *bubble context*. A bubble context is a small bounded context that is created inside the legacy solution, but separated from legacy code using a separate top-level module (e.g., Java package or C# namespace) and an anticorruption layer (ACL). The bubble context is worked on by a small development team that feels like they are in a bubble when programming in this bounded context, which is facilitated by the separate module and the ACL (hence the name "bubble"). This bubble isolates the work so that the team can develop a true domain model that it optimizes for the chosen bounded context, relatively free from the existing concepts of the legacy system.

The paper recommends working with a bubble context if you want to make a relatively small enhancement to your software that can be easily separated.

At the same time, it assumes that working in the bubble context is temporary. After the work is completed, the bubble context remains in the legacy code, and it may be that, over time, the ACL will get holes and the bubble will burst because there is no longer a reason to maintain the separation. Although the bursting of the bubble means that the evolved DDD domain model loses some clarity, the functionality that was developed in the bubble will live on long after the bubble is gone. Since the boundary is no longer so clear, the fossil model footprint will be just one of many in the mosaic of the legacy system.

Evans goes on to write that a bubble context is a great way to get started with DDD while coexisting with legacy systems. If this first attempt is successful, it could be that other areas of the legacy system might also benefit from DDD.

The bubble context is a means to bring new functionality into a legacy system by using DDD. In this book, we will not use it because we are more focused on transforming the existing functionality of the legacy system with DDD.

The Domain Model

In DDD, the *domain model* of a bounded context is based on the ubiquitous language of a subdomain (see "Ubiquitous Language"). To develop a domain model, consider what domain concepts and work items the domain experts use in their work, what they are called, and (even more importantly) what the domain experts do with them (i.e., the actions with them).

The developers therefore let themselves be driven by the domain when developing the domain model. The following rules of thumb apply:

- The concepts and objects formulated in the ubiquitous language become types in the domain model (i.e., classes in object-oriented programming).
- The actions become operations of the domain model (i.e., methods in these classes in object-oriented programming or functions in functional programming).

What does this mean in the cinema example?

IN THE CINEMA: THE DOMAIN MODEL

In the Metropolis movie theater, a central object for the cashier is the seating plan, which they use to manage the seats for a show. So the developers implement corresponding classes `SeatingPlan` and `Seat` in the domain model.

How is the seating plan used? When the cashier sells a seat, they mark it in the seating plan. Therefore, the class `SeatingPlan` is given a corresponding method.

In Java, this results in the code in Listing 3-1.

Listing 3-1. A domain model for the seating plan

```
public class SeatingPlan {
    public void markAsSold(Seat soldSeat)
    //...
}
```

The ubiquitous language is expressed in the following ways: the terms `SeatingPlan`, `markAsSold`, and `Seat` are all part of the language that has been expressed in code.

A domain model that is created in this way is expressive and has rich behavior. Not only the types, but also each operation has a name from the domain. A domain model with rich behavior is useful because it helps the team to think and converse in familiar domain terms.

There are other benefits as well. A rich domain model creates a *structural similarity* between the domain and the source code (see [Züllighoven 2005], p. 104). If a user has difficulties with the software and needs a change or an adjustment to the software, then the relationship between domain language and source code can be quickly realized due to the structural similarity.

IN THE CINEMA: STRUCTURAL SIMILARITY

When the old CEO of Clintwood Cinemas retired, there was only one type of seat in the theaters of the chain. Now, the new management modernizes the theaters so that they are both more luxurious and barrier-free. Booking a ticket will provide a choice of standard seats, upscale seats, and seats for wheelchair users. These different types of seats are shown accordingly in the seating plans for the cinemas.

Once Cine2001 has been transformed to use the true ubiquitous language, it will be much easier for the team to identify the source code where the adaptation around the new type of seating must be done; that is, in the classes `SeatingPlan` and `Seat` and the user interface in which seats can be reserved.

Domain model and bounded contexts

Building an expressive domain model is great, but in a large system this model will quickly grow quite large and reach a size where it is hard to handle. That is why with strategic design, designing one large model is purposely avoided. Rather, the goal is to design a different model for each bounded context.

Unfortunately, in many systems, teams try to build one canonical model that is intended to address all (existing and future) business use cases. In such a case, the subdomain boundaries are not expressed in the system's architecture, leading to what Vaughn Vernon calls an unbounded context (see Figure 3-14). Here, the one same huge domain model is used for all the business use cases and contains all the business logic in a strongly coupled way. Because different teams work on the same unbounded context, no one can work autonomously. Rather, they are continuously stepping on each other's toes.

For this reason, according to DDD, a separate ubiquitous language with a matching domain model is developed for each bounded context. This means that the crystallization of domain terms is carried out depending on the bounded context; in other words, the language is purely contextual based on domain expertise and the business capability under development. The individual terms are reduced to their contextual significance and thus become sharper and clearer.

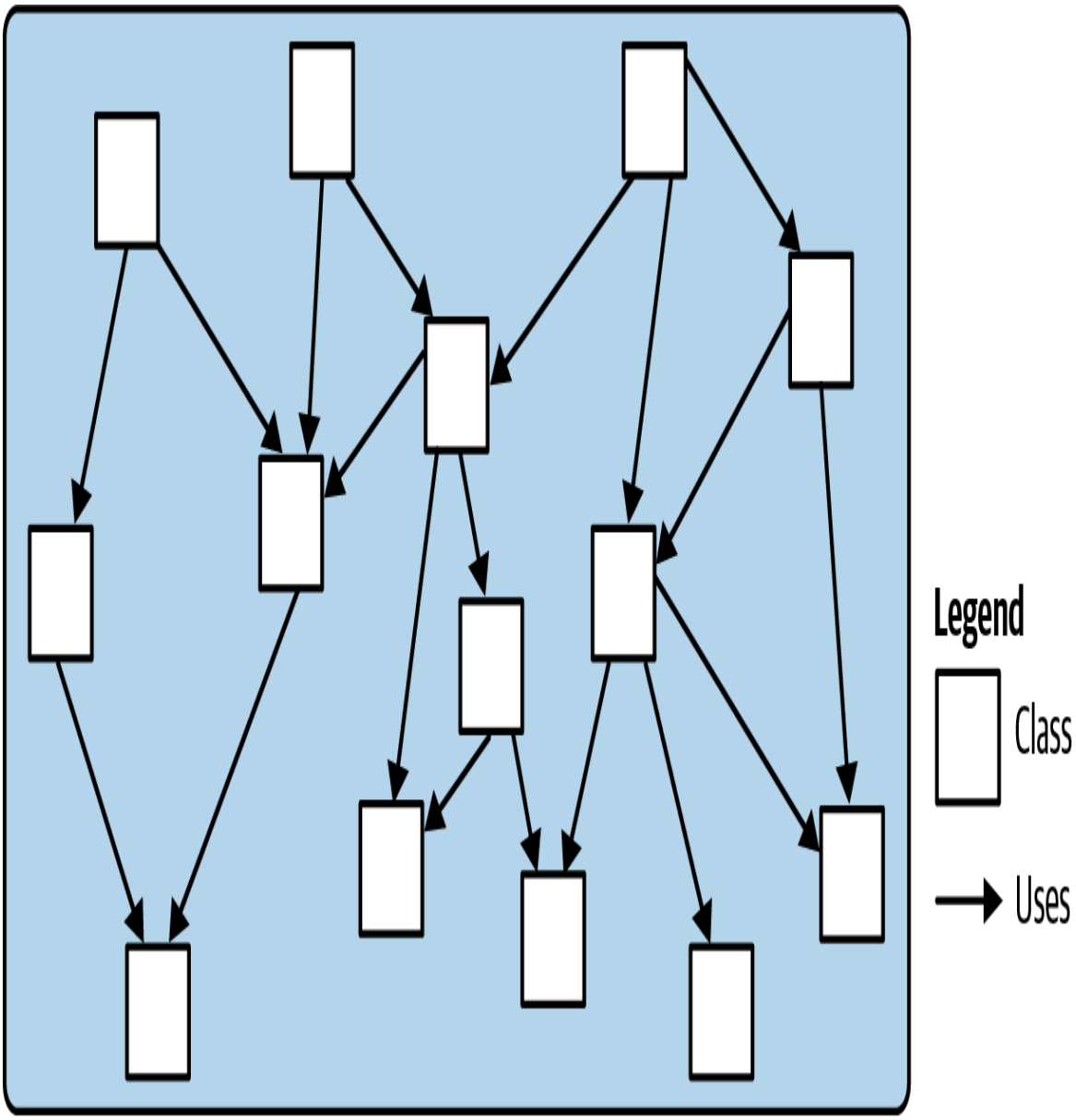


Figure 3-14. Traditional domain model: tendency toward gigantic scale and strong coupling.

So, it can happen that the developers implement the same domain term in different bounded contexts with different classes—different classes with the same name. This takes some getting used to! (See Figure 3-15.)

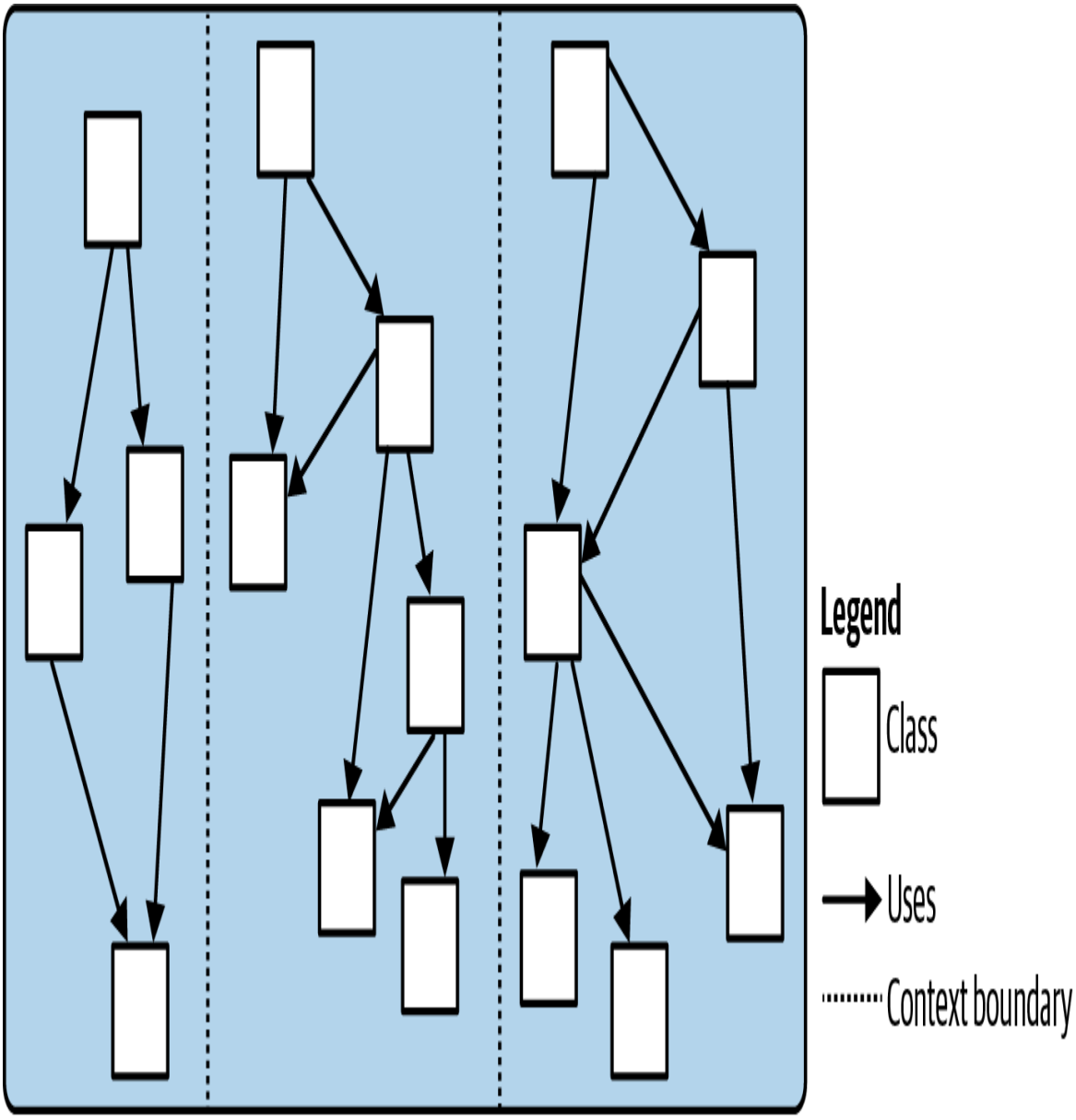


Figure 3-15. Domain model(s) in bounded contexts: smaller models, looser coupling.

A bounded context contains a domain model in a consistent language that is tailored to a specific purpose.

IN THE CINEMA: DOMAIN MODELS AND BOUNDED CONTEXTS

In the transformed cinema software, there will be two `WeeklySchedule` classes: one in the bounded context of program planning and a second in the bounded context of ticket sales. In the bounded context program planning, the class `WeeklySchedule` is certainly more extensive, as already discussed, and will provide methods to modify the schedule. In the ticket sales bounded context, the weekly schedule is only available for viewing when searching for shows with movies, and modifying it is no longer possible.

As an extension, let's imagine that the work scheduling of the cleaning staff and ice cream vendors is also based on the weekly schedule and is transferred to its own bounded context. Here, too, a `WeeklySchedule` class is needed, but the information about which movie is playing in which theater is irrelevant for this bounded context. Here, for ice cream sales, only the part of the "weekly schedule" that provides information about the beginning and end of shows (including commercials), as well as the pause between commercials and the beginning of the movie, is needed. The individual domain models of the three bounded contexts that need the weekly schedule each map only those properties of the domain concept that are essential in the particular bounded context.

Reuse, duplication, and redundancy

With this partitioning of types that represent the same domain concept to the respective bounded contexts in which they are only partially needed, DDD takes a radically different point of view on the principle of reuse. As stated above, a canonical model is intended to be able to solve all existing and potential business challenges and thus be highly reusable. So much for the theory; in practice, it is not only not reusable, but potentially not even

usable. In reality, the canonical model forces all uses to adhere to the least common denominator of all potential uses.

DDD tells us that domain model types should not be reused between bounded contexts! Rather, specialized (and smaller) types are to be developed for each bounded context, which contain only exactly the functionality required for the respective bounded context. The point is to create models that serve a specific purpose, and they will then be highly reusable for that special purpose.[3]

IN THE CINEMA: SAME TERM IN TWO CLASSES

For the new structure of the cinema software, the team envisions two classes called `WeeklySchedule`: one for the ticket sales bounded context and one for the program planning bounded context (see Figure 3-16). In the bounded context program planning, the class `WeeklySchedule` has much more functionality because there the cinema manager develops the weekly schedule with all additional aspects, like advertising, cleaning, and expected attendance.

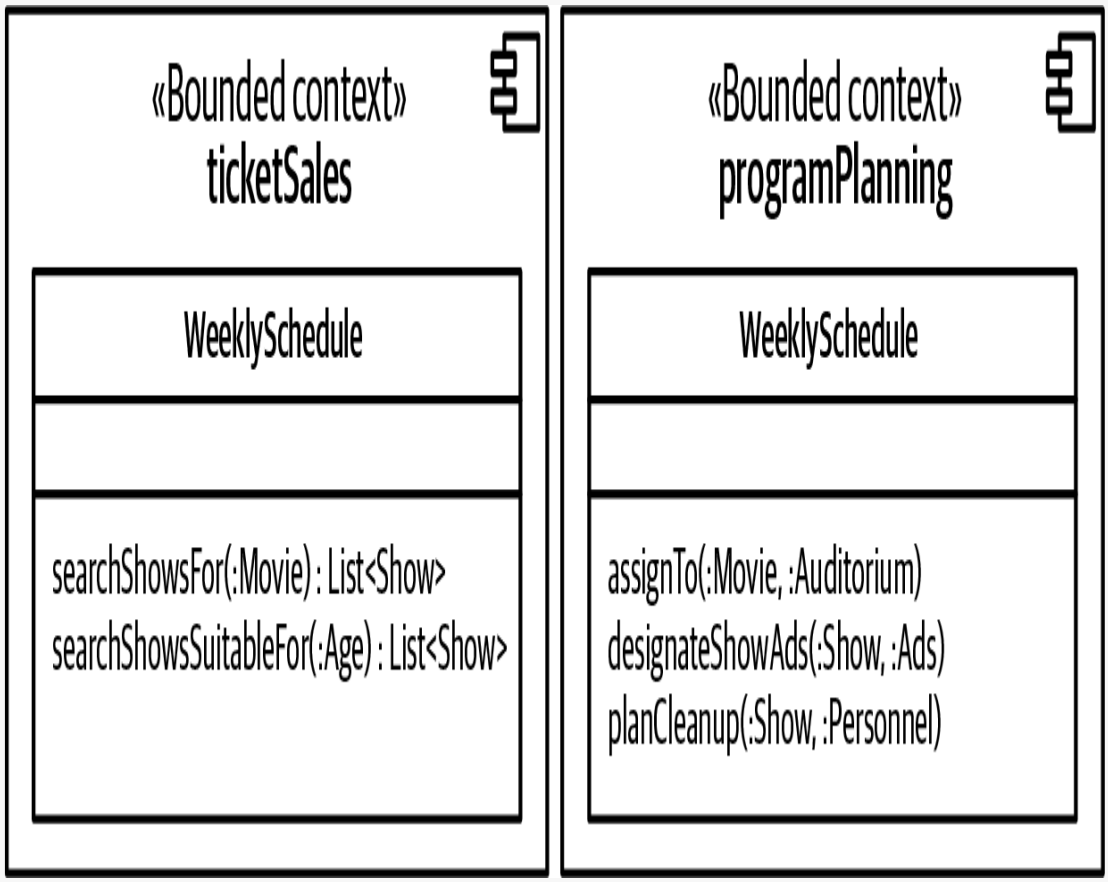


Figure 3-16. Domain model(s) in bounded contexts: smaller classes with higher cohesion.

There may be fear that separating models into bounded contexts will cause redundancy. This could be a justified concern. Yet this will not be the case when determining contextual boundaries using domain-driven criteria. In

the example of the movie theater, we can see that the weekly schedule is present in two bounded contexts, but certainly not in all the bounded contexts that have been thought about so far.

The concept of "customer" is central in many systems and can occur in several bounded contexts if it makes sense. But even such a central concept will not occur in all bounded contexts. And, more importantly, only a small part of the customer data and typically none of the behavior of the `Customer` type will occur in all bounded contexts. Duplicating behavior across contexts will leak business rules and business operations into multiple codebases, making changes error-prone or at least very difficult. We first look at the behavior (mapped in the methods) and then at the data needed to support it. While data redundancy might happen, redundancy will be much rarer in the behavior; that is, in the methods at the interface of the class. If the code contains business rules and operations, it must follow the DRY principle (don't repeat yourself) as the Pragmatic Programmers state (see [Hunt and Thomas 2019, p. 31, Tip 15]):

> *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*

So, it's not about code. It's about knowledge. And domain knowledge is something that is expressed in domain models.

IN THE CINEMA: DUPLICATION AND REDUNDANCY

The management of the Clintwood Cinemas group wants to be able to track the moviegoers' visits to the cinema at every phase. A customer card is to be introduced for this purpose. Certainly, the two bounded contexts "ticket sales" and "concessions" will want to know the identity and name of the customer so that the respective sales staff can address the customer by name. But an email address is only important for sending the movie tickets, not for concessions. And the drinks and food chosen by the customers in the past are only relevant in the bounded context of concessions and not in ticket sales.

Thus, each bounded context has its own data and associated methods. For each bounded context, the data of the customer class is reduced to the required part and stored in the bounded context's own database schema. One could also say that each bounded context follows the principle of data minimization.[4] Should the customer change their name, for example, because they get married, then surely both bounded contexts need to know this. But this change of name is a relatively rare event (as most people marry less often than they go to the movies), for which we then propagate the change from one bounded context to all other bounded contexts interested in it. Also, this propagation might take some time; i.e., the system might implement eventual consistency.

The Layered Architecture in DDD

In the "blue book," DDD is introduced on the basis of a four-layer architecture with the layers *User Interface*, *Application*, *Domain*, and *Infrastructure* (see Figure 3-17).

Over time, it has become apparent that the independence of the domain layer from the technology fits better with Hexagonal Architecture (more

accurately also called Ports & Adapters Architecture) or Onion Architecture. More on these architectural styles can be found in Chapter 5.

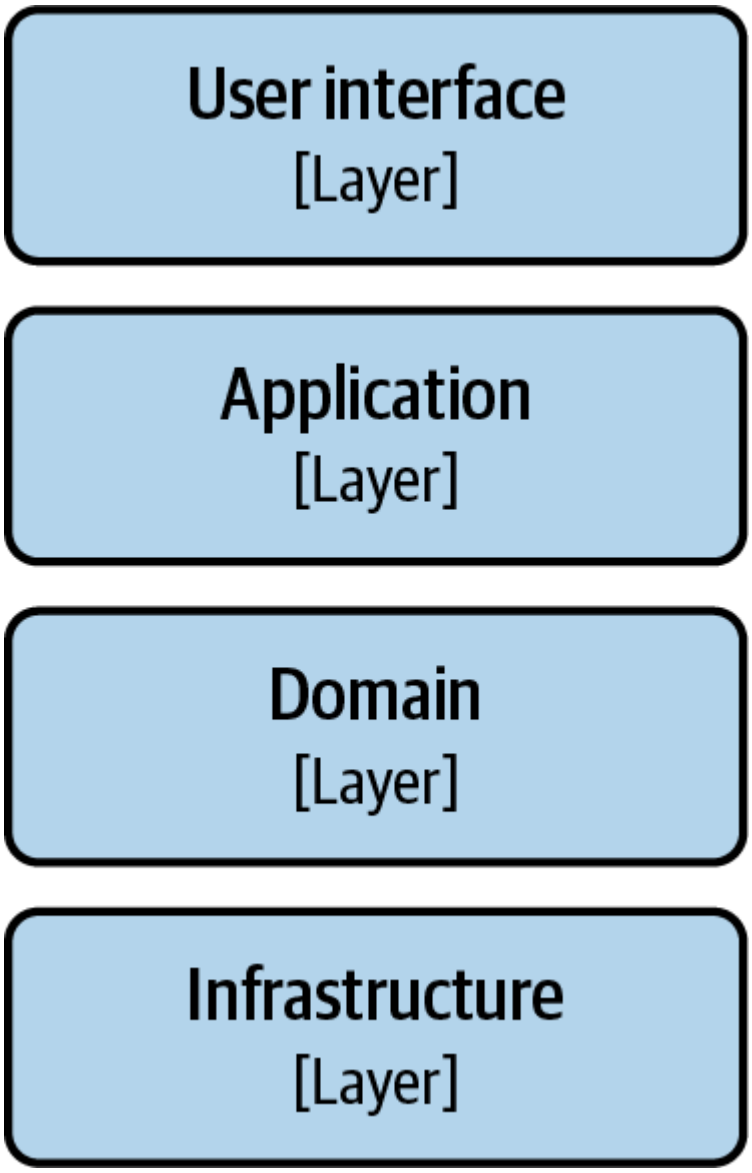


Figure 3-17. The four-layer architecture of DDD.

Unsurprisingly, the most interesting layer for *Domain*-Driven Design is the *Domain layer*, where the *domain model* resides.

Tactical Design

With tactical design, DDD describes patterns for realizing concepts technically within a bounded context in the domain layer, called *building*

blocks. They are often combined with tactical patterns for the application layer (see Figure 3-18).

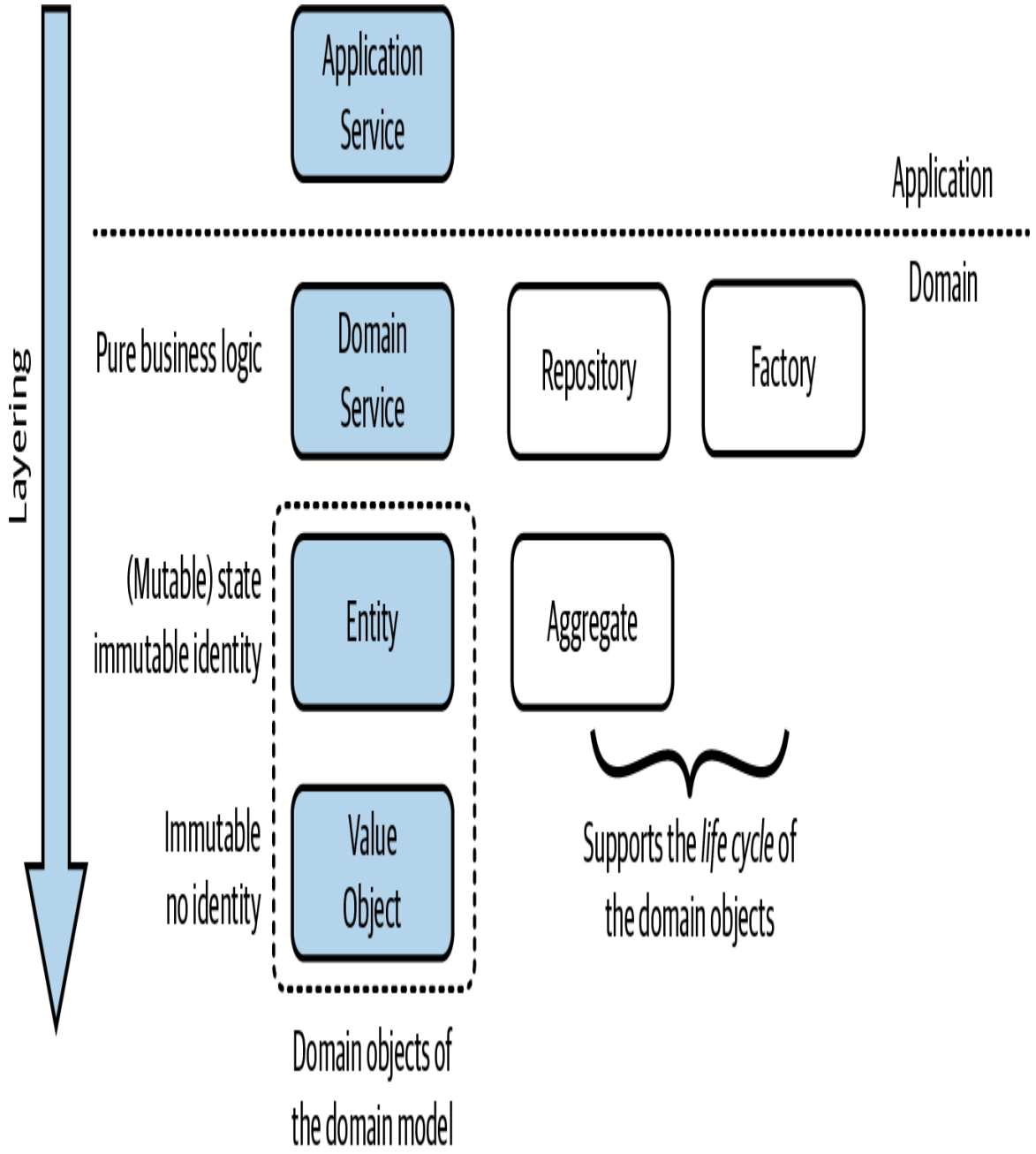


Figure 3-18. The building blocks of tactical design.

Building blocks are design patterns for programming.[5] In Figure 3-18 you can see that the individual building blocks are arranged hierarchically; i.e., there is a layering. Let's look at the individual building blocks in detail. We start with the building blocks that model the domain in the code:

Entity

An entity models a domain concept that has a unique identity. Its name is part of the ubiquitous language. Often, an entity represents something from real life that can be seen and touched. An entity has an immutable *identity* and a well-defined *life cycle* during which the *state* of the entity usually changes over time.

The identity of an entity is usually represented by its own domain type and implemented as a value object (see "In the Cinema: Entities").

Entities contain business logic, often logic that ensures that the respective entity remains in a correct state. Entities represent their state with value objects and may hold other entities.

CASE STUDY "CINEMA"

IN THE CINEMA: ENTITIES

On the seating plan, which is an entity, sold seats are marked and thus the state of the seating plan is changed with each new marking. However, the seating plan itself always remains the same seating plan for a particular show in a particular auditorium. Thus, the seating plan has an identity that does not change. Other typical examples of entities are: weekly schedule and movie ticket.

In the bounded context of program planning, the show is also an entity that has a mutable state. Thus, the movie and the time of a show can be changed during scheduling.

Value object

A value object represents an unchangeable value in a domain that has *no identity of its own*. This means that values are not things in the model, but rather describe, measure, or quantify, entities in the model.

Value objects are always *immutable* and thus easily reusable and distributable.

The state of entities is often described, at least partially, by value objects. We recommend avoiding the use of "naked" strings or numbers in entities. Instead, introduce real types (i.e., value objects), thereby fighting the bad smell of Primitive Obsession (see [Fowler 2019, chap. 3]).

Value objects can consist of other value objects, but should (at least in our opinion) never hold references to entities.

Like entities, value objects often contain business logic, such as for calculations or business rule validation.

A value object forms a conceptual whole around a group of attributes, also called a *whole value*. This is the case, for example, with a numeric value object that contains not only the property measured, but also the unit of that property.

CASE STUDY "CINEMA"

IN THE CINEMA: VALUE OBJECTS

Examples of simple value objects in the cinema are the auditorium number, the ticket price, and the seat number.

In the ticket sales bounded context, a more complex value object can be found. The concept "show" is modeled differently than in the program planning bounded context. In this case, it is immutable and identity-less—a value object, and one that is composed of the further value objects date, start time, and movie title.

Domain service

A domain service contains business logic that does not belong to an entity type or a value type. For example, it represents business functionality that is conceptually entity-overarching as a separate concept.

Domain services hold no business operational state. The parameters passed to a domain service's behavioral operations and their results are entities and value objects. Note, however, that the domain service can hold technical state, such as a reference to a repository[6] used to look up entities.

CASE STUDY "CINEMA"

IN THE CINEMA: DOMAIN SERVICES

At Clintwood Cinemas, when a ticket is sold, the system works with both the sold ticket and the seating plan. The logic that models the selling in the system belongs partly in the entity `Ticket` and partly in the entity `SeatingPlan`, but the knowledge that the sales process involves both of them belongs at a higher level, not in either entity. For this purpose, we introduce a domain service, `SalesService`.

Then we look at the building blocks that support the life cycle of the domain model's objects:

Aggregate

An aggregate establishes a transactional boundary around a graph composed of entities and value objects. The aggregate's transactional boundary ensures that all data within the boundary remains consistent and correct according to business rules when the transaction commits. Sometimes this unit of consistency includes only one entity; sometimes

multiple entities need to be maintained consistent as a single unit. The object graph is directed and should be acyclic.

An aggregate always has a single entity as its parent and entry point (called *root entity*, *aggregate root*, or simply *root*), and is regarded as a unit with regard to changes. All data held within the root entity must be consistent. The root entity is the only direct access point for the entire aggregate. Whether any internal elements of the aggregate can be accessed from the outside is a matter of design preference. We strongly recommend that at least state-changing operations of the internal entities not be directly accessible from the outside. Accessing value objects is not problematic, since they cannot change their state and thus the consistency of the aggregate is not endangered.

CASE STUDY "CINEMA"

IN THE CINEMA: AGGREGATES

The previously described `SeatingPlan` is the root entity to an aggregate with the same name. It could be modeled as a single-entity aggregate. Alternatively, the individual seat located on the seating plan could additionally be modeled as a separate entity with the possible states "not yet sold" and "sold." Then the aggregate boundary would be drawn around this seat entity as well.

Repository

A repository is a place where aggregate instances of a specific type are persisted. How do you implement persistence in an IT system? Typically, in a database. A database is a kind of technology, and we don't want technology in the domain layer. To prevent any such technology from corrupting the domain model, a repository is used to abstract away the technical details of the outside infrastructure

(technical source code) from the business source code inside the architecture.

Typically, a repository is implemented in two parts: the interface becomes part of the domain layer on the inside, while the implementation(s) is part of the outside infrastructure layer. In terms of Hexagonal Architecture (see Chapter 5), the repository interface is a driven-side port, and the repository implementation is an adapter.

CASE STUDY "CINEMA"

IN THE CINEMA: REPOSITORY

The `SeatingPlan` instances are stored in a `SeatingPlanRepository`. This provides methods for reading and storing a seating plan.

Factory

A factory is a program element that encapsulates the creation of aggregates, entities, or value objects. It can be a type in its own right or just an operation, and is used when a simple constructor is no longer sufficient. Factories work exclusively within their bounded context and have no access to the technical building blocks. Here the typical creational design patterns of the Gang of Four can be found, such as Factory Method, Abstract Factory, and Builder (see [Gamma et al. 1995]).

Finally, a building block from the application layer (see "The Layered Architecture in DDD") is also interesting:

Application service

Although not specifically a DDD tactical pattern, an *application service* orchestrates the elements that are in the domain layer. It offers an

external interface that can be used to call the business functionality implemented in the domain services, entities, and value objects. In this kind of service, processes and use case coordination are implemented. For this reason, the application services belong to the application layer, while the domain model and the domain services belong to the domain layer (see “The Layered Architecture in DDD”).

CASE STUDY “CINEMA”

IN THE CINEMA: APPLICATION SERVICE

If, for example, the entire process of a seat reservation is to be supported by the software via a web frontend, then a corresponding application service is the right place for this sequence of steps.

In “Documenting Architecture in Code”, we will see how to make the architectural rules imposed by DDD’s tactical design testable using xMolecules and ArchUnit.

CASE STUDY “CINEMA”

IN THE CINEMA: OBJECT GRAPH

Let's take a look at a typical object graph according to Domain-Driven Design in Figure 3-19—the object graph in the bounded context of ticket sales. Here we see the entities "seating plan," "film," "row," "seat," and various value objects. The seating plan and the film are root entities to aggregates of the same name.

Some interesting points can be seen in this object graph of Figure 3-19:

"Show" as entity and as value object

> In the program planning bounded context, the movies are scheduled in shows. There, the show is modeled as an entity (and aggregate). In the other bounded context, ticket sales, no changes are made to the weekly schedule. It is immutable and the show has no identity here. Therefore, in the ticket sales bounded context, the show is implemented as a value object. (In the case here, the value is an identity that references the entity/aggregate in the originating program planning bounded context. In other words, the value *is* an identity even though it *does not have* an identity.)

"Film" as its own aggregate

> If the film were part of the seating plan, then it would also be saved again every time the seating plan is saved. This makes no sense. Instead, the film is an independent aggregate that provides its own consistency.

"Film" by title in the show

> The film for the show is referenced in the seating plan only on account of its identity, the "film title,"[7] because the film is an aggregate in its own right.

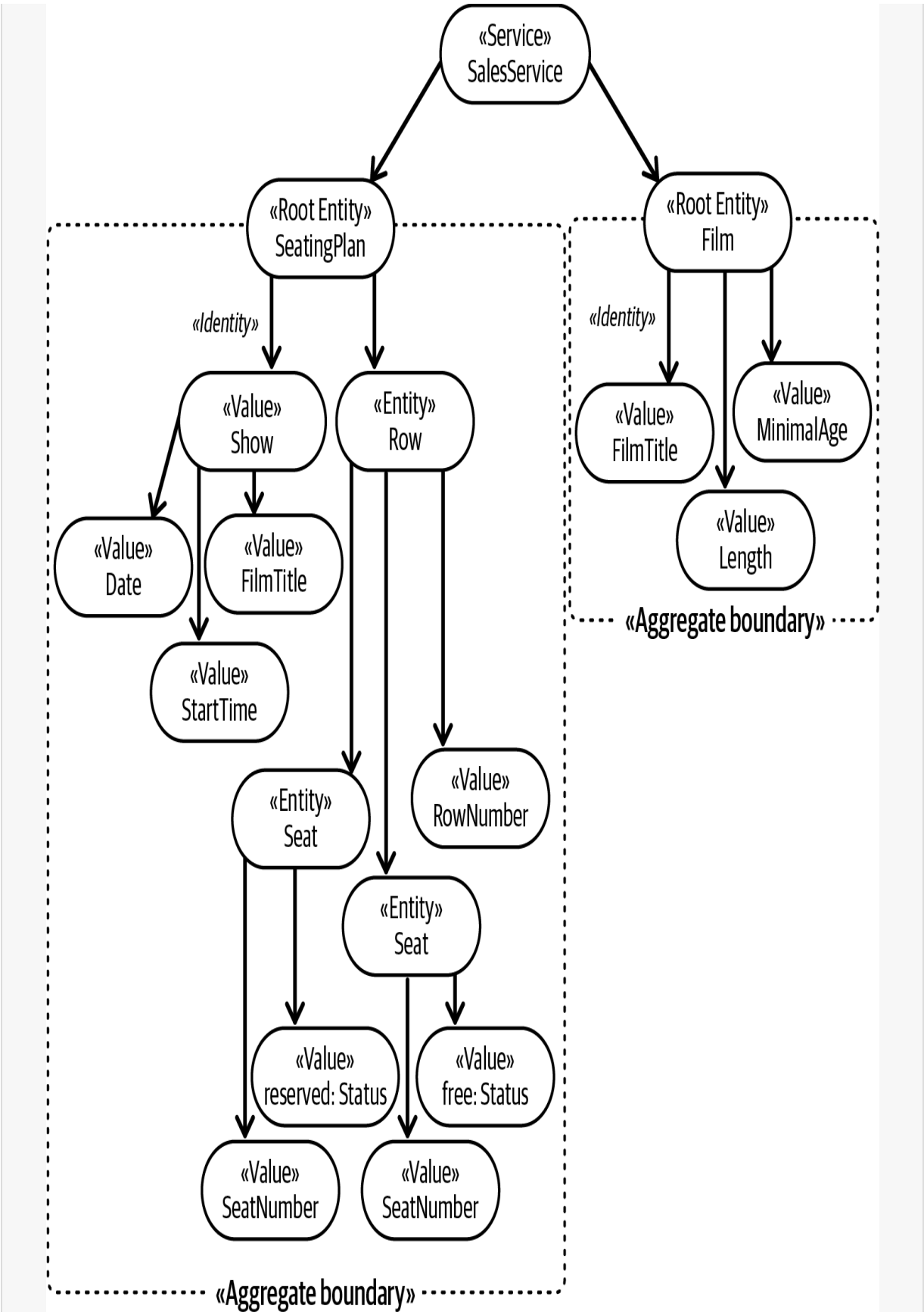

«Service»
SalesService
«Root Entity»
SeatingPlan
«Root Entity»
Film
«Identity»
«Identity»
«Value»
Show
«Entity»
Row
«Value»
FilmTitle
«Value»
MinimalAge
«Value»
Date
«Value»
FilmTitle
«Value»
Length
«Aggregate boundary»
«Value»
StartTime
«Entity»
Seat
«Value»
RowNumber
«Entity»
Seat
«Value»
reserved: Status
«Value»
free: Status
«Value»
SeatNumber
«Value»
SeatNumber
«Aggregate boundary»

Figure 3-19. Object graph in the bounded context ticket sales.

Domain Events

In the “blue book” the concept appeared only marginally, but over time, the *domain event* was added to the set of DDD building blocks (see [Vernon 2013] and [Evans 2015]). This was a logical step, since domain events are used to communicate between bounded contexts in many systems built according to DDD.

Domain events make changes in the state of entities and aggregates become visible. They represent an important occurrence (something that happened) in the past. Therefore, domain event names should always express that something has already happened.

Generally, applying a form of the behavior that caused the event using the idiom “noun followed by a verb in past tense” works well. However, sometimes naming events is more awkward than applying a common idiom, and an appropriate name must be more carefully chosen.

CASE STUDY “CINEMA”

IN THE CINEMA: DOMAIN EVENTS

In Clintwood Cinemas, the bounded context program planning publishes a domain event "weekly schedule finalized" when the cinema manager has completed their planning for the next week. The bounded context ticket sales subscribes to this event so that next week's shows can be offered and sold at the box office and online.

When a ticket is sold, the ticket sales bounded context publishes a "ticket sold" event. The accounting bounded context has subscribed to this event and will process the relevant information (see Figure 3-20).

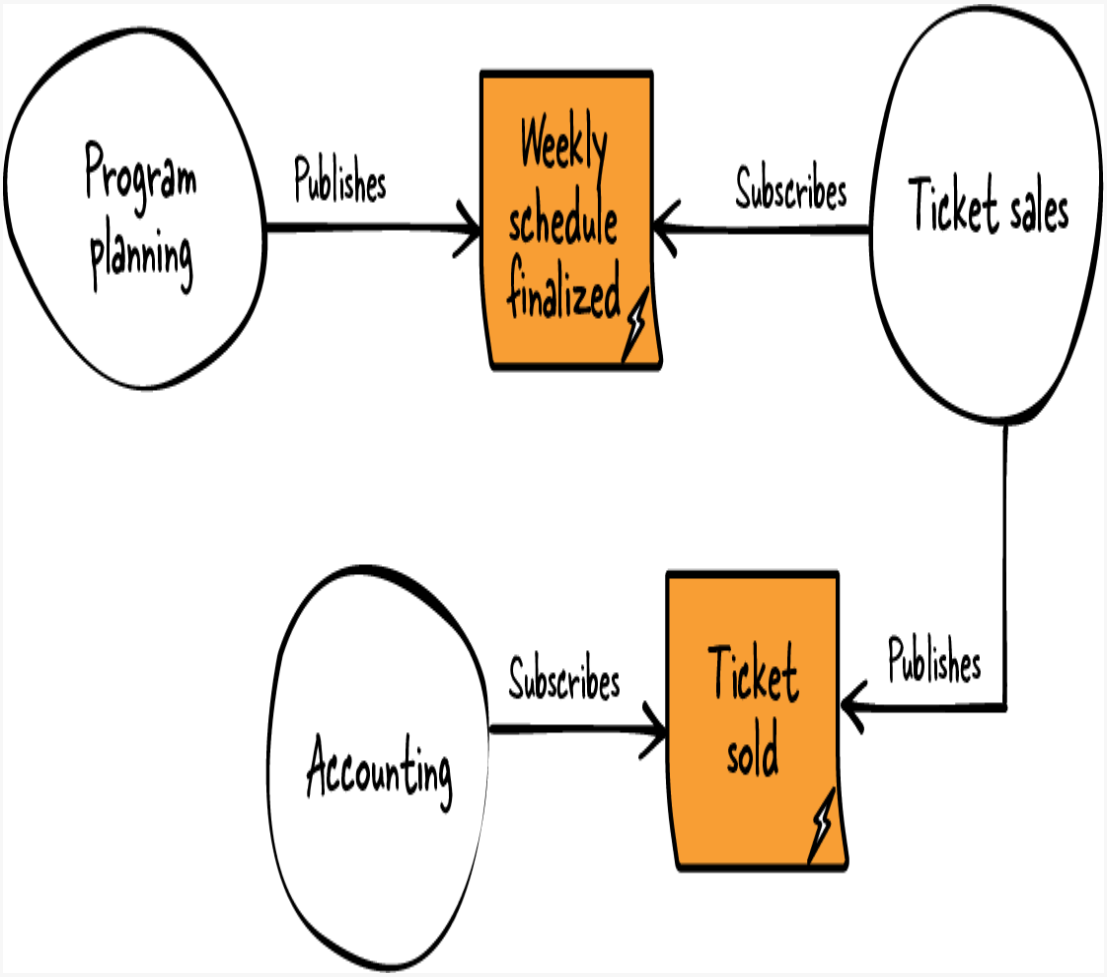


Figure 3-20. Events in the cinema context map.

"Weekly schedule finalized" is published only once per week and is always different. "Ticket sold" occurs many times per day and often

occurs as a whole series of similar events.

Both these events follow the idiom "noun followed by a verb in past tense." Considering the behavior `assignMovieToTheater(movie, theater)`, the event `MovieAssignedToTheater` works best, rather than the idiomatic form of "noun followed by verb in past tense." In this case there are two nouns and one verb. The name `MovieTheaterAssigned` could work, but it doesn't sound as natural as `MovieAssignedToTheater`.

Domain events, just like value objects, are always immutable and usually contain a timestamp of when the event happened in addition to their name. This allows the recipient to see the sequence of events with the same name.

Events can be implemented as "thin" or "rich." A "thin" event contains only the identity of the entity (or entities) involved in the state change, while a "rich" event contains all information about the state change.

CASE STUDY "CINEMA"

IN THE CINEMA: RICH AND THIN DOMAIN EVENTS

The domain event "weekly schedule finalized" is implemented as a thin event that only contains the number of the cinema week for which it is valid. The actual weekly schedule must then be fetched from the program planning via an open host service.

"Ticket sold" is "rich" and contains the amount paid, whether it was paid in cash or by card, the film for which the ticket was sold, and of course, a timestamp for the booking.

Domain events help not only in modeling domain communication between bounded contexts, but in other ways as well:

- If the state changes of entities are to be tracked, a list of domain events is a helpful way to travel in time. Elsewhere, the developers

have to build additional possibilities into their systems with history functions or use the change tracking features of the database. This is where domain events fill a frequently occurring gap. This approach is known as *Event Sourcing*.

- In distributed systems, the difficulty arises that the state of different parts of the system cannot be kept completely consistent at all times. If updates do not arrive in the correct sequence or from different sources, we can possibly resolve this issue using the timestamps in the events. For more on this topic see [Vernon 2015].

Domain events are a great help in the decomposition of legacy systems because they decouple the software and at the same time increase the domain-oriented expressiveness of the models.

Summary

In this chapter, we have introduced you to the various concepts of Domain-Driven Design. As a starting point, we have taken the question of reducing complexity in software development and have shown that Domain-Driven Design offers both for complexity in the problem space and for complexity in the solution space.

---

1 Thanks to Vaughn Vernon for this analogy.

2 In fact, those are additional good reasons not to use “get” and “set” in the domain model, but rather to prefer fluent expressions that are closer to a true ubiquitous language. Do domain experts ever say “get X” or “set Y”? In most domains they do not, which means they are poor words to introduce because they are programmer technical rather than business-centric.

3 In some cases, a shared kernel can be used, e.g., for common value types. But a larger unique model should not be shared.

4 Being German native speakers trying to find the right English word, we agree with Martin Fowler, who observes: “Datensparsamkeit is a German word that’s difficult to translate properly into English. It’s an attitude to how we capture and store data, saying that we should only handle data that we really need.” See

https://martinfowler.com/bliki/Datensparsamkeit.html. We're using the translation of the European Union laws here (see *https://iate.europa.eu/entry/result/3520793/de-en*).

5 Of course, programming is best seen in code. To not let this chapter become too long, we left out code examples here. You can find examples in code for the building blocks in Chapter 8.

6 We recommend holding such references in application services, if possible.

7 Of course, there can be different films with the same title. Since the cinema never shows films of the same title at the same time, the film title is unique enough. In other situations, a combination of film title and release year might be better suited to model the identity of the film.

Chapter 4. Collaborative Modeling

> **TIP**
>
> This chapter presents knowledge that is foundational for Domain-Driven Transformation. If you are already familiar with a Collaborative Modeling method presented here, you can skip that section and come back to it later for reference.

In recent years, a group of new workshop formats has received a lot of attention in the DDD and Agile communities, and has become more and more widespread: methods grouped under the label *Collaborative Modeling* (also known as *CoMo* for short).

In this chapter, we will look at CoMo in general and three CoMo methods in particular: Domain Storytelling, EventStorming, and Scenario Casting. An overview of Domain Storytelling and EventStorming will be presented, like the rest of the content in this first part of the book. Scenario Casting will be presented in depth, since this is new content that has never been published in any book before.

Collaborative Modeling stands behind many successful methods that focus on "requirements analysis" in the problem space[1] (see Figure 4-1).

CoMo helps to solve an age-old sociotechnical problem that derails collaboration. Where does this problem come from?

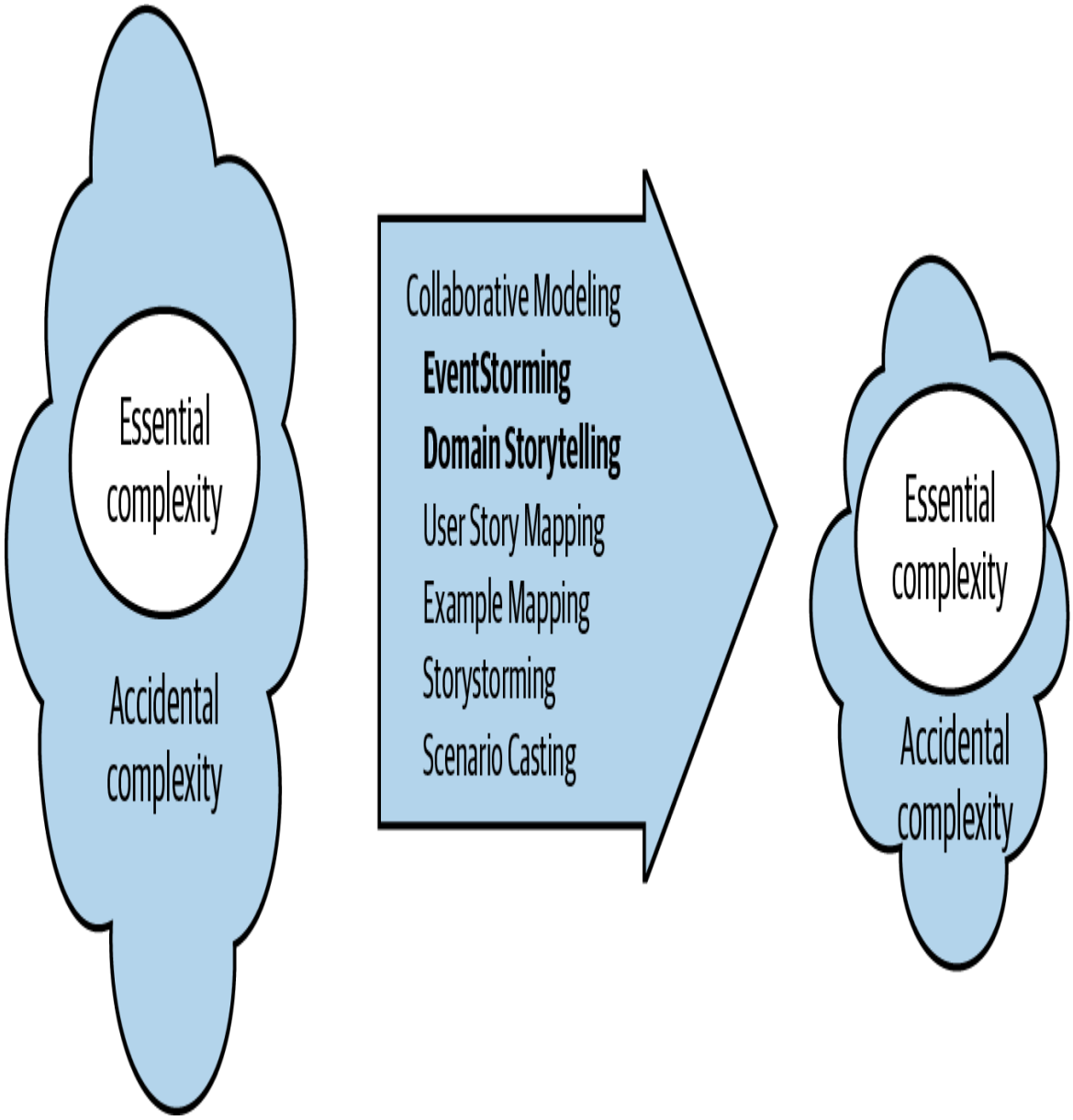


Figure 4-1. Complexity and CoMo.

Modeling Tools and the Model Monopoly

The crucial point in all methods for Collaborative Modeling is that the people who are not software developers should not have any problems with understanding the notation in the discussion about the domain. If you choose a technical notation, such as UML or BPMN, for analyzing the domain with the domain experts, a phenomenon called model monopoly in sociology occurs [Bråten 1973].

A *model monopoly* occurs when a notation is used for a model that one group of participants is more experienced in than another in a discussion or understanding process. Domain experts usually have no training in technical notations, so they are at a distinct disadvantage compared to software developers or business analysts when analyzing the domain based on these notations. If you are not aware of this problem and use a notation that poses a comprehension hurdle for the domain experts, then you will get less feedback and corrective comments when developing and presenting the results. This will ultimately lead to bad-fitting software.

The bottom line is that with a notation that is difficult for the domain experts to understand, you run into a frustrating and expensive trap in software development: the domain experts agree to the results of the analysis of their business processes without many comments because they have difficulties understanding the notation and cannot/will not question the expertise of the other side. Or they just get frustrated and refuse to participate in the future. The software is (further) developed on the basis of these analyses and the resulting requirements specification. But when the software is delivered, it becomes clear that the software or its extension cannot be used in this way. The users need a slightly different domain model for their work, and the processes defined in the software are missing steps, or the sequence of steps does not make sense in some cases. Many workarounds have to be built in, and the chosen domain model has to be revised. All in all, the reconstruction of the software in a meaningful direction at the time of delivery is very time-consuming and in part no longer feasible. None of this describes domain-driven collaboration and design.

If only, when analyzing the domain, a notation had been chosen that the domain experts could understand and thus get valuable feedback before implementation!

Of course, the modeling tools and notations developed in software engineering, such as BPMN, UML, and others, are important and usable, just not for communicating with domain experts. These tools and notations are intended as a means of communication between developer and

developer. CoMo is a means of communication between developers and domain experts. Formal correctness is important for source code-related communication between developers. In Collaborative Modeling, however, we deliberately refrain from it.

Collaborative Modeling Methods for Transformation

In Figure 4-1 you can see the following CoMo methods currently used in the community:

- EventStorming [Brandolini 2021]
- Domain Storytelling [Hofer and Schwentner 2022]
- User Story Mapping [Patton 2014]
- Example Mapping [Wynne 2015]
- Storystorming [Schimak 2019]
- Scenario Casting [Koch 2022]

EventStorming and *Domain Storytelling* are bolded in Figure 4-1 because we use them most often in our transformation work.

The advantage of Collaborative Modeling is that these methods make it possible for all participants to engage directly in the requirements analysis process in a lightweight way, using only a pen and paper or sticky notes and perhaps a whiteboard (see [Floyd 1993]). Domain experts and users alike have access to the model because they create it together and later use it together. The barrier to entry is low for both domain experts and users because the notations used can be explained in five minutes. The beauty of the various CoMo methods is that they are all easy to understand, even for people who are not familiar with IT (see “Modeling Tools and the Model Monopoly”).

Due to their lightweight nature, these methods quickly provide answers to, for example, the following questions:

- What is really going on in a domain?
- What roles do the various domain experts have?
- In which terms do the domain experts speak?
- How do the domain experts work, and which workflows (business processes) do they perform?

These questions (and Collaborative Modeling in general) help us to find out what the problem really is and guide us to design the right solution for the right problem. This corresponds to the ideas of "divergence-convergence" and "double diamond design" from Design Thinking (see [Design Council n.d.]) and is visualized in Figure 4-2.

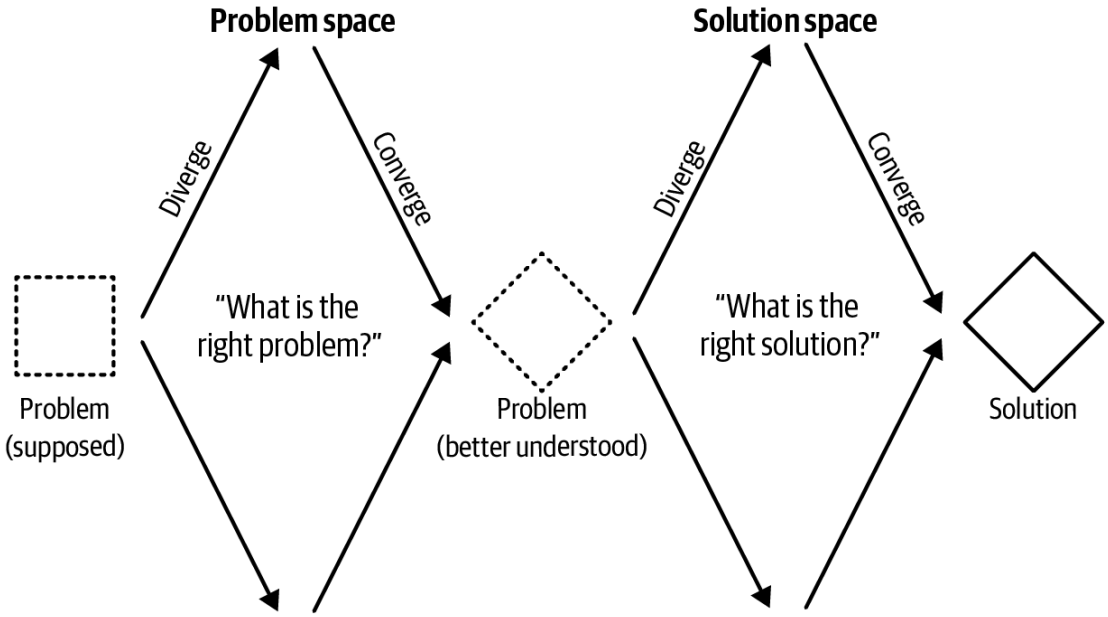


Figure 4-2. Double diamond design model.

To build new software and understand and improve old software, IT experts need to understand the domain, i.e., the tasks and workflows (business processes) of the domain experts. Conversely, domain experts should understand what opportunities the software opens and how digitalization will affect their daily work.

Developers and domain experts need a *common language* (see "Ubiquitous Language") to be able to talk about the domain and the resulting requirements for the software. Only in this way will the software reflect what the developers have understood from the requirements and what the domain experts expect from digitalization and automation of the business processes.

Bringing the Right People Together

With Collaborative Modeling, we bring the business and technical people together and make sure that they exchange ideas directly:

- Those who need or want to learn (typically the developers)
- The knowledge holders (typically users, domain experts, and business analysts)

CoMo brings domain expertise from the minds of domain experts to the minds of developers, testers, product owners, product managers, and business analysts—to everyone involved in software development. The bottom line is that through direct communication and working together, Collaborative Modeling supports the learning process that is essential to understanding a domain (see Figure 4-3).

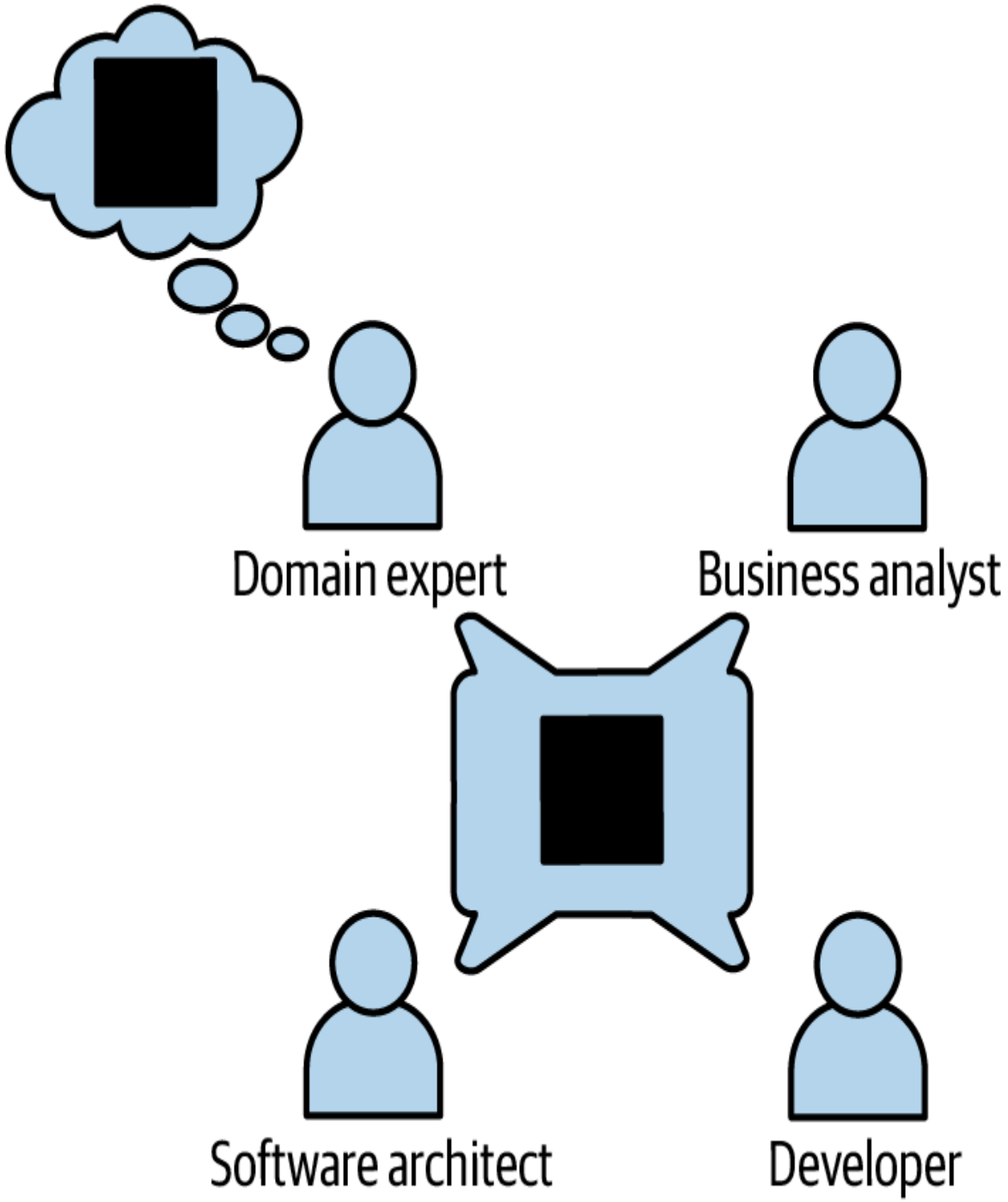


Figure 4-3. Collaborative Modeling: direct communication leads to shared understanding.

In classic requirements engineering methods, interviews are conducted and requirements are subsequently derived from them. In contrast to CoMo methods, the various participants, who all have their own view of the domain and/or the software used, are *not* brought together in a workshop in classic requirements engineering. Not achieving a common understanding

in a large group is like playing the telephone game, which leads to a variety of misunderstandings (see Figure 4-4).

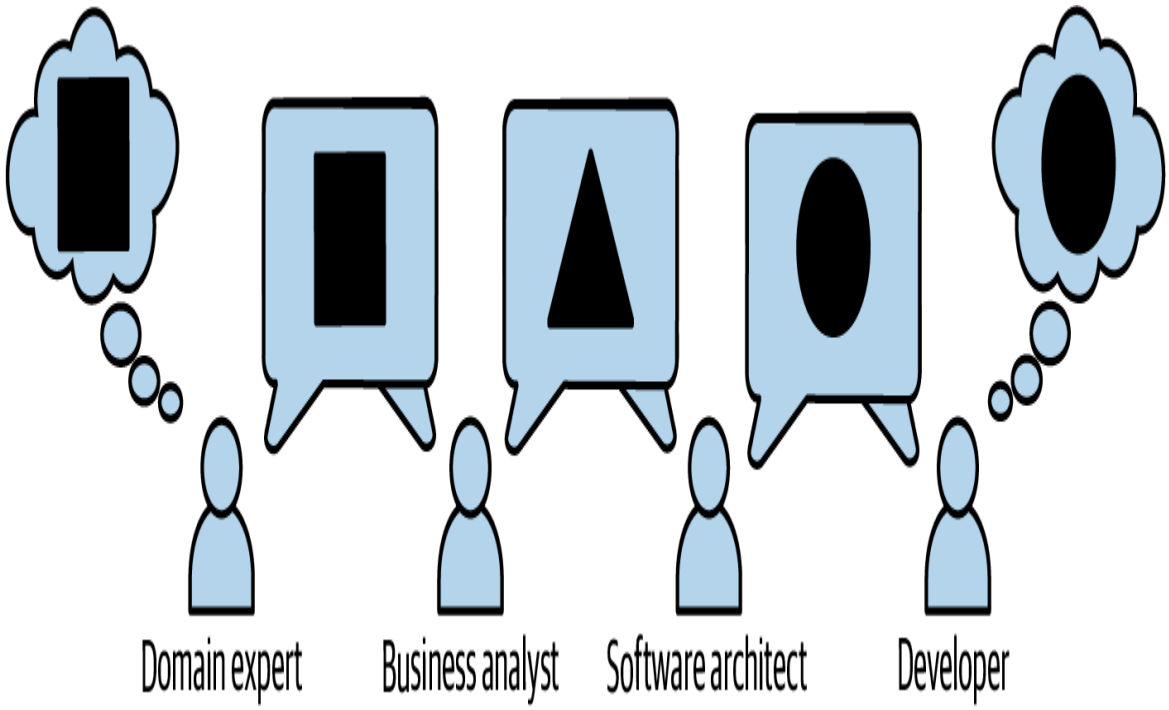


Figure 4-4. Not *Collaborative Modeling: playing the telephone game leads to misunderstandings.*

In the quite different CoMo methods, similar concepts can be discovered: they make visible the spirit of CoMo to iteratively gain new insights in open and respectful discussions by skillfully asking questions and crossing different perspectives on the same problem. Let's take a look at these concepts.

Fundamental Concepts of Collaborative Modeling

The various techniques of Collaborative Modeling build on the following common concepts (see [Hofer et al. 2020]):

Group work of all stakeholders

> The basic approach is that the various stakeholders on the business side and the development side work together to clarify business processes and requirements, rather than individual specialists specifying these

issues on the basis of interviews and document analyses. All approaches provide for joint workshops, sometimes in large groups, sometimes with a few representatives of the groups.

Telling stories

In the workshops, participants discuss concrete scenarios instead of abstract descriptions. Scenarios are told like stories. Once the scenarios are understood by everybody, the abstractions and "complete" images of rules and case distinctions necessary for software development are built. There are differences in whether a separate image is generated for each case or all relevant cases end up in one image.

By using a common language, strive for mutual understanding among all those involved

People often complain that the business departments and IT live in their own worlds and misunderstand each other. All CoMo approaches, by contrast, prioritize the language of the users; the various workshop techniques are intended to help those involved to understand each other better.

Building understanding through visual means

Direct conversation and listening has been, is, and will continue to be the best way to communicate with each other. Often when communicating without visualization, the other person's contribution is misunderstood. Visualization opens an additional feedback channel and helps to really get to the heart of the discussion and look at different ideas.

Creating shared models of the application area

Our experience with CoMo shows that joint work should have a tangible result. The various models made from drawings, index cards, or sticky notes are more than a mere byproduct. They show all participants the state of the discussion, and their graphic representation offers a new

view on the topic and marks a tangible result at the end of a workshop, which can often be used in further work.

Exploratory learning in modeling

Complex modeling takes place in an initially unknown knowledge space. We do not know beforehand who will ask the right questions or who will give the decisive answers. There will also not be that one expert who is perfectly familiar with all parts of the domain; usually each participant has a different depth of understanding of different parts of the domain. In addition, in traditional software development, the development team often receives user feedback late in the project. Collaborative Modeling is a very good starting point to ensure that the necessary discussions take place at the very beginning of a project.

In Collaborative Modeling, the problem is visible to everyone on the whiteboard, canvas, wall, or blackboard, and participants work with, rather than against, each other to find a solution. This requires all participants to be humble and recognize that their knowledge, while essential, only goes so far, and to give way to those who have specific areas of expertise. Arguing a point that merely qualifies as an opinion will only serve to short-circuit collaboration.

Collaborative Modeling and DDD

In Evans' "blue book" there are several chapters in which he talks about something he calls *knowledge crunching*. Concrete methods for crunching the knowledge are missing in his book as well as in the "red book." This gap is filled by Collaborative Modeling.[2]

Fortunately, CoMo can be combined very well with DDD. With CoMo we discover the language of the domain experts, events, actions, work items, and structures of the domain. All these elements influence the domain model, which the developers map in the software. A valid domain model can only be created jointly by developers and domain experts. DDD

experts, such as Paul Rayner, therefore see Collaborative Modeling as an important pillar of DDD (see Figure 4-5).

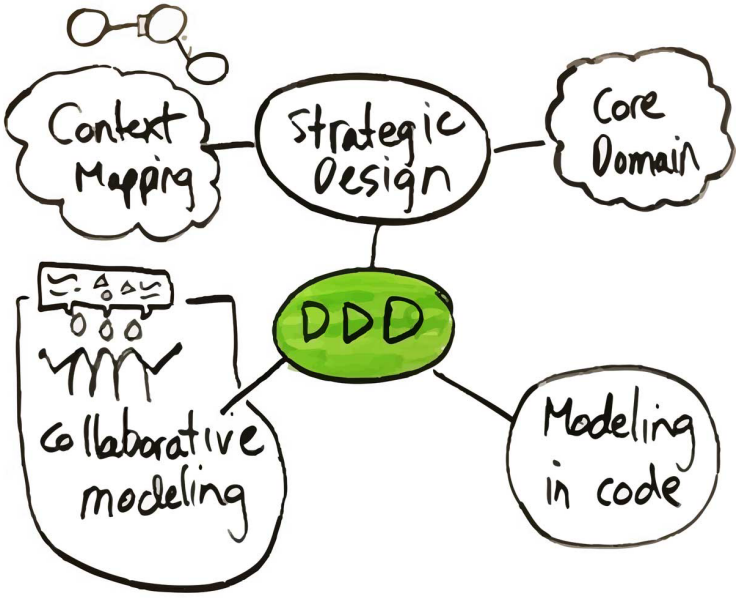


Figure 4-5. Collaborative Modeling as a pillar of DDD.[3]

In our approach, *Domain-Driven Transformation*, we rely on two methods from Collaborative Modeling when decomposing legacy systems: Domain Storytelling and EventStorming.

Domain Storytelling

Domain stories [Hofer and Schwentner 2022] combine a simple graphical notation with a workshop format to capture business processes. In doing so, we try to address the fundamental question:

> *Who (actor) does what (activity) with what (work items) when (sequence number) with whom (other actors) and why (goal)?*

The notation for domain stories specifies two types of icons and one type of arrow:

Actor (icon)

> An actor is an acting person or organizational unit (or sometimes an acting IT system).

Work object (icon)

> A work object is a thing/item from the domain with which something is done.

Activity (arrow)

> An activity is an action that one or more actors perform on or with one or more work objects.

These symbols are used to form sentences. If you want to read a domain story, you follow the *sequence numbers* on the sentences. Everything that cannot be expressed with these icons is written down in *annotations*.

Domain stories are created by letting domain experts tell us what is happening in the domain. During the narrative, the story as we understand it is recorded as a diagram. In this way, we provide direct feedback on understanding, and any misunderstandings are quickly cleared up. This point is especially important when moving quickly. Exchanging information about a domain in writing is far less effective than direct exchange. Documenting the common understanding is very important for a progressive learning process—for this, domain stories are a solution that is understandable to all participants.

CASE STUDY “CINEMA”

IN THE CINEMA: FIRST DOMAIN STORIES

In the Metropolis movie theater, we conduct our first workshop to analyze Cine2001 with the domain experts and developers. We ask them to give us a broad view of how a typical cinema visit happens. The domain experts tell us the following:

1. A moviegoer buys a ticket from a cashier.
2. The moviegoer shows this ticket to the usher.
3. The projectionist starts the movie.
4. The moviegoer watches the movie.

While we hear this story, we visualize it sentence by sentence in a diagram (see Figure 4-6).

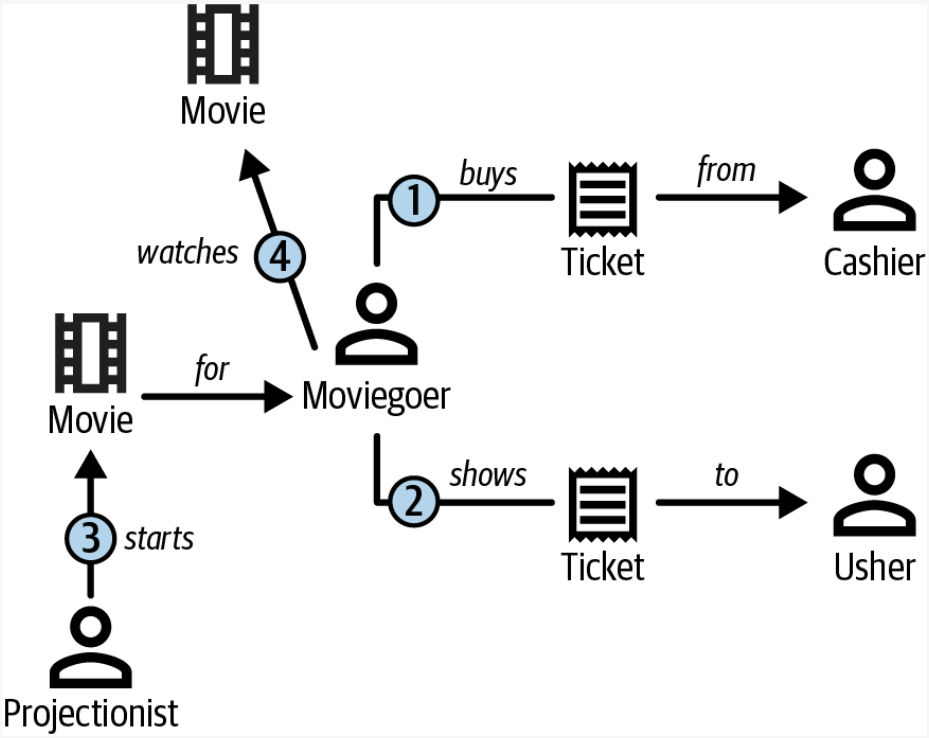


Figure 4-6. Typical visit to the Metropolis as a domain story.

The Domain Storytelling notation can be found in Figure 4-6: the "moviegoer" is an actor, "ticket" is a work object, and "buys from" is

an activity (and has the sequence number 1). To read the domain story, just follow the sequence numbers.

Next, the team goes deeper into step 1 from Figure 4-6 (selling tickets and what is necessary to do so). They draw another domain story that describes which work steps the cinema manager and the cashiers in the box office do today with Cine2001 and how they work together in this business process.

Figure 4-7 shows that the cinema manager at the top left prepares the weekly schedule (step 2) for the coming week in collaboration with the advertising agency (step 1) and the film distributor (step 3). Once the weekly schedule is ready, they finalize it (step 4), thereby releasing the schedule for ticket sales. The cashier can now sell movie tickets to the moviegoer (steps 5 to 11) by searching the seating plans for suitable seats (step 7) and offering them to the moviegoer (step 8). Finally, the moviegoer pays (step 9) and leaves satisfied with their printed tickets (step 11).

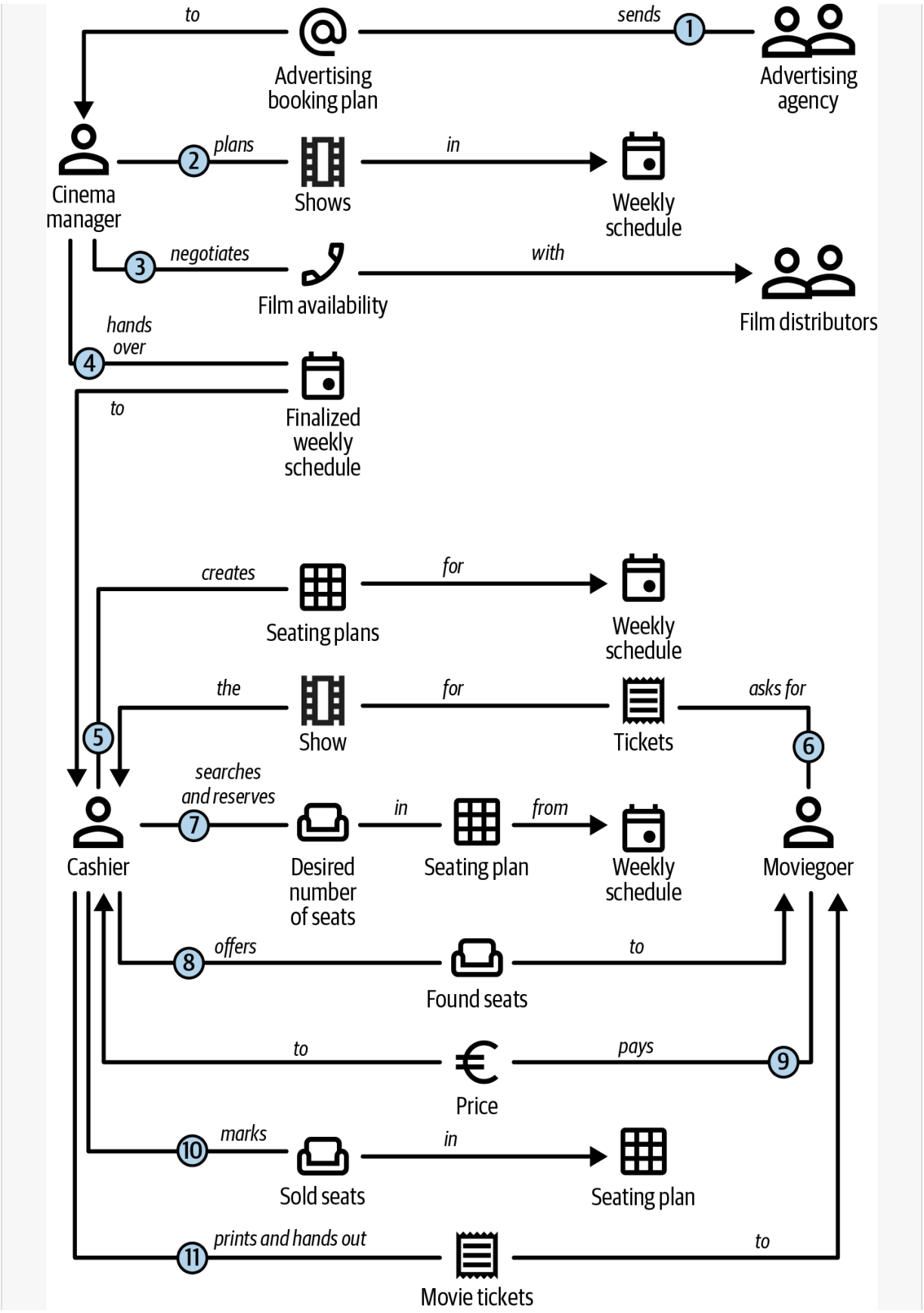

to
Advertising booking plan
sends
1
Advertising agency
Cinema manager
2
plans
Shows
in
Weekly schedule
3
negotiates
Film availability
with
Film distributors
hands over
4
to
Finalized weekly schedule
creates
Seating plans
for
Weekly schedule
the
Show
for
Tickets
asks for
6
5
searches and reserves
7
Cashier
Desired number of seats
in
Seating plan
from
Weekly schedule
Moviegoer
8
offers
Found seats
to
to
Price
pays
9
10
marks
Sold seats
in
Seating plan
11
prints and hands out
Movie tickets
to

Figure 4-7. Overall process in the cinema.

One Picture—One Story

Domain Storytelling is a scenario-based approach, which means that each diagram represents only one case. This way, the single picture remains understandable. For further (interesting) cases, additional domain stories are drawn.

CASE STUDY "CINEMA"

In the cinema, we have so far considered the "happy path." The next step is to draw domain stories for the relevant special cases, such as:

- The show the moviegoer wants to book a ticket for is fully booked. (Then the cashier offers a show for a different film at the same time or a different show for the same film.)
- The cashier finds the desired number of available seats, but these are not next to each other. (Then the cashier offers to either book the nonadjacent seats or a different show.)
- The moviegoer cannot pay the price directly because they forgot their wallet. (Then the cashier reserves the tickets for a certain time.)

We omit the domain stories for these and other cases here for the sake of brevity.

Drawing Boundaries

Domain stories offer *groups* as a further element of the pictographic language. These can be used, among other things, to make subdomains visible.

CASE STUDY “CINEMA”

IN THE CINEMA: BOUNDARIES IN THE DOMAIN STORY

In this domain story of the movie theater, we draw the same subdomains that we had already found back in Figure 3-7 based on the core concepts: the subdomain "weekly schedule" and the subdomain "ticket sales." In step 4 of Figure 4-7, the cinema manager hands over the finalized weekly schedule. This is an instance of both indicators completing work results and passing the work results one-way (more on that in "Indicators for Subdomain Boundaries"). This confirms that the found boundaries are feasible (see Figure 4-8).

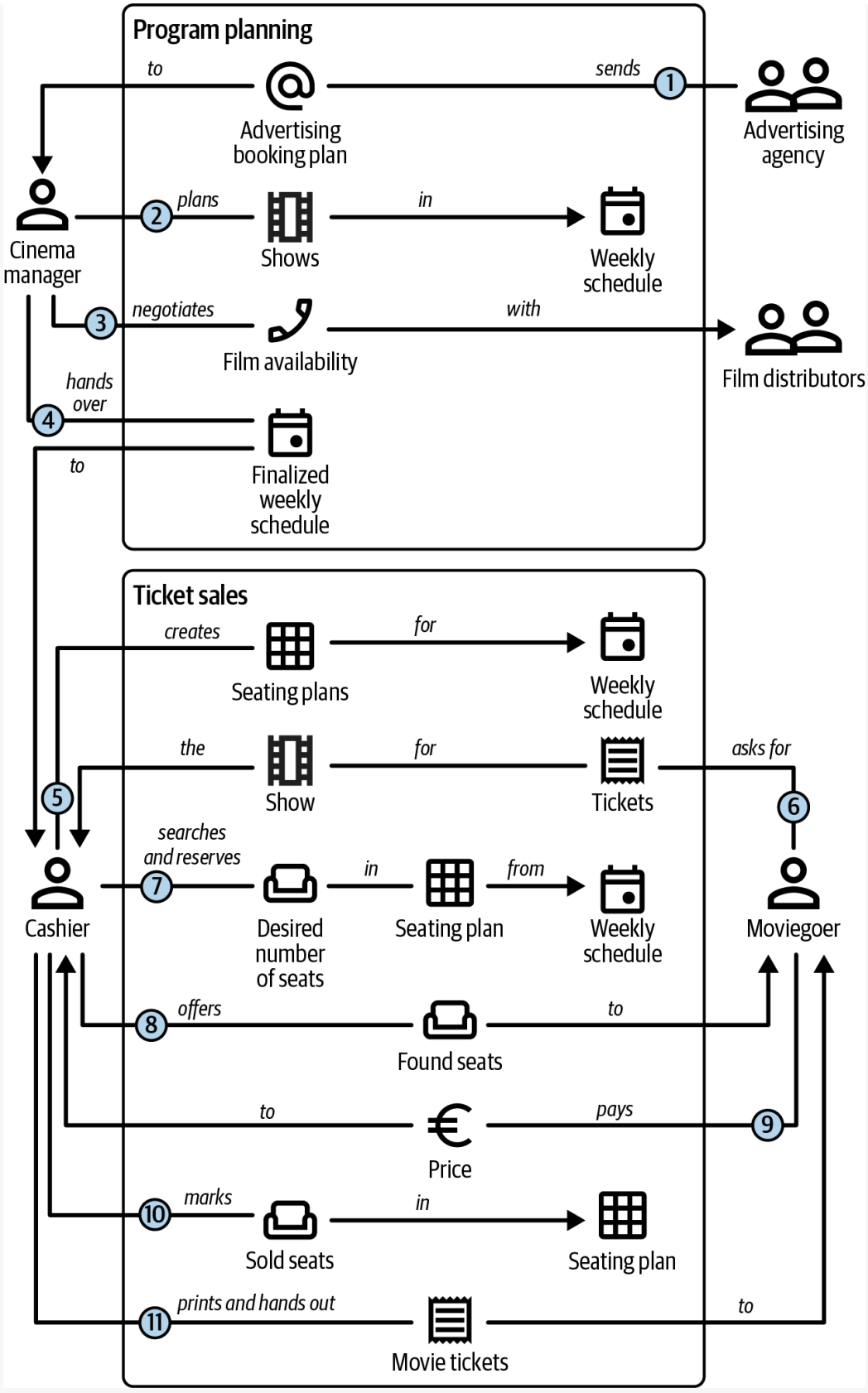


Figure 4-8. Overall process in the movie theater with subdomains.

We show more about finding subdomains with domain stories in Chapter 10.

In addition to the basic technique of developing domain stories and dividing them into subdomains, Henning and his coauthor Stefan introduce different types of domain stories in their book ([Hofer and Schwentner 2022]), which are categorized based on *scope factors*.

Scope Factors for Domain Stories

Domain stories have different characteristics, which are determined by the *scope*. The scope of a domain story is made up of the following factors:

Granularity

> Domain stories can be coarse-grained, fine-grained, and everything in between. The spectrum is continuous. It is important to achieve a consistent level of detail in a domain story. Mixing fine-grained and coarse-grained activities is confusing and usually indicates that some knowledge holders were missing when the domain story was created.

Domain purity

> Domain stories can be modeled in a purely domain-oriented way, so that only human actors and no software systems appear in them. Such domain stories that are purely domain-oriented are called *pure*. If one or more software systems appear in the domain story, it is called *digitalized*.

Point in time

> Domain stories can be modeled for the current situation, the *as-is*, or for the future, the *to-be*. Depending on whether you want to first understand a domain as it is, or whether you are planning the next step in the future, the domain story has a different point in time.

For each domain story, a triple of these three factors should be specified and what their respective values are (X, Y, Z). For Figure 4-6, this triple would look like: (coarse-grained, pure, as-is), and for Figure 4-7 it would look like: (medium-grained, pure, as-is). The scope factors will help us later in the Domain-Driven Transformation to crystallize the domain and decompose the legacy system on this basis.

CASE STUDY “CINEMA”

IN THE CINEMA: DIGITALIZED DOMAIN STORY

The difference between pure and digitalized domain stories can be seen in the two domain stories that show only the sales process in the cinema example (see Figure 4-9 and Figure 4-10). The digitalized story in Figure 4-9 shows the sales process in which the cashier sells tickets for a show to the moviegoer. The cashier uses the software system Cine2001, which is shown on the left side of the domain story.

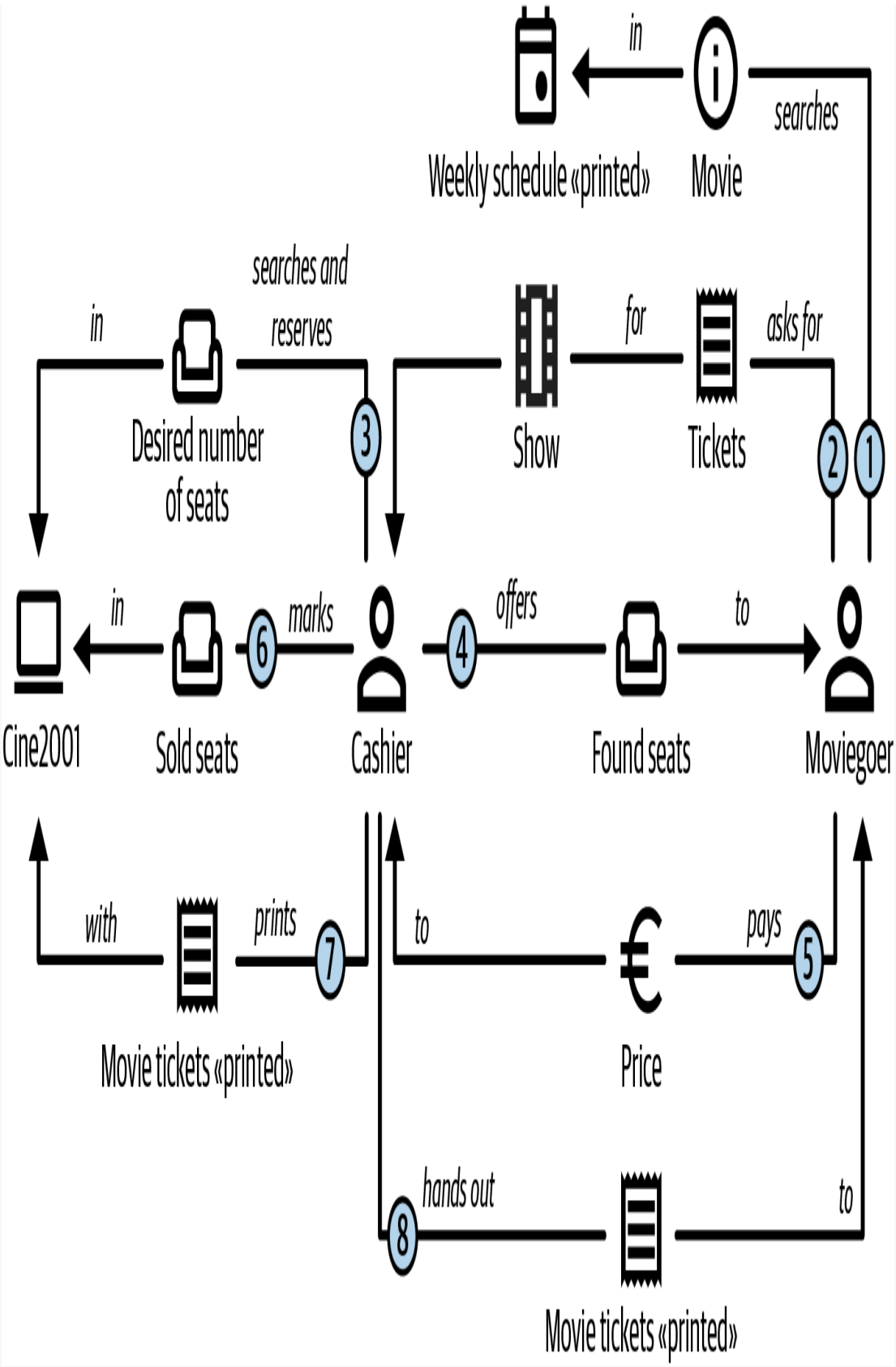


Figure 4-9. Sales process in the cinema with a system.

If you compare this domain story with the coarse-grained process from Figure 4-7, you will see that the seating plan does not appear here at all.

Something like this often happens when you take up an already digitalized process with the domain experts. The domain experts are so used to working with their software system that they no longer think of the original domain concepts or have even forgotten them. Our task is then to distill out the pure business process.

CASE STUDY “CINEMA”

IN THE CINEMA: PURE DOMAIN STORY

If we now look at the next story in Figure 4-10, which has been "purified" of technology, we see that Cine2001 no longer appears, but instead the seating plan appears as an IT-supported artifact with a domain-specific name. This artifact, which is after all very important for the cinema, only became visible in the requirements analysis when we motivated the domain experts to talk about how things would run if all activities were motivated only by the domain (and not by the existing software systems).

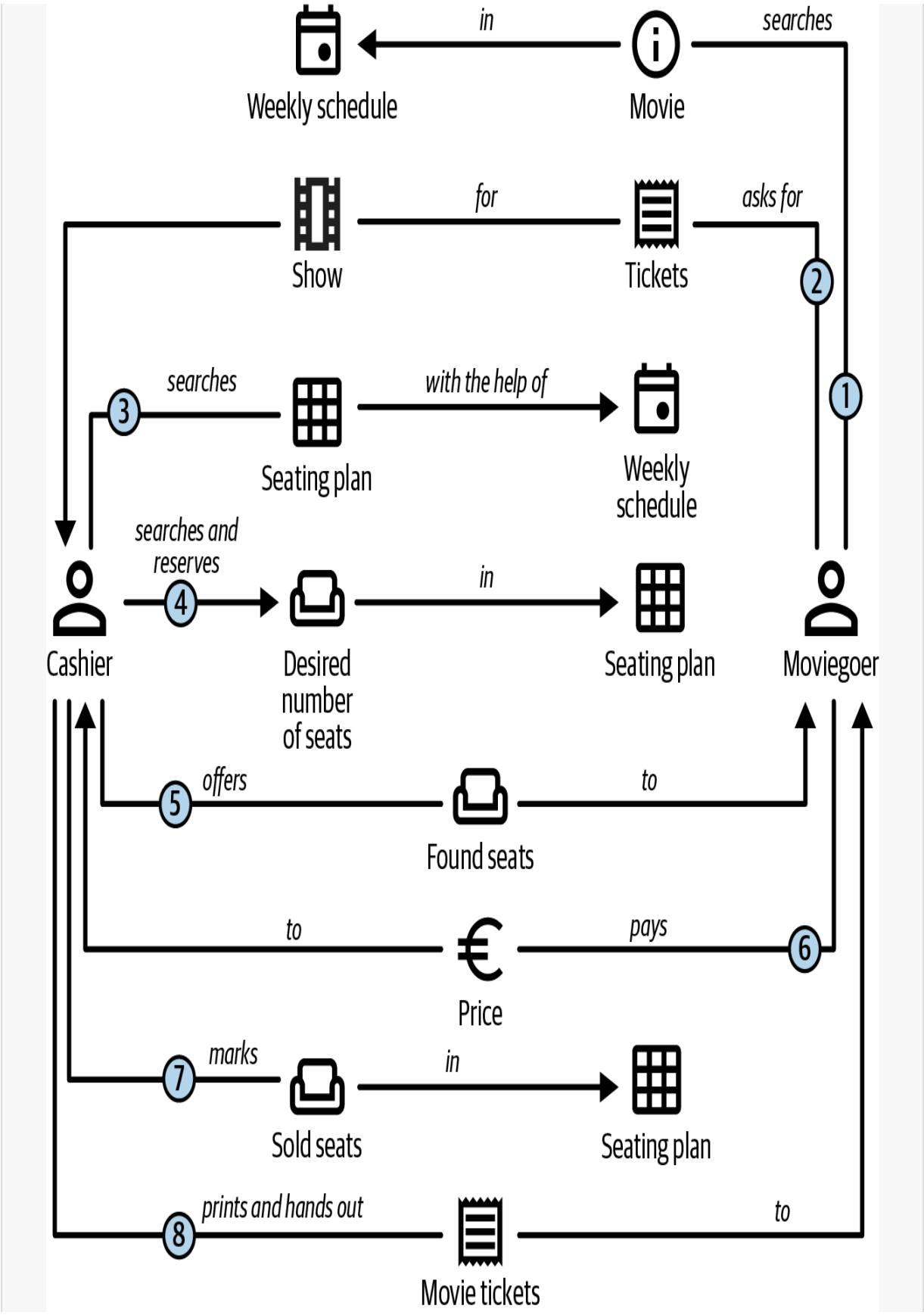

in
searches
Weekly schedule
Movie
for
asks for
Show
Tickets
2
searches
with the help of
1
3
Seating plan
Weekly schedule
searches and reserves
in
4
Cashier
Desired number of seats
Seating plan
Moviegoer
offers
to
5
Found seats
to
pays
6
Price
marks
in
7
Sold seats
Seating plan
prints and hands out
to
8
Movie tickets

Figure 4-10. Sales process in the cinema without an IT system.

Finding the Domain Model

If we apply Domain Storytelling in such a way that we have pure domain stories, then these stories should contain only domain terms and no technical terms. If we have achieved this, then we have really focused on the domain and found many terms for the ubiquitous language and thus for the domain model.

CASE STUDY “CINEMA”

IN THE CINEMA: DOMAIN LANGUAGE

In the pure domain story of the sales process in the last section (see Figure 4-10) and also in the overall process (see Figure 4-7), we are exclusively using the domain language of the cinema. All the terms used there are candidates for the ubiquitous language of the cinema system: “weekly schedule,” “show,” “ticket,” “seating plan,” “seat,” and so on. We are interested not only in the nouns, but also in the verbs: “offer,” “pay,” “plan,” etc.

With these terms and verbs, we can go directly into modeling and derive the domain model from the domain story. We design two classes, `SeatingPlan` and `Seat`, and add the method `markAsSold()` to the `SeatingPlan` class in order to model a typical usage of the seating plan by the cashier (see Figure 4-11).

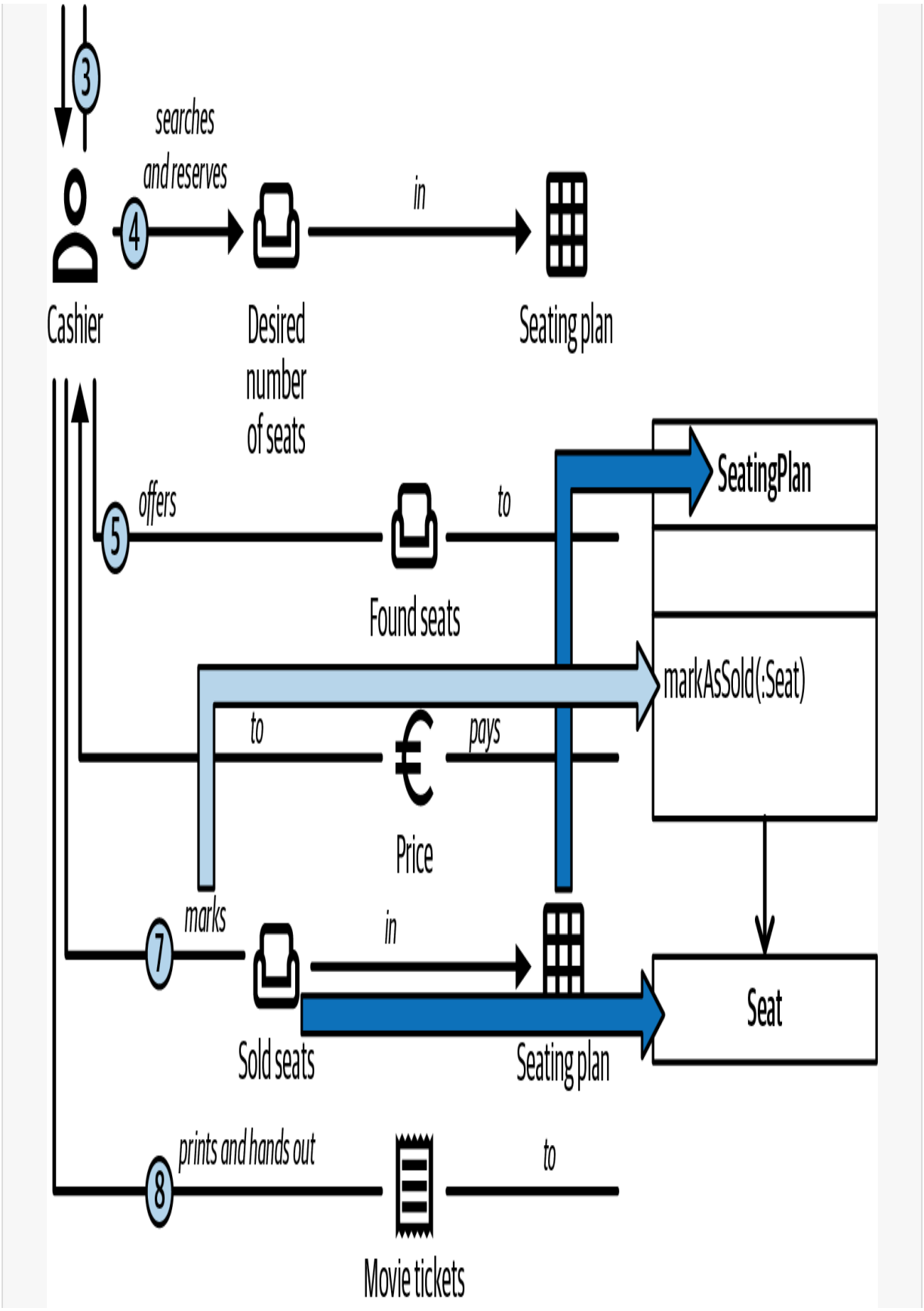

3
searches
and reserves
4
in
Cashier
Desired
number
of seats
Seating plan
SeatingPlan
offers
5
to
Found seats
markAsSold(:Seat)
to
pays
Price
marks
7
in
Seat
Sold seats
Seating plan
prints and hands out
8
to
Movie tickets

Figure 4-11. From domain story to domain model.

Outlook

If you want to learn the whole Domain Storytelling technique, we recommend the book of the same name by Henning and Stefan Hofer [Hofer and Schwentner 2022] and the Domain Storytelling website (*https://domainstorytelling.org*).

EventStorming

EventStorming is a family of workshop formats that explore complex domains and model the supporting software collaboratively. EventStorming was invented by Alberto Brandolini.[4] The first article about EventStorming appeared in 2013 and initially focused on discovering aggregates as a natural consequence of an event-driven flow (see [Brandolini 2013]).

Later, the formats evolved into three different recipes that can be seen as a zoom-in progression:

Big Picture

Used primarily for large-scale discovery (the typical scope is an end-to-end business line)

Process Modeling

Used for the discovery of intricate business processes or for a software-agnostic design session

Software Design

Used for collaborative design of complex software processes supporting key business interactions

Exploring with Big Picture EventStorming

In a Big Picture format, the participants (usually 20–30 people from different departments) are asked to use orange sticky notes to visualize the key events happening on a timeline.

In Domain Storytelling the fundamental question was: "Who does what (activity) with what (work objects) when, with whom, and why?" Meanwhile, in EventStorming, this fundamental question is:

> *"Which events are happening in the domain, and when?"*

In EventStorming, a common understanding of the domain progressively emerges from the discussions necessary to sort out the initial *chaotic exploration*.

Big Picture EventStorming requires a specific room preparation: participants gather in front of a wide wall,[5] to which about 10 meters of a paper roll is attached with masking tape as a *modeling surface*. Tables and chairs are moved aside or removed altogether. This ensures that everyone has equal access to the wall.

The workshop is moderated by a *facilitator*. On a lectern, standing desk, or similar, the facilitator provides the necessary modeling material: sticky notes in different colors (see "Modeling Tools and the Model Monopoly") and markers to write on the sticky notes. At the beginning, each participant gets a stack of orange stickies and a marker.

Chaotic Exploration

First, all participants write what they think is happening in the domain as events. Similar to brainstorming, the initial aim is to gather knowledge and ideas, and not yet to correct others. The whole thing can happen quietly but tends to ignite the moment participants stop their personal brain dump and start sharing the same modeling surface.

Participants write *domain events* on orange stickies and stick them on the modeling surface. Reminder (see Chapter 3): a domain event is an occurrence that is relevant to the domain experts. The default is to write down domain events in the past tense. Some stickies can appear more than

once, often in slightly different wording. This is OK and forms the basis for the discussions that are starting now.

CASE STUDY “BANK”

ANOTHER EXAMPLE DOMAIN: BANKING

As we modeled the cinema processes already with domain stories and don't want to just show the same processes in another modeling format, at this point we introduce our next example—the Big Bank. In this bank, there is a mainframe system that is used to support all processes in the bank. This mainframe system is to be replaced, but it is clear to all parties involved that this cannot happen all at once. Subareas in the old system must be found that can be replaced by new software.

We (the consultants called in) propose an EventStorming to the bank's management to decompose the domain into subdomains. To do this, we invite domain experts from different departments and the development teams to a workshop. Together, we first find the events shown in Figure 4-12.

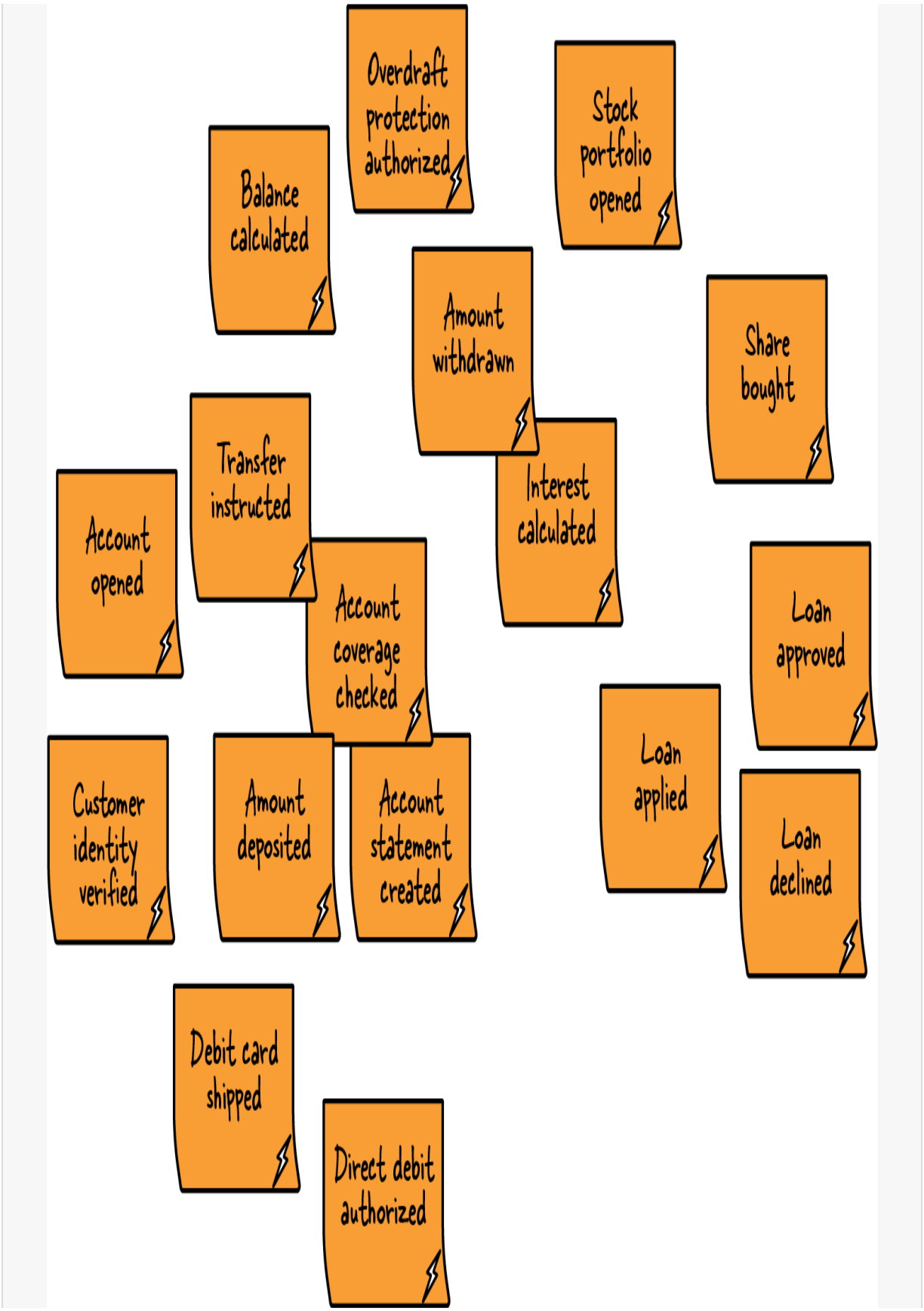

Overdraft protection authorized
Stock portfolio opened
Balance calculated
Amount withdrawn
Share bought
Transfer instructed
Interest calculated
Account opened
Account coverage checked
Loan approved
Loan applied
Customer identity verified
Amount deposited
Account statement created
Loan declined
Debit card shipped
Direct debit authorized

Figure 4-12. Chaotic exploration: all participants let their knowledge and ideas come bubbling out.

Enforcing the Timeline

To bring some structure to the evolving model, as a next step, events are placed in a roughly chronological order from left to right. This reassigns many stickies and makes the discussion broader and deeper.

Putting events in chronological order forces collaborating participants to uncover gaps in the story where more events are missing. Duplicate stickies are discussed, sometimes removed, and other times, what looked like the same event leads us into a more sophisticated wording. Rewriting and discarding notes is part of the game.

While the workshop was fairly quiet at the beginning, it is now getting louder. At the beginning, everyone is busy with their own ideas, but now they are discussing them with each other. Typical phrases you hear at this stage are:

- “Does this really happen before that?”
- “Oh, that’s how you do it!”
- “I’ve always wondered why we send this email!”

In this phase, many participants have “aha moments.” They build a shared understanding of what is actually happening. This is also when the facilitator starts adding hot spots to the model. Emerging issues are captured this way and made a colorful part of the model.

When several people work together to reach a common insight or decision, three phases can be observed. In the beginning, the space is opened and the participants start at the same point, often drifting apart at first (divergence). Then it starts to creak and something new is born (emergence). Finally, everyone comes together again and something common emerges (convergence). That is exactly what happened here.

CASE STUDY “BANK”

IN THE BANK: THE TIMELINE

In the bank, after extensive engaged discussions, the workshop participants end up with Figure 4-13.

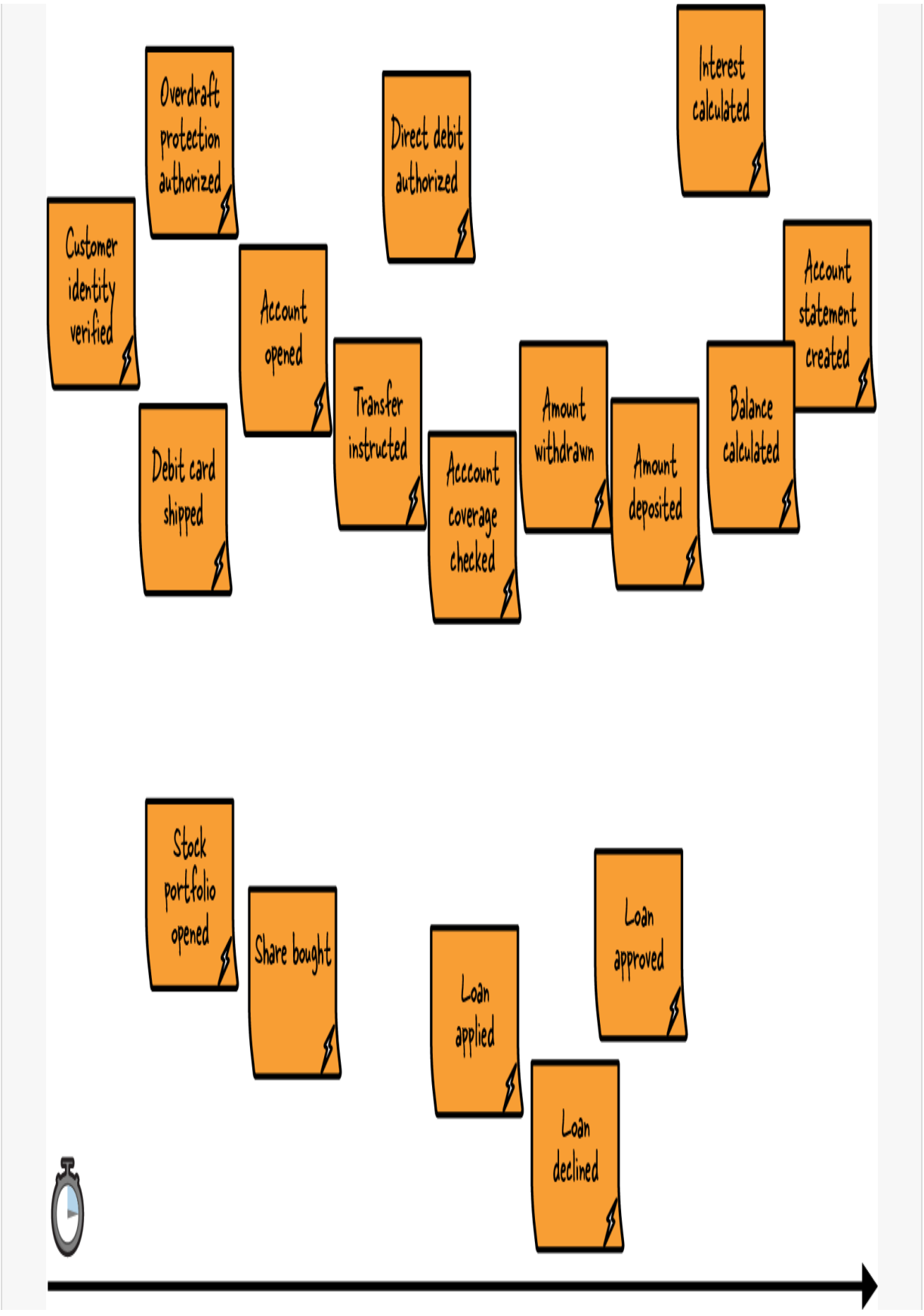

Customer identity verified
Overdraft protection authorized
Debit card shipped
Account opened
Transfer instructed
Direct debit authorized
Acccount coverage checked
Amount withdrawn
Amount deposited
Interest calculated
Balance calculated
Account statement created
Stock portfolio opened
Share bought
Loan applied
Loan declined
Loan approved

Figure 4-13. Time: the events are put in a sequence.

Sorting out many events—some workshops can flood the modeling surface with 300–500 orange stickies—requires some strategies:

- A particularly useful one is to rely on *pivotal events*: the ones that are part of many versions of the story and often act as natural boundaries between *phases* or areas of responsibility.
- Another strategy is to separate substories in the picture into *swim lanes*.

This approach simplifies the sorting because it provides some emergent structure and gives some information to draw some boundaries, usually with colored (and replaceable) masking tape.

People and Systems

Once we have a basic tentative structure, we can expand our grammar by adding *people* and *systems* and adding some more related events if necessary.

People and systems are relatively open elements of the grammar: people can be seen as *roles*, *actors*, *personas*, and sometimes first names and last names are needed. Systems are open too: they can represent software services, external departments and companies, government entities, and market conditions.

The goal of this stage is to provide enough ingredients for validating the emergent narrative.

Explicit Walkthrough

Now the participants need to make sure that what was chaotically added to the wall, based on many parallel conversations, makes sense to the eyes and ears of every participant. During *explicit walkthrough* a designated *narrator* will try to tell the story in front of the whole audience. The initial story is

never good, so a lot of rewriting and rearranging is necessary. A *scribe* ensures that the conversations and the model are in sync.

This step looks daunting at first, but progressively accelerates and leaves us with a more powerful artifact: a *validated narrative*.

Value-Added Steps

The validated narrative can now serve as a robust background for a variety of value-added steps, such as:

- Visualizing *value* on multiple currencies (money, time, happiness, stress, reputation) in a way that is similar to Value Stream Mapping
- Highlighting existing *problems* and *opportunities* for improvement
- Discovering and extracting *boundaries*
- Highlighting *areas of influence* for different teams

Let's look for emerging boundaries.

Drawing Boundaries

Next, we divide the now much better understood domain into subdomains. These subdomains are candidates for later becoming bounded contexts in the software system or guiding the decomposition of the software system.

CASE STUDY “BANK”

IN THE BANK: BOUNDARIES

In the banking example, “account opened” is identified as a pivotal event which helps to find a first boundary. The events “stock portfolio opened” and “share bought” are recognized to happen in another swim lane than the other events and that helps to find another boundary. Further discussion with the domain experts results in boundaries for subdomains, as seen in Figure 4-14.

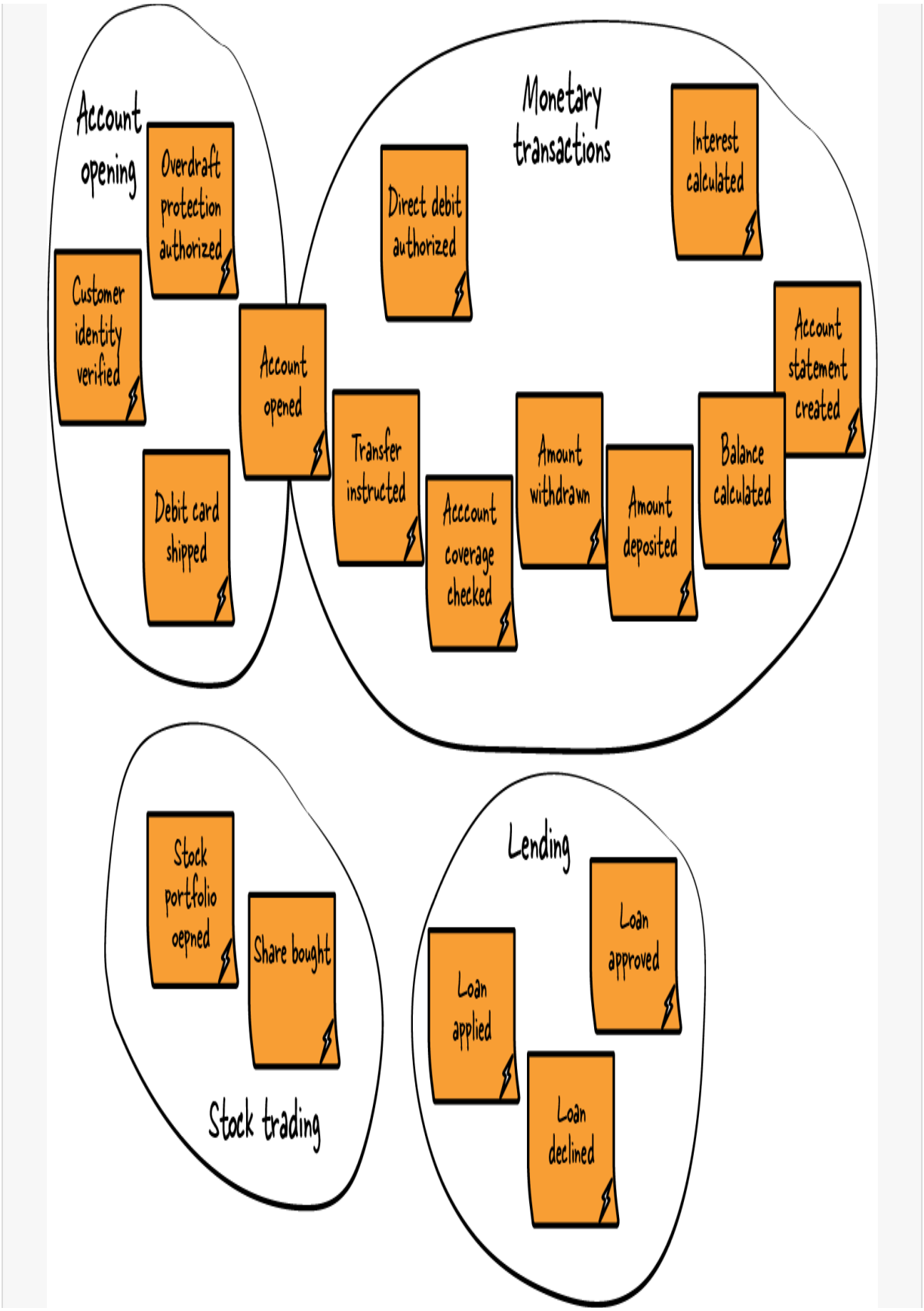

Account opening
Overdraft protection authorized
Customer identity verified
Account opened
Debit card shipped
Monetary transactions
Direct debit authorized
Interest calculated
Account statement created
Transfer instructed
Acccount coverage checked
Amount withdrawn
Amount deposited
Balance calculated
Stock portfolio oepned
Share bought
Stock trading
Lending
Loan approved
Loan applied
Loan declined

Figure 4-14. Boundaries: events are grouped together.

What we have seen so far is the first level of EventStorming—that all EventStorming workshops begin with: orange sticky notes for events that are described ("stormed out") and sorted. If necessary, more sticky notes with other colors are introduced throughout the process or later.

The Color Coding

Figure 4-15 summarizes the types of sticky notes in Big Picture EventStorming. In the middle, you can see the already familiar orange sticky note used for domain events. Next, hot spots are usually used in an EventStorming:

Hot spots

> Whenever open questions or controversial discussions slow down the modeling process, mark the disputed point with a red sticky note as a hot spot for later clarification, and redirect energy to the rest of the process.

People as well as external systems often become interesting only later:

People

> People are represented with yellow sticky notes in portrait orientation, on which the name of the role is written and a small symbol for a human person is depicted.

Systems

> Systems are shown as a wide pink sticky note.

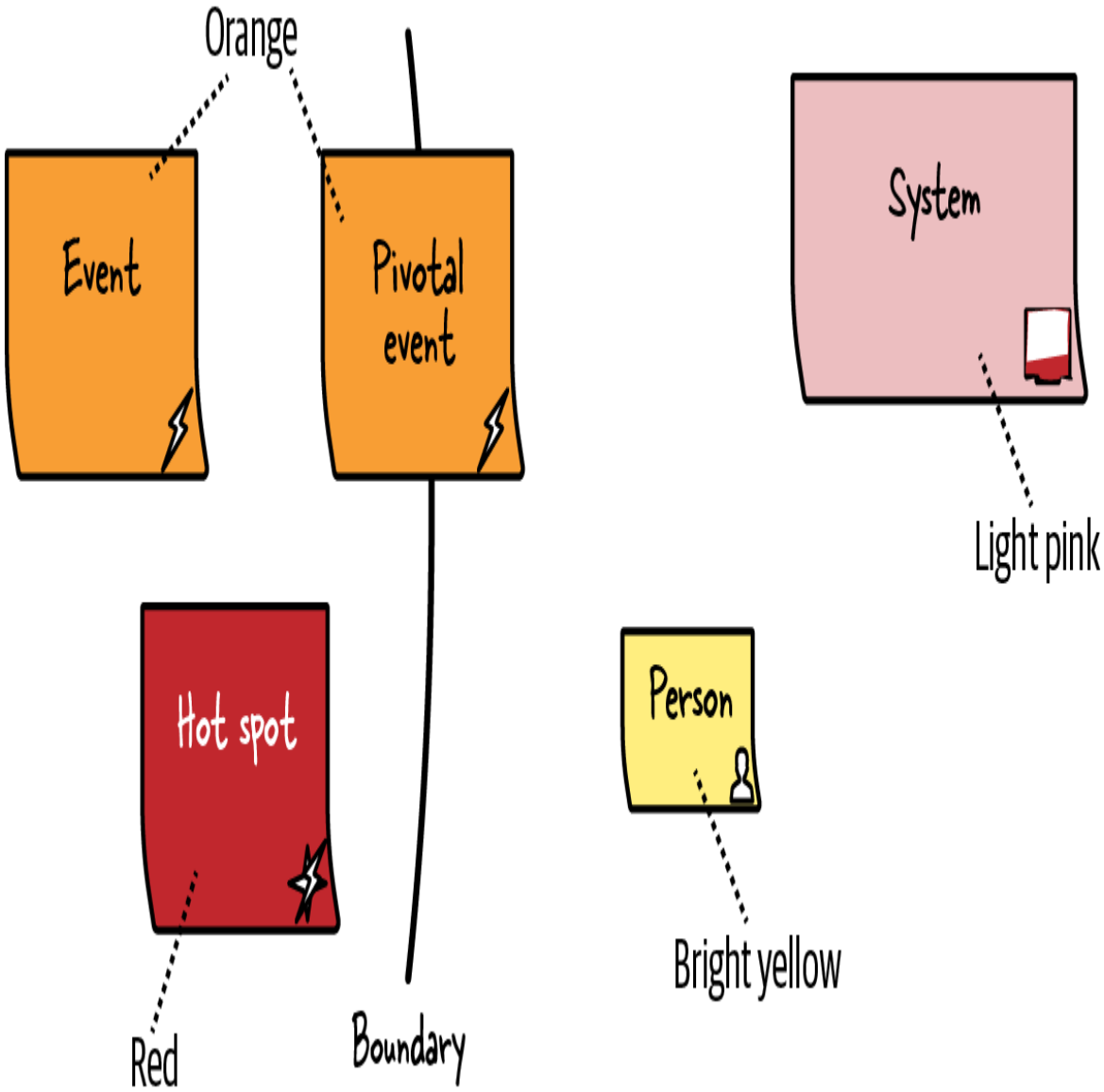


Figure 4-15. Different colors/symbols of EventStorming sticky notes (see "When Color Is Not Appropriate, Use Symbols").

In Domain Storytelling, we learned the difference between domain stories with software systems as actors and those without. When the scope factors for domain stories were added, it became clear that our goal in Domain-Driven Transformation is to model *pure* domain stories, that is, without software systems as actors. In EventStorming, pink sticky notes can be used to model software systems that users use in their work; nevertheless, in EventStorming too, it usually makes sense for Domain-Driven Transformation to model the process without technology.

WHEN COLOR IS NOT APPROPRIATE, USE SYMBOLS

EventStorming relies heavily on color, which is not always appropriate, e.g., in a vintage movie, when colorblind people attend a workshop, or when a book (like the paper version of this book) is printed in black and white. That is why here we add symbols (inspired by [Vernon and Jaskuła 2022, Fig. 3.6]) to the EventStorming sticky notes. In a workshop, symbols like these are usually omitted unless using only color reduces the accessibility.

To help workshop participants, the different colors of the stickies with their meaning are provided on a flip chart next to the modeling area as a *visual legend*. Typically, we start with events (orange color/flash icon) and hot spots (red color/explosion icon). Then we might add people and possibly systems. So the legend grows with more colors during the session, depending on what is needed.

The meaning of the colors is important, but if it doesn't suit us in a given situation, we change it.

Deeper into the Details: Software Design EventStorming

So far, we have developed a coarse-grained picture of what is happening in the domain. That is why such a workshop is also called a *Big Picture EventStorming*. You get an overview of the events together. The Big Picture EventStorming often leads to a desire to go into more detail for individual sections and to design a domain model for the software with a *Software Design EventStorming*.

For a session on more detailed work, you often no longer need all the domain experts at the same time. Instead, conduct a separate session for each development team/bounded context that is discovered as a subdomain and that can possibly be extracted from a legacy big ball of mud or existing development team. Within a bounded context, the team that owns it drills down further based on the events already found. The next question to ask is:

what triggered the individual domain event? This is when we evolve the color code and add even more colors[6] (see Figure 4-16):

- For *aggregates*,[7] rectangular pale yellow sticky notes are used in landscape orientation.
- We formulate an action of a person as a *command* on a blue sticky note.

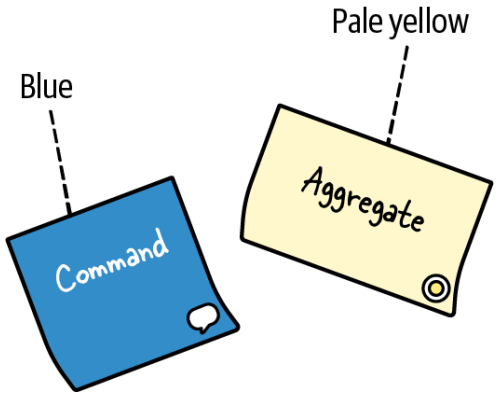


Figure 4-16. More colors for Software Design EventStorming.

If we analyze the domain events we have already found with this refinement, we sharpen our understanding of the domain and find gaps in our previous analysis. Often, at this stage we also discover additional events that we missed during the big-picture session. Let's take a look at this in our banking example.

CASE STUDY “BANK”

IN THE BANK: SOFTWARE DESIGN EVENTSTORMING

In the bank, the next day we start a new session for a Software Design EventStorming with the team responsible for the bounded context "monetary transactions." (We found this in Figure 4-14.) An excerpt from this, which models the substeps of a bank transfer, is shown in Figure 4-17.

Readers who are familiar with the banking domain will surely see immediately that there is more detail behind the actual implementation. For example, the excerpt from Figure 4-17 only works for transfers executed within one bank. If the account of the recipient of the money is with another bank, clearing between the banks has to be done, a process that is too complex for this example.

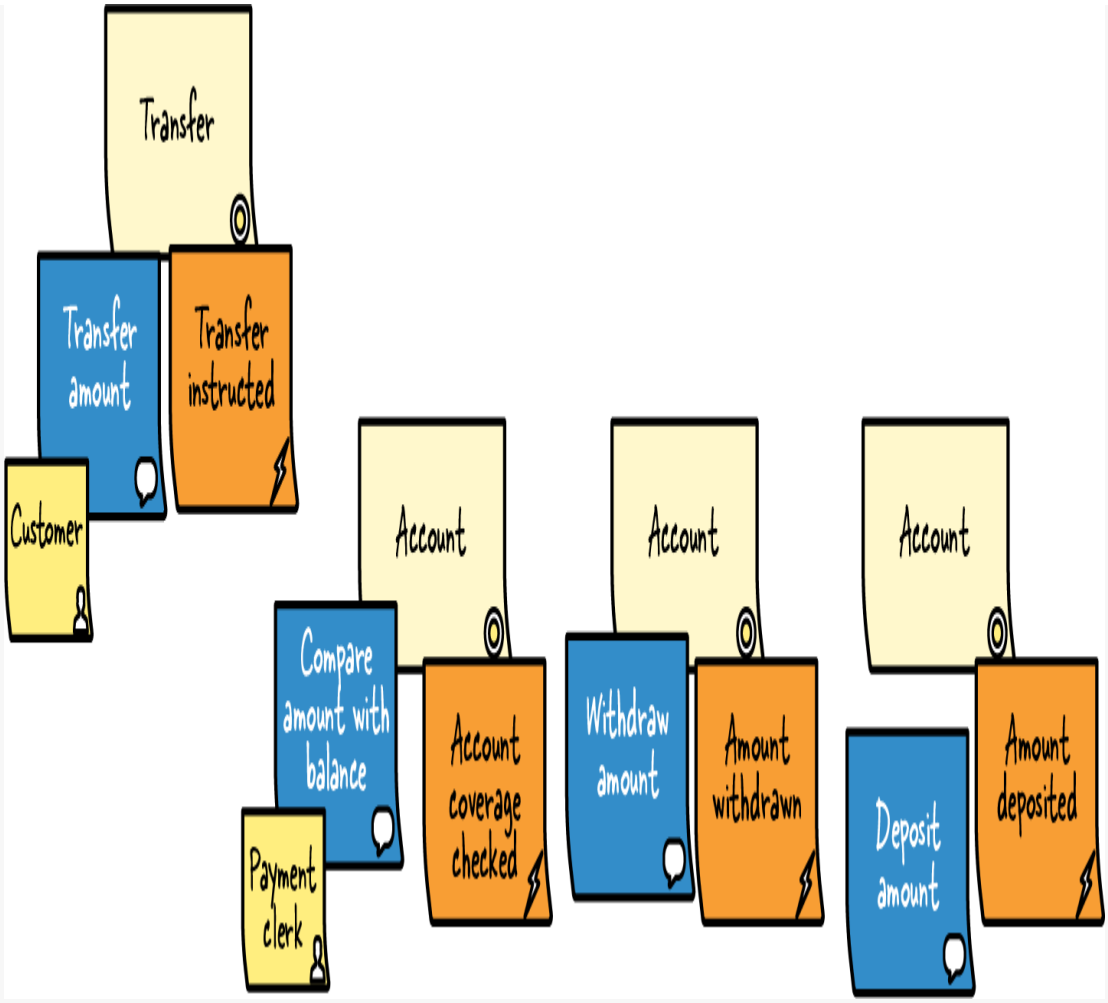


Figure 4-17. Software Design EventStorming in the bounded context "monetary transactions" (extract).

In Figure 4-17 you can see the commands that trigger the events, as well as the actors "customer" and "payment clerk" that trigger these commands. You can also find the aggregates "transfer" and "account."

Finding the Domain Model

Similar to Domain Storytelling, a first draft for the implementation of the domain model can be derived from the stream of events on the wall: a class diagram made of sticky notes, so to speak. In object-oriented design, aggregates become classes and commands become methods.[8]

CASE STUDY “BANK”

IN THE BANK: THE DOMAIN MODEL

In our bank example, we move the yellow and blue sticky notes around and come to a first idea of what a class `Account` with domain-driven methods could look like (see Figure 4-18).

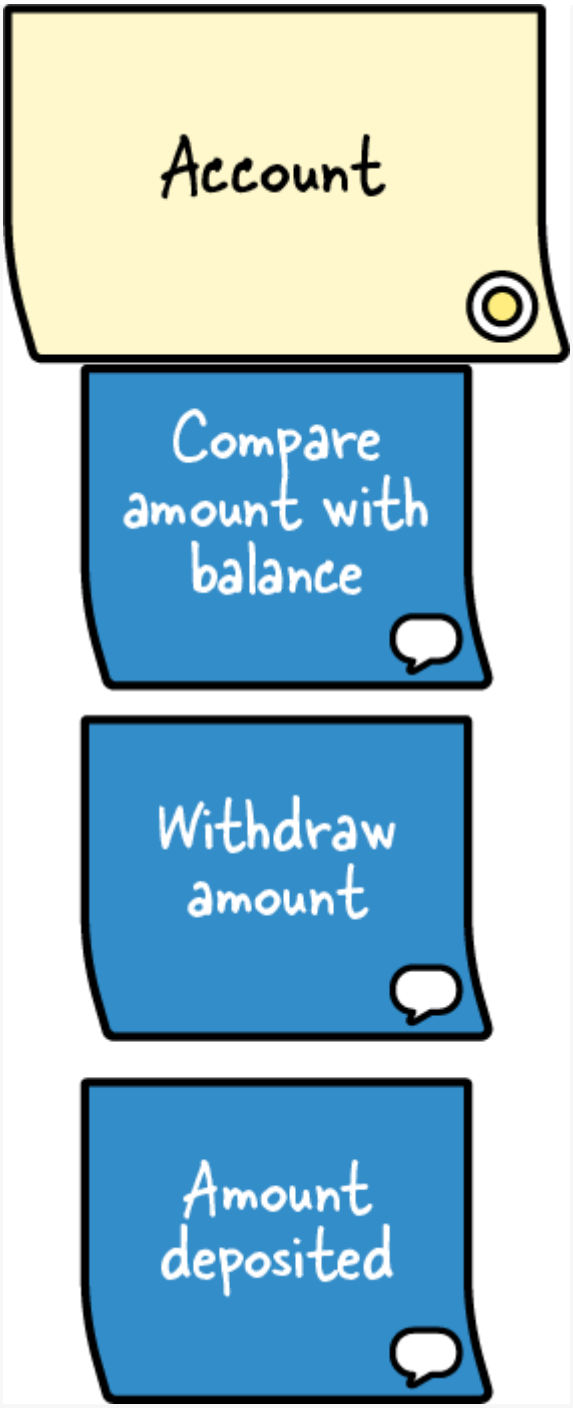


Figure 4-18. Domain model for the account.

In Chapter 8 we will revisit the bank account and discuss various ways to enrich domain classes and their behavior with business knowledge.

Other Formats and Outlook

We have seen that EventStorming comes in different formats. Brandolini describes this with the analogy of pizzas that all have the same base but different toppings. For this book:

Big Picture EventStorming

> Big Picture EventStorming is the main format for decomposition with strategic Domain-Driven Transformation, which is the topic of Part III.

Software Design EventStorming

> Software Design EventStorming goes into more detail and helps introduce more domain knowledge into the domain model with tactical Domain-Driven Transformation, as we will describe in Chapter 8.

Additional formats

> Additional formats, like Process Modeling EventStorming, are interesting in other situations, for which we refer to the literature tips that follow.

The definitive reference on EventStorming is [Brandolini 2021]. For practical tips on facilitating EventStorming workshops, see [Rayner 2024].

Scenario Casting

Guest author: Jörn Koch

When using Domain Storytelling or EventStorming, it is not always clear where to start and which parts or processes of the domain should be modeled. In such cases, Scenario Casting can help. It is a technique developed by our WPS colleague Jörn Koch, who contributed this section to the book.

How Does the Method Work?

Scenario Casting helps teams make sense of a complex problem space by systematically exploring it through example scenarios. These scenarios illuminate key aspects of the domain and provide a strategic focus for each development increment. By iteratively selecting and implementing the most relevant scenarios, teams progressively shape software that aligns with domain needs.

Why use example scenarios? Concrete example scenarios are a powerful tool for making complex problems understandable. We already use them informally to justify decisions. Here's an example: "New orders are sent to logistics after 15 minutes. This way, if a customer quickly cancels a duplicate order, we avoid reversing a fulfillment process." If such scenarios help us understand a solution *after* the fact, they can also guide us *toward* better solutions. Scenarios give context, clarify constraints, and expose the business logic we must support. They help us explore, evaluate, and communicate solution ideas.

The Core Idea

Scenario Casting[9] collects such example scenarios in a *scenario backlog*. From this pool, we select a strategically meaningful subset of example scenarios and combine them into a vivid orientation scenario. This composite scenario serves as the shared focus for an increment.

One small example scenario alone is rarely enough. But by assembling several related ones, we gain insight into underlying patterns, relationships between subdomains, and the broader domain structure. Orientation scenarios highlight what matters now and provide rich input for analysis and design.

Tools like EventStorming and Domain Storytelling are excellent for analyzing processes and refining understanding. Scenario Casting happens *before* that: it helps us decide *where* to focus. We use it to select scenarios worth elaborating, not to elaborate them in detail.

This makes Scenario Casting especially useful in large, unfamiliar, or fast-changing domains. It supports strategic navigation and allows teams to

explore the problem space incrementally without getting lost in complexity.

Workshop Flow: From Exploration to Implementation

Scenario Casting unfolds across collaborative workshops with all relevant stakeholders. Using sticky notes and physical or digital boards, it follows three core steps:

1. Brainstorming

Stakeholders share concerns, ideas, and questions. These are translated into domain language as concrete example scenarios and collected in a scenario backlog.

2. Focusing

The team selects the most impactful scenarios based on business goals or emerging insights.

3. Assembling

Selected examples are composed into one or a few coherent narratives that define the next increment's scope. These orientation scenarios serve as central reference points for all further activities.

With a clear orientation scenario in place, the team proceeds:

4. Analyze

Using methods like EventStorming or Domain Storytelling, the orientation scenario is broken down into events, subdomains, work objects, and other relevant structures.

5. Design and implement

The software is extended or adapted to adequately support the scenario.

6. Validate

> The orientation scenario serves as a primary acceptance test: can the described process be executed as intended?

This structured flow connects strategic focus with executable development, ensuring that each increment delivers meaningful progress grounded in real domain needs.

After each increment, we revisit the scenario backlog:

- We add new scenarios that were discovered along the way.
- We remove scenarios that are now supported or irrelevant.
- If needed, we conduct another round of brainstorming.

Typically, implementing one orientation scenario also resolves several related example scenarios from the backlog. As the backlog clears, the remaining scenarios tend to be less relevant—a natural burn-down effect.

By focusing on one or a few orientation scenarios per increment, multiple Agile teams can stay aligned without rigid coordination. Each team makes autonomous tactical decisions within a shared strategic frame.

Scenario Casting turns the messiness of complex domains into navigable terrain—step-by-step, orientation scenario by orientation scenario.

Introducing a Case Study: Green Urban Supplier

The company *Green Urban Supplier* takes orders and delivers boxes of organic produce to end consumers in large cities. To increase the attractiveness of the offering, products from private producers should also be included—from hobby chicken farmers to passionate urban gardeners. Instead of an official organic label, the company is discussing using its own organic label for these producers. The products of very small producers are often of exceptional quality and therefore particularly popular with customers.

Over the years, the company has built a complex software landscape that no longer meets the requirements. Management has therefore decided to comprehensively modernize the existing software landscape.

In the various departments and branches of the company, there are numerous domain experts, few of whom have IT project experience. The IT teams are currently fully occupied with maintaining the existing software landscape. However, some developers are supposed to accompany the project from the beginning and support the setup of the project teams.

The entire company, and especially the project manager Paulina, face a huge task, and many questions are still completely unanswered. Where do we even start? How do we get everyone to pull in the same direction? How do we avoid misdevelopments? How can we achieve relevant results quickly?

Paulina invites all project participants to a Scenario Casting—and we accompany her.

Step 1: Brainstorming

In the kickoff workshop, Paulina explains the procedure. “I know you all have very different questions, ideas, and concerns about this project. We want to collect these points today. To do this, consider the following guiding question: in what domain situation does my point come into play? Think of a specific example situation and note it on a yellow sticky note. Each example scenario serves as a marker for us so that we don’t forget your point during the project!”

Paulina demonstrates, “When a new order comes in, I currently have to enter the data in four different systems. In the new software, entering once should be enough! That’s important to me! Maybe the customer can even enter the order digitally themselves. So, I note the following specific example situation: ‘The Smith family orders a small vegetable box for the weekend.’ That should be enough as a marker for us to consider improving order processing in the project—even without me already writing down my initial solution ideas.” (See Figure 4-19.)

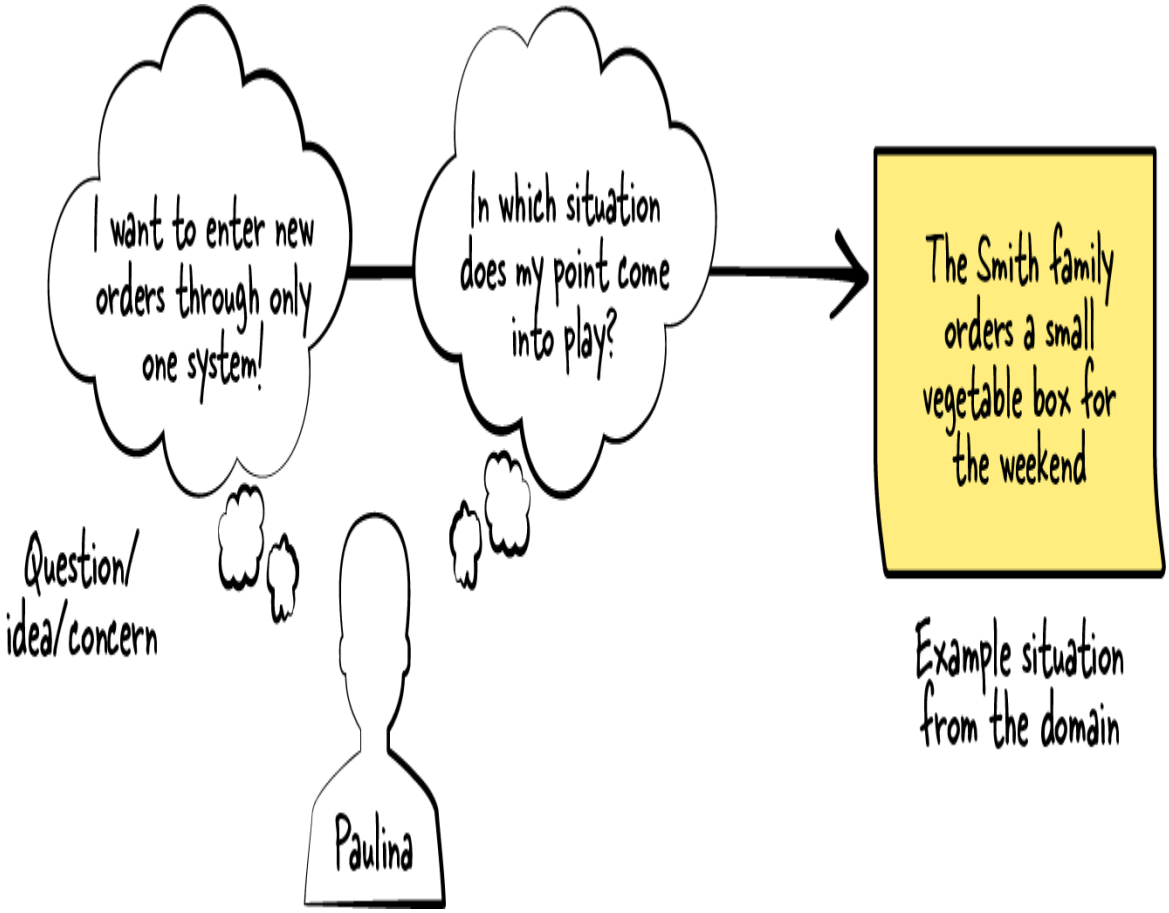


Figure 4-19. The path to the first sticky note with a vivid, domain-oriented example scenario.

“That will quickly become a lot of example scenarios. Why don’t we just write down the point ‘Order Processing’ on the sticky note?” asks a domain expert.

“Because then we lack the domain context. Order processing encompasses much more than just entering a single small order, which is sufficient for my point—I don’t want more, but also not less than my example scenario considers.”

Another domain expert adds, “It often happens that customers place old orders again. That would be the next sticky note: ‘The Smith family orders the vegetable box from last weekend again for the next one.’”

“But you can still modify this new order!” the first domain expert argues.

“True, so let’s add ‘… and adds a liter of milk,’ so that we also have refrigerated products included. For those, we need to record an exact

delivery time so they can be personally received. So, I continue writing: '… and specifies Saturday, 9–10 a.m. as the delivery time.'"

"We shouldn't do it like that anymore," notes the first domain expert, "because our drivers can hardly manage that. Let's just say 'morning.'"

"But what if the family isn't home after 10?"

Nobody comes up with a solution right away, but the actual problem is evident in the example scenario. In the end, the sticky note reads: "The Smith family orders the vegetable box from last weekend again for next Saturday and adds a liter of milk, but can only accept the box at home until 10 a.m."

"I can think of a few nice orders right away, such as self-pickup," says a domain expert excitedly.

That is a sign for Paulina that the group has understood the principle. Now everyone is allowed to write their own sticky notes.

"Don't spare the details. There's no 'too concrete'!" Paulina calls out. "Every detail can be a good clue to better understand the domain! Please write your name or initials on the top of the sticky notes so we can ask you questions if necessary when working with the notes."

In just a few minutes, the area around Paulina's two example scenario sticky notes is already filled with many more example scenarios.

Stakeholder Sinan writes: "The new software must be customer-oriented!"

This is too general for Paulina, and she asks for a specific example scenario where customer orientation comes into play.

"Oh, there are many," says Sinan. "For example, ordering via mobile phone should also be possible."

This, in turn, is too solution-focused for Paulina. "In what situation is ordering via mobile phone important?" she asks.

"Well, for example, when I realize while looking into the fridge that it urgently needs to be refilled. I'm not going to run to the PC for that!"

“Perfect, that’s a very vivid example scenario!” says Paulina, suggesting to note exactly that on the sticky note. Sinan then comes up with many more specific example scenarios (see Figure 4-20).

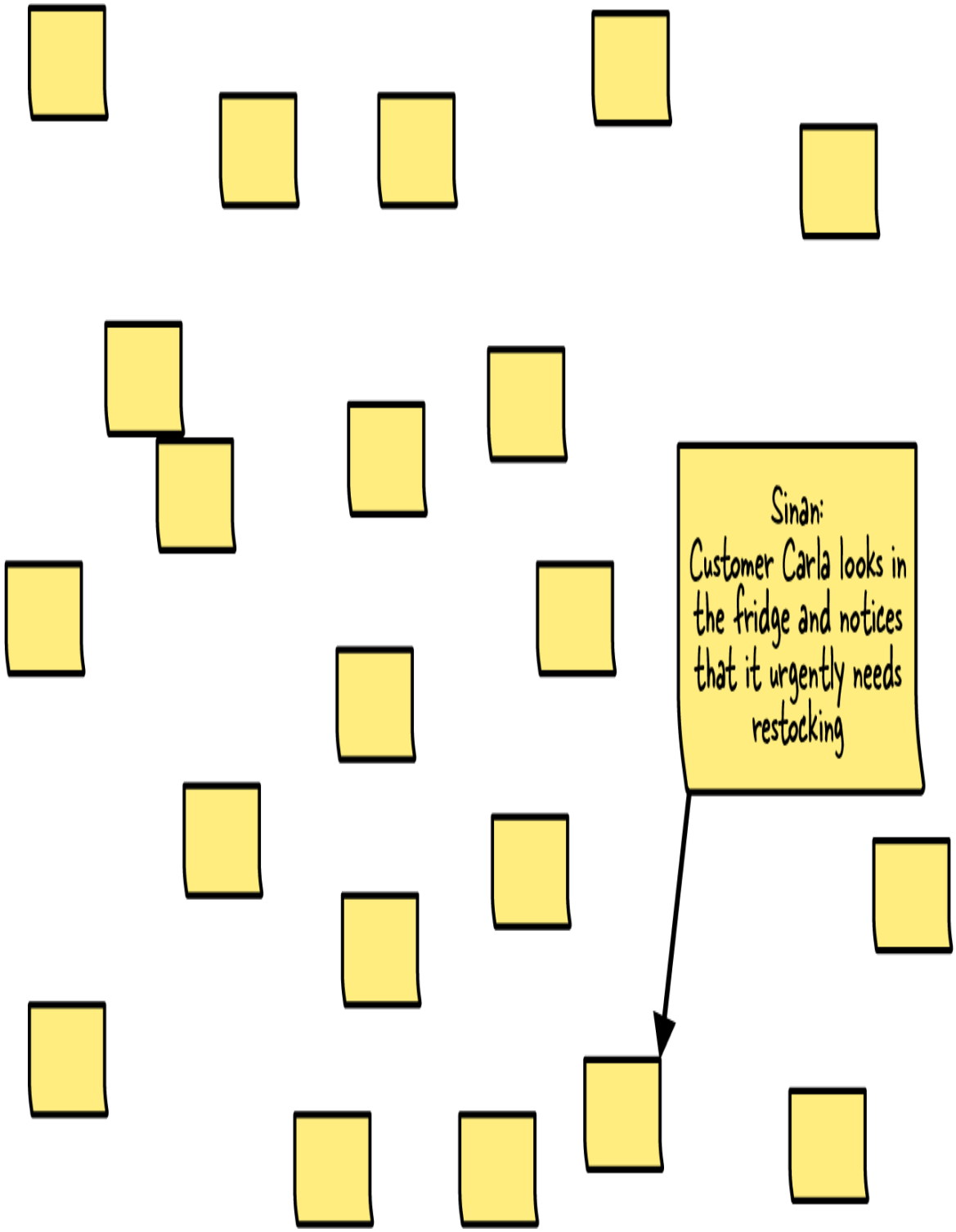


Figure 4-20. Freshly brainstormed example scenarios.

After half an hour, only a few new sticky notes are being written. This is a sign for Paulina to move on to the next phase. “Read through the sticky notes of others. If one of your sticky notes fits thematically, take it and stick it next to the others. This way, topic clusters emerge. We need them to gain a better overview and to find our stickies later—no need to overthink it!”

Indeed, the first sticky notes are being moved, and the first topic clusters are forming naturally. Paulina helps to separate the clusters spatially and sticks green notes on the clusters.

“Write the topic of the cluster on the green notes. This makes clustering easier,” she says.

A developer notes that the clusters overlap thematically, and some sticky notes fit into several clusters. Paulina is aware of the problem but knows that the clustering will improve with each round of Scenario Casting.

Therefore, she says, “Just hang the sticky notes in the clusters where you think they fit best. It doesn’t have to be perfect right away.” (See Figure 4-21.)

After a short time, all the yellow sticky notes are assigned to a cluster. Some allocations were discussed more than others. A developer asks Paulina whether the clusters are subdomains.

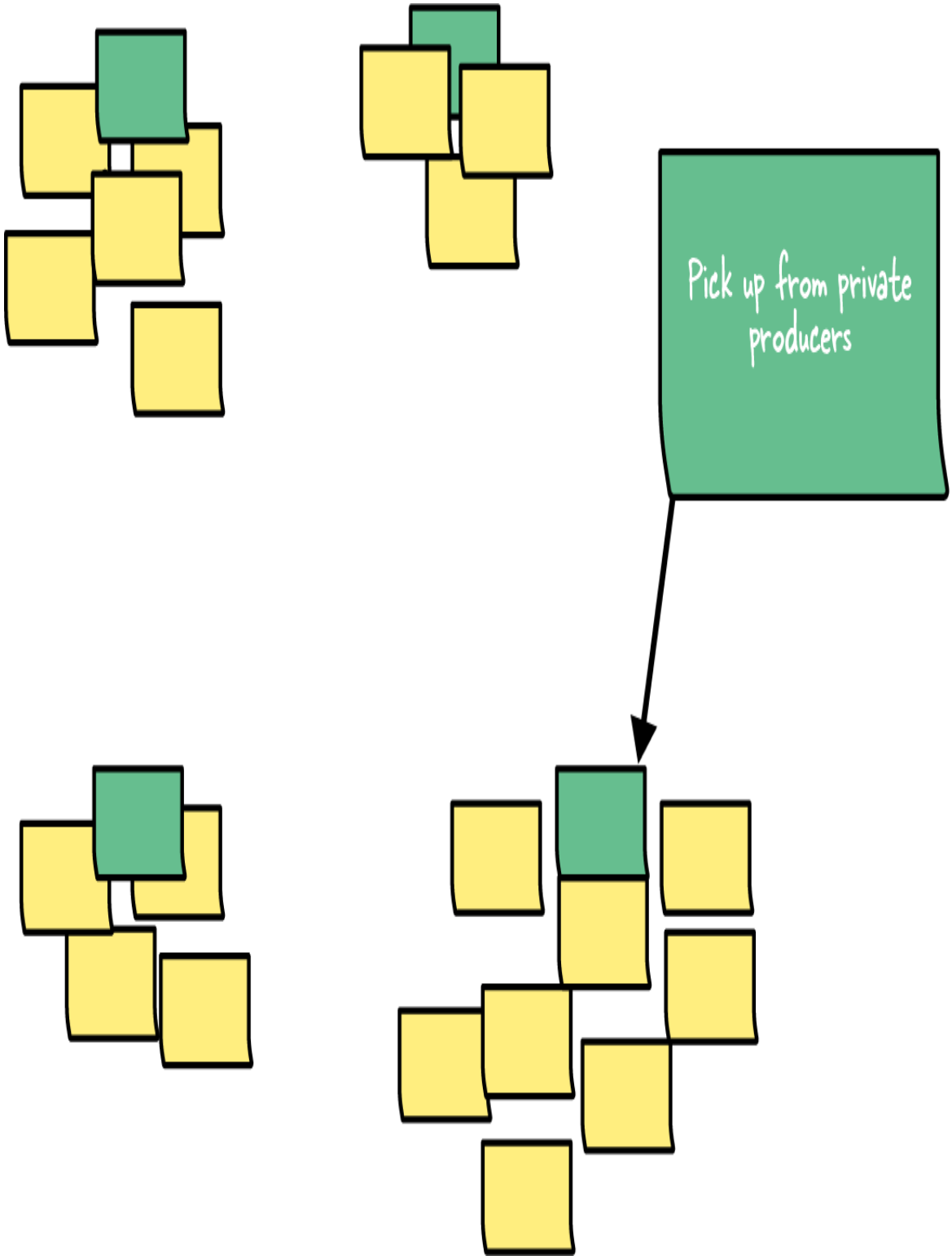


Figure 4-21. Content-clustered scenario backlog.

She replies, “That would be nice—but no, these clusters are only meant to bring some order to the example scenarios. We’ll identify subdomains

later."

Paulina now goes through all the clusters with everyone, without looking closely at the individual yellow sticky notes—there are simply too many.

"Are the clusters as expected? Is anything important missing?" she asks.

A few more sticky notes are moved, and another domain expert comes up with two more example scenarios to add—but essentially, the group is satisfied with the result. Almost 100 example scenarios have been gathered and grouped into 12 clusters. Some clusters are very large, while others only contain one or two example scenarios.

"You can already guess where the problem lies," says Paulina, pointing to the largest cluster.

She explains, "We call all of this the scenario backlog. We'll take care of all these topics here!"

With that, Paulina concludes the kickoff meeting. It lasted about two hours. Everyone contributed and was able to place their important points. And everyone now has a very concrete impression of what will actually drive the project from a domain perspective.

Step 2: Focusing

The second workshop takes place the following week. Paulina opens the second workshop. Above and below the area with the example scenarios, she has delineated two new areas: the area above is called *Focus* and the area below *Out of Scope*. Out of Scope are all example scenarios that are invalid, off topic, irrelevant, uncommissioned, or not actionable in the foreseeable future.

Paulina explains, "Today, we will roughly prioritize our scenario backlog and focus on a few example scenarios. We will do this by hanging sticky notes that we believe should be addressed immediately in the upper area, and sticky notes that are considered outside the scope of our project in the lower area. Just leave all the others hanging."

Paulina emphasizes that participants can decide on their own. “There is only one rule. If you think a sticky note from someone else is misplaced, speak up, and we’ll discuss it.”

Knowing that most sticky notes tend to move upwards, Paulina has drawn the upper area significantly smaller and adds, “Please only hang the sticky notes in the upper area if you believe they are absolutely essential for us to address now!”

A domain expert hangs some special cases up top and immediately encounters opposition.

“Let’s first look at the standard cases!” Paulina suggests focusing initially on the most representative example scenarios, like a “sightseeing tour through the domain,” to get a rough overview.

The participants find this to be a good idea and start selecting suitable example scenarios.

It becomes apparent that there are many special cases for products that need cooling. The domain experts agree to leave these out.

“That’s a separate, larger topic.” Paulina notes for prioritization on a sticky note: “Assumption: No products needing cooling.” Assumptions establish general guidelines for the scenarios considered and should be regularly reviewed for validity.

A domain expert points to an example scenario and asks what products like cheese and sausage count as, which don’t need continuous cooling but stay fresh longer when cooled. The domain experts are unsure and note this on a red sticky note (analogous to the hot spot sticky notes from EventStorming). They stick the red note with their question directly to the example scenario: “Always cool cheese/sausage?” (See Figure 4-22.)

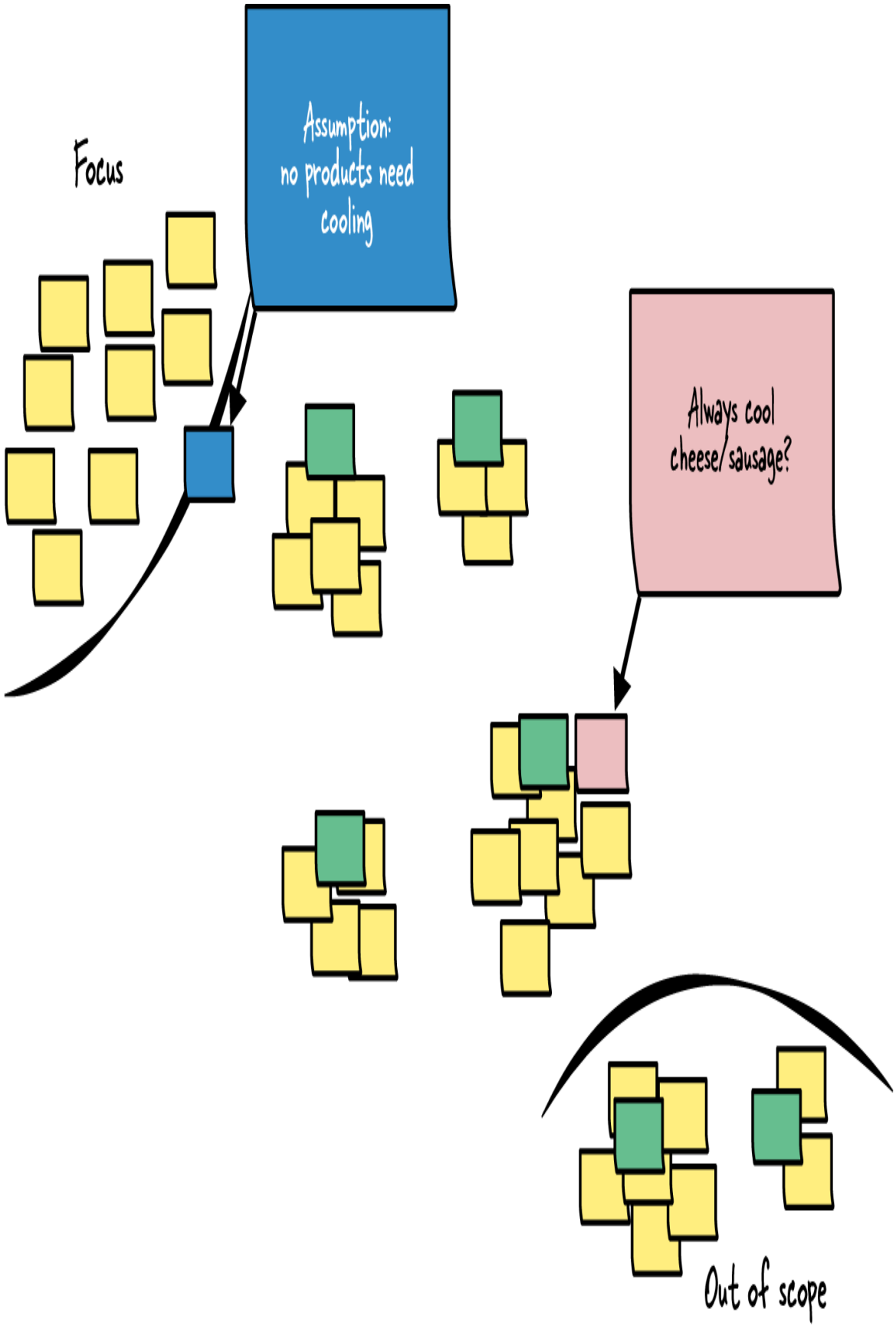

Focus
Assumption:
no products need
cooling
Always cool
cheese/sausage?
Out of scope

Figure 4-22. Prioritized scenario backlog.

At the end, about a dozen sticky notes are hanging in the focus area. The "organic label for private producers" cluster has been completely moved to the Out of Scope area. The topic is too large and does not need to be pursued within the project. The "Producer contracts" cluster has also been removed because the contracts will continue to be managed with the existing software—which most participants were not aware of. "Good that we talked about it!" says Paulina. Two invalid example scenarios, which cannot occur, have also been removed and have caused "aha" moments for two developers.

Paulina is pleased with the result and goes through the individual sticky notes in the focus area with the participants. One sticky note is pushed back into the scenario backlog as it no longer seems so important.

"These are quite a few example scenarios," notes one of the developers. "Do they somehow connect?"

"Yes, they do," replies Paulina, "and that's why we are now combining them into longer stories." (See Figure 4-23.)

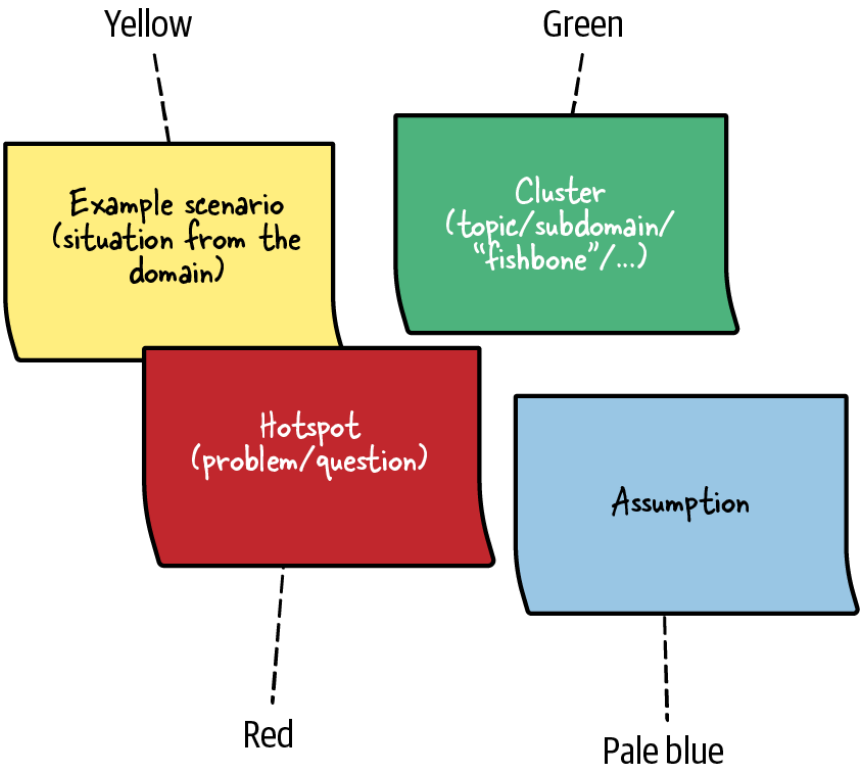


Figure 4-23. Visual legend of sticky notes in Scenario Casting.

Step 3: Assembling

Paulina takes the sticky notes from the focus area and distributes them above a timeline from left to right. The timeline serves as a guide for the chronological order in which the example scenarios can follow each other.

"It's important that the stories we assemble from these example scenarios are plausible and relevant!" emphasizes Paulina. "Because, of course, the example scenarios can be arranged into many different stories." (See Figure 4-24.)

"What would be a good starting point?" asks Paulina.

A domain expert says, "Here, this sticky note: 'The private producer Jones offers 20–25 fresh eggs for the weekend.'" The domain expert hangs it at the front of the timeline.

"'Family Smith orders a small vegetable box for the weekend' would also be a good starting point," notes another domain expert and hangs the note further back on the timeline.

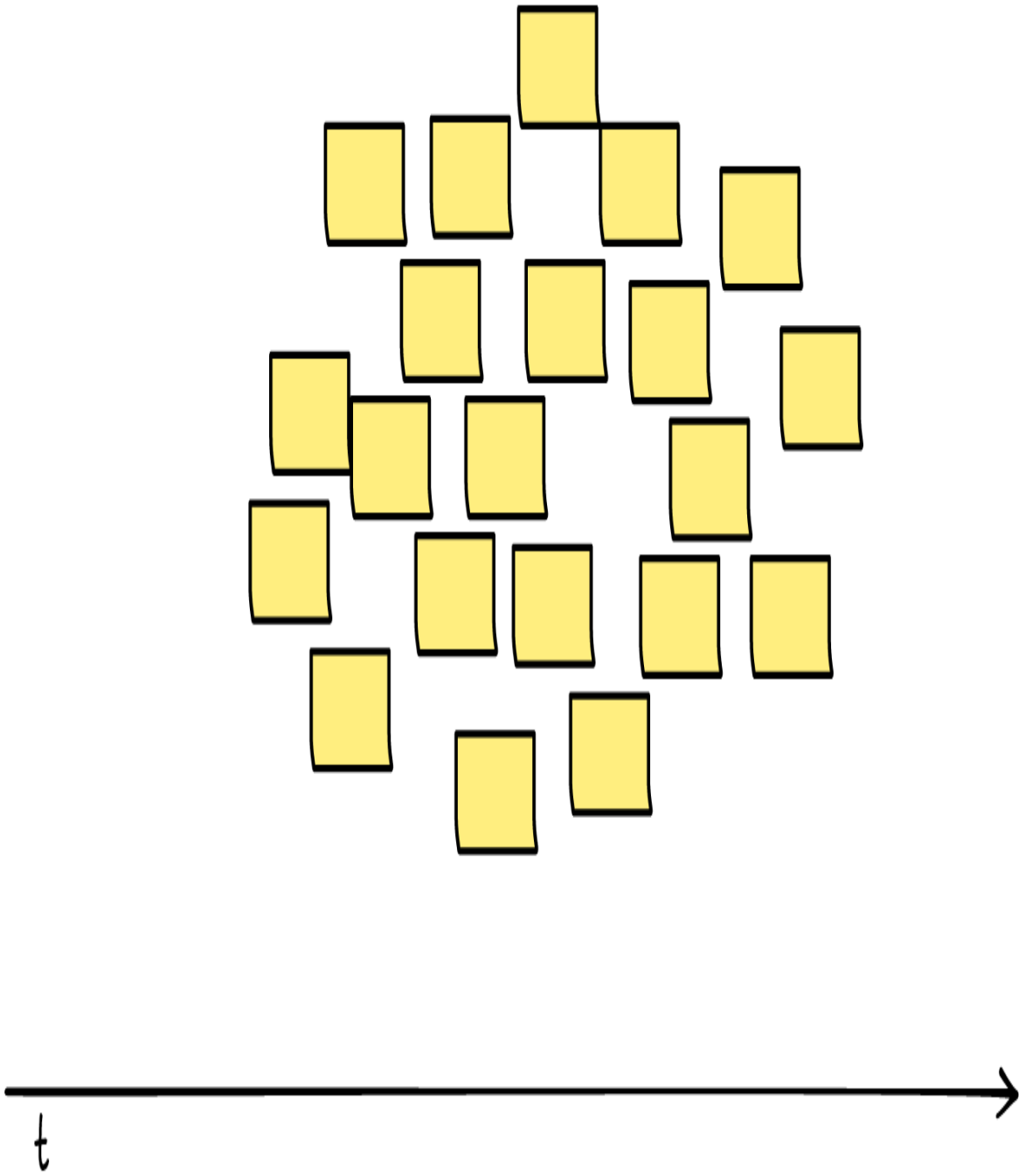


Figure 4-24. Unordered example scenarios placed above the timeline.

Additional notes describing how a box is assembled, driven to the customer, and delivered are loosely added. Likewise, additional notes describing how a price is calculated for the producer, which he accepts, and so on. Eventually, all the notes are grouped into three loose clusters around the timeline.

Paulina asks what each of these clusters is about.

“The first cluster is ‘Private producer offers products,’ the second is ‘Family orders box,’ and the third is ‘Closing of the day,’” suggests a domain expert. (See Figure 4-25.)

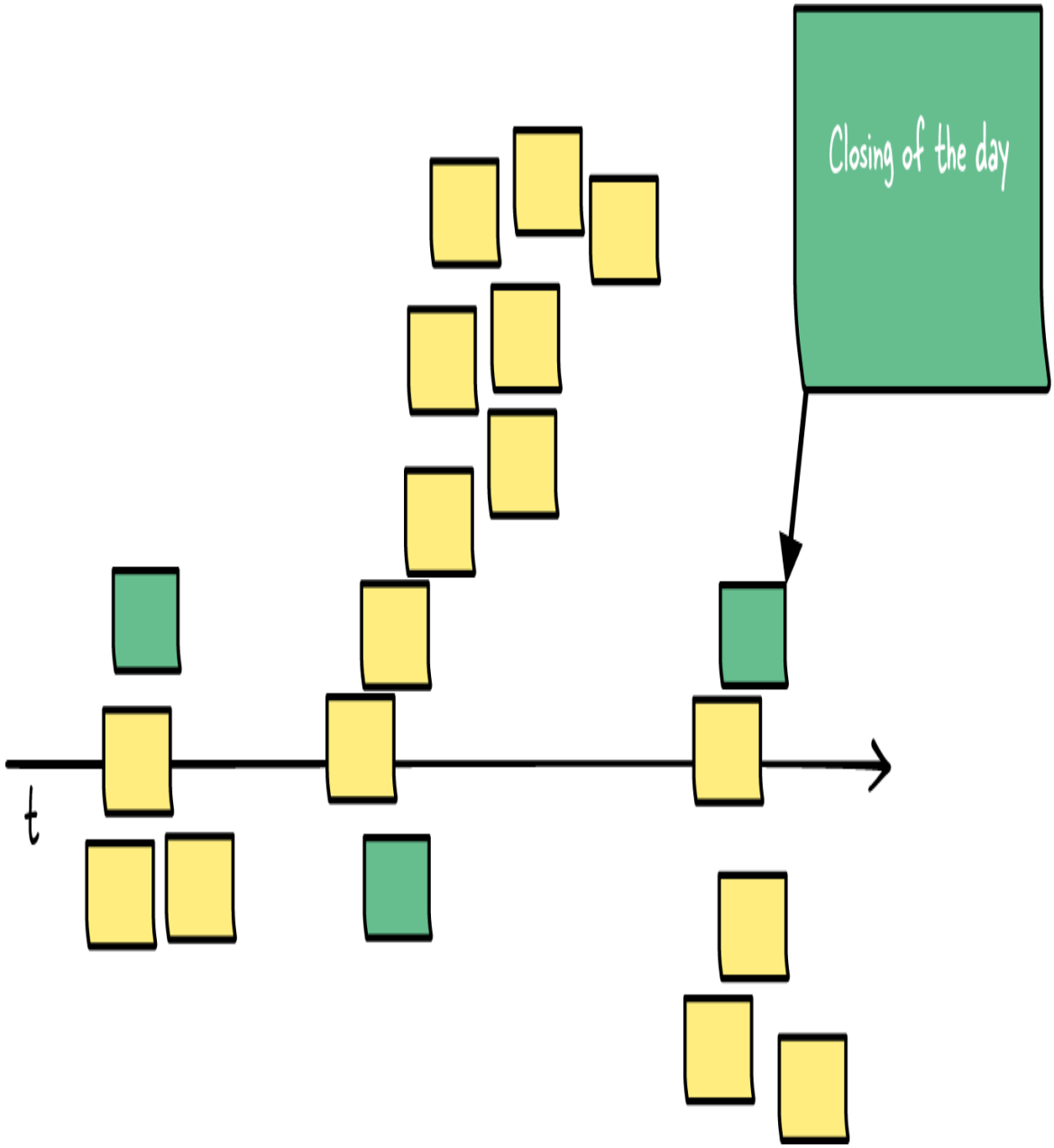


Figure 4-25. Example scenarios assembled into loose clusters over timeline.

“Let’s turn these clusters into coherent stories!” suggests Paulina.

Two domain experts begin to arrange the example scenarios from the second cluster in succession. One of them muses aloud, “Family Smith could order a vegetarian box instead, then the eggs offered by producer Jones can also be included.”

Paulina likes this idea because it makes the connection between the stories clearer.

“The example scenario of family Garcia ordering cheese and eggs can then be removed, right?”

“Yes, we’ve covered the animal products with the vegetarian box.”

“But dairy products are still interesting because they are always ordered separately and not included in the vegetarian box.”

In the end, the example scenario reads: “Family Smith orders a small vegetarian box and 200 grams of wild herb cheese for the weekend.”

The remaining example scenarios were also aligned in the first two clusters and put into a final order. In this way, two small stories emerged from the loose clusters. The third cluster, “closing of the day,” remains unsorted as the participants do not find the topic as exciting compared to the other two stories. Paulina later pushes it back as a whole into the scenario backlog (see Figure 4-26).

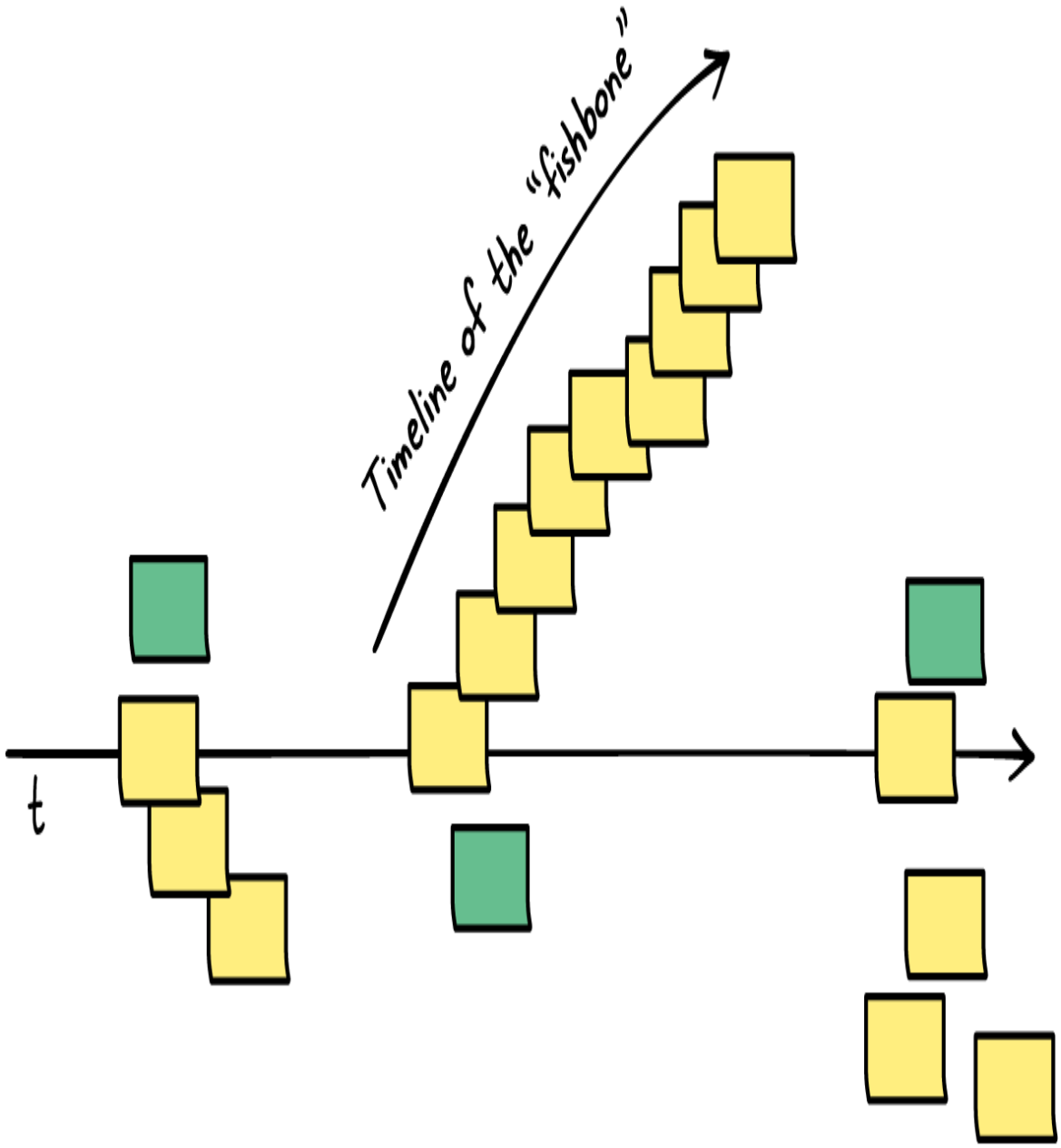


Figure 4-26. Two consolidated stories ("fishbones") and one loose scenario cluster.

Paulina explains, "Great! We've found two stories that cover the most important standard cases. We can now orient ourselves around them further in the project. That's why we call them orientation scenarios." Paulina smiles. "Most people, however, say 'fishbones' because they branch off from the timeline like fishbones—the term is here to stay!"

"Shouldn't we divide the second fishbone into ordering and delivery?" asks a domain expert.

The others agree because new actors come into play during delivery.

“And we can then combine offer and order,” suggests the same domain expert, but the others disagree.

So, there remain three orientation scenarios, and one only loosely assembled “cloud.”

Paulina provides an outlook on the next steps. “After outlining three central orientation scenarios, we want to analyze them in more detail. We’ll do this in separate sessions so that those not affected by the respective orientation scenario can opt out—so no one needs to fear missing out.”

What’s Next in the Scenario-Based Approach?

In the next sprints, the orientation scenarios will be put into action. Only then will the next Scenario Casting be due to set a new focus for the following sprints. Paulina is now particularly concerned with identifying the subdomains touched by the orientation scenarios. She wants to quickly assemble teams to handle each subdomain and develop solutions autonomously.

The description of the orientation scenarios so far only summarizes the main aspects but hardly goes into detail. Paulina knows that this is insufficient for separating subdomains from each other or diving into an Agile solution development. Therefore, she wants to dig deeper with the domain experts and tell the orientation scenarios in detail.

To do this, Paulina primarily uses two methods: Domain Storytelling and EventStorming. Both methods are equally suitable for better understanding the domain outlined in the orientation scenarios, identifying subdomains, and filling product backlogs.

To tell the first orientation scenario in detail, Paulina has organized another workshop. With the domain experts, she now examines the details, following the orientation scenario.

During the storytelling, the domain experts repeatedly encounter situations where the orientation scenario could develop differently than outlined.

“How do we deal with such alternative storylines?” asks one of the domain experts.

Paulina suggests documenting the example situation that leads to the alternative storyline on a yellow sticky note, similar to the beginning of the Scenario Casting during brainstorming.

“We’ll stick the new notes here in a dedicated area nearby and move them later into our scenario backlog!” explains Paulina. “We’ll analyze them once they are prioritized in one of the next rounds of Scenario Casting.”

This way, the domain experts accomplish two things: telling an engaging, coherent story and ensuring that additional example situations are not overlooked and addressed later.

The developers find the story in this vivid and detailed form very understandable. They already have initial ideas on how to support the orientation scenario with software. But Paulina first wants to know which subdomains are involved in the story to build the teams accordingly (see “Drawing Boundaries”).

She explains, “To break down our domain into subdomains, we can initially break down our orientation scenario into subprocesses. Each subprocess aims for a specific outcome. And only when this mission is accomplished do other subprocesses come into play.”

Paulina goes through the fully narrated orientation scenario with the group and asks, “At which points have we accomplished a mission and thus permanently completed a subprocess in the orientation scenario?”

The domain experts find it easy to name the points. In fact, domain experts usually know the central outcomes in their processes very well, as they are crucial from a domain perspective. These outcomes usually trigger subprocesses in other subdomains and are thus useful for delineating and identifying these subdomains from each other.

Paulina visually separates the subprocesses with lines. She writes on green sticky notes (the subprocesses) which mission each subprocess fulfills.

In the delivery scenario, the developers find the missions “Determine the next pickup point of the tour” and “Determine the next delivery point of the tour” very similar. They suspect they belong to the same subdomain and propose “Determine the next destination point of the tour” as a mission that encompasses both subprocesses.

“‘Tour coordination’ would be a good name for the subdomain,” suggests a domain expert.

Gradually, the subdomains emerge.

“Nice to see how everything we need for solution development suddenly falls into place after just a few workshops,” one of the developers remarks happily.

Another developer adds, “I’ve only been here for a short time, but I could understand the orientation scenarios well and why we are focusing on them.”

The project developers form two teams in follow-up workshops, assign the subdomains among themselves, and prepare for solution development.

Paulina defines the goal for the next sprints precisely. “Let’s make sure the orientation scenarios become reality!”

There’s still a lot to do for that—but everyone agrees: This goal no longer feels like an overwhelmingly large project but rather like a task that can be tackled in a few sprints.

To fill the product backlogs, Paulina suggests further splitting the fully narrated orientation scenarios into meaningful user stories. “The easiest way is to split the fully narrated orientation scenario into user stories. The section each user story refers to is directly the first acceptance test for it.”

“That seems to call for a test-driven approach like TDD (Test-Driven Development) or BDD (Behavior-Driven Development) directly,” says a developer.

After each sprint, Paulina marks the implemented parts in the fully narrated orientation scenarios, enabling cross-team tracking of progress. The domain

experts also appreciate this overview as they have a good insight into what is happening in the implementation.

During this, Paulina notices that the implementation of the payment subprocess supposedly takes longer. She talks to the responsible team. Apparently, the third-party system that processes the payments cannot be easily integrated due to licensing reasons. The team has come up with a very simple workaround: initially handling payments manually by invoice. Although this does not correspond to the fully narrated orientation scenario, which involves paying by direct debit, it would allow the orientation scenario to be tested as a whole in a timely manner. Paulina brings in the domain experts, who approve the proposal (see Figure 4-27).

One sprint later, Paulina marks all orientation scenarios in green. Now comes the exciting moment of integration testing of the entire orientation scenario as a whole.

Since the two teams have developed precisely for this test and have already tested their parts using the orientation scenarios, there are no major problems. Only a few minor things need to be fixed. Paulina has experienced this differently before and is pleased that the teams had a concrete testable example process with the orientation scenario rather than an abstract solution idea as a specification (see [Adzic 2011]).

The project's testers now verify alternative test scenarios derived from the orientation scenarios.

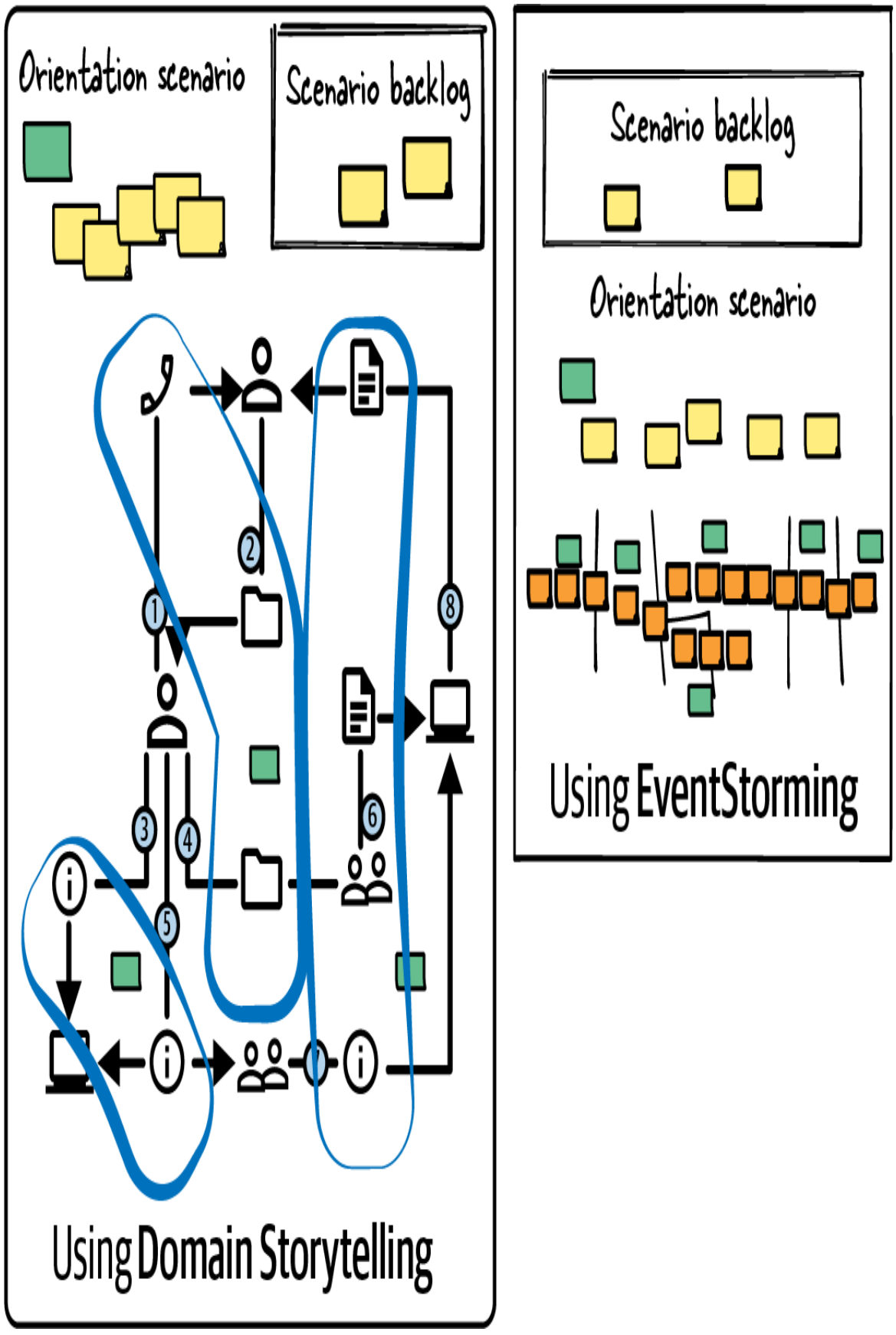

Orientation scenario
Scenario backlog
1
2
3
4
5
6
7
8
Using Domain Storytelling
Scenario backlog
Orientation scenario
Using EventStorming

Figure 4-27. Fully narrated and broken down into subprocesses orientation scenario plus scenario backlog for newly discovered scenarios.

The Second Round of Scenario Casting

Paulina is relieved that the second round of Scenario Casting can begin after just a few sprints. When she first used Scenario Casting, she didn't realize that there are many different strategies you can pursue in Scenario Casting. At that time, she had been driven largely by her curiosity in selecting example scenarios. Although it was very enlightening, the orientation scenarios were too rich to implement in a manageable time! Since then, Paulina has always kept in mind during Scenario Casting that the assembled orientation scenarios should be manageable within a short time. It's difficult to estimate, so the rule here is always: less is more!

The same applies to the implementation of the orientation scenarios. Only what is described in the current orientation scenario needs to be supported now—nothing more! Everything else will be added gradually with further orientation scenarios. The solution is progressively extended and generalized, making it increasingly powerful. Particularly exciting scenarios from a developer's perspective can be deliberately prioritized (provided they are sufficiently relevant from a domain perspective).

Paulina starts the second round of Scenario Casting by reviewing and updating the scenario backlog together with some domain experts:

1. Newly discovered scenarios during storytelling are added to the scenario backlog.
2. Completed example scenarios are removed from the scenario backlog.
3. The thematic clustering of the scenario backlog is adjusted as needed based on new insights.

The vast majority of example scenarios and clusters remain unchanged. Unlike solution ideas, which can quickly become outdated through

continuous development, example scenarios from the problem space often remain valid and relevant for a long time.

Now Paulina invites everyone to the next Scenario Casting, which this time will take place online. Paulina has transferred the scenario backlog to a virtual whiteboard.

The group skips brainstorming because the scenario backlog is already well filled. Should any scenarios still be missing, participants are welcome to add new example scenarios to the scenario backlog at any time.

In the first round of Scenario Casting, it became evident that integrating existing systems into the new architecture is challenging, not just the payment system. Paulina suggests using a different strategy in this Scenario Casting and focusing more closely on the identified problem areas through shorter scenarios. The group now examines the ordering process, as it is unclear how products from small private producers can be integrated into the legacy system. In the payment process, invoicing will be examined more thoroughly to put the implemented workaround on a solid foundation.

At the end of this second round of Scenario Casting, there are seven small, largely parallel orientation scenarios describing the domain of the previously mentioned problem areas. Since the seven orientation scenarios revolve around the same themes and are manageable in total, they will all be supported together with the next increment.

The new orientation scenarios will also be modeled using EventStorming and Domain Storytelling. And so, the group continues its collective journey through the domain.

Outlook

We, Carola and Henning, sincerely thank Jörn for this guest contribution. We consider Scenario Casting to be a promising method worth adding to the Domain-Driven Transformation (DDT) toolkit.

Which Method to Use When?

We have introduced two methods that are important for our daily work on legacy systems, Domain Storytelling and EventStorming, and one that is a promising companion to them, Scenario Casting. In the rest of the book, we will use Domain Storytelling more often because it is more suitable for presentation in a book and for documentation in general. In our workshops, however, we use the methods equally. Which method to use is a topic of popular discussion in the community. Some indicators are:

- EventStorming is particularly suitable when we analyze a process of a (seemingly) not very structured domain or when the timeline should be visible at once.
- Domain Storytelling shows its strengths when we analyze processes that focus on cooperation between different actors and roles.
- Scenario Casting is helpful as a pre-exercise when the domain is huge and it is unclear where to start, or you want to find scenarios that should be explored with the other two.
- In Chapter 14 we will see how different types of domains behave.

HENNING'S FIELD STORY: USING TWO COMO METHODS ON THE SAME DAY

I was working with a large e-commerce company. One particular team was responsible for two subdomains. Since we didn't have a lot of time, a one-day workshop was planned in which one subdomain was supposed to be explored in the morning and the other in the afternoon.

In the morning session, we did an EventStorming with great success. Many dozens of sticky notes just flew to the wall. We achieved a lot in just a few hours: a deep common understanding of the domain and a good plan for the implementation of the subdomain.

With this great success behind us, we returned with high expectations from lunch. Enthusiastically, we started to explore the other subdomain. Of course, we used EventStorming again, since it had worked so well in the morning. But strangely, we didn't get a grip on the business process. After three-quarters of an hour, there were still only a handful of sticky notes on the wall.

When the frustration reached a climax, we switched methods. I drew what I had heard (and that couldn't be well expressed in domain events) into a domain story on a whiteboard. Soon, it "clicked" and we became productive again. At the end of the day we had drawn a handful of domain stories for different cases and in different scopes. They served as a good start for the domain model to implement.

Being in that moment where one method failed and everybody was disappointed wasn't a particularly enjoyable experience. But witnessing that another method did the trick was great.

The moral of the story is of course not that "one method is better than the other," but that it is a good idea for a professional to have more than one tool in their toolbox.

For me, it was very interesting to see that even with the same prerequisites (same team, same facilitator, same knowledge), a particular method will not work in every domain.

The effectiveness of the tool also depends on who picks it up. Thus, different CoMo practitioners have and develop different styles. As an interesting mash-up, we would like to mention Storystorming by Martin Schimak (see [Schimak 2019]). *Communication Patterns* help us understand what we technical people do when we talk with each other and with domain experts (see [Read 2023]). In addition, we would like to mention a book that just came out while we were writing this: *Collaborative Software Design: How to Facilitate Domain Modeling Decisions* [van Kelle et al. 2025].

Collaborative Modeling Remote

If you can't bring people together in real life, you can work in a video conference with digital tools such as Egon.io or Miro. However, a lot is lost here because, for example, you can't read the body language of the other participants. Nevertheless, both Domain Storytelling and EventStorming can also work in the online setting. We advise against hybrid workshops: either all participants are on site, or all are online in a video conference.

Collaboration Modeling and AI

The methods for Collaborative Modeling are all designed to enable developers to work together with users or domain experts to reflect on their work and model it. This is a human activity and it is about human learning. Still, AI can support these modeling workshops: it can obtain a transcript and, based on that, a summary of the discussion in the workshop. AI can also sometimes provide missing explanations of domain terms in the workshop, if necessary.

The situation is different when it comes to analyzing and processing the diagrams created during Collaborative Modeling. We are currently experimenting with presenting results of Collaborative Modeling workshops (such as domain stories as Egon.io files) to an AI and asking it

to find subdomains in the domain story. The initial results with smaller domain stories look very promising. The AI often arrives at a similar result as we do when breaking down the story.

Other Collaborative Modeling Methods

In the course of a Domain-Driven Transformation, it can also make sense to use other CoMo methods. For example, many teams plan implementation and realization within a bounded context with User Story Mapping (see [Patton 2014]). However, in order not to overload the book, we do not go deeper into methods that are not directly used to decompose legacy systems. Instead, we recommend that the interested reader take a look at [Baas-Schwegler and Rosa 2020], visit the DDD Crew repository on GitHub (*https://github.com/ddd-crew*), or attend the conference series *ComoCamp* (*https://comocamp.org*).

Summary

In this chapter, we have introduced you to three different requirements analysis methods that are easy for users to understand: Domain Storytelling, EventStorming, and Scenario Casting. All three methods belong to Collaborative Modeling, a group of workshop formats that have evolved around DDD in recent years. In Chapter 8 we will use the result of the EventStorming we've seen, and in Part III, we will pick up Domain Storytelling again.

---

1 These methods also help in the solution space with designing the system.

2 As DDD's founding father Eric Evans says again and again in his keynotes (e.g., Explore DDD 2018 in Denver; DDD Europe 2019 in Amsterdam): "DDD isn't done! Domain-Driven Design is not a dogma, it's a method that always needs to be extended, refined and improved." He called for integrating more good approaches and methods into DDD and in this way making software development better. This is exactly what has happened with the Collaborative Modeling methods.

[3] We thank Paul Rayner for permission to reproduce this graphic here. Source: Workshop at Explore DDD 2018. Photo by Martin Schimak.

[4] We're very grateful to Alberto, who has intensively reviewed and extended this section.

[5] Sometimes there is no wall available that provides enough space. In this case, elongated tables can be set up in a row on which the paper roll can be rolled out. Ideally, the tables are then positioned so that the modeling surface is easily accessible from both sides.

[6] Here, we're using only a restricted version of the Software Design EventStorming colors to demonstrate our example. In other situations, more colors are added, e.g., *read model*, *policy*, and *value* stickies.

[7] Many people in the EventStorming community aren't particularly happy with the term *aggregate*, as it does not make a lot of sense to business stakeholders. So what to call the yellow sticky? Some folks prefer the term *constraint* (see e.g., *https://github.com/ddd-crew/eventstorming-glossary-cheat-sheet*) or *business entity*. Your authors sometimes use *work object*, *domain concept*, or just *thing*.

[8] In functional programming, the aggregates are modeled as types and the commands as functions. With an event-sourced implementation (see [Fowler 2005]), the domain events also become architectural elements. With Command-Query-Responsibility Segregation (CQRS) this also applies to the commands (see [Fowler 2011] and [Fowler 2017]).

[9] "Scenario Casting" as in (1) "casting a fishing rod" to reel in new scenarios, (2) "casting show" to pick the best scenario candidates, and (3) "casting metal" to fuse many scenario splinters into a compact story.

Chapter 5. Concepts of Software Architecture

> **TIP**
>
> This chapter presents knowledge that is foundational for Domain-Driven Transformation. If you are already familiar with an architecture concept presented here, you can skip its section and come back to it later for reference.

For a transformation to be successful, it is important to visualize the architecture of the to-be transformed system, which is why we start this chapter with a brief discussion of *architecture documentation* with the *C4 Model*.

In every company there are large software systems that have been developed over many years and whose maintenance becomes more and more tedious and expensive year after year. One of the main causes of these difficulties is the interwoven structure of these old systems. This arises when two basic principles of *modular* architecture have not been sufficiently considered in the development of the system: *cohesion* and *coupling*, which we will cover in this chapter.

> **NOTE**
>
> This book focuses on the structural aspects of software architecture, i.e., on the quality attributes maintainability, extensibility, and portability, which are closely related to cohesion and coupling. We do not address other quality attributes, such as usability, performance, and security, in our considerations. That would go beyond the scope of this book.

In the context of Domain-Driven Design, we will also discuss some *architectural styles* that will have an impact on our further discussion:

- Monoliths
- Microservices
- The good old layered architecture
- Hexagonal Architecture (a.k.a. Ports & Adapters),[1] Onion Architecture, and Clean Architecture
- Service-oriented architectures
- Self-contained Systems

The chapter ends with the presentation of the Modularity Maturity Index (MMI), which can be used to evaluate whether a legacy system has high cohesion and loose coupling. Chapter 6 shows how to use the MMI to clarify which path through Domain-Driven Transformation is the right one: should the team start directly with strategic Domain-Driven Transformation? Or is tactical Domain-Driven Transformation the first thing to do?

> **TIP**
>
> If you are already familiar with all these concepts, we recommend that you skip to Chapter 6, where we discuss how the transformation can be approached.

Architecture Documentation with the C4 Model

There are various approaches to documenting architecture, such as UML, ArchiMate, and Arc42. We use the C4 Model in this book to familiarize you with our examples, so we explain the C4 notation here.

For the visualization of architecture, a lightweight variant has established itself in recent years in addition to UML: the *C4 Model* (see [Brown, forthcoming]). The idea is to approach the software architecture as a kind of drill-down from the coarse-grained to the fine-grained level. Four *levels* are

defined whose names begin with the letter C: context, containers, components, and code. These are the four Cs in C4.

People, *architectural elements*, and *relationships* can be represented on all levels of the C4 Model. Each finer-grained level describes an architectural element from the coarser-grained level in its parts.

All diagrams should be as self-explanatory as possible and have a legend to help with clarity. The architectural elements typically contain explanatory text.

Level 1: Context

Here the system of focus as a whole is represented in its environment with users and connected systems.

CASE STUDY “CINEMA”

C4 IN THE CINEMA: CONTEXT

At Clintwood Cinemas a context diagram had already been drawn, as shown in Figure 3-2. Now, the team elaborates on it to show the relationships to external systems (see Figure 5-1).

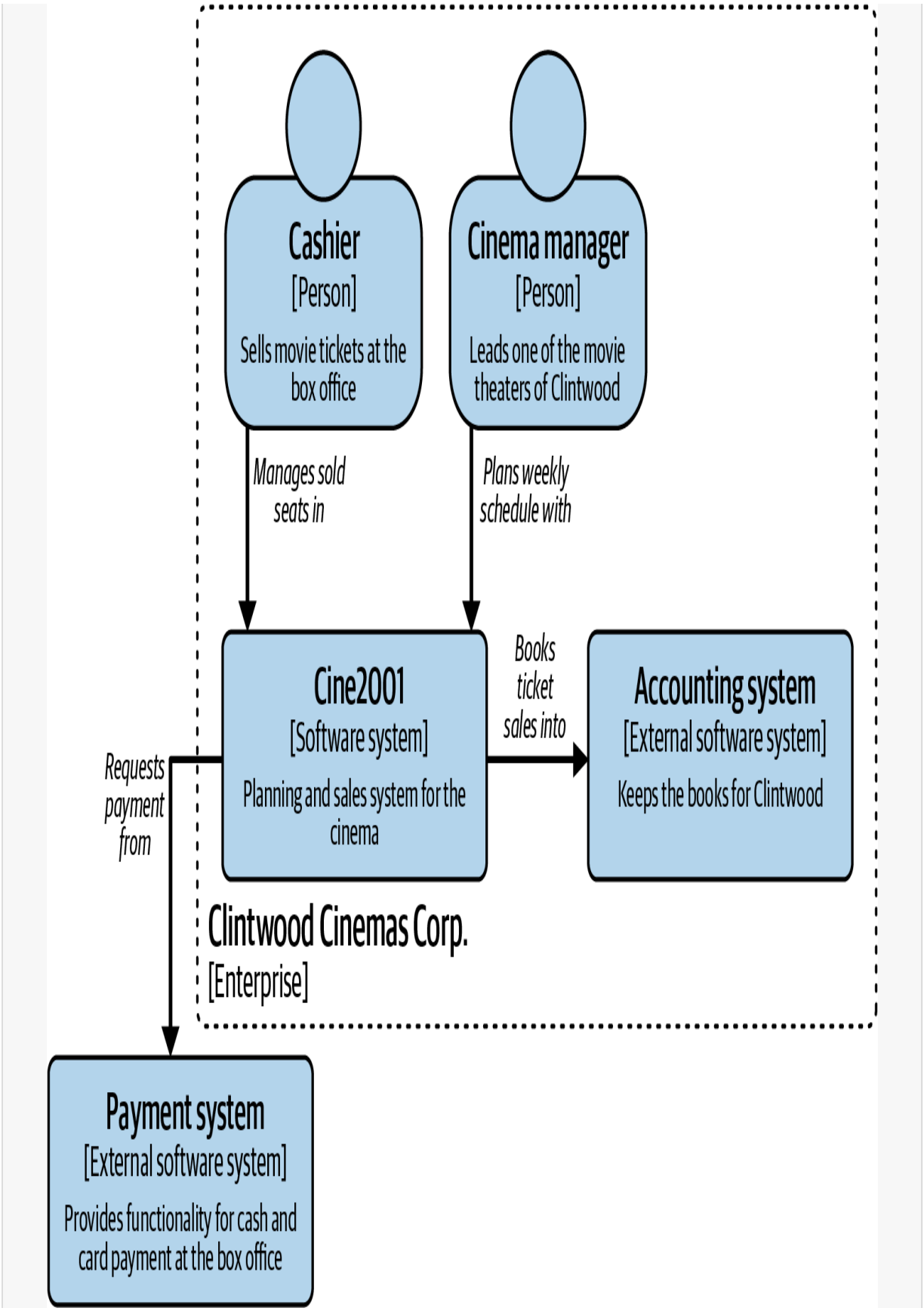

Cashier
[Person]
Sells movie tickets at the box office
Cinema manager
[Person]
Leads one of the movie theaters of Clintwood
Manages sold seats in
Plans weekly schedule with
Cine2001
[Software system]
Planning and sales system for the cinema
Books ticket sales into
Accounting system
[External software system]
Keeps the books for Clintwood
Requests payment from
Clintwood Cinemas Corp.
[Enterprise]
Payment system
[External software system]
Provides functionality for cash and card payment at the box office

Figure 5-1. Cine2001 system in context (elaborated).

Level 2: Containers

Zooming into the system level displays deployment units—called containers—that the system consists of. A container is a separate process.

> **WARNING**
>
> Containers in the C4 sense can, but are not required to, run in containers in the meaning of Docker.

CASE STUDY “CINEMA”

C4 IN THE CINEMA: CONTAINERS

The transformation team at Clintwood zooms into Cine2001 and visualizes the containers of the system in Figure 5-2.

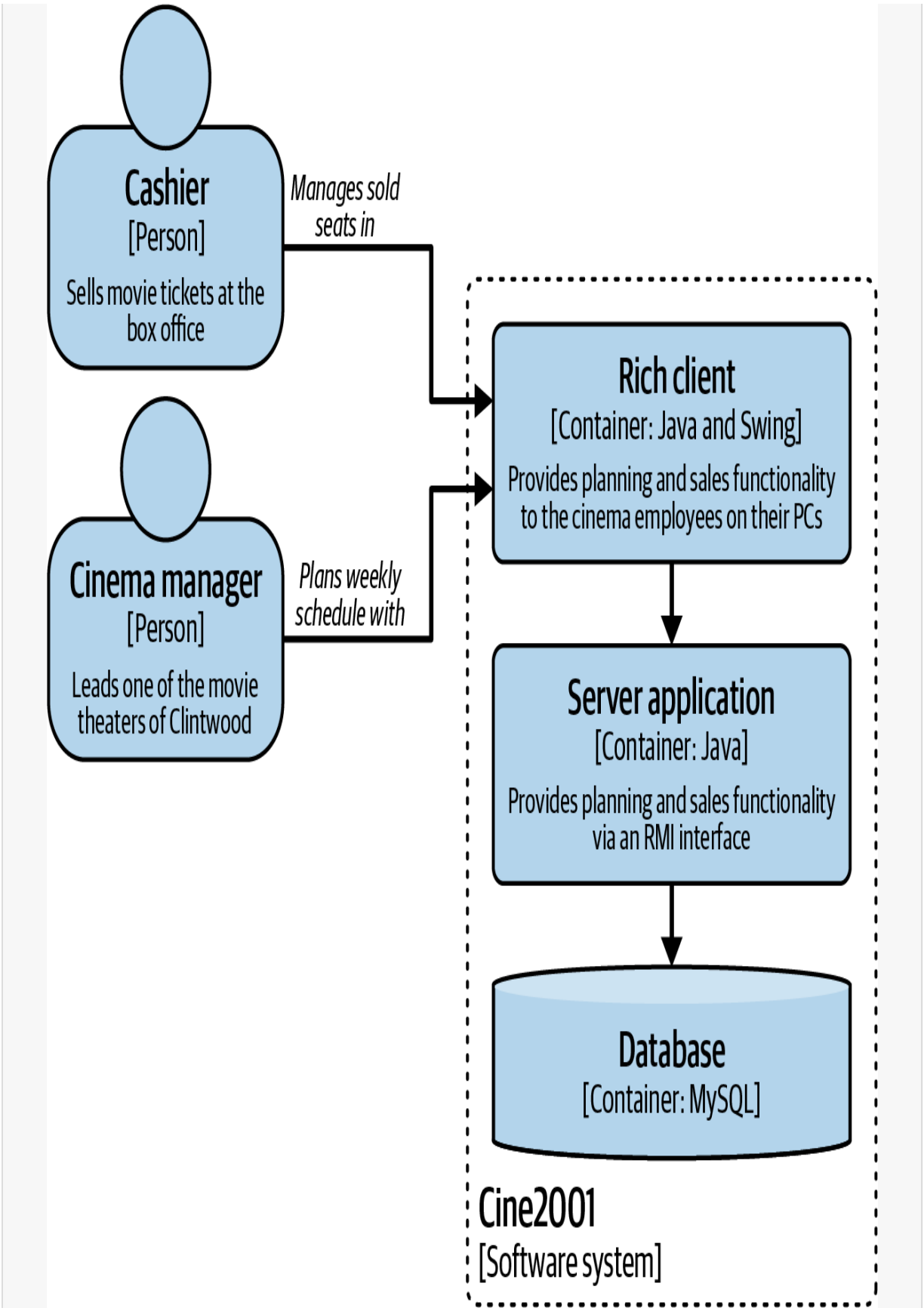

Cashier
[Person]
Sells movie tickets at the box office
Manages sold seats in
Cinema manager
[Person]
Leads one of the movie theaters of Clintwood
Plans weekly schedule with
Rich client
[Container: Java and Swing]
Provides planning and sales functionality to the cinema employees on their PCs
Server application
[Container: Java]
Provides planning and sales functionality via an RMI interface
Database
[Container: MySQL]
Cine2001
[Software system]

Figure 5-2. Containers of the Cine2001 system.

Level 3: Components

By drilling down into a single container, the components within the container are seen. Several components run in the same process.

CASE STUDY “CINEMA”

C4 IN THE CINEMA: COMPONENTS

Focusing on the container *server application*, its components are shown in Figure 5-3.

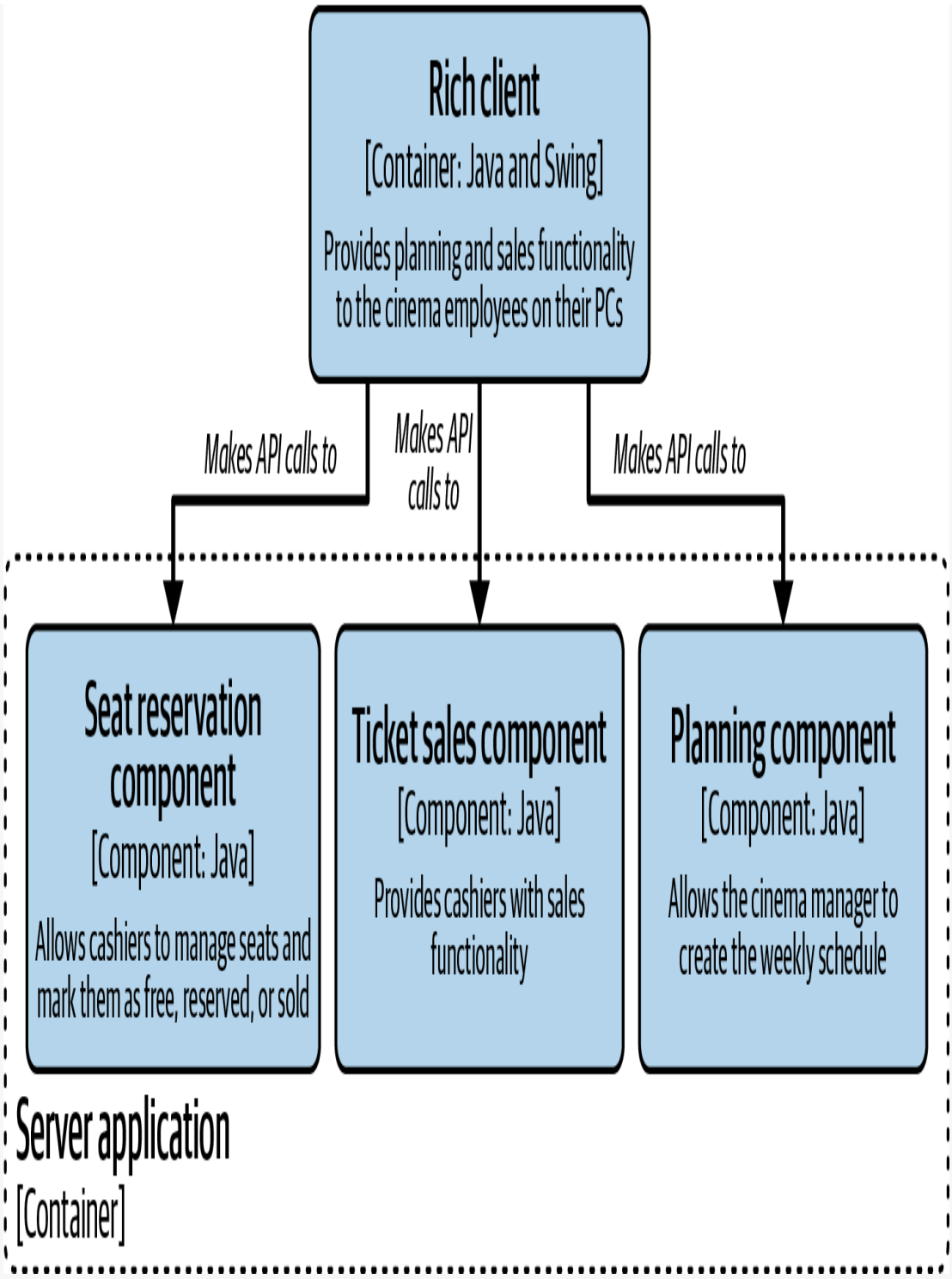


Figure 5-3. Components of the container server application.

Level 4: Code

Finally, drill down into the individual component and look at the code level, i.e., the classes and methods that the component consists of. Usually, UML class diagrams are used here. (If you're not familiar with them, [Fowler 2004a] is a good reference.)

C4 IN THE CINEMA: CODE

Zooming into the server application component, the class diagram in Figure 5-4 is drawn.

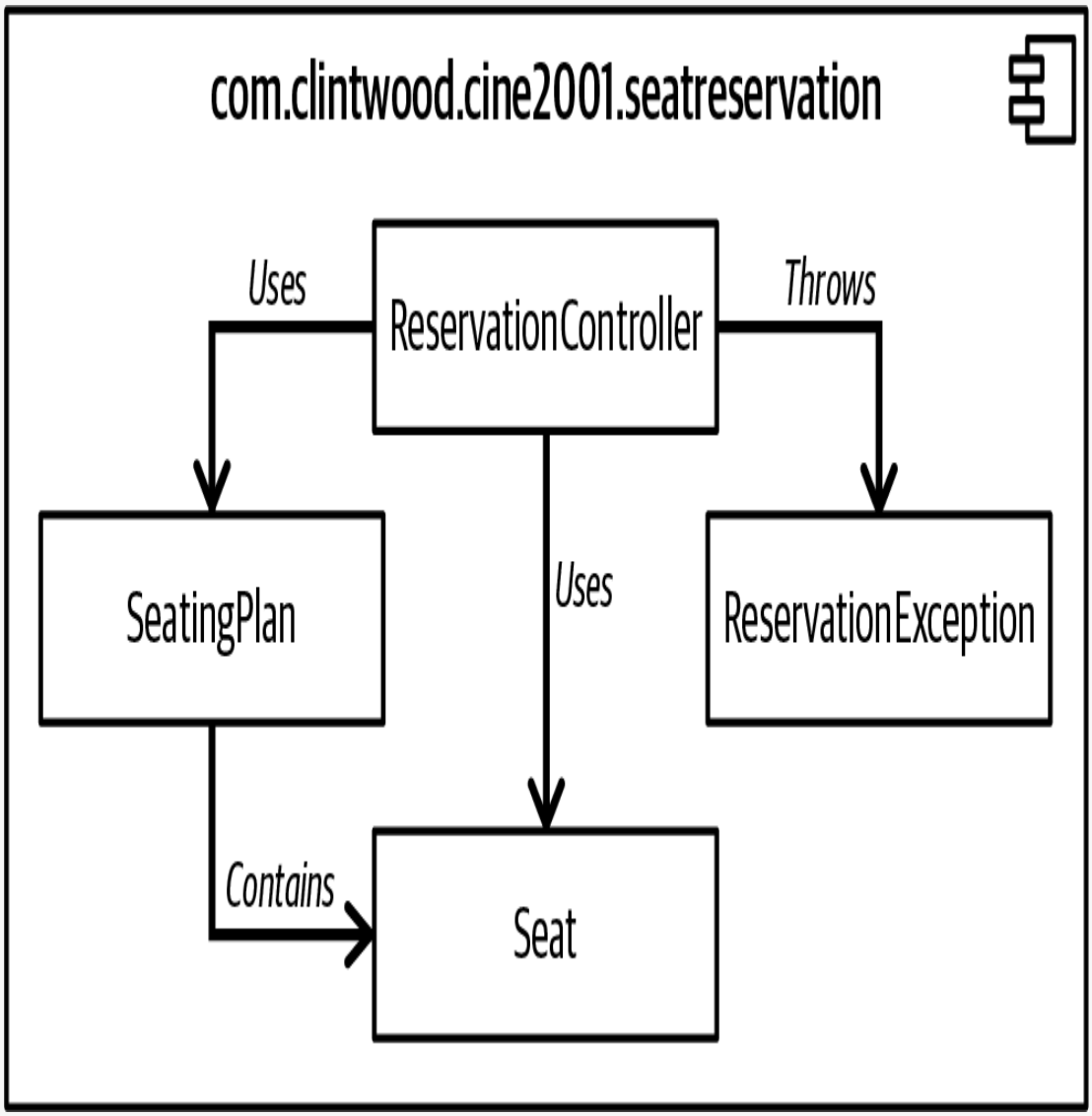


Figure 5-4. Code in the server application component.

More examples of C4 diagrams can be found in Chapters 1, 3, and 10.

Modularity

In the ancient days of programming, a program was *one* piece of code in which one instruction after another was processed, and one could jump blithely back and forth between the lines and access all data from everywhere. It quickly became clear that this kind of programming leads to chaos, and it is better to structure the program and divide it into units that can call each other (see [Dijkstra 1968]).

These program code units are also called *modules*, and David Parnas was the first to describe that their implementation should be hidden behind a narrow interface (see Figure 5-5 and [Parnas 1972]). With this *information hiding*, the caller's cognitive load decreases (see "Team Topologies") because they do not need to understand the details of the implementation. Moreover, the implementation can be easily changed as long as the interface does not change with it.

This fundamental concept of modularity is universal and is applied at various levels, e.g., to methods, classes, packages, and assemblies. In DDD, a bounded context and the tactical building blocks each form a module in this sense.

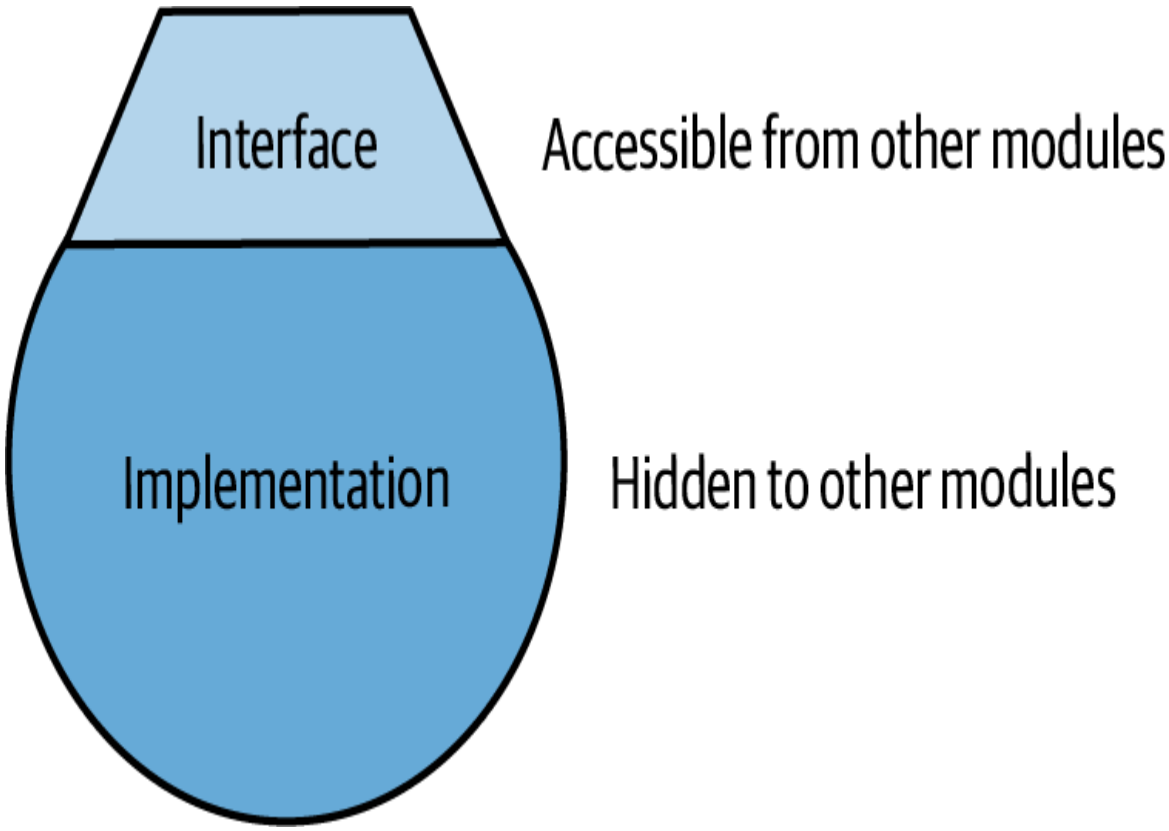


Figure 5-5. A module in the sense of Parnas.

Cohesion and Coupling

To understand and change a module (*module* here is meant in the sense of the previous section) in a software system, developers need to get an overview of the module to be changed and its neighboring modules. It is important to gain an understanding of the task that the module performs internally and how the module works with other modules. To make a change, the developers must also understand the other modules to some extent. The more concentrated the module is inside and the smaller the number of modules it cooperates with, the easier this task becomes (see [Lilienthal 2019]).

Modules with high attraction within and little cooperation with other modules are said to have *high cohesion* and *loose coupling*. Cohesion is a measure that describes how much the parts inside a module belong together conceptually and functionally. Coupling refers to the degree of dependency

between modules in a software system. The more dependencies that exist in a system, the stronger the coupling. High cohesion can be a way to loose coupling (see [Stevens et al. 1974], [Myers 1978], and [Yourdon and Constantine 1978]).

Different kinds of coupling and cohesion can be distinguished and defined. For a comprehensive treatment of the topic, we recommend Vlad Khononov's *Balancing Coupling in Software Design*. The book discusses the close relationship among modularity, complexity, and coupling, and explains dependencies and different types of coupling in great detail using his new model to understand coupling (see [Khononov 2024]).

Figure 5-6 illustrates high cohesion and loose coupling with four modules (four big circles), each containing a different number of parts (small circles inside the big circles) that use each other extensively (connecting lines in the big circles). Each module interface is shown as a trapezoid at the hull of its module and is used by other modules (arrows).

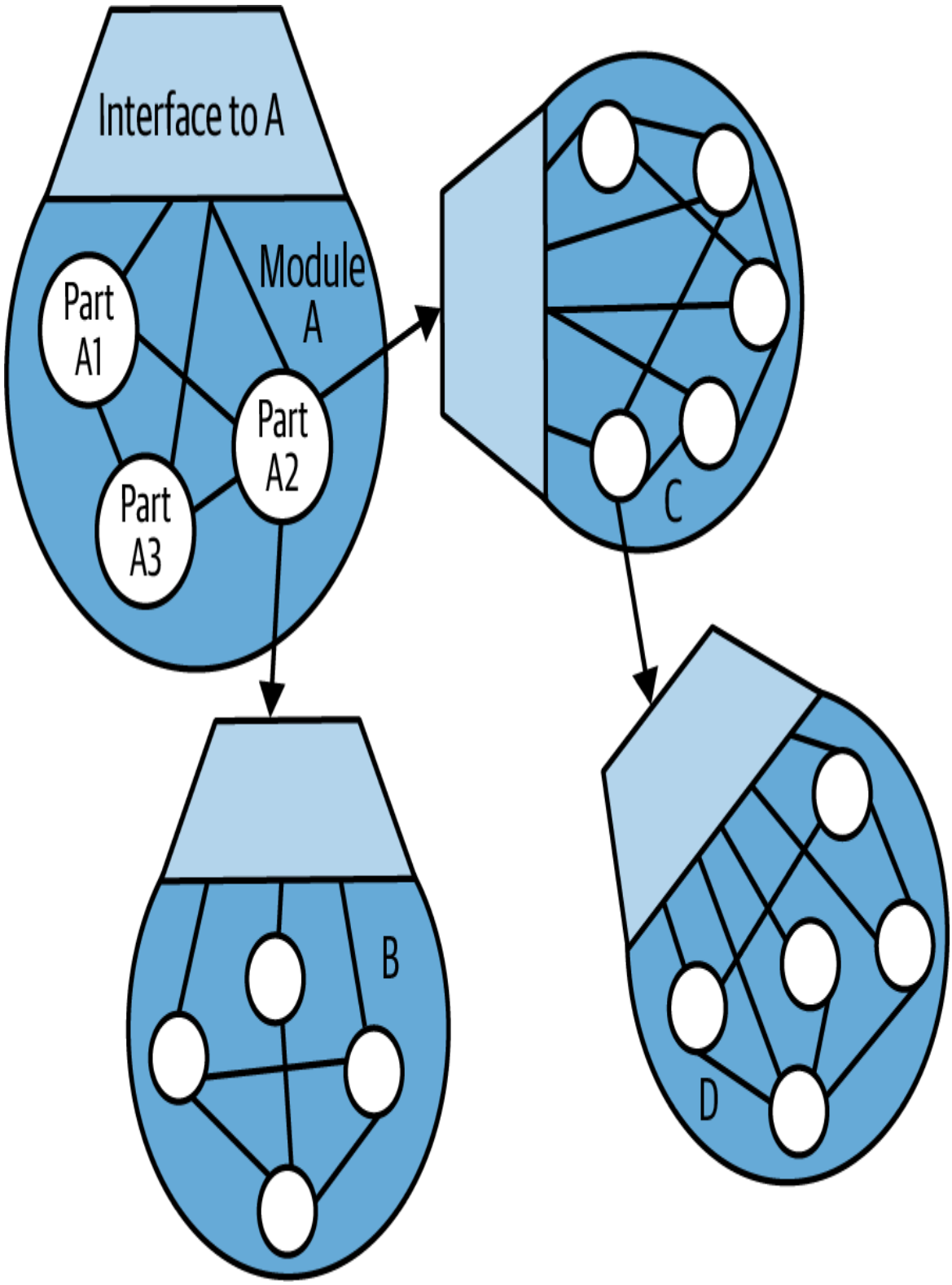


Figure 5-6. Cohesion and coupling.

The four modules in Figure 5-6 have high cohesion inside and loose coupling to the other modules. The parts in the modules use each other extensively to perform the task offered at the narrow interface of the module. There are some relationships between the modules. So there is a coupling between the connected modules, but the coupling is unidirectional and related to a narrow interface. If an architecture is structured in this way, many changes can be implemented locally and thus independently of other modules. Only a limited section of the system then needs to be understood for a change, which also increases the comprehensibility and changeability of the source code.

Cohesion and Coupling in DDD

Very similar statements can be read in the DDD literature. On a strategic level we like a quote from Michael Plöd:

> *A model in a bounded context should have high cohesion, meaning that it forms a coherent whole around a single unified purpose. [Plöd 2025]*

For the tactical level, in the "blue book" in the section on "Standalone Classes," Eric Evans makes clear that interdependencies between classes or modules make models and design hard to understand, test, and maintain:

> *Low coupling is a basic way to reduce conceptual overload. A standalone class is an extreme of low coupling. [Evans 2004, 267].*

And in the glossary of the same book:

> *A standalone class is a class that can be understood and tested without reference to any others, except system primitives and basic libraries. [Evans 2004, 513].*

Of course, this has to be balanced with the size of the classes and must not be used as a justification to put entire systems into one class.

Value objects are the building blocks that logically best meet this default of standalone classes because they are the leaves in an object tree. But all

classes and modules should be built with this in mind: as little coupling as possible. Or as Evans puts it:

> *Low coupling is fundamental to object design. [Evans 2004, 266]*

Cohesion and Coupling in the Domain

By focusing on the ubiquitous language and the domain-motivated decomposition for bounded contexts, Domain-Driven Design supports this goal in an excellent way. Domain-specific concepts and subdomains, which come from the business world of domain experts, inherently have high cohesion and loose coupling. As a result, bounded contexts based on these (from a business perspective) well-decomposed subdomains have significantly fewer interdependencies than modularization based on technical criteria could ever achieve, such as through a client-server decomposition. The DDD concepts *aggregate* and *standalone class* also aim at high cohesion and loose coupling.

WARNING

Care should be taken with the term *loose coupling*.

Some software developers and architects understand it to mean that there should be no direct calls (e.g., via remote procedure call), but sending messages or events over a bus/broker automatically leads to loose coupling.

At this point we must disagree: loose coupling in a domain sense means that there is as little coupling as possible. So, if anything, there is a narrow domain-oriented interface.

Systems in which messages are sent excessively are certainly loosely coupled from a technical point of view, but not from a domain-oriented point of view. In such a case, the context that needs to be understood when changes are made is much larger than the chatty module itself. This is because, once again, the message receiver or the reason for the message and, potentially, the responses must also be understood.

High cohesion and loose coupling are also given by Eric Evans, Vaughn Vernon, and other DDD authors as the rationale for and goal of Domain-Driven Design. The absolute opposite of a cohesive, loosely coupled architecture is the *big ball of mud* often cited in the realm of DDD.

Big Ball of Mud

Since the publication of the eponymous article by [Foote and Yoder 1997], interwoven structures with high coupling and low cohesion have been referred to as a *big ball of mud* (BBoM) in the DDD community, as mentioned earlier in the book. The dependencies between the parts in a BBoM are not acyclic—instead they form dependency cycles.

At the Class Level

At the class level, a big ball of mud looks like Figure 5-7. This class dependency cycle comes from a Java system with 650,000 lines of code (LOC). This and the following screenshots are diagrams that have been generated automatically from the source code with the tool Sonargraph (*https://www.hello2morrow.com*). What can be seen here is a big ball of mud with a very large class dependency cycle of 463 classes spanning 50 packages. Each rectangle corresponds to a class, and the arrows between the rectangles are the relationships between the classes. At the class level, this cycle is tightly interwoven. This can be seen from the fact that a majority of the classes are very close together and there are few satellites. In addition, classes from the same packages—identifiable by the same color or grayscale scheme—are distributed throughout the cycle, also indicating little order at the higher levels.

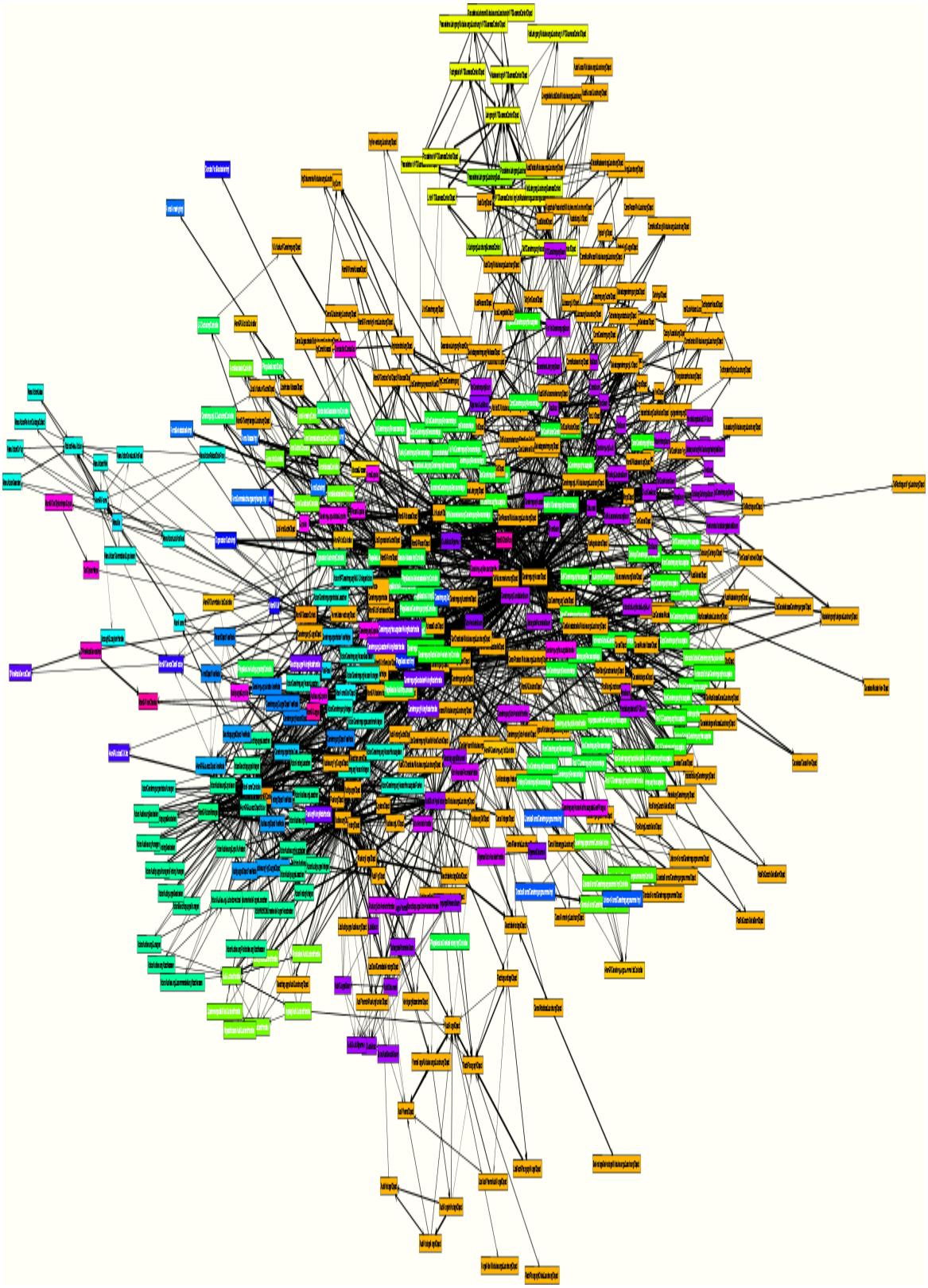


Figure 5-7. Screenshot of an architecture review tool: many classes in a big ball of mud.

In a well-designed architecture, we would expect packages to be used as structuring means with high cohesion (see "Cohesion and Coupling"). All classes in a package work closely together because they perform a common task. There is high cohesion between these classes, but they do not need anything from the classes in other packages. So there is no strong coupling between classes in different packages. If the packages in Figure 5-7 had been used like this, then different clusters of classes that all have the same color would be seen, as in Figure 5-8.

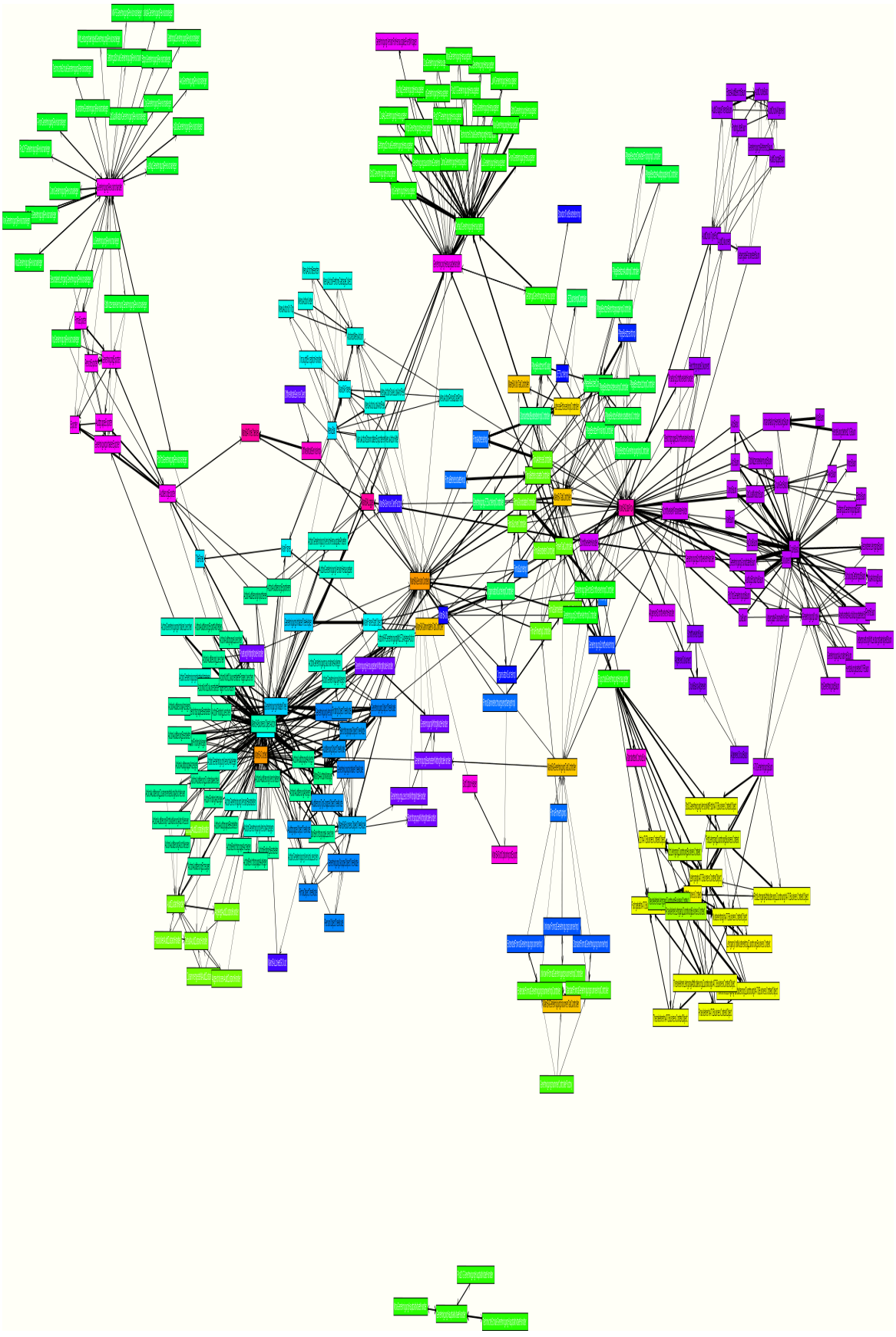


Figure 5-8. Screenshot of an architecture review tool: big ball of mud with satellites.

This class structure is also a dependency cycle; however, the packages can be identified as clusters here by the colors or the grayscale scheme of the classes. Unfortunately, there are still cyclic relationships between the individual clusters—so the whole thing is still a big ball of mud. However, the connections that have to be worked on for a decomposition can be clearly seen. t is therefore important to put classes that process the same business function into the same package.

At the Architecture Level

At the architectural level, for example, a big ball of mud looks like Figure 5-9. This shows 50 projects from a 10 million LOC Java system. The individual projects are shown as squares in the center of the diagram, and their relationships to each other are represented by the arcs on the left and right sides of the axis. The arcs on the left side of the axis represent relationships from top to bottom, and the arcs on the right side represent relationships from bottom to top.

The squares on the center axis, each representing a project, have been ordered with a graph-optimizing algorithm such that all projects that are not needed by any other project are arranged on top. All projects that are used a lot are sorted further down. Still, many arcs remain on the right side because the projects further down need functionality from the projects further up. This is exactly what a big ball of mud is—many dependencies between elements of the architecture at all possible levels.

The bottom line is that a big ball of mud is a software system in which the number of dependencies, i.e., the total coupling, was never an important consideration and has continued to increase uncontrolled. In summary, you could say that if you don’t have your dependencies under control, you don’t have your architecture under control.

How could this come about?

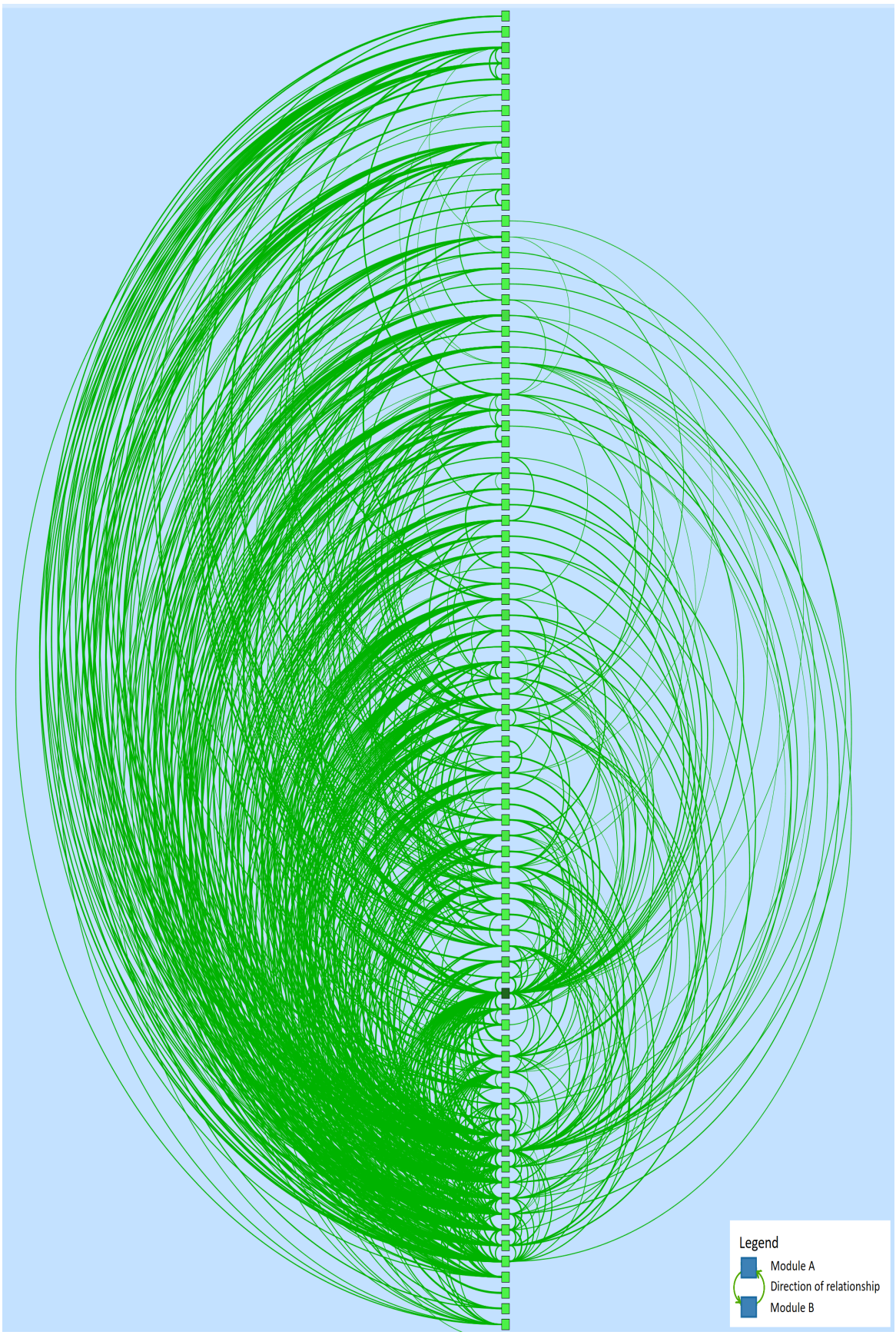

Legend
Module A
Direction of relationship
Module B

Figure 5-9. Screenshot of an architecture review tool: big ball of mud at the architectural level.

Where Does the Big Ball of Mud Come From?

In many systems, while maintainability and understandability decrease over time, the cost of change increases. This typical degradation of a system over time is visualized in Figure 5-10.[2]

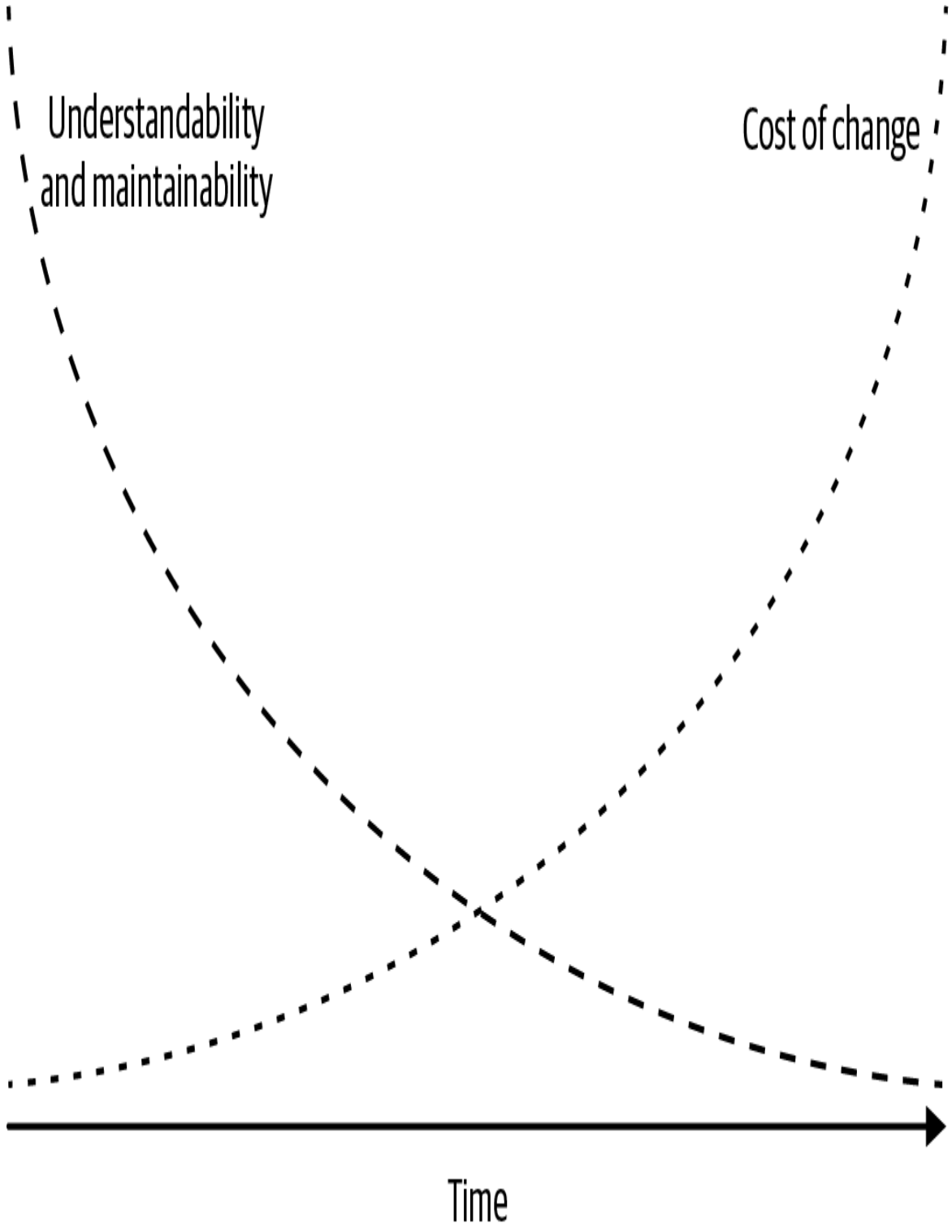


Figure 5-10. Typical change of the system situation over time.

Let’s assume that at the beginning of a software development project, the team designed a high-quality, modular architecture that has high cohesion

and loose coupling. Ideally, the software system had a small amount of accidental complexity at the beginning. Recall from Chapter 2 that accidental complexity arises from misunderstandings, poor design, and poor architecture. Thus, the system appears in Figure 5-11 in the lower left as a small dot (light gray in the print book, green in the ebook). In this initial stage, the software system is in the corridor of good legacy systems.

The longer the system lives, the more the team extends it and adds new functionality. So the initially small dot gets bigger and bigger over time because it contains more and more business value in the form of features. If the developers do not succeed in maintaining the modular architecture during these extensions, the architecture erodes more and more over time. The system takes on more accidental complexity (arrows pointing upwards in Figure 5-11 going up from the corridor of good legacy systems; in the ebook, the arrows are initially yellow and then even turn red as the complexity increases).

Eventually, the system gets closer and closer to a big ball of mud with high accidental complexity and ends up in the corridor of bad legacy systems. Figure 5-11 makes this slow decay clear by showing that the arrows pointing upwards become shorter and shorter. The developers create less and less functionality per time period or budget. Consequentially, errors become harder to track, to the point where every change becomes a painful effort.

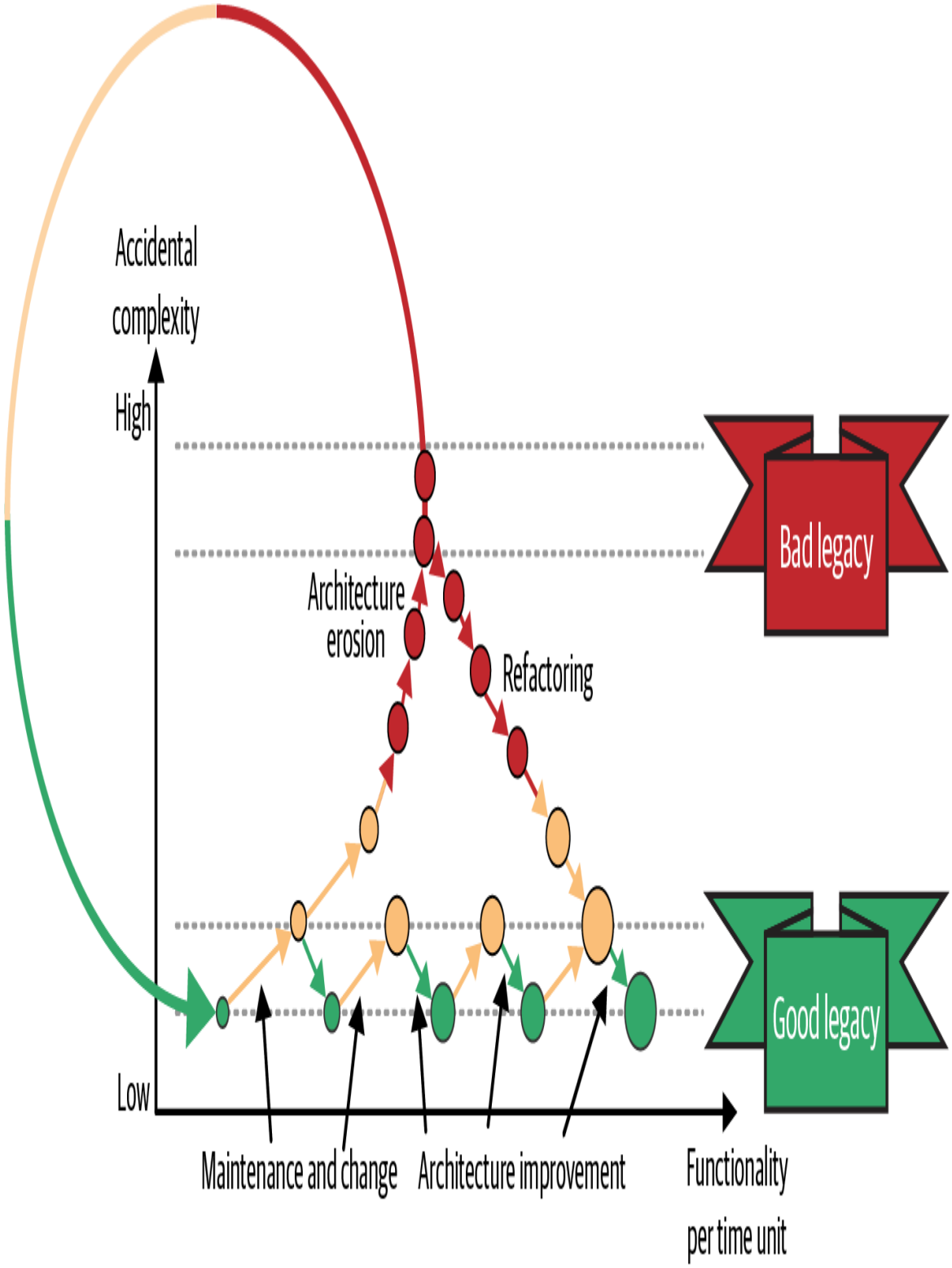


Figure 5-11. Corridors of accidental complexity.

It would be better if management and the team accepted that software development is a constant learning process in which the initial version of a

solution is rarely the final one. An extension in the corridor of good legacy systems therefore always leads first to more accidental complexity (see the arrows pointing upwards marked “maintenance and change” in Figure 5-11). The revision of the architecture (see the arrows pointing downwards marked “architecture improvement” in Figure 5-11 in the corridor for good legacy systems) must be carried out at regular intervals to avoid a big ball of mud. If this principle is followed, development proceeds in a steady up-and-down sequence of extension and refactoring. If a team can perform this sequence of extension and refactoring on a sustained basis, the system will remain in the corridor of good legacy systems (see the arrows in the corridor of good legacy in Figure 5-11).

Coupling and Reuse

When analyzing older software systems, we regularly find that the development teams and architects are surprised at how high the coupling has become in their systems. We are less surprised because our discipline and our development environments are geared toward reuse.

The credo in almost all projects is: what is already there should be used and not implemented again. Duplicated code must be avoided at all costs, even if the existing source code does not quite fit what is needed at the moment. Then the developers abstract, parameterize, or work with conditionals.

Our opinion: what a big misunderstanding this is, and how much additional coupling it adds to our software systems! Whenever you have time to revise your architecture, please make sure that you reduce the coupling among your modules, packages, and classes. Your successors at the keyboard will thank you for it with every extension and also with the eventual decomposition of the legacy system.

Architectural Styles for Legacy Systems

In Figure 5-12 you can see the architecture concepts that we think make sense for large legacy systems: microservices and modular monoliths as

well as hexagonal architectures and Self-contained Systems.

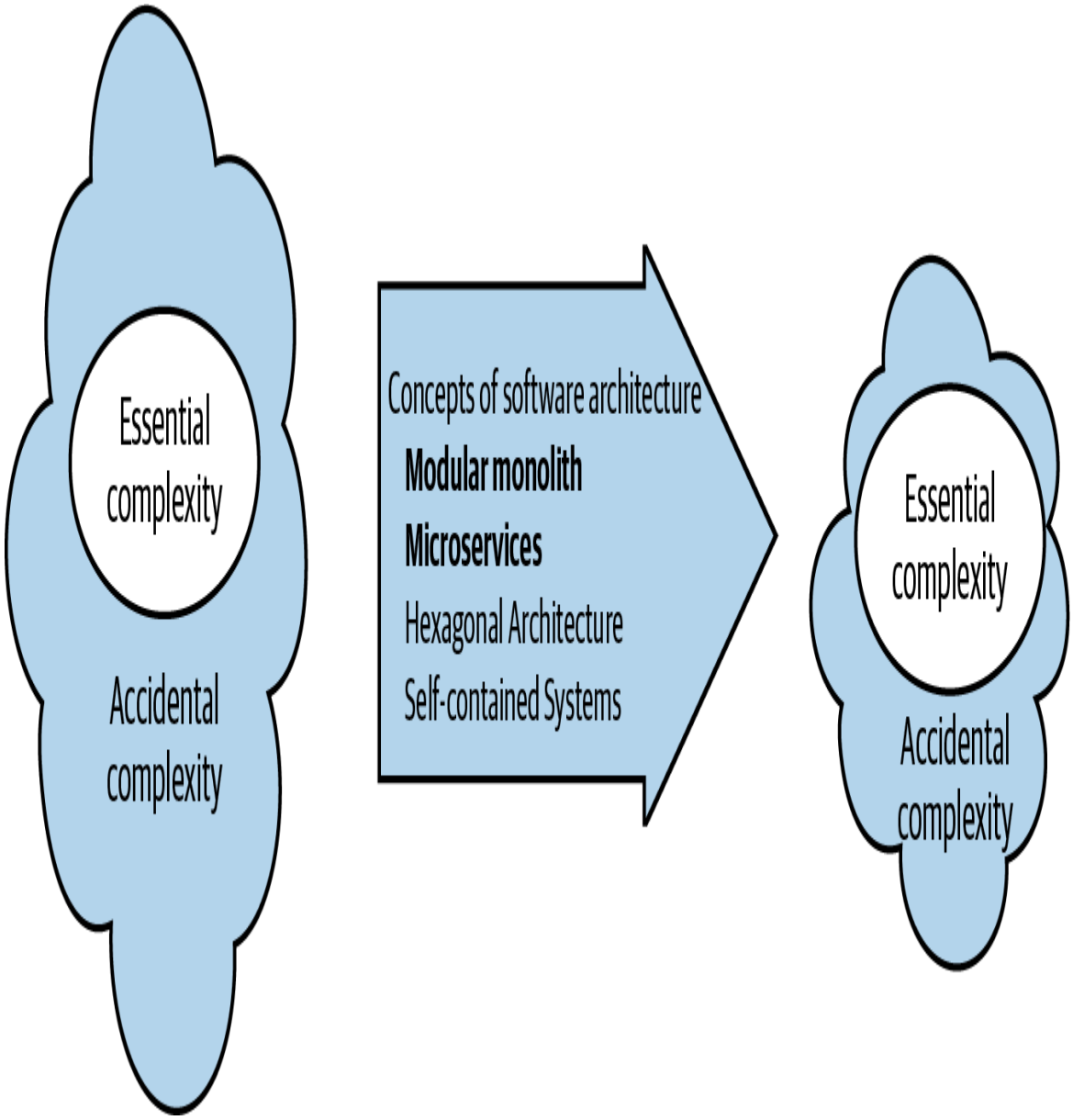


Figure 5-12. Concepts of software architecture can help reduce accidental complexity.

Two architectural styles are missing from Figure 5-12:

Service-oriented architecture

> Service-oriented architecture (SOA) in the misunderstood way that it is implemented in most projects (see "Service-Oriented Architectures") fits poorly with the idea of loosely coupled contexts.

Layered architecture

Layered architecture is the foundation of any software architecture today and was already used in the "blue book" (see "The Layered Architecture in DDD"), so we do not list it again here.

Monolith

The term *monolith* has taken on a very negative connotation in recent years. It is used to describe legacy systems that have a high level of coupling and are thus a big ball of mud. However, the term really only describes *deployment* to begin with.

> **NOTE**
>
> Monolithic architecture means that a software system is to be built and delivered as a single deployment unit.

This is in contrast to microservices architecture, which consists of multiple deployment units (see Figure 5-13 and [Newman 2022] and [Wolff 2017b]).

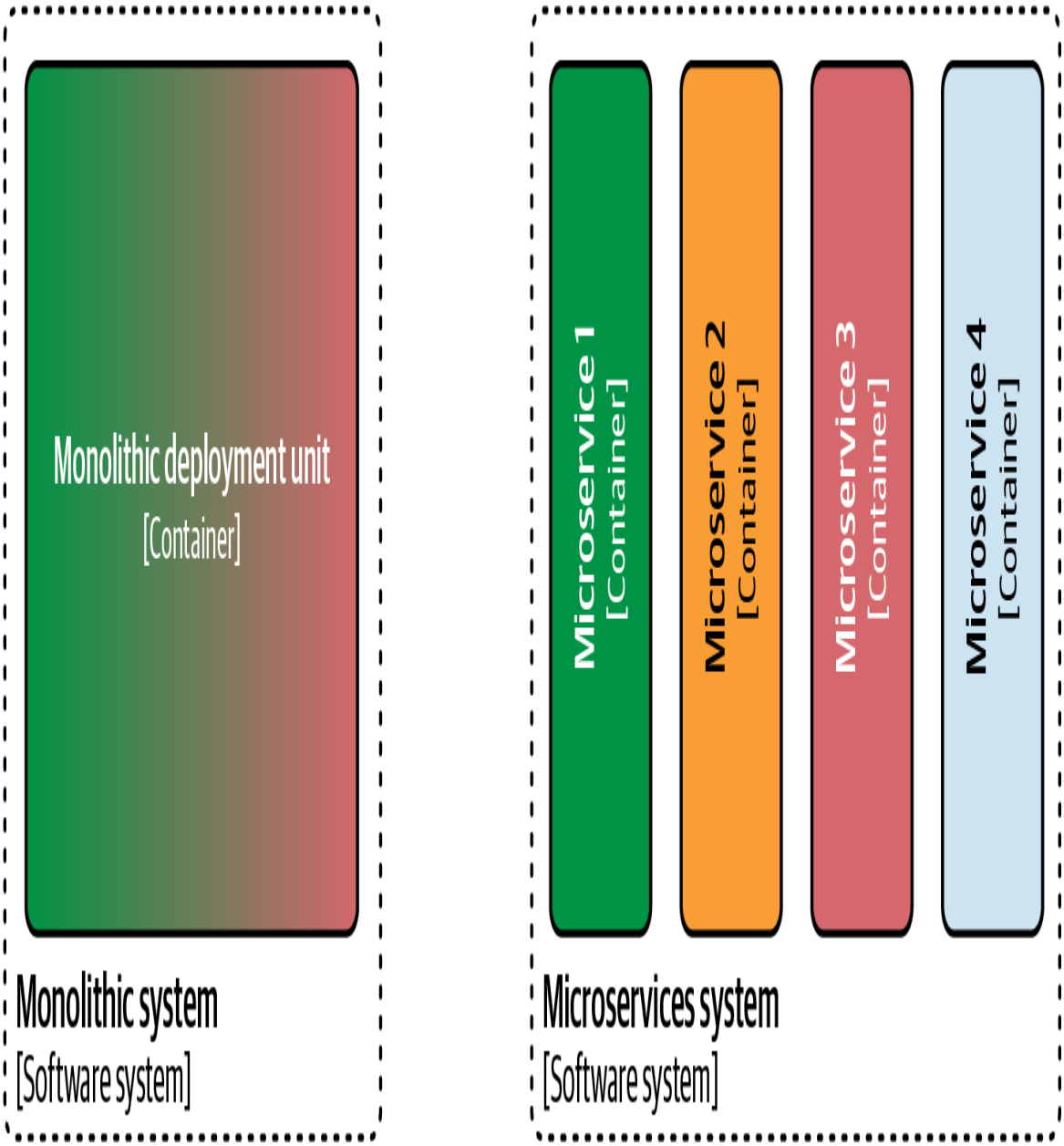


Figure 5-13. Deployment: one large unit versus many small units.

Consisting of only one deployment unit is not a problem for many systems; in fact, it is the simplest solution. However, it becomes problematic in the following cases:

- A team working on a part of the system has a rate of change greater than any other teams working on the whole system.
- The system exceeds a certain size (making testing and rollout slow).

- The system should be scaled horizontally or different modules have different scaling requirements.

When dealing with any of these requirements, microservices can be helpful.

Microservices

In a *microservices architecture*, a software system is not built from just one program, but from several programs that can be deployed independently of one another (the microservices).

> **NOTE**
>
> Many developers and architects initially thought that this was only about the resulting technical advantages, such as scalability. But we believe the main benefit is different. Microservices make it possible to divide software among development teams in such a way that the teams can develop independently and autonomously.

Thus, changes don't require immediate corresponding changes in another microservice, and slow testing and slow rollout in another microservice don't slow down testing and rollout in the current microservice. This allows every team to work at its own pace. Overall, the development will be faster than before. So, microservices are first and foremost not about technology, but about people and their performance.

For an effective development team, it is desirable to have enough team members so that vacation and sick leave do not lead to a standstill, i.e., at least three people. In addition, communication should remain manageable, so no more than seven ± two people who have to keep each other up to date. Such a team should own a piece of software, a module, for (further) development, which has as few and as stable dependencies on the rest of the system as possible. Because only then can the team make independent decisions and develop its module further at its own pace without waiting for other teams and their deliveries.

In a microservices architecture, the focus is on dividing the system into individual microservices (see Figure 5-14). The technical layering is implemented locally in each microservice and is therefore a subordinate concept to the division into microservices. In Figure 5-14, the technical layers in the microservices are shown as the UI layer, application layer, and domain layer. However, there may well be more or other technical layers in each microservice.

Each microservice is exclusively owned by one team, and that team should be responsible for all the microservice's source code, database, technologies, deployment, and arrangements with users. The teams should be cross-functional in their composition so that they can do everything on their own. This requires the microservice to be of a size that a team of about seven people can be completely responsible for it. This kind of acceptance of responsibility by a team brings the hope that software development in large companies can match the rate of speed of start-ups. Chapter 9 will discuss more arguments for why cross-functional teams lead to better results.

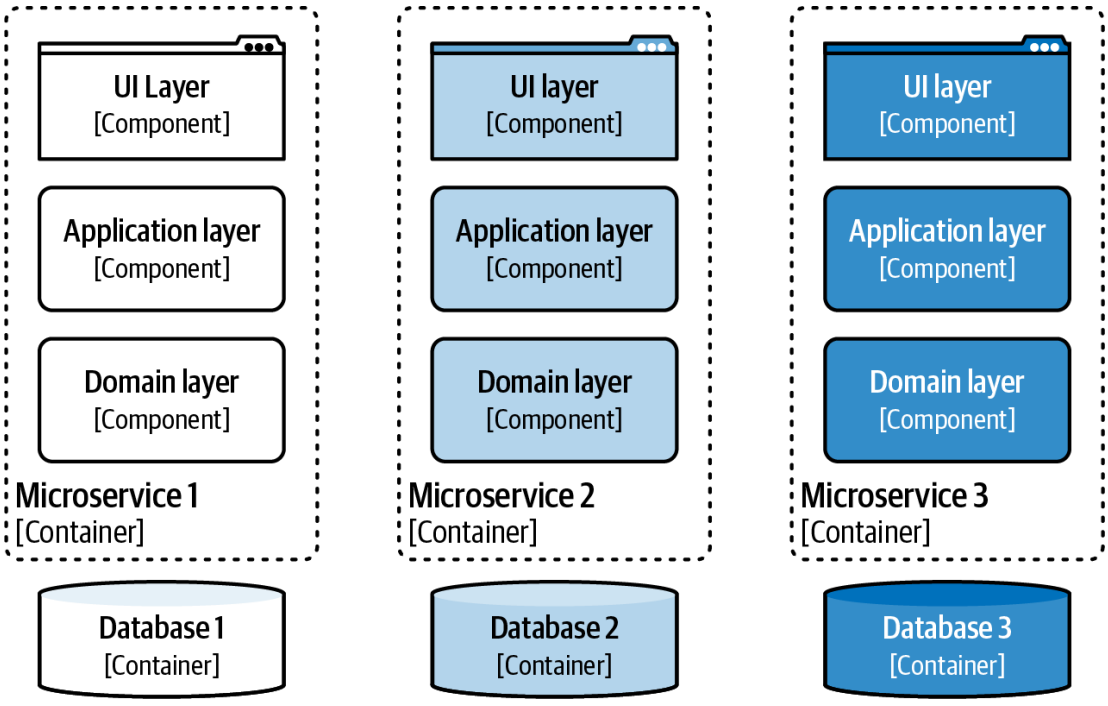


Figure 5-14. Microservices.

Domain-Driven Design supports this idea through its specifications for dividing a domain into bounded contexts. In this respect, Domain-Driven Design and microservices fit together very well. A microservice should implement one (or several) bounded context, and one bounded context should be deployed in only one microservice (maybe together with other bounded contexts). Since bounded contexts represent independent parts of the domain (assuming the goal that each be a subdomain), teams can work independently of each other. They can further develop their microservice without having to wait for the other teams. But what can be done with the big ol' monoliths?

From Monolith to Microservices

There is no simple path from a monolith that has a strongly coupled architecture to a microservices architecture. In Figure 5-15, we have adapted a graphic that C4 model author Simon Brown posted on Twitter[3] to summarize the possible paths from a big ball of mud to a microservices architecture.

In Figure 5-15, the monolithic big ball of mud can be seen in the lower left quadrant, shown here as an architecture without boundaries between possible modules with only one deployment unit (x-axis: distribution). Modularity (y-axis) is low for a monolithic big ball of mud.

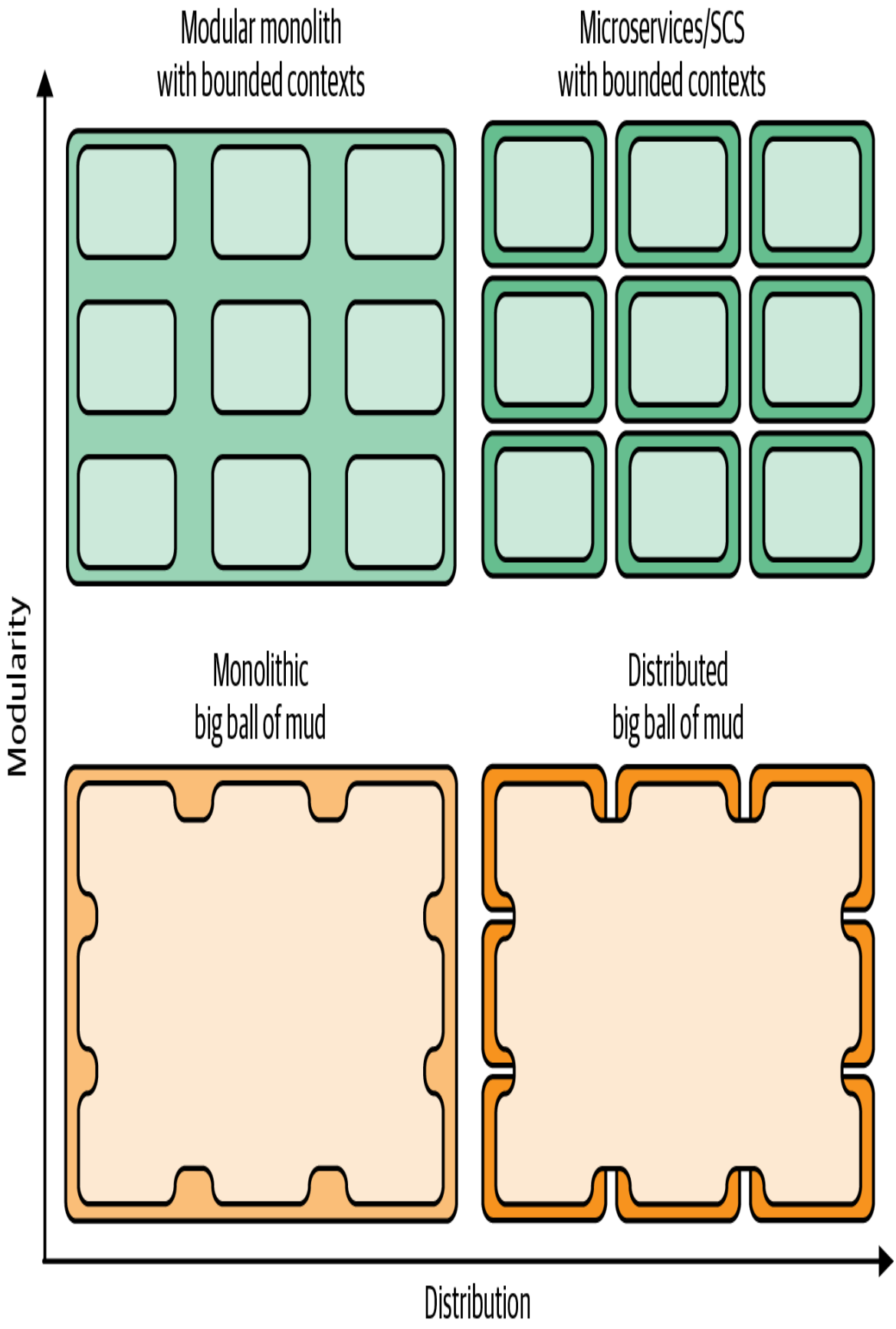

Modular monolith
with bounded contexts
Microservices/SCS
with bounded contexts
Monolithic
big ball of mud
Distributed
big ball of mud
Modularity
Distribution

Figure 5-15. Paths from monolith to microservices (adapted from Simon Brown).

If you simply break this big ball of mud into several deployment units, you end up in the lower right quadrant. You get a *distributed big ball of mud*—the combination of all evil. In contrast to the big ball of mud within a deployment unit, which was "only" difficult to maintain due to the high internal coupling, you have now also "gained" (read: digressed into) the complexity of distributed computing.

If you want to convert your big ball of mud into a microservices architecture without ending up with the distributed big ball of mud, you can first step-by-step move your architecture from the quadrant on the bottom left to the quadrant on the top left. That is one of the better ways to transition a monolithic big ball of mud to microservices. First, break the tangle into a modular monolith by forming and extracting independent bounded contexts into modules inside the deployment monolith. Once you have transformed your system into a manageable monolith, the step to the microservices architecture can be made from the top left to the top right by moving bounded contexts into their own deployment units/microservices.

CAROLA'S FIELD STORY: A "DISTRIBUTED BIG BALL OF MUD"

Some time ago, a team from our company, WPS, received a system from London for maintenance, which caused great expectation when it was announced. The system was still comparatively young (five years of development) and was announced as a microservices architecture. Everyone was very excited to see what was in store for them!

What we got was 500,000 LOC split across 100 microservices sending 250 different types of events to each other. Maintenance turned out to be extremely complicated because almost every algorithm in one of the 100 microservices sent out an event that one or more of the other microservices would act on. Debugging and extension involved moving from one microservice and its runtime environment to another microservice almost every time. What we actually got was a "distributed big ball of mud" with all its drawbacks.

Strategic Domain-Driven Transformation will show us the way to evolve a big ball of mud into a modular monolith and then also into a microservices architecture in Part III.

Inside a monolith or a microservice, different architectural styles can be used. Let's have a look at the different options.

The Good Old Layered Architecture

In many areas of computer science, abstractions build on each other; for example, network protocols, where TCP is based on IP and HTTP is based on TCP. The *layered architecture* (see Figure 5-16) was derived from this as a general pattern (see [Buschmann et al. 1996]).

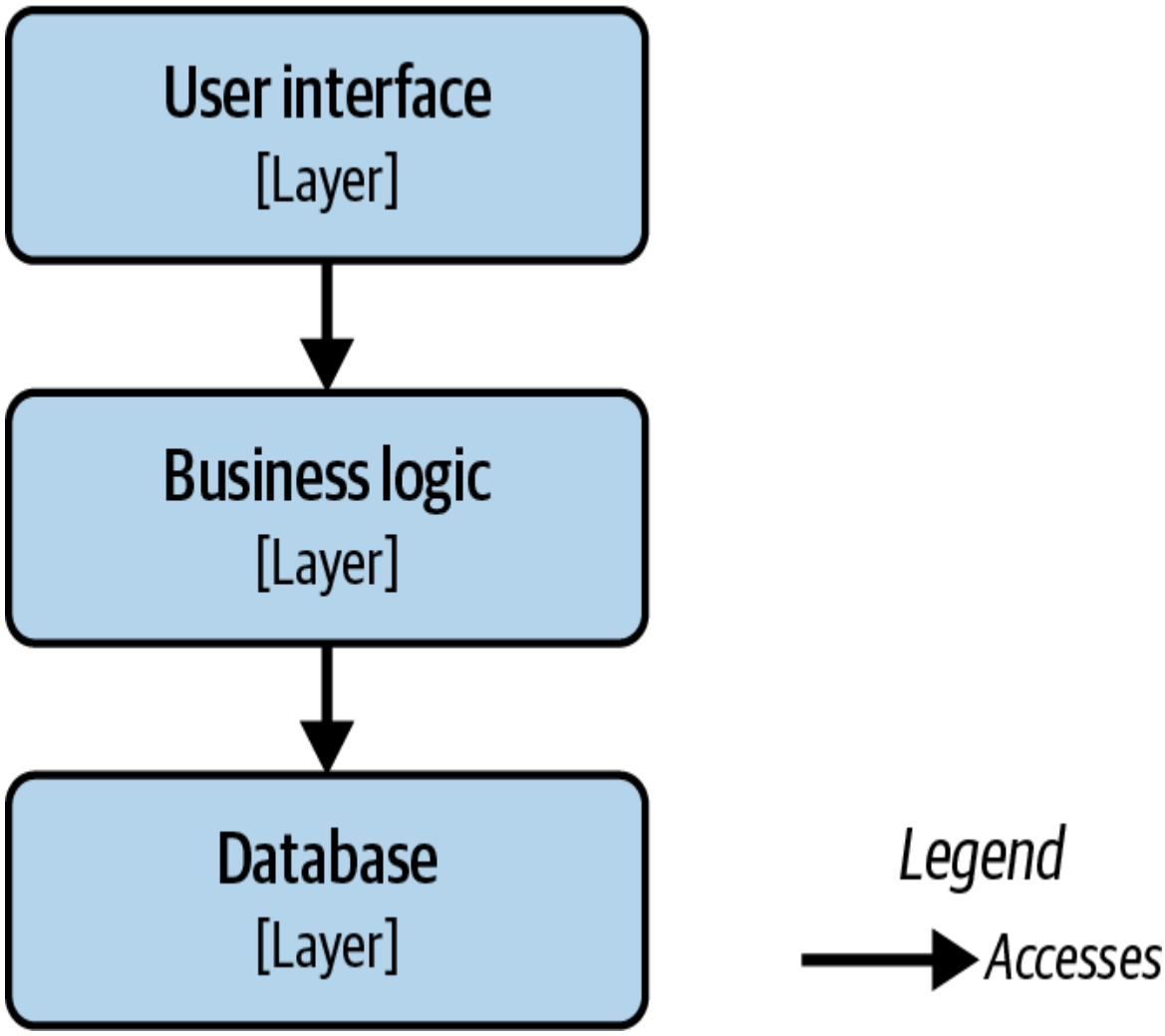


Figure 5-16. Three-layer architecture.

As a general rule, higher-level layers may access lower-level layers, but lower-level layers may not access higher-level layers. This creates a hierarchy, and the layers follow the Acyclic Dependency Principle (ADP, see [Martin 2018], p. 112).

Buschmann et al. distinguish strict layered architecture (a layer may only use the layer directly below it) from relaxed layered architecture (a layer may use all layers below it). When it became apparent that the "above-below" of layered architecture was not sufficient to describe how modern systems are built, it evolved to an "inside-out."

Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More

As an evolution of the layered architecture, several new architectural styles emerged:

- In 1998 the first book came out about the *Tools & Materials Approach* by Heinz Züllighoven, in which a U-shaped architecture was recommended. The enclosing U should contain the technical source code, and the business source code should be enclosed in the U (see [Züllighoven 2005]).
- In German-speaking countries there was another development in this direction. Johannes Siedersleben, with the company sd&m, developed the Quality Software Architecture (briefly *Quasar Enterprise*), in which there are different so-named blood groups of software parts: A-software (application code) may not be mixed with T-software (technical code) (see [Siedersleben 2004] and [Voss et al. 2006]).
- *Hexagonal Architecture* was introduced by Alistair Cockburn in the 1990s and renamed to *Ports & Adapters* in 2005. The inside of the hexagon (the business source code) offers ports to the outside, to which the outside technical source code accesses with adapters. The inside knows only the ports, but not the outside (see [Cockburn 2005] and [Cockburn and Garrido de Paz 2024]).
- Jeffrey Palermo started writing about *Onion Architecture* in 2008, where several rings are inside each other like an onion. Access may only ever be from the outside in (see [Palermo 2008]).
- Robert C. Martin published the *SOLID principles* in the year 2000 and expressed with the S (from SOLID) the Single Responsibility Principle—that a class should be changed only for one reason (see [Martin 2018], p. 57): either for a technical or for a business

reason. Martin eventually summarized these principles and more as *Clean Architecture* and published a book of the same name (see [Martin 2018]).

Although these architectures differ in their details, they all have the same goal: reduction of complexity through modularity. All of these styles achieve this goal by emphasizing the separation of the business source code from the technology and the hierarchical structure of the domain-related modules.

Different Styles Together—Explicit Architecture

In Figure 5-17 we summarize these different architectures following the *Explicit Architecture* of Herberto Graca (see [Graca 2017]).

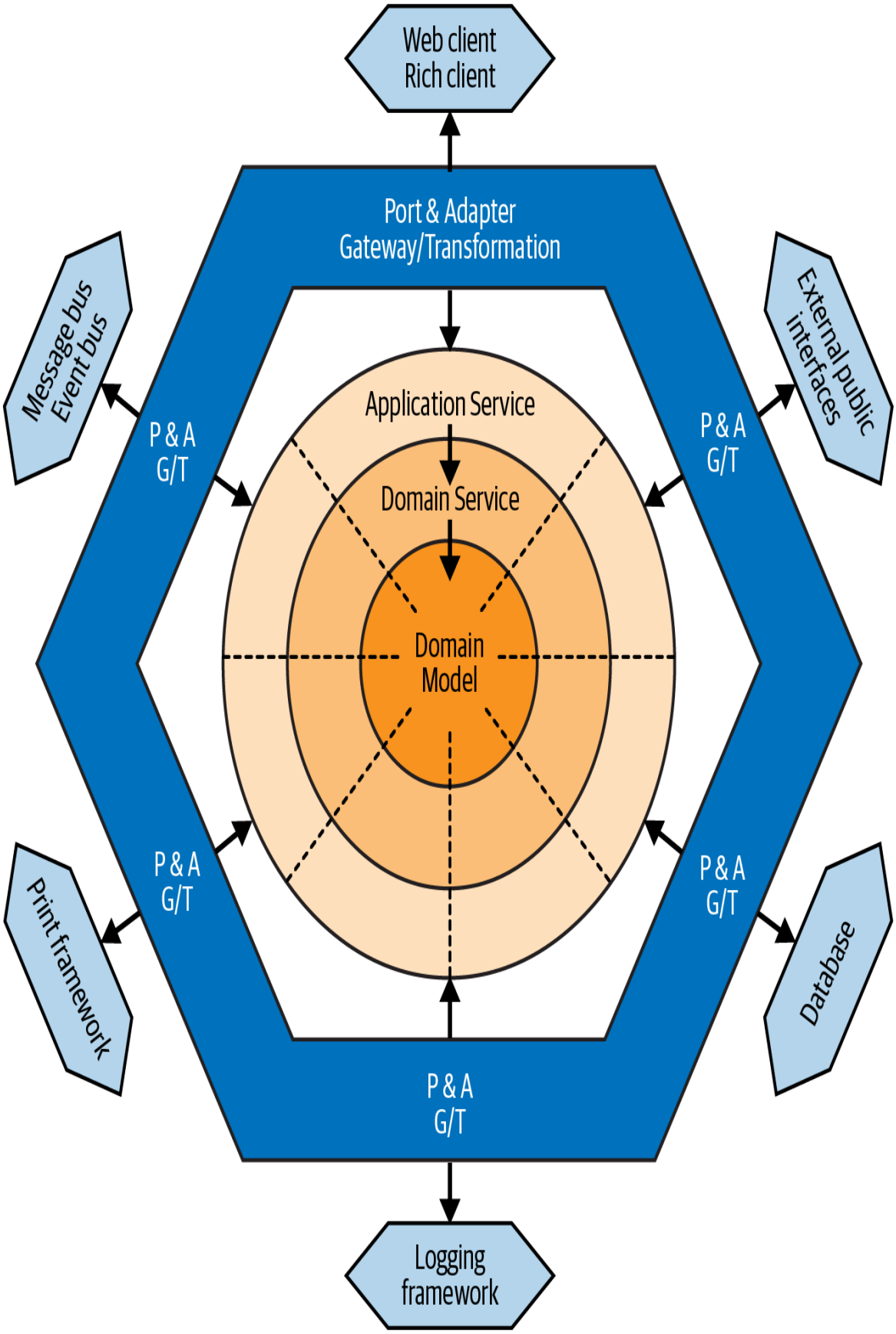

Web client
Rich client
Port & Adapter
Gateway/Transformation
Message bus
Event bus
External public
interfaces
P & A
G/T
P & A
G/T
Application Service
Domain Service
Domain
Model
P & A
G/T
P & A
G/T
Print framework
Database
P & A
G/T
Logging
framework

Figure 5-17. Hexagonal, Onion, and Clean Architecture.

In the center, you can see the application core, the business source code, as an onion consisting of three layers whose relationships go from the outside to the inside. The decomposition of the onion into bounded contexts is indicated by dashed lines. Surrounding the onion is the hexagon, which enables collaboration with the technical parts of the software via port and adapter, gateway, or transformation.

With these architecture styles, the application core becomes truly independent of the technical parts of the software. The application core can be tested without the user interface, database, web server, or other external interfaces. The user interface, database, and interfaces can be changed or replaced without affecting the rest of the system, because the application core knows nothing about the rest of the world.

Inside-Outside Architectures and Tactical DDD

Chapter 3 taught that Domain-Driven Design only wants to take care of the application core; however, there are also building blocks, such as repositories, that need to be linked to the technology in some way. Consider Figure 5-18, where the DDD building blocks are distributed in the different rings of the hexagonal onion architecture from Figure 5-17. We omit the partitioning into bounded contexts here for space reasons. Actually, all these kinds of building blocks exist in every bounded context.

In Figure 5-18 you can see how the building blocks from DDD are distributed across the different layers of the domain-oriented core. The inner circle contains the aggregate with its root entity; other entities belonging to the aggregate and value objects; and the domain events, factories, and interfaces for repositories. Technical repository implementations are connected via a repository adapter that implements a given repository interface.

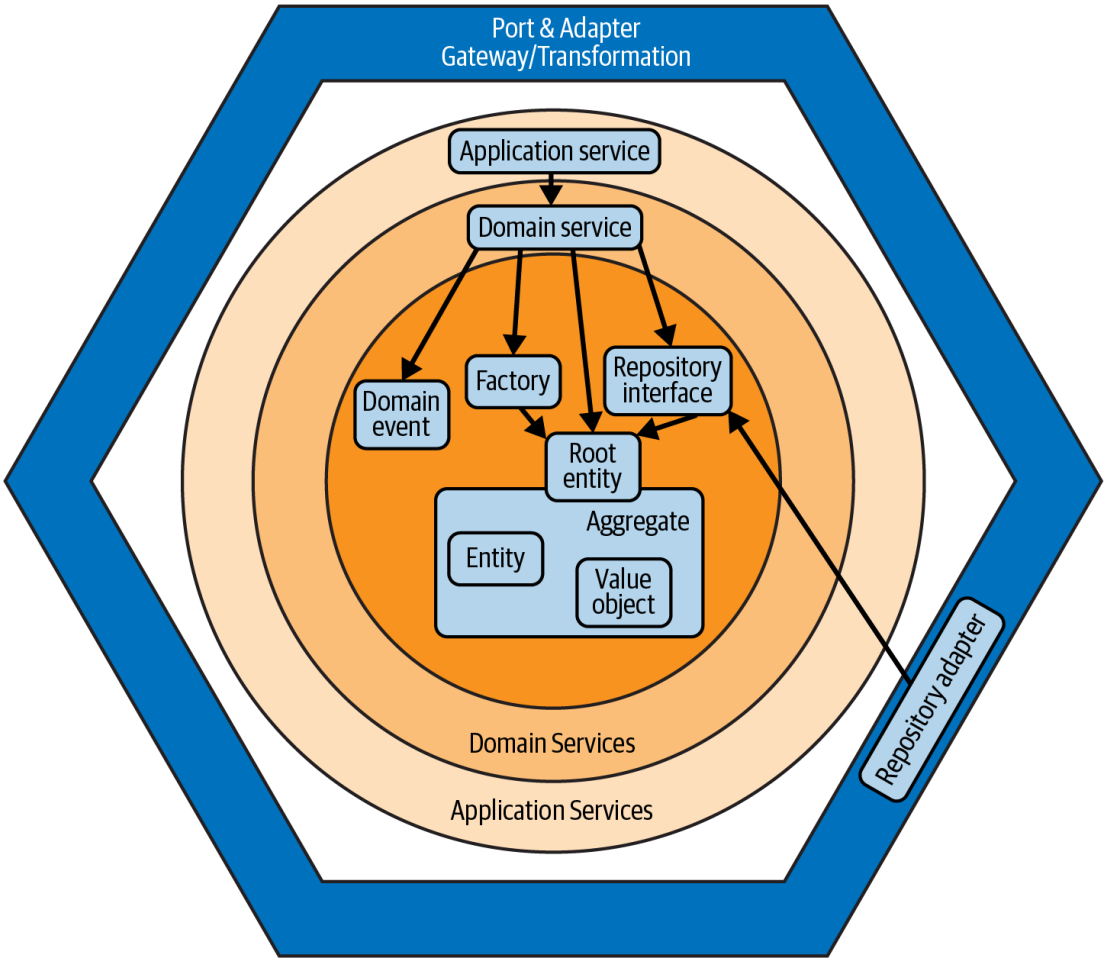


Figure 5-18. Hexagon with DDD building blocks.

To understand this construct in a bit more detail, we omit the rings for the domain and the application services in Figure 5-19 and focus only on the connection of the repository interface to various adapters and their implementation.

Figure 5-19 shows four repository adapters and implementations that are connected to the repository interface:

- An adapter for output to a CSV file
- An in-memory implementation for the repository
- An adapter for an Oracle database
- An adapter for a REST API

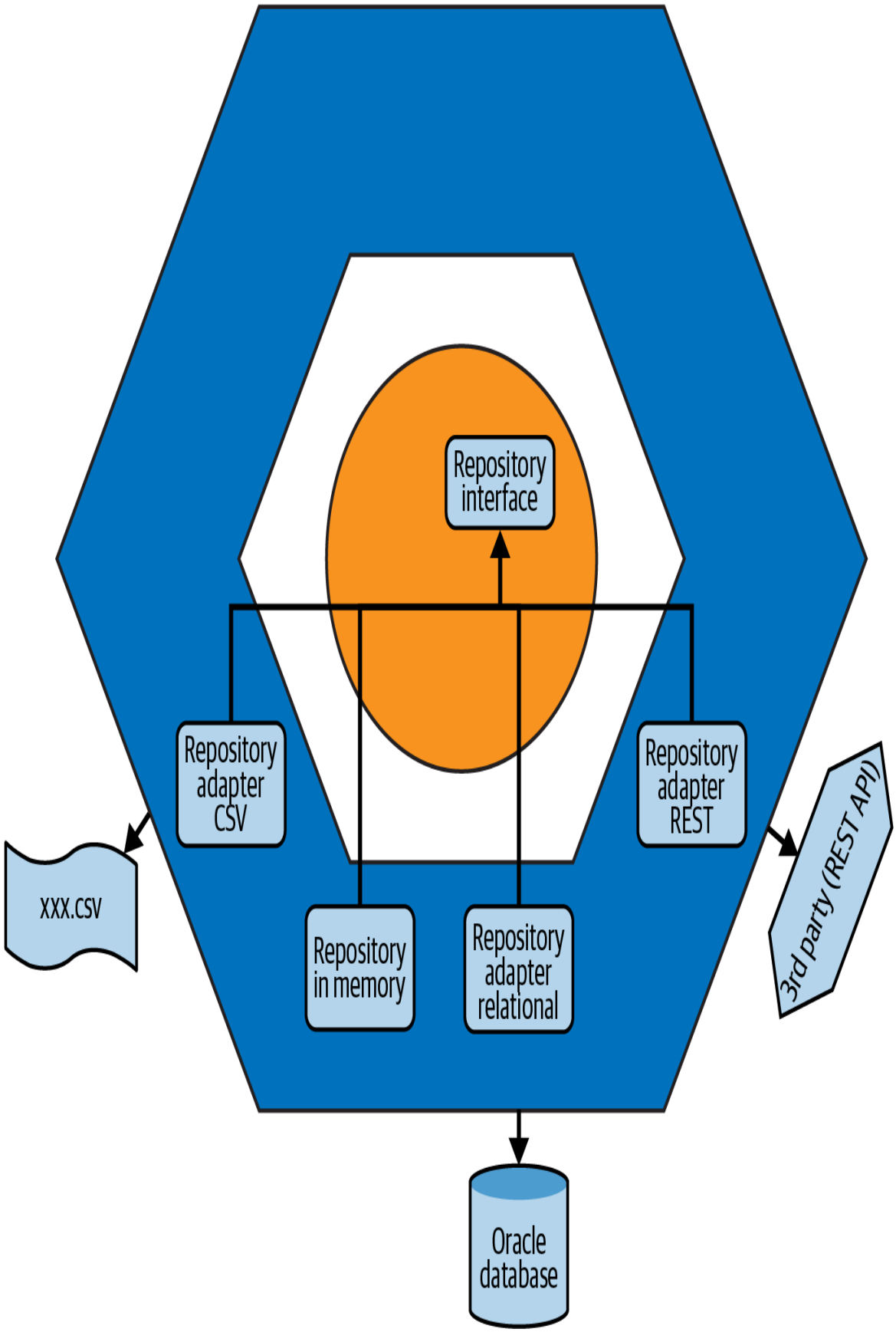

Repository interface
Repository adapter CSV
Repository adapter REST
XXX.CSV
Repository in memory
Repository adapter relational
3rd party (REST API)
Oracle database

Figure 5-19. Hexagon with repository and adapters.

Certainly, in a real system you will usually not find so many different adapters for a repository, but to illustrate the construction we have shown many variants. The in-memory implementation serves as a reference implementation for the repository interface and is suitable as a mock object for unit tests. It has no persistence and thus offers significant performance advantages during development.

Like DDD, the Hexagonal, Onion, and Clean Architecture aim to make the application core independent of the technical source code and to describe mechanisms for how these two parts should work together. As this book moves forward, the two parts will serve as an architectural template for the structure of a software system developed or restructured according to DDD.

Software Architecture Hamburger

Henning's opinion is that the inside-outside of the Hexagonal Architecture and similar is a great advancement, but that there is also a need for the above-below of layered architecture. Also, UI and database are represented together on the outermost ring in the Onion architecture, etc., while Henning thinks they should not have any dependency on each other at all.

Henning therefore considers the four layers of DDD to be the *Software Architecture Hamburger*[4] (roughly represented in Figure 5-20).

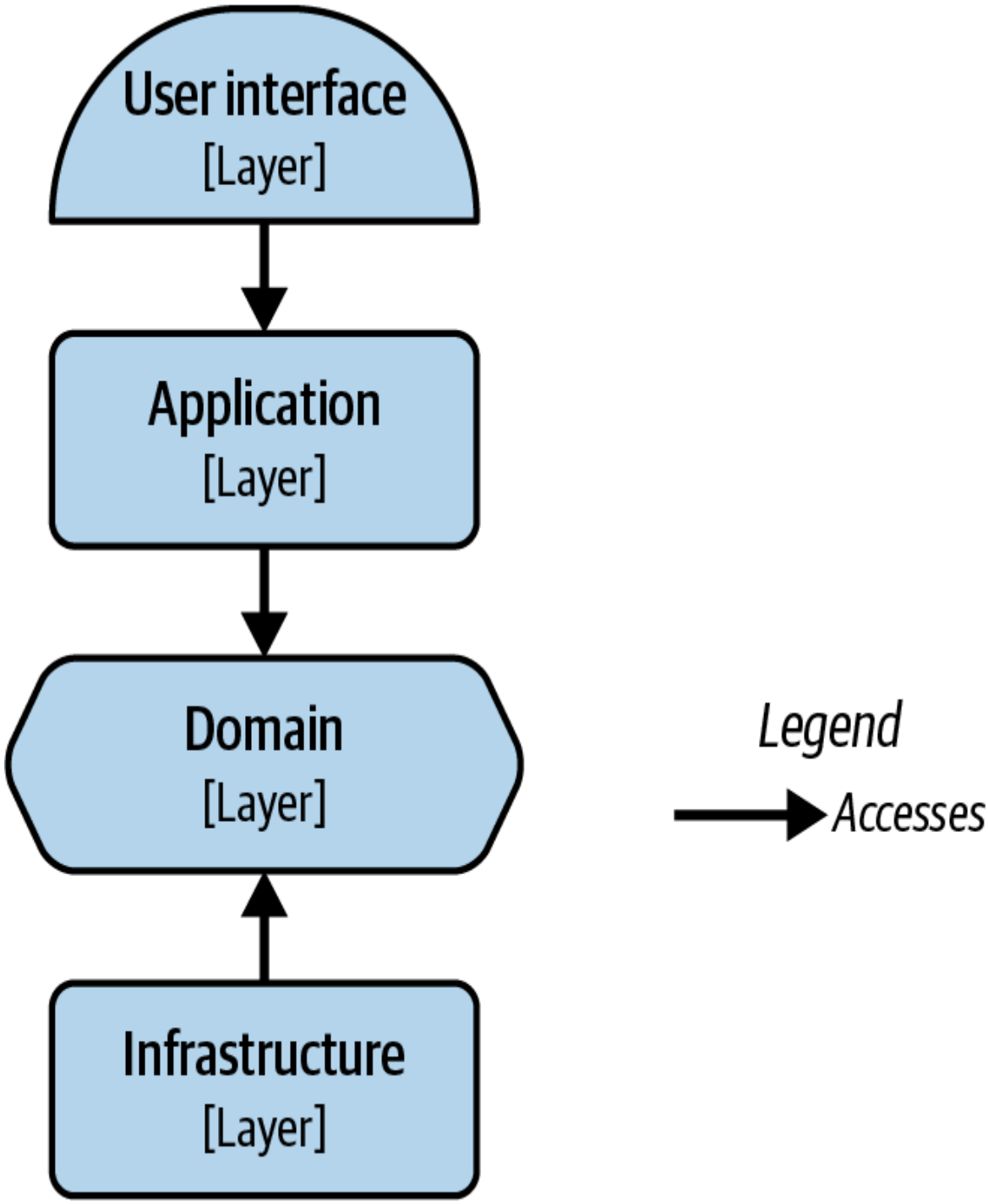


Figure 5-20. Architecture Hamburger—coarse-grained.

The Software Architecture Hamburger has the following properties:

- The most important thing is the patty, which is the domain core around which everything else is built. All dependencies point in its direction.

- The domain core can be garnished with a variety of infrastructure or UI technologies; it always remains the same regardless.
- Many, but not all, systems have an application layer, just as many, but not all, burgers are cheeseburgers.
- Within the individual layers, their own rules apply, which are described in more detail by pattern languages.
- The layers build on each other and everything is formed around the core. So all three principles are here: above-below, inside-out, and architectural rules of pattern languages.

To make the architecture style complete, we need the ingredients for the individual layers, namely the pattern languages:

- In the domain layer, the tactical DDD building blocks are used.
- The UI layer uses Model-View-Controller (MVC) or a variant of it, such as Model-View-Presenter (MVP) or Model-View-ViewModel (MVVM).
- In the application layer there are application services or use cases as orchestrators of the elements of the domain layer.

If you zoom in on the fine-grained layer and add the allowed relations, you get Figure 5-21. This diagram shows again: all dependencies point inwards.

Such an elaborate architectural style is not without effort. It is typically only worthwhile in a core domain.

To avoid going beyond the scope of this book, we will stop here with a short overview of the Software Architecture Hamburger. Maybe it will become a book of its own one day.

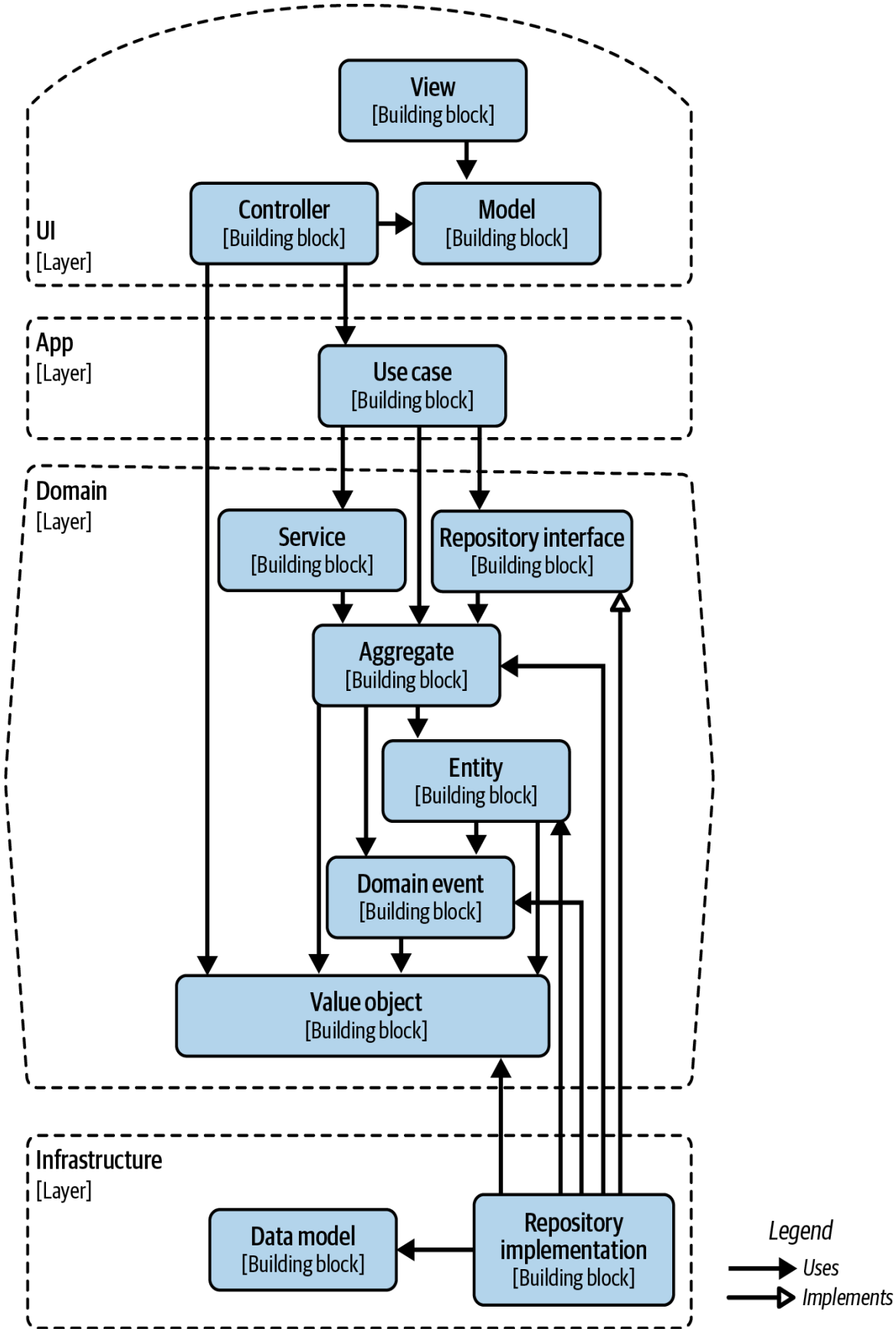

View
[Building block]
Controller
[Building block]
Model
[Building block]
UI
[Layer]
App
[Layer]
Use case
[Building block]
Domain
[Layer]
Service
[Building block]
Repository interface
[Building block]
Aggregate
[Building block]
Entity
[Building block]
Domain event
[Building block]
Value object
[Building block]
Infrastructure
[Layer]
Data model
[Building block]
Repository implementation
[Building block]
Legend
Uses
Implements

Figure 5-21. Architecture Hamburger—fine-grained.

Service-Oriented Architectures

In many larger enterprises, we regularly encounter service-oriented architectures, and we are always asked how this should go hand in hand with microservices and DDD.

Basically, an SOA is an enterprise-wide IT architecture with services as the central concept. The services each implement or support a business function for the entire architecture.

CASE STUDY “BANK”

IN THE BANK: SOA

Let's revisit the bank case study introduced in Chapter 4. Going a bit further from the EventStorming there, in Figure 5-22, you can see the services of the bank which are connected to an enterprise service bus (ESB).

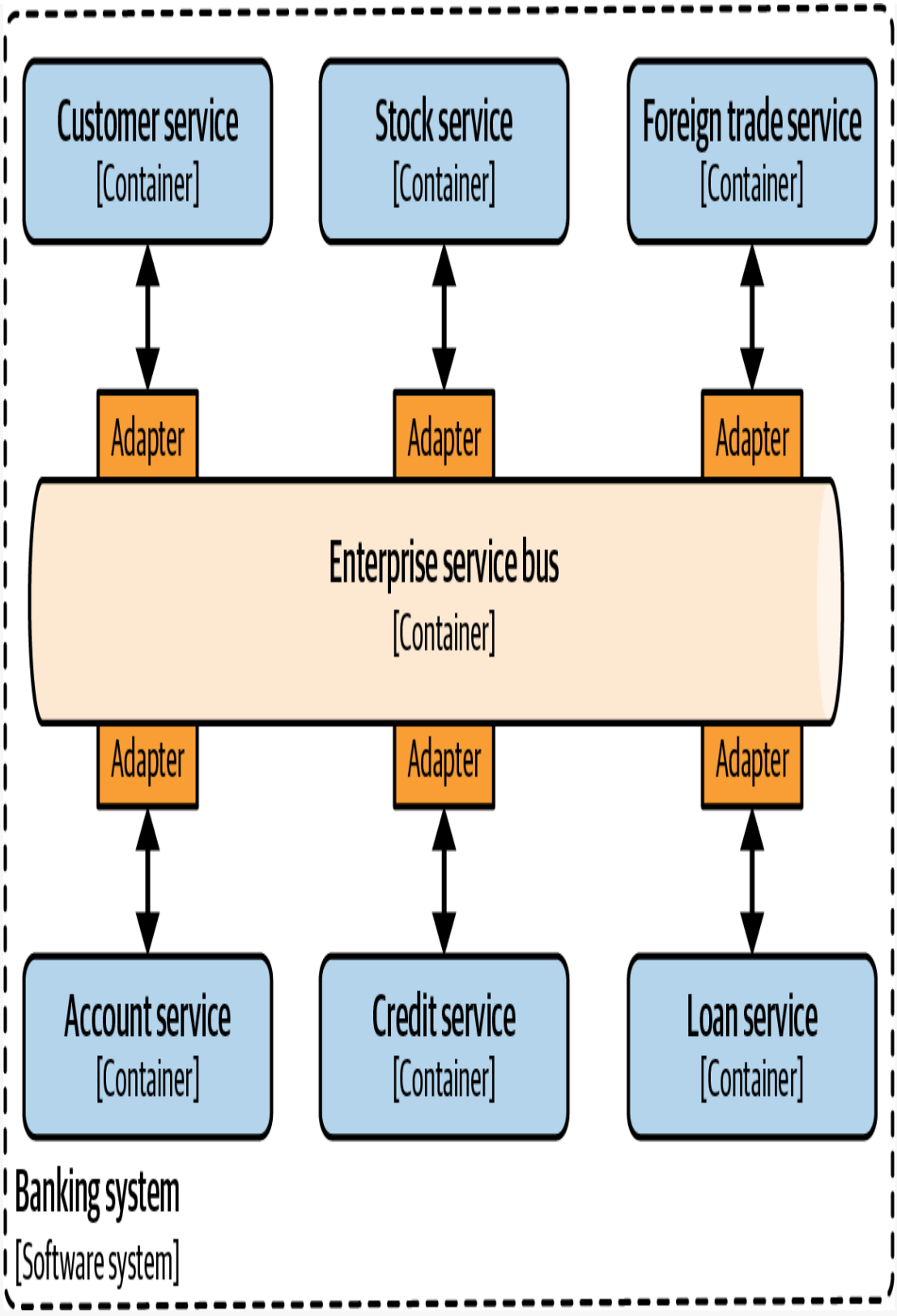


Figure 5-22. Service-oriented architecture with ESB.

Applications built on top of the ESB can use messages to make requests to the various services, and the applications can then further process and display the results.

> **WARNING**
>
> Unfortunately, most of the SOAs that we see follow the entity-service antipattern, where each service is designed around one of the central entities. This typically leads to gigantic services.

Such a misguided service-oriented architecture with its central services has the disadvantage that all development teams that require a change to one of the services in order to implement their extended functionality can no longer work independently. Each dependent team must wait for the implementation in the corresponding service and possibly adapt their own ideas because the team responsible for the service finds that the desired adaptations should not be implemented in this way. This means that the teams based on the SOA lose their individual effectiveness. In this respect, a service-oriented architecture is a long way from Domain-Driven Design, where the focus is on self-contained modules that can be worked on independently by a team.

Nevertheless, every now and then we meet teams that—even though they want to build something much smaller than an SOA—run into a SOA-like trap. These teams believe that they should have one microservice per aggregate because, after all, an aggregate is a transactional wrapper for a tree of entities. In our view, this admittedly seductively simple rule is not necessarily in the spirit of Domain-Driven Design.

CASE STUDY “BANK”

IN THE BANK: SOA MICROSERVICES

Let’s take a look at the example from the bank domain. An SOA-like solution that we encounter in some discussions of microservices looks like the one shown in Figure 5-23. Each aggregate is processed by its own service, which is in turn used by the applications built on top of it.

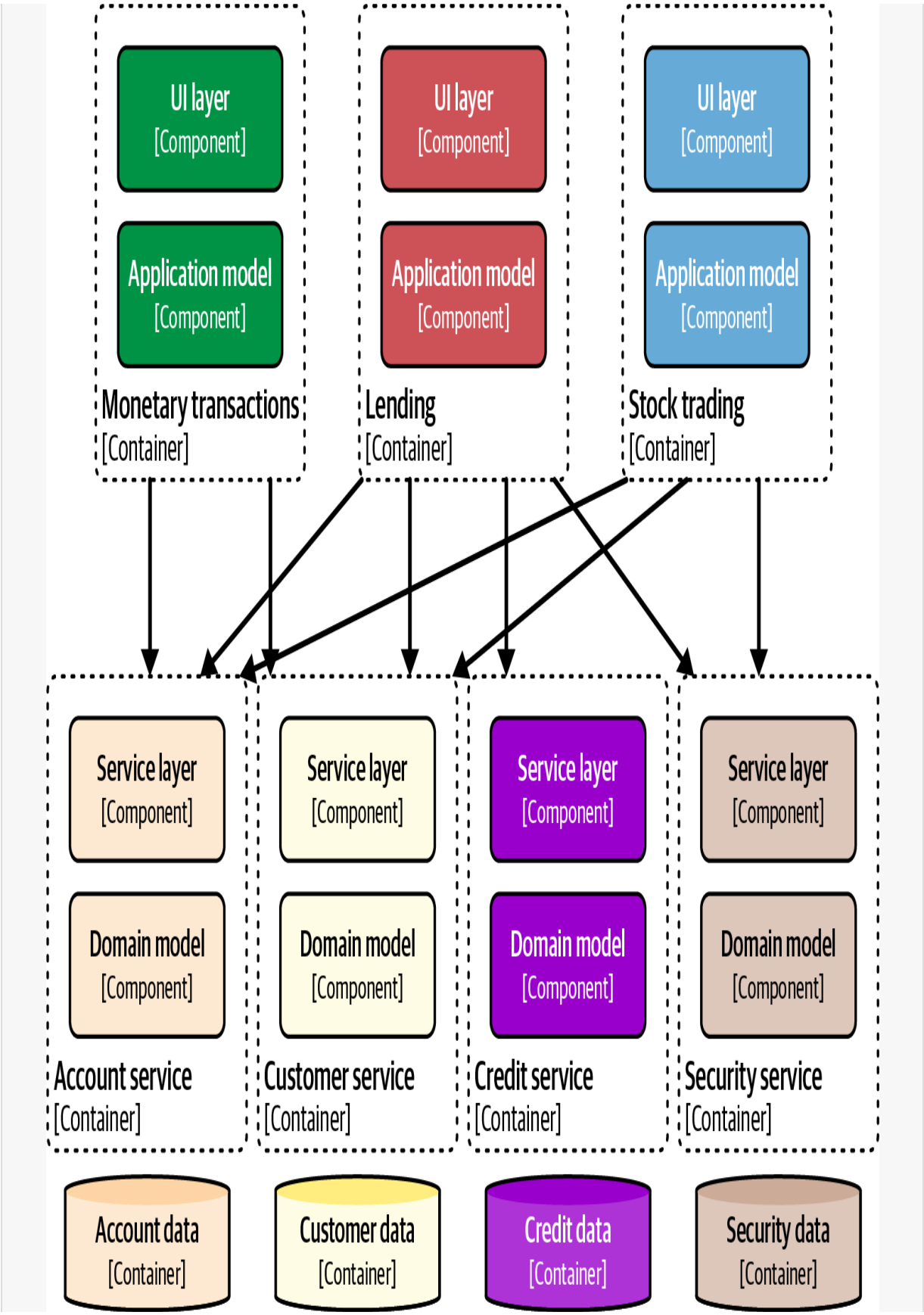

UI layer
[Component]
Application model
[Component]
Monetary transactions
[Container]
UI layer
[Component]
Application model
[Component]
Lending
[Container]
UI layer
[Component]
Application model
[Component]
Stock trading
[Container]
Service layer
[Component]
Domain model
[Component]
Account service
[Container]
Service layer
[Component]
Domain model
[Component]
Customer service
[Container]
Service layer
[Component]
Domain model
[Component]
Credit service
[Container]
Service layer
[Component]
Domain model
[Component]
Security service
[Container]
Account data
[Container]
Customer data
[Container]
Credit data
[Container]
Security data
[Container]

Figure 5-23. Not DDD: one microservice per aggregate.

At this point, we would like to make it very clear that this architecture with one microservice per aggregate is not what we consider a decomposition according to Domain-Driven Design. Domain-Driven Design guides practitioners toward implementing only a part of an overall business process, such as a single business capability, per bounded context. If distribution is required, each bounded context is deployed within a single microservice—with all the aggregates and entities that are needed for the given context. An implementation of this idea can be seen in Self-contained Systems, which are described in the next section.

At the same time, it is presumptuous to expect or demand that all mainframe systems, in which central data management is ensured in many companies, and which are addressable as SOA services, be converted to microservices architectures. In such cases, we often encounter a mixture of SOA and microservices architecture, which can work quite well if the microservices are granted their own data management. More appropriately, you can mix microservices, which are decomposed according to DDD, on top of an SOA. Unfortunately, this in itself does not transform the service-oriented architecture into an architecture of DDD-based services.

Self-Contained Systems

A concept that receives somewhat less attention than microservices, but is highly valued by us, is Self-contained Systems (SCS) (*https://scs-architecture.org*). SCS was developed because the term "microservice" is so fuzzily defined, and the "micro" in it sounds like something small. SCS is an architecture that focuses on dividing functionality into many independent systems so that the complete system consists of a collaboration of many smaller software systems. Communication between SCSs is asynchronous wherever possible. This decouples the systems, reduces the impact of errors, and thus supports autonomy.

From IT Landscapes to SOA and Back Again

When a system is decomposed in the SCS or microservices way, the result is a structure that is surprisingly similar to an IT landscape from the early 2000s. Different systems (possibly also from different manufacturers) are connected via interfaces but continue to operate autonomously on their own database. In Figure 5-24, the customer exists in each of the four systems shown and customer data is exchanged between the systems via interfaces.

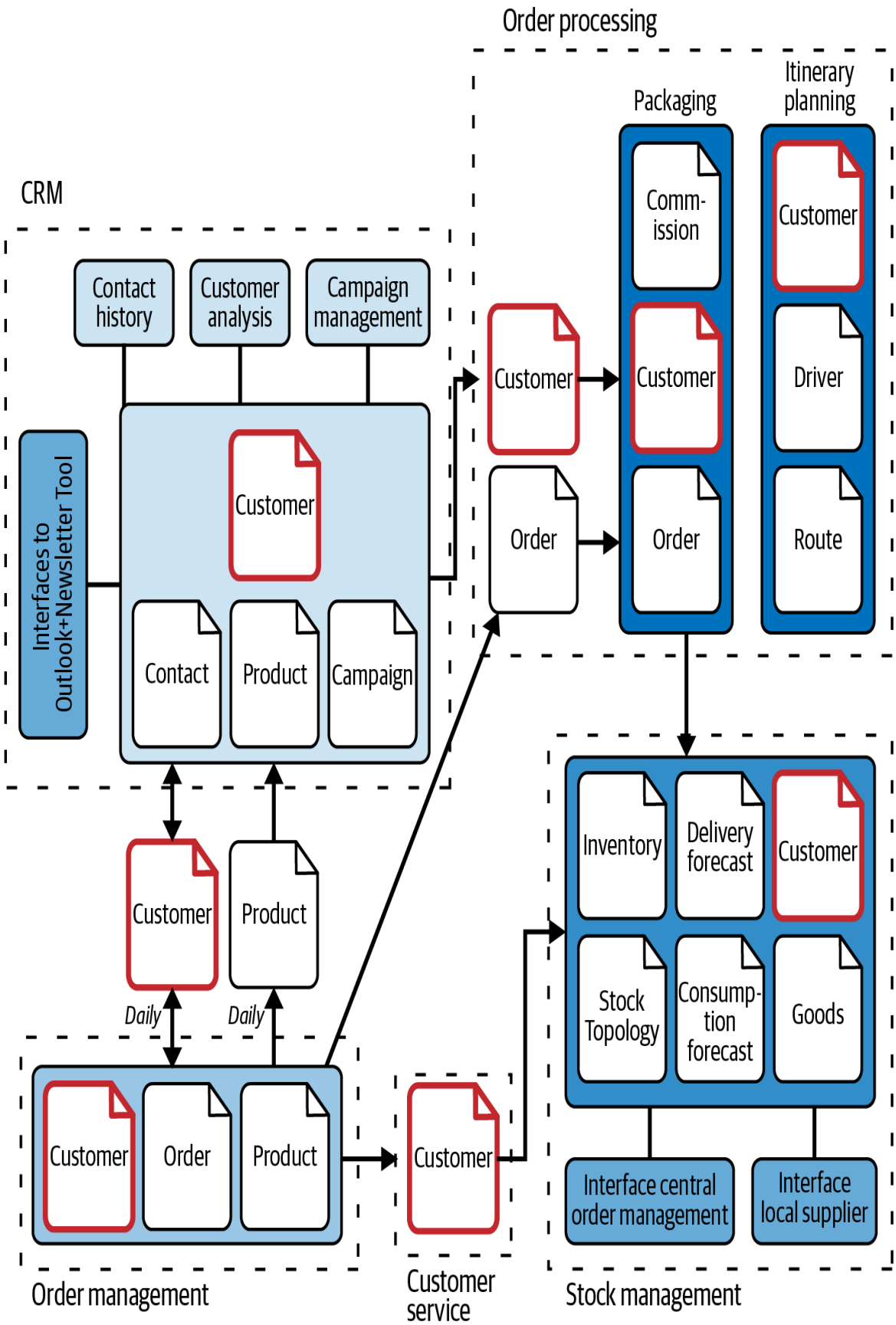

Order processing
Packaging
Itinerary planning
CRM
Contact history
Customer analysis
Campaign management
Comm-ission
Customer
Customer
Customer
Driver
Interfaces to Outlook+Newsletter Tool
Customer
Order
Order
Route
Contact
Product
Campaign
Inventory
Delivery forecast
Customer
Customer
Product
Stock Topology
Consump-tion forecast
Goods
Daily
Daily
Customer
Order
Product
Customer
Interface central order management
Interface local supplier
Order management
Customer service
Stock management

Figure 5-24. Several independent systems in one IT landscape connected via interfaces.

Amusingly, in the early 2000s, people wanted to counteract the distribution of customer data from Figure 5-24 across different systems by pursuing service-oriented architecture (SOA) as an architectural style. People wanted to have customer data in only one place and not have to constantly ensure that it was reconciled. The result was IT landscapes in which there is a central customer service that all other systems use via a service bus. Figure 5-25 illustrates how the IT landscape from Figure 5-24 was transformed into an SOA by pulling together the customer data into one customer service.

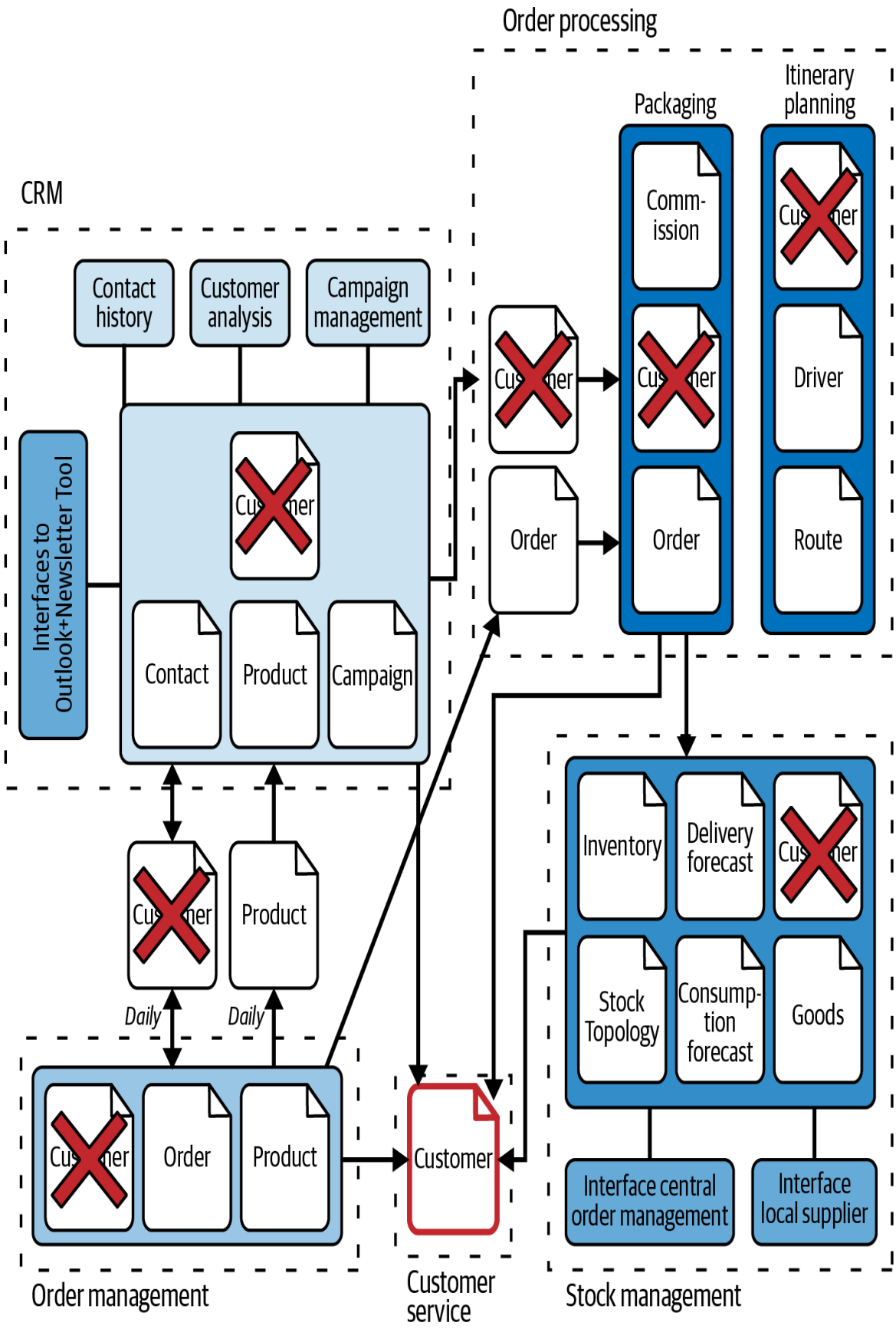

Order processing
Packaging
Itinerary planning
CRM
Contact history
Customer analysis
Campaign management
Comm-ission
Driver
Interfaces to Outlook+Newsletter Tool
Order
Order
Route
Contact
Product
Campaign
Product
Inventory
Delivery forecast
Stock Topology
Consump-tion forecast
Goods
Daily
Daily
Order
Product
Customer
Interface central order management
Interface local supplier
Order management
Customer service
Stock management

Figure 5-25. Conversion of an IT landscape to an SOA.

This design is known as a *canonical model*. A canonical model is intended to be a one-size-fits-all service. Yet, in practice, it inevitably becomes a one-size-fits-none conglomeration of generalities. Any one service that requires a given concept, such as Customer in Figure 5-25, must understand the parts that it doesn't need. The perilous journey can be described as 30 years of shifting from single systems with their own database, to IT landscapes made up of single systems with their own database that learned about data changes via interfaces, to service-oriented architectures and ever-growing applications built on top of SOA. Today, the industry is moving in the other direction again—combining the advantages of the two approaches—toward IT landscapes made up of individual subsystems (bounded contexts with their own database. Yet today they are called microservices, and perhaps more accurately Self-contained Systems. These are developed by independent teams, but with a common idea of forming an overall whole system.

The Modularity Maturity Index: Where Do You Stand?

To create a way to evaluate legacy systems and new developments, and to find how well the source code is prepared for a domain-oriented decomposition, Carola and her team have developed the Modularity Maturity Index (MMI) in recent years (see [Lilienthal 2019, chap. 4.5] and [Ciceri et al. 2022, chap. 4]).

The MMI evaluates a software system—whether monolith or microservice—on the basis of various criteria in the categories of modularity, hierarchy, and pattern consistency, and maps the evaluation on a scale of 1 to 10. Let's take a look at the three categories that make up the MMI in the next sections.

Modularity in the MMI

In the *modularity* category, the MMI expresses whether the program units in a software system have high cohesion internally and loose coupling to other program units at all levels. This is just as we required at the beginning of this chapter in "Cohesion and Coupling".

Modularity can only be judged qualitatively; i.e., a human must make this judgment. This qualitative judgment is supported by various measured values:

Well-balanced proportions

> Program units that are on one level, i.e., the layers, the technical modules, the packages, the classes, or the methods, should have well-balanced proportions. Well-balanced proportions here mean that program units on the same level should be similar in size (lines of code) and have a similar number of subunits (sublayers, submodules, packages, classes, or methods). It is worth examining the very large program units to see if they are eligible for decomposition.

Responsibilities of the program units

> If the program units of a system are modular, you should be able to answer this question for each program unit: what is its job? It is important that the program unit has exactly one responsibility and not several. Good clues to unclear responsibilities are the names of the program units—the name of a program unit should describe its job. If the name is vague, it should be revised.

Cohesion through coupling

> Blocks should contain subblocks that belong together. For example, a class should contain methods that together make a whole. The same is true for larger program units such as packages, components, modules, and layers. If the subblocks have more to do with other blocks than with their "sisters and brothers," then the question arises whether they belong in another block.

Mapping of the target architecture to the actual architecture

In a software system you will find a structuring by layers, packages, and maybe modules, projects, or deployment units. This implemented structure—the actual architecture—usually does not correspond to the plan that the architects once had for this system—the target architecture. How much and in what way the target and the actual deviate from each other also enters into the MMI.

Overall, modularity accounts for 45% of the result in the MMI.

Hierarchy in the MMI

In the *hierarchy* category, the MMI expresses how closely the system resembles a big ball of mud. A big ball of mud consists of cyclic structures at all levels that we have to break up for a modular system.

Hierarchy can be measured quantitatively through metrics. The following metrics are used in the MMI:

Percentage of classes and packages in cycles

For a system with less than 10% of classes in cycles, the chaos at the class level is not high. So we are looking at a system with a hierarchical design at the class level. If the result is between 10% and 20%, we can expect some work to untangle the cycles. If a system has more than 30% of its classes in cycles, then the system is already severely eroded or never had a meaningful class-level design.

For package cycles, 20% is a good value; anything over 40% is a serious problem.

Number of classes and packages per cycle

Systems with class dependency cycles in the range of 2 to 5 classes are in good condition. In a big ball of mud, on the other hand, you will find cycles in the range of 300, 400, or 500 classes. Untangling these big monsters means a lot of work. At the package level, there are similar orders of magnitude to deal with.

Percentage of violations in technical layering

How well is technical layering adhered to?

Percent proportion of violations in relationships between bounded contexts

How well is the planned coupling between bounded contexts adhered to?

Overall, hierarchy accounts for 30% of the result in the MMI.

Pattern Consistency in the MMI

In the *pattern consistency* category, the MMI expresses whether the architecture and design patterns planned for the system are followed. Patterns specify rules for the responsibilities of classes or packages, thus increasing cohesion. They also use rules to restrict the relationships that are allowed in a pattern, which reduces coupling.

Pattern consistency can only be judged qualitatively; i.e., a human must perform this judgment. This qualitative judgment is supported by various measured values:

Percentage mapping of the source code to the pattern

The higher the proportion of source code that follows a pattern, the better the results for cohesion and coupling. Conversely, the part of the source code that cannot be assigned to a pattern usually causes the most difficulties.

Explicit mapping of patterns

If the patterns are made visible in the source code through name additions, inheritance, or annotations, their use and consistency are much more likely.

Overall, pattern consistency accounts for 25% of the score in the MMI.

Calculation of the MMI

Modularity, hierarchy, and pattern consistency are each included in the overall calculation of the MMI with an individual percentage share. Modularity has the strongest influence on the MMI at 45%, because modularity is the basis for hierarchy and pattern consistency. Hierarchization accounts for 30% and pattern consistency for 25%. We give pattern consistency a slightly lower weighting than hierarchization because patterns have not yet arrived in all areas of software development as a useful means of structuring architectures. In the case of modularity and hierarchization, there are two to three subprinciples below the basic principle, each of which accounts for a percentage of the basic principle. The values are finally calculated using the instructions in Table 5-1.

Table 5-1. Criteria of the MMI

No.	Criterion	Ratio
1.	*Modularity*	*45%*
1.1.	*Technical and domain-oriented modularization*	*25%*
1.1.1.	Allocation of the source code to the domain modules as a % of the total source code	
1.1.2.	Allocation of the source code to the technical layers as a % of the total source code	
1.1.3.	Size ratios of the domain-oriented subsystems $((LoCmax/LoCmin)/number)$	
1.1.4.	Size ratios of the technical layers $((LoCmax/LoCmin)/number)$	
1.1.5.	Domain-oriented modules, technical layers, packages, and classes have clear responsibilities	
1.1.6.	Mapping of technical layering and domain-oriented modules through packages or projects	

No.	Criterion	Ratio
1.2.	*Internal interfaces*	*10%*
1.2.1	At least one type of module (domain-oriented/technical) has internal interfaces (% violations)	
1.2.2.	Mapping of internal interfaces through packages or projects	
1.3.	*Size ratios*	*10%*
1.3.1.	% of the source code in large classes	
1.3.2.	% of the source code in methods	
1.3.3.	% of classes in large packages	
1.3.4.	% of the methods of the system with a high cyclomatic complexity	
2.	*Hierarchy*	*30%*
2.1	*Technical layering and domain-oriented structure*	*15%*
2.1.1	Number of architecture violations in the technical layering (%)	

No.	Criterion	Ratio
2.1.2	Number of architecture violations in the domain-oriented structure (%)	
2.2.	*Class and package cycles*	*15%*
2.2.1	Number of classes in cycles (%)	
2.2.2.	Number of packages in cycles (%)	
2.2.3.	Number of classes per cycle	
2.2.4.	Number of packages per cycle	
3.	*Pattern consistency*	*25%*
3.1.	Assignment of the source code in % to patterns	
3.2.	Relationships of the patterns are cycle-free (% violations)	
3.3.	Explicit expression of patterns (via class names, inheritance, or annotations)	
3.4.	Separation of the technical and domain-oriented source code	

The MMI is calculated by determining a number between 0 and 10 for each criterion using Table 5-2. The resulting numbers per section are added together and divided by the number of criteria in question. The result is recorded in the MMI with the percentage share of the respective principle, so that a number between 0 and 10 can be determined.

Table 5-2. Detailed calculation instructions for the MMI

No.	0	1	2	3
1.1.1	≤54%	>54%	>58%	>62%
1.1.2	≤75%	>75%	>77.5%	>80%
1.1.3	≥7.5	<7.5	<5	<3.5
1.1.4	≥16.5	<16.5	<11	<7.5
1.1.5	no			
1.1.6	no			
1.2.1	≥6.5%	<6.5%	<4%	<2.5%
1.2.2	no			
1.3.1	≥23%	<23%	<18%	<13.5
1.3.2	≥23%	<23%	<18%	<13.5
1.3.3	≥23%	<23%	<18%	<13.5
1.3.4	≥3.6%	<3.6%	<2.6%	<1.9%
2.1.1	≥6.5%	<6.5%	<4%	<2.5%

No.	0	1	2	3
2.1.2	≥14%	<14%	<9.6%	<6.5%
2.2.1	≥25%	<25%	<22.5%	<20%
2.2.2	≥50%	<50%	<45%	<40%
2.2.3	≥106	<106	<82	<62
2.2.4	≥37	<37	<30	<24
3.1	≤54.5%	>54.5%	>59%	>63.5
3.2	≥7.5%	<7.5%	<5%	<3.5%
3.3	no			
3.4	no			

Some criteria, such as large classes or large packages, can be determined with metric tools, and their comparability depends only on the comparability of the metric tools used. Other criteria, such as the question of whether the modules, layers, etc., have clear responsibilities (1.1.5), cannot be measured but are at the discretion of the reviewer. To ensure the greatest possible comparability here, we always carry out the reviews in pairs and then discuss the results in a larger group of architecture reviewers.

To be able to evaluate these unmeasurable criteria, our reviewers rely on discussions with the developers and architects of the respective system. We conduct this discussion in on-site or remote workshops, where we discuss

the architecture of the system from different architectural perspectives using an analysis tool.

If you cannot evaluate the MMI yourself, you can alternatively use the questionnaires at the end of each chapter in Part II (technical, tactical, and team organizational Domain-Driven Transformation) to determine where you stand.

Summary

In this chapter, we looked at the big ball of mud, monoliths, and microservices, starting with cohesion and coupling. We looked at how monoliths become a big ball of mud.

The various architectural styles that we repeatedly encounter in connection with Domain-Driven Design—Hexagonal, Clean, Onion, and service-oriented architectures as well as Self-contained Systems—were categorized and related to DDD.

At the end of the chapter, we learned about the Modularity Maturity Index (MMI) as a way to determine the degree of entangledness or, conversely, the degree of modularity of a piece of software.

---

1 When this pattern was first described, Alistair Cockburn used the name Hexagonal Architecture. Later, he came up with the better and more accurate name Ports & Adapters. When he published his book on the topic, the old name was used as the title again, since it is so widely used in the industry. For more on the history, see that very book [Cockburn and Garrido de Paz 2024].

2 We lean on a diagram by Gernot Starke, first published in [Starke 2024].

3 Back in the day, when X was still called Twitter: *https://x.com/simonbrown/status/847339104874381312*.

4 Carola doesn't like food metaphors in architecture.

Chapter 6. How to Approach the Transformation

What is the right course of action when approaching a legacy system? That is the topic of this chapter.

From a broad perspective, there are basically two options: either replace the legacy system completely or retain it by "merely" reshaping/refactoring it. Also, for replacement there are two variants: big bang replacement and step-by-step replacement.

On a more detailed level, different approaches can be used for different subsystems of a system.

We'll stay at the broad level for the moment; the options are the following:

Replacement

Big bang

The new system is built from scratch on a greenfield site while the legacy system continues to operate. At some point in time, the switch is flipped from the legacy system to the new system.

Step-by-step

The new system is developed piece by piece and each piece is put into production when possible. At the same time, the corresponding functionality in the legacy system is shut down.

Reshaping

The big ball of mud is reformed with refactorings and remains in operation all the time.

Considering the advantages and disadvantages of the different types of transformation, you and other stakeholders will be in a good position to make the best decisions for your unique situation. To be sure, none of these options is easy or even simple to accomplish. There is, however, a greater or lesser degree of complexity from one to another.

Replacement—Big Bang

This first variant of replacement is shown in Figure 6-1. Users continue with the old system (and only the old system) until the new system is completely built (arrows 1 and 2). Then "you *just* flip the switch" (arrow 3).

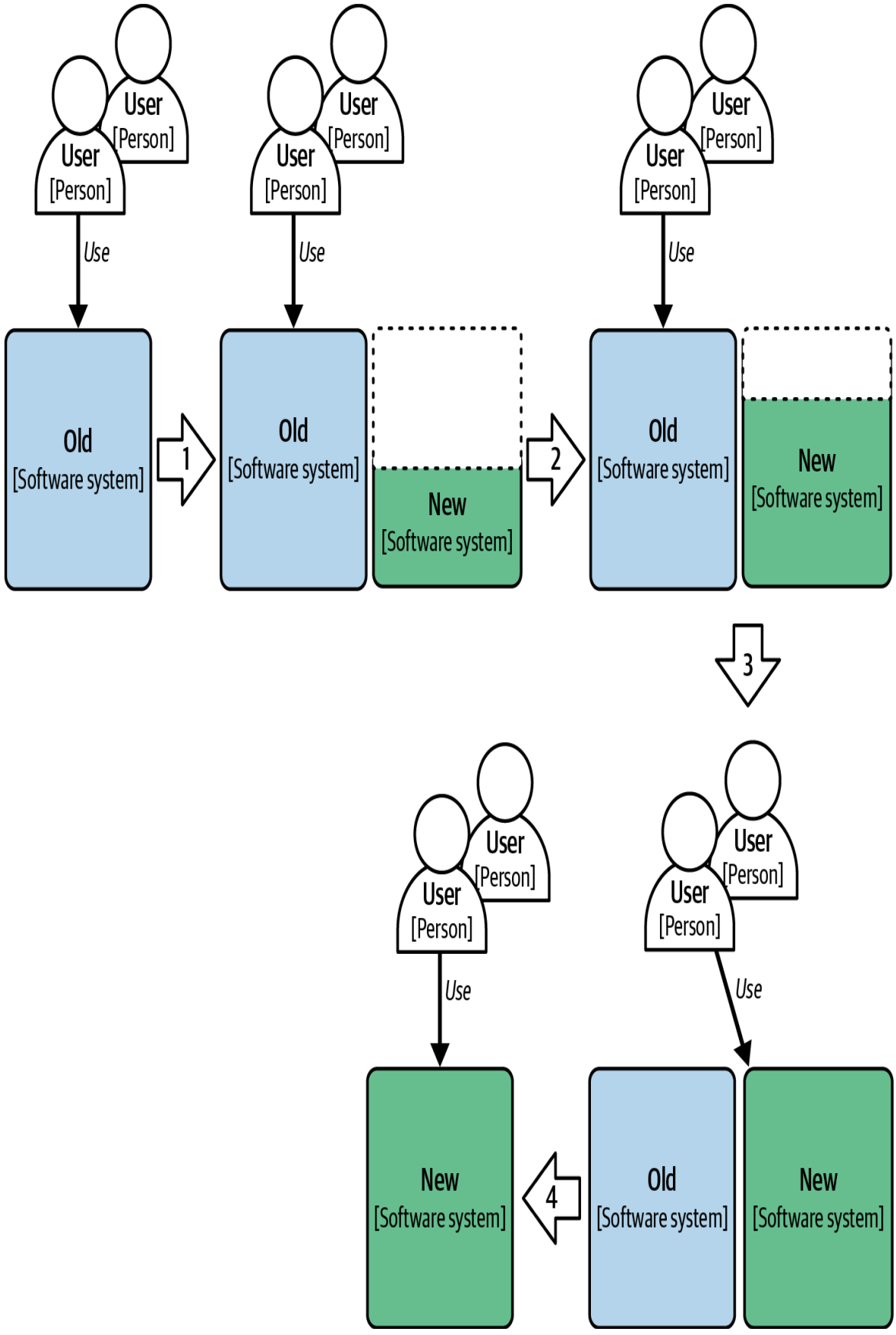

User
[Person]
User
[Person]
Use
Old
[Software system]
1
Old
[Software system]
New
[Software system]
2
Old
[Software system]
New
[Software system]
3
User
[Person]
User
[Person]
Use
New
[Software system]
4
Old
[Software system]
New
[Software system]

Figure 6-1. Steps of a big bang replacement.

The procedure is called *big bang replacement* because, like with the big bang,[1] a whole new world is suddenly there. Unfortunately, the name is also apt because there will be a big bang. What sounds good in theory, namely, to raise up the new system next to the old one, does not work in practice for various reasons:

Unknown unknowns (overlooked functionality)

Analyzing the functionalities that a legacy system offers to perform different tasks is usually costly. This is because a great deal of functionality has been criss-crossed into the system over the years by a wide variety of teams. Often, one only has the standard business processes in mind and overlooks the fact that there are many specialized and often hidden business processes as well as the manifold use of interfaces. Typically, business processes that are rarely executed are forgotten, e.g., only at the turn of the year or employed by a few smaller departments. Besides, there are almost always users who use the system in a way that was never intended during development and expansion and that nobody knows about.

In addition, shadow IT has formed around most legacy systems over the years. This shadow IT, implemented by users themselves in Access databases or large Excel files, for example, is usually not managed by the IT organization and is therefore overlooked. The users supply the shadow IT they have created themselves with data from the existing export interfaces of the legacy system, which were actually created for a different purpose. Most of the time, the IT organization does not know about the special use of the interfaces.

Standstill on the one hand/moving target on the other hand

For the users, a feeling of standstill arises because, on the one hand, further development of new requirements that the users wish for in the old system is halted so that enough capacities are free for the development of the new system. On the other hand, the development of

a new system often takes longer than originally planned because more and more unknown unknowns become apparent over time and have to be additionally implemented.

Duplicate work

While a freeze of the old system might be an initial goal, later it is learned that at least some changes must be made. For example, changes to the legal regulations must be resolved or implemented in the old system while it is still in use. The new system must also be adapted to the changes in the legal framework. The same applies to critical bugs. This leads to duplication of work and further delays the new development.

Loss of know-how

Developers who continue to maintain the legacy system often feel left behind because they are not allowed to participate in the new development. Often, developers who are demotivated quit as a result, leading to overall staff shortages.

Late deployment

The new system is intended to be deployed in one fell swoop rather than incrementally, so the development team will get late feedback on its new development. Since a new development is almost never a one-to-one replica of the old system, there will be parts of the system where the development team would actually need early feedback. Because of the late feedback, bugs and misunderstandings are found late and the development subsequently suffers from all the problems of the waterfall model.

User frustration

Users came out of a big bang implementation disappointed and frustrated in the cases we know of. A complete redevelopment with big bang replacement results in development teams being much more

tempted to redo everything than if they developed the new system incrementally.

The users were familiar with the legacy system and knew its capabilities and weaknesses. They were able to implement their work quickly within this framework and knew what the result would look like. With the big bang, users lose this certainty for the whole functionality in one fell swoop.

In Figure 6-2, the life of a legacy system with its maintenance phases 1 and 2, as well as the big bang replacement with previous standstill due to new development, is schematically shown on a time axis.

In Figure 6-2, the dashed line shows that the business requirements for a software system continue to increase during its use, because maintenance and modernization create business value and thus corresponding requirements. As a result, significantly more functionality must be implemented in the new development than was necessary during the development of the legacy system. This difference is often underestimated, so that development takes longer than planned and the downtime leads to the new system being introduced when it is not ready. At that moment, when the organization members think, “We have to roll out now and this will probably work,” the unknown unknowns strike and a system that can do far too little is put into operation with a big bang rollout.

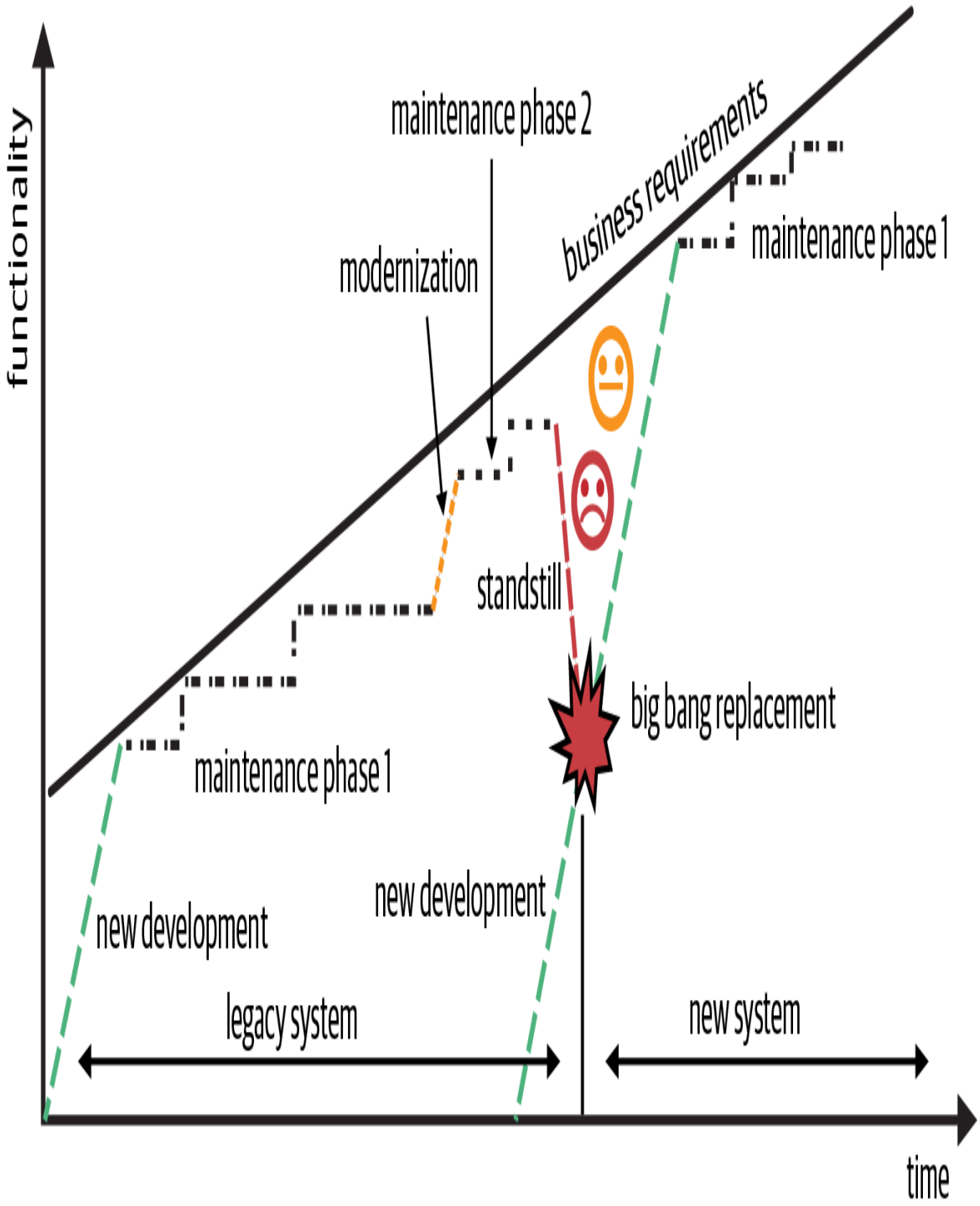


Figure 6-2. Life cycle of a software system.

From these remarks, you can readily understand why we are not big fans of a big bang rollout. Our impression is that we are not the only ones who think there are other (often better) ways for replacing legacy systems.

Replacement—Step-by-Step

The collective experience of our industry shows that the second option—replacing the legacy system step-by-step with a slowly growing new system—works better than the big bang (see Figure 6-3).

In this process, parts of the old system are removed step-by-step and placed next to the old system in a new guise. After a short time, users are directly confronted with the new system and work productively with it. When all functionality is available in the new system, the old system can be switched off (arrow 4). Martin Fowler gave this procedure a very figurative name: *Strangler Fig Application* (see [Fowler 2004b]). He chose this name because the old system is entwined by the new system and finally strangled, like the host tree by a strangler fig.

This type of modernization fits in with the Agile approach preferred in IT, where you learn from iteration to iteration and thus constantly evaluate and adjust the path you want to take based on the latest insights.

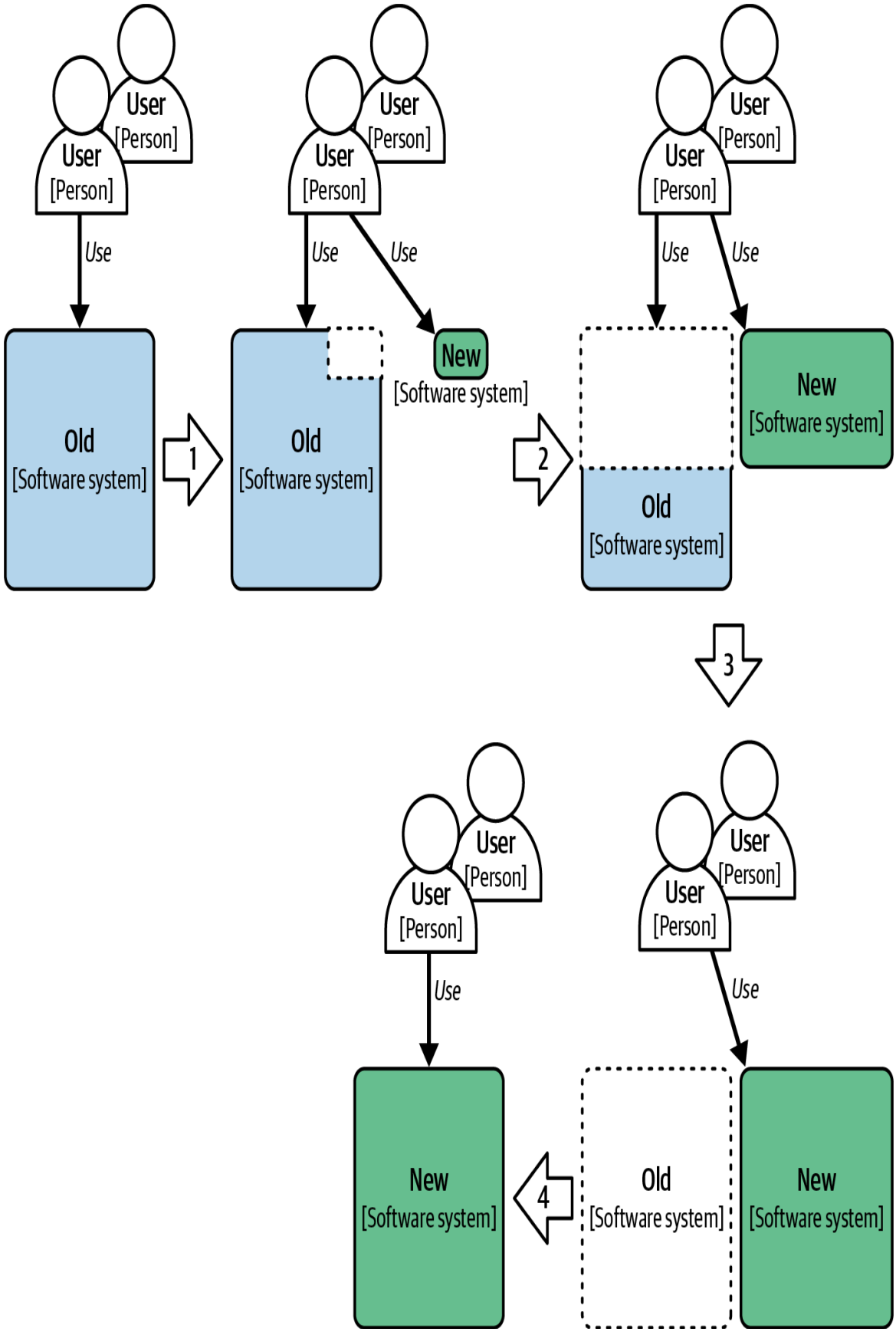

User
[Person]
User
[Person]
Use
Old
[Software system]
1
User
[Person]
User
[Person]
Use
Use
Old
[Software system]
New
[Software system]
2
User
[Person]
User
[Person]
Use
Use
New
[Software system]
Old
[Software system]
3
User
[Person]
User
[Person]
Use
New
[Software system]
4
Old
[Software system]
User
[Person]
User
[Person]
Use
New
[Software system]

Figure 6-3. Steps of a step-by-step replacement.

The reasons listed in the last section as to why big bang replacement often does not work turn into advantages here:

Unknown unknowns (overlooked functionality)

The analysis of the functionalities that are performed with a legacy system is only carried out for the part of the old software that is to be replaced. This reduces the complexity, and you learn a little more about the unknown unknowns with each partial replacement. When you start with the next section you want to replace, you have already identified some business processes and discovered some shadow IT, so you can expect fewer pitfalls. Step-by-step, the picture is completed.

Standstill on the one hand/moving target on the other hand

For the users, a feeling of standstill can also arise during the step-by-step replacement, because development capacities must of course also be withdrawn from the legacy system for this conversion. However, since the old system has to be touched again and again in order to integrate the partial replacements by the new system, the hope is quite justified that functional adjustments will also continue to be made to the legacy system.

Duplicate work

In step-by-step replacement, duplicate work arises from fixing critical bugs and changes to legal regulations only in the part of the old and the new system that is currently being replaced. The amount of duplication is therefore significantly less than in big bang replacement.

Loss of know-how

If, during step-by-step replacement, the teams are organized according to the respective subfunctionalities to be replaced, e.g., on the basis of bounded contexts, the same problem arises for the subsystems. The subsystem in the legacy system must continue to be maintained while

the new system part is developed. However, since the amount of duplication is less, the amount of duplicate work needed here will also be less. If you additionally set up a rotating system for maintenance in the development team for the subsystem currently being worked on, you can avert the loss of know-how.

Late deployment

Step-by-step replacement means that the respective finished parts of the new system are deployed quickly. The development team gets early feedback compared to the big bang. This often gives the impression that step-by-step replacement takes longer and is therefore more expensive than big bang replacement. But appearances are deceptive. In the case of big bang, significantly higher follow-up costs occur compared to step-by-step replacement, because the newly developed system requires rework in many places.

User frustration

Users are introduced to the new system, or parts of it, much sooner in a step-by-step replacement. This gives users the opportunity to slowly get used to the new system in smaller chunks than with big bang replacement. There might be new bugs and nuances in the replacement that are wrong or at least inconsistent that will make users unhappy. But the feeling of not knowing the system at all will only apply to parts of the software, so overall frustration should be less.

Reshaping

Many organizations assume that there is a lot of know-how hidden in their legacy system that they would like to save for the future. In this case, the legacy system is not replaced but is reshaped step-by-step through refactorings, and remains in operation the whole time (see Figure 6-4).

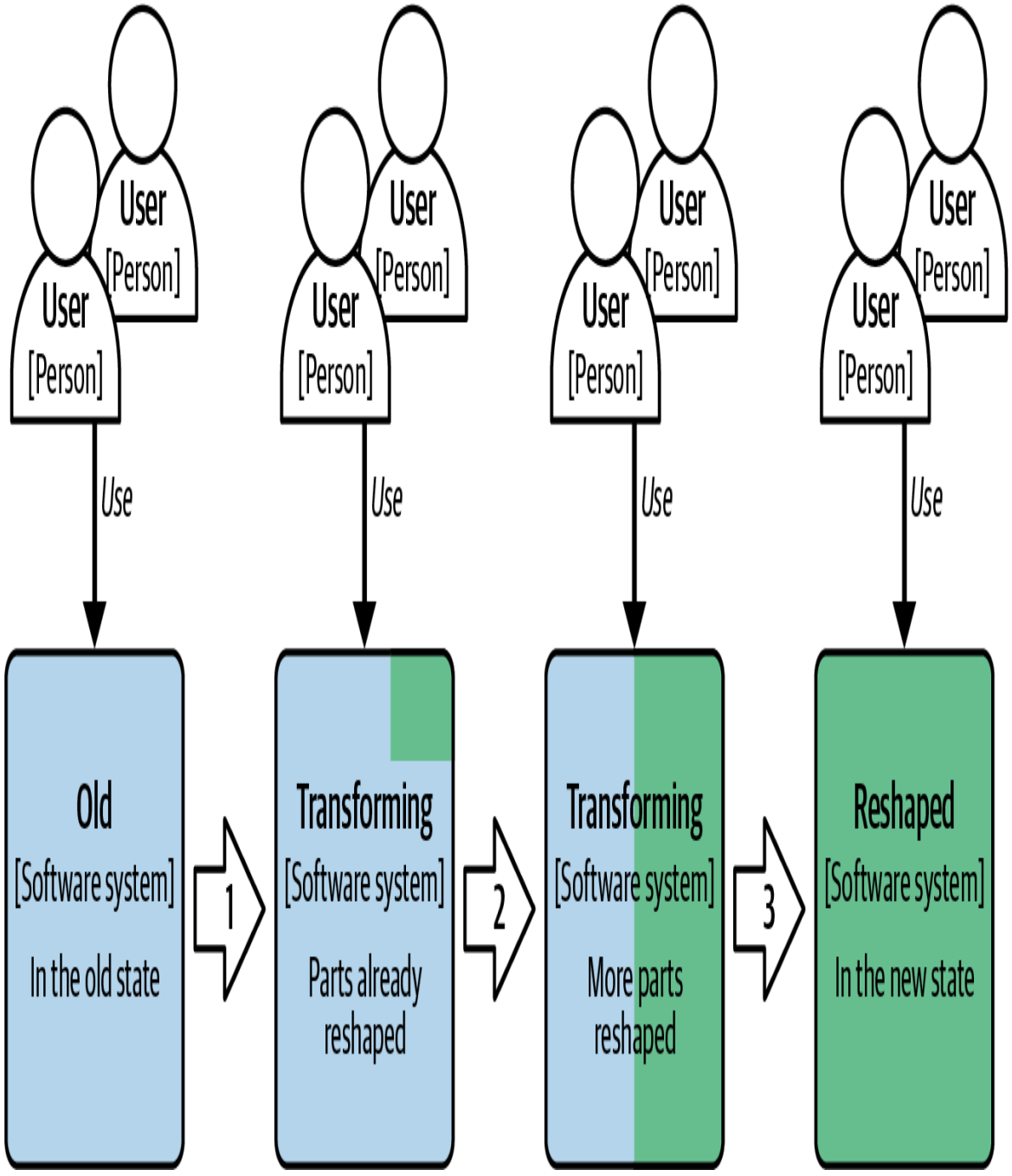


Figure 6-4. Steps of a reshaping.

This procedure can also be evaluated using the criteria from the last sections:

Unknown unknowns (overlooked functionality)

> During the refactoring of a legacy system, the developers quickly encounter things that they do not yet know, because the old source code

will react to every change. To support this procedure, it is advisable to create automated tests in advance for each of the parts that you want to refactor in order to safeguard the refactorings.

Standstill

For the users, a feeling of standstill can also arise during refactoring, because development capacities for extensions must be reduced in favor of refactorings for this conversion. However, since the old system is the subject of all work, users will perceive progress much faster than with step-by-step replacement and even more so compared to big bang replacement.

Duplicate work

With refactoring, duplicate work is completely avoided if the development team is able to find sufficiently small refactorings.

Loss of know-how

If new technologies are integrated into the legacy system during refactoring, the developers will learn new know-how with each subsequent refactoring. In this way, the loss of know-how should remain at a normal level.

Late deployment

Due to the step-by-step transformation, the revised parts of the system are deployed very quickly. The development team gets feedback very early. Often, just as with step-by-step replacement, the impression is created that this process will take longer and therefore be more expensive than big bang replacement. We have already seen with step-by-step replacement that this impression is deceptive, because a big bang is usually followed by a long phase of rework.

User frustration

At first the users will not notice the system changing during a redesign. It is only when new technologies are implemented, or parts of the system are rethought, that something changes for the users. So again, users have the opportunity to slowly get used to the changes.

Sticking to old solutions

The big disadvantage of this approach is that there is no clear-cut way for developing a new solution for parts or the whole system. The development team is not confronted with the question of whether the task that the software supports could be done differently or with a different system. This is exactly the disadvantage that Domain-Driven Transformation addresses!

Summary

In this chapter, we have provided you with concepts and arguments on how to select the most sensible procedure for converting your legacy system. You have learned about the relationship between big bang replacement, step-by-step replacement, and reshaping.

We have shown that while big bang replacement might look like a good choice in the first place, it is less often than expected. Far more often, step-by-step replacement or reshaping of the legacy system or combinations of these are the better option.

This last chapter in Part I, "Fundamentals for Domain-Driven Transformation," is the link to Part II and Part III. Now we can turn to the next part: Domain-Driven Transformation at the technical, tactical, and team levels.

---

1 The event that according to the eponymous physical theory created the universe.

Part II. Technical, Tactical, and Team-Organizational Domain-Driven Transformation

Decomposing a legacy system with Domain-Driven Transformation is a *major operation*. Similar to a human before major surgery, you should make sure that the patient is in the best possible general health. How to do that is the topic of this second part of the book.

Most legacy systems we see are in poor general health—not only do they consist of one or more *big ball(s) of mud*, but they also suffer from the following problems:

- They have become *rigid* and *inflexible* and are difficult to change, and changes are very error prone due to the convoluted, difficult to understand structure.
- Overall, compared to systems with a good architecture, they have many *undetected or even known errors*, *poor test coverage* leading to the former, and *many security gaps* due to outdated libraries.
- They are *difficult to build and deploy*.
- They lack quality even at the level of simple code metrics, such as length of classes and methods, cycles between classes, number of parameters, etc.

If a legacy system is in such a *distressed state*, then we cannot immediately start with the strategic Domain-Driven Transformation (Part III). Rather, we must first improve the general condition of the legacy system to an appropriate level with appropriate measures that we describe in this second

part of the book. In doing so, we will work on three levels to improve the general condition of our legacy system:

- Technical stabilization (see Chapter 7)
- Strengthening the source code with domain knowledge (see Chapter 8)
- Improving team organization (see Chapter 9)

For each level, we will reference the appropriate *domain-driven refactorings* from our catalog in the appendix in various places.

Where Do You Stand?—Made Easy

At the end of each of the three chapters that follow, you will find a section entitled "Where Do You Stand?" There you can evaluate the state of your legacy system and the organization in which it is being processed. You will find 12 statements in each case for which you can enter your own assessment on a scale of 1 to 5, between (1) = "I do not agree at all" and (5) = "I agree completely."

If your Xs look like those in Table II-1, then your system is in good shape with respect to the chapter's question.

Table II-1. Completed evaluation of a system in good condition

No.	Statement	1	2	3
1	<statement 1>			
2	<statement 2>			
3	<statement 3>			
4	<statement 4>			
5	<statement 5>			
6	<statement 6>			
7	<statement 7>			
8	<statement 8>			
9	<statement 9>	X		
10	<statement 10>	X		
11	<statement 11>	X		
12	<statement 12>	X		

If your Xs look more like those in Table II-2, then your system is in bad shape with respect to the chapter's question.

Table II-2. Completed evaluation of a system in poor condition

No.	Statement	1	2	3
1	<statement 1>	X		
2	<statement 2>	X		
3	<statement 3>	X		
4	<statement 4>	X		
5	<statement 5>	X		
6	<statement 6>	X		
7	<statement 7>	X		
8	<statement 8>	X		
9	<statement 9>			
10	<statement 10>			
11	<statement 11>			
12	<statement 12>			

Chapter 7. Technical Stabilization of Legacy Software

This chapter covers how to technically stabilize legacy software systems before undertaking deeper transformations. We will discuss practical measures that improve robustness, reduce technical debt, and lay the foundation for sustainable change.

Most of the time, it is advisable to start not with just some measures, but with those that cause the greatest improvement as quickly and cheaply as possible—the often-cited Low-Hanging Fruits (see [Manns and Rising 2015]). The following sections are arranged in the order in which these low-hanging fruits can be picked.

Updating the Integrated Development Environment

Like any craftsperson, a programmer can only do their job if they can work with tools. Having the right tools usually drastically increases the programmer's productivity. Nevertheless, time and again we come across teams that have stuck with an outdated version of an integrated development environment (IDE) and thus do not use the modern possibilities and the many small aids that state-of-the-art IDEs offer today, such as automatic refactoring. An easy step to take is to update the IDE (or, if necessary, introduce an IDE).

As we write this book, more and more IDE-integrated AI assistants are becoming publicly available to programmers. If you're not using them in your project yet, you should do so soon!

Automating Build and Deployment

As one of the first measures, we work with the development team to ensure that the software deployment pipeline meets current standards. Depending on the status of the development team and its software, different measures make sense:

Using a modern version control system

In some cases, the team needs to introduce a version control system because they have previously only worked with versions on individual computers and transfer to other computers via a drive or USB stick. Fortunately, today we rarely encounter such situations and only in smaller organizations that do not have professional IT.

More commonly we find outdated version control systems and migrate to a more modern system with the team. As we write this (it's 2025), in most cases that should be Git. (There are some special cases, like the games industry, where Perforce is dominant.)

Introducing automatic build

More and more rarely, we encounter teams that do not have an automated build process, but perform the individual steps of the build by hand. In these cases, we motivate teams to introduce an automated build. Tools like Maven, Gradle, MSBuild, Composer, and the like help with this—depending on which technology stack you are on.

Other teams do have an automated build, but an unreliable one. In such situations we advise considering if now is the right time to stabilize it.

Optimizing build

Typically, legacy systems are large, which is why we want to disassemble them. With large systems, the build process takes a correspondingly long time. The consequences are that team members start the build less often because they have to wait so long to continue. This reduces the frequency with which developers get feedback from

the build process, which in turn leads to more bugs in larger units and thus frustration.

There are several ways to go about optimization. You can treat yourself to a hardware upgrade for the build server and thus significantly shorten the duration of the build process. You can change the deployment model by introducing partial deployments, or use software for hot deployment.

Introducing CI/CD server

Once you have arrived at a performant build process, the next step worth taking is to use a CI/CD server that automatically builds, runs the tests, and deploys after each push. Frequently used tools are Jenkins, TeamCity, Bamboo, GitHub Actions, or GitLab CI. More information can be found in [Humble and Farley 2011] or [Wolff 2017a].

Infrastructure as code

The goal of all automation levels is to achieve repeatability and determinism. For this purpose, it makes sense to also regard configurations, infrastructure, etc., as code, i.e., to check them into the repository and put them under version control. This is what makes the next steps possible in the first place, since experiments can now be made easily and, in the event of failure, it is simple to roll back.

Once the deployment pipeline has been developed and is performing accordingly, the first sign of relief in the development team becomes noticeable—finally, you can switch back and forth between changing source code and building and testing the result in short cycles. Wonderful!

Updating Software Dependencies

The next step we recommend is to close the security gaps caused by outdated libraries. In modern projects, teams use dependency trackers to keep track of their dependencies on libraries and frameworks and to install updates in a timely manner.

Many legacy applications do not use these tools, which means that these parts of a system become outdated without anyone noticing and do not receive security updates that would keep them up to date.

One way to make rapid progress here could be to use AI tools. We hear from various sources that teams dealing with legacy software are using AI to:

- Determine which technologies are currently used in the legacy system and whether they are up to date.
- Identify which updates can be implemented without much effort because the new versions are largely compatible with the old ones.
- Get support by instructing a coding assistant to adapt all areas that are necessary for the update.

Updating Libraries and Frameworks

The first level with improved security and infrastructure can usually be achieved without major reworking of the source code. It is sufficient to use the new versions and usually make corrections in a few places, because most libraries and frameworks are delivered in a backward-compatible manner. But even for versions with breaking changes, the effort for the update is usually worthwhile.

It becomes difficult when the frameworks and libraries used are so outdated that you have to or want to switch to completely new frameworks. For example, a number of Java-based web systems from the early days were developed with the Struts user interface library. If you want to replace Struts with something modern, such as Angular or React, a major rebuild is necessary (in this case even with a change of programming language).

A new framework leads to major changes in the processes between the frontend and backend, and to completely different interpretations of the states in the frontend, so that a lot of source code has to be modified. In this case, it is better to continue with stabilization on the other levels first and to tackle the conversion to another framework, just like the decomposition

according to Domain-Driven Transformation, when the general state of the legacy system is better.

Another case that makes it hard to upgrade a framework is when the same version of the framework is used in many places in the code base that are owned by different teams. Then it might be better to upgrade the framework after strategic transformation when it is possible that different bounded contexts in the system can use different versions of the framework.

Updating the Programming Language

Especially with the programming language you use, an update is important because sometimes features are added that make the transformation easier. Examples are:

Java 9

: Modules were added as a new feature that can help with the implementation of bounded contexts.

C# 8

: Nullability can be disabled.

PHP 8.1

: Adds enums as a feature that makes implementation easier for some value objects.

PHP 8.1 and 8.2

: Introduce readonly properties and readonly classes, respectively, which make value objects easier to implement.

Java 14 and C# 9/10

: Introduce record types that make the implementation of value objects leaner and faster.

To upgrade the programming language (i.e., the absolute basis), automation up to CI/CD and Infrastructure as Code level is essential. The language version should be adjustable in a configuration file, and the rest should run automatically. We want to get feedback quickly if the version change causes us problems, and be able to undo it easily. Tools like asdf, SDKMAN, NVM, pyenv, or rbenv help with this.

Similar to updating libraries and frameworks, AI tools can also be a time-saving aid here. Currently, we can envision the following use cases:

- Identify which updates can be implemented without much effort because the new versions are largely compatible with the old ones.
- Get support by instructing a coding assistant to adapt all areas that are necessary for the update.

Resolving Compiler and IDE Warnings

The modern developer has a variety of tools at hand to be warned of potential bugs:

- The compiler of statically typed languages like Java or C#
- Static analysis tools for dynamically typed languages like PHPStan for PHP, Flake8 for Python, or RuboCop for Ruby
- IDEs like IntelliJ and its descendants
- Special bug finders like SpotBugs or StyleCop Analyzers

In legacy software, there is often a large number of these warnings, a form of technical debt. As a measure to stabilize, you need to resolve these warnings. Once you have done that, if possible, set the warning level to be even more sensitive and start over.

If you are faced with a large number of warnings, an AI tool can be useful. The task here is to specify a suitable prompt that identifies warnings that should be fixed first.

HENNING'S FIELD STORY: ONLY CHECK IN CODE WITHOUT WARNINGS TO THE CODE REPOSITORY

I was working as a developer on a project at a leasing company. When I started on the project, the system was about 15 years old. During this time, so many compiler warnings had been created in the code that you didn't notice when you created more warnings with a change. In turn, this meant that the code quality became poorer and poorer.

At some point, the IT manager decided to turn things around, and he introduced the following rule: only code that no longer contains any (!) warnings may be checked in. This was incredibly cumbersome at the beginning because there were so many warnings. Even the smallest changes became time-consuming because you spent much more time cleaning up old bugs (of other developers) than building new features. Every single developer hated the rule.

Within two or three months, however, at least the part of the code that was changed most frequently had been cleaned up. Here, the compiler warnings could finally be used again as a tool to protect the developer from careless mistakes. Life as a developer improved rapidly. After half a year, everyone was happy to finally be able to program on a tidier system.

Increase Test Coverage When Fixing Bugs

The next step in stabilization is to reduce bugs in the system. In most legacy systems we get to see, there is a well-stocked bug tracking system and poor test coverage. It is frustrating how many software systems without automated testing still exist today!

Often, poor test coverage and a high number of bugs are mutually dependent: the development team fixes critical bugs and extends the system at the request of the users. In doing so, the development team introduces

new bugs because they don't have a safety net of automated tests to alert them to the new bugs. Michael Feathers writes about this:

> *The big deal is that we often don't know how much of that behavior is at risk when we make our changes. [Feathers 2004]*

Therefore, he explicitly recommends tests to secure the existing functionality against changes.

The Scout Rule (Formerly Known as the Boy Scout Rule)

Of course, no one can afford to write tests after the fact for the entire existing source code base of a legacy system. But we can follow the scout rule! The real scout rule is that when a group of scouts pitches their tents for the night, its members must clean up the next day and leave the place where they spent the night in a better condition than when they arrived. Trash they have found, therefore, must be collected and taken away (see [Martin 2010]).

If we apply the scout rule to software development, then pitching tents for the night means the following: a software developer needs to change the source code in one place to fix a bug, perform a planned refactoring, or implement a new functionality. Their cleanup efforts then look like this: as they make the changes, they must implement tests and perform the necessary refactorings to leave the source code in a better state than when they started. If you follow the scout rule in your team, then tests are continuously added to the source code as bugs are fixed and enhancements are made, so that test coverage increases with each bug fixed, and the system's vulnerability to unintentionally introduced new bugs decreases.

Here, too, it is possible to make use of AI tools and have them write tests for existing software. If you do that, you must be aware that AI tools will initially only write tests against existing functionality, meaning they will not find any errors. However, this is valuable for the planned transformation of the legacy system, because the system will then have a larger test bed with which the refactorings run in the transformation can be checked.

CAROLA'S FIELD STORY: INTRODUCING TESTS INTO A LEGACY SYSTEM

I well remember a legacy system where we had secured a complicated algorithm step-by-step with unit tests. The rest of the source code was still without tests because we had changed little there recently. Now, if someone found another bug in our already well-tested algorithm, we were happy! We could make this new bug visible with another test (red bar) and then fix the bug. If our error correction was successful, then our new test turned green and all other tests continued to run as well. The new test was now another thread in our safety net.

If our error correction had side effects on other parts of the algorithm, we could find these side effects with our test suite. One of our existing tests failed in this case and reported the side effect to us, so we could think again about how to fix the original error without a side effect. In this well-tested part of the system, to our initial surprise, it was actually fun to do refactorings. We knew every time we made a change, the tests would alert us if we had changed the original functionality through our refactoring.

The experience that a comprehensive test suite makes both finding more bugs and refactoring a positive experience was a very good motivation for the whole development team to increase the number of unit tests in the rest of the source code as well.

It is often not at all easy to introduce unit tests in a highly interwoven legacy system with many cyclic connections. In order to make such legacy code testable (and later also decomposable) in the first place, the dependencies must be reduced. You can read more about this in "Breaking Dependencies".

Different Kinds of Tests

The good experiences with automated unit tests have led to the fact that, today in the software industry, we not only work with unit tests on the class level, but also secure our work on the component level with automated integration tests, and on the frontend level with automated UI tests. All these tests help to stabilize a legacy system.

The main focus in your quest for higher test coverage should be on local unit tests. They test small pieces of source code compactly in a well-defined test bed. Integration tests and UI tests span multiple classes or entire parts of a system, so debugging them becomes much more involved than unit tests. In Figure 7-1 you can see the test pyramid, which illustrates the weighting between the different types of tests. So over time, try to use the scout rule to build up as broad a base of unit tests as possible.

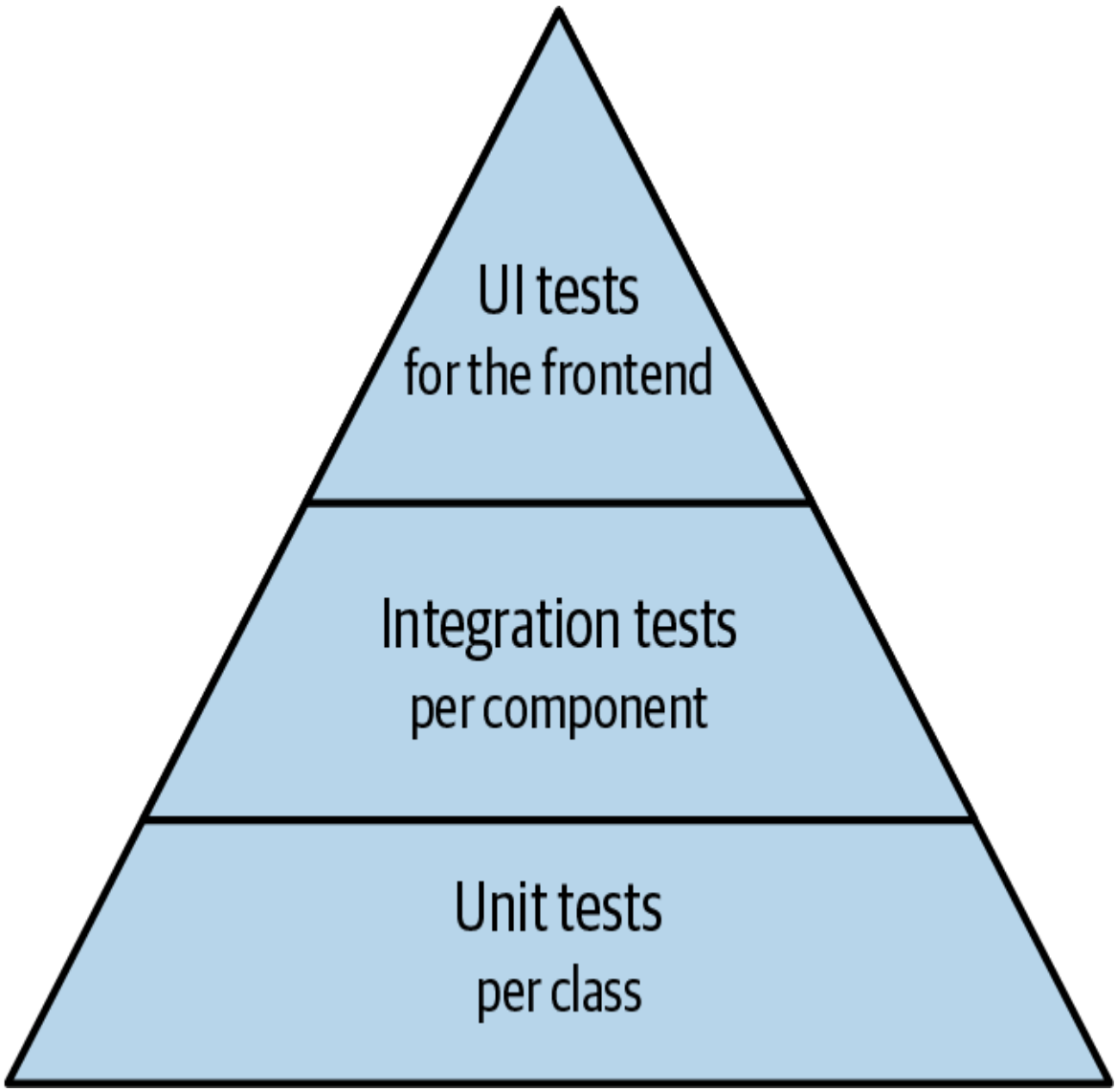


Figure 7-1. Test pyramid.

There is a discussion in the community about whether it should be a test trophy or something else instead of a test pyramid. The problem with this, as Martin Fowler describes well, is that the terms unit and integration testing are not cleanly defined (see [Fowler 2021]). We do not necessarily think of "unit test" in terms of "a unit is a class." In the DDD sense, a unit that deserves its own unit tests can be the following:

- A value object
- An entity with associated value objects

- An aggregate with inner entities and value objects
- A service with aggregates/entities/value objects, whereas repositories are mocked

When adding unit tests to the system, it can make sense to start with the leaves in the object tree and then move up in the tree.

The next level at which you should at least partially stabilize your legacy software before using tactical or strategic Domain-Driven Transformation is the implementation. An improvement in the source code is achieved via refactorings.

Cleaning Up the Code as You Go

In a decades-old system, a lot of dirt has accumulated in many corners. While we work on the code, we should also clean it up at the same time so that (a) the later transformation becomes easier and (b) it becomes clearer which parts of the code should be changed in the transformation. Anyone who has ever worked with a legacy codebase knows that readability and comprehensibility are crucial prerequisites for enabling change. This is why tidying up before the transformation is important. Kent Beck also calls this kind of cleanup *tidyings*; for a deeper insight, see [Beck 2024].

A legacy system usually contains a lot of unused, dead code (i.e., code that is never called). Now is the time to remove it (i.e., apply the refactoring *Remove Dead Code* [Fowler 2019, p. 237])! If you are not sure whether a piece of source code is really no longer needed, then temporarily add a log that shows its usage. Don't forget, however, that there is also source code that is only called up very rarely, e.g., at the end of the month, at the turn of the year, or at the end of the accounting year.

You should also remove duplicated code before you break up the system during a transformation. Carola has written more on this topic in [Lilienthal 2019].

Refactoring

In his seminal work on the subject of the same name, Martin Fowler defines *refactoring*[1] as "changing a software system in a way that does not alter the external behavior of the code yet improves its internal structure" [Fowler 2019, p. xiv]. Refactoring includes the tidyings from the last section and more.

Now, in a legacy system, there is usually a lot of bad implementation. And of course, the question arises: where should we start improving? Just as with the introduction of unit tests, the scout rule is a good place to start. Wherever we are fixing a bug or making an enhancement, we improve the implementation by refactoring first. As of 2025, AI code assistants have become so good that they can provide significant support during refactoring. They are able to handle small refactorings entirely on their own in agent mode. Of course, the programmer has to review and commit the changes. For large refactorings, it is up to the programmer to break them down sensibly to get meaningful suggestions from the AI tools.

Refactorings are part of a philosophy to always keep the code compilable, executable, and correct even while changing it. A typical procedure for renaming a method is therefore not:

Rename method—bad example

1. Change the name of the method first. Now all code that calls the method does not compile (and worse: does not work) anymore.

2. Fix the compiler errors step-by-step by using the new name.

Only after all these steps are complete will the software compile, test, and run again. During the steps we will thus have no feedback if the respective small step did not break anything.

Instead, we will proceed as follows:

Rename method—good example

1. Add a new method with the new name. It should have the same signature as the old one. Implement the new method as simply forward by calling the method with the old name.

2. Swap the names of the methods.

3. Mark the method with the old name as `@Deprecated`, or `[Obsolete]`, or similar.

4. Step-by-step go through the compiler warnings complaining about the deprecation warnings and change the calls to the old method name to calls to the new name.

5. Delete the (now not-called anymore) old implementation.

After each of these refactoring steps, the software compiles, tests, and runs as before. This has the consequence that you can integrate after each step. This gives you the additional possibility to interrupt the refactoring after each step and continue it at a later time.

The idea of the baby steps is so important that we show it here with a simple example. Of course, renaming a method is something that a modern IDE should support automatically, so if you've followed our advice from "Updating the Integrated Development Environment", you will simply use automatic refactoring.

Let's look at two typical categories of refactorings for which such a baby-steps approach is important: breaking dependencies and improving internal structures.

Breaking Dependencies

> *For each desired change, make the change easy (warning: this may be hard), then make the easy change.*
>
> —Kent Beck[2]

When (legacy) software is hard to change (and unit test), Kent Beck's tweet quoted here recommends that we first make it easy to change, and then make an easy change afterwards. Exactly. We try to do that by breaking our refactorings into the smallest steps possible. Unfortunately, in legacy systems this is usually a rather frustrating endeavor, especially because of the many dependencies involved. With every attempt to find a small refactoring, you go from the hundredth to the thousandth, and in the end you don't even know where you started. Just as Kent Beck says in his quote, "warning: this may be hard."

Fortunately, others have already dealt with this problem. An excellent book that describes how to break dependencies down into small parts and get back to testability and modifiability is *Working Effectively with Legacy Code* by Michael Feathers (see [Feathers 2004]). The *seam model* presented there is particularly helpful. The idea behind it is that we look for seams in our source code, where the code can be split up without being changed.

DEFINITION: SEAM

"A Seam is a place where you can alter behavior in your program without editing in that place" [Feathers 2004, p. 31].

At first, this sounds like a contradiction—how can I split code without changing it? But a closer look reveals several possibilities for this; one of them is polymorphism. You can think of a seam as a zipper on a coat. This is opened and the two zipper sides are no longer brought together, but a piece of fabric with a similar zipper counterpart is "zipped in" (see Figure 7-2).

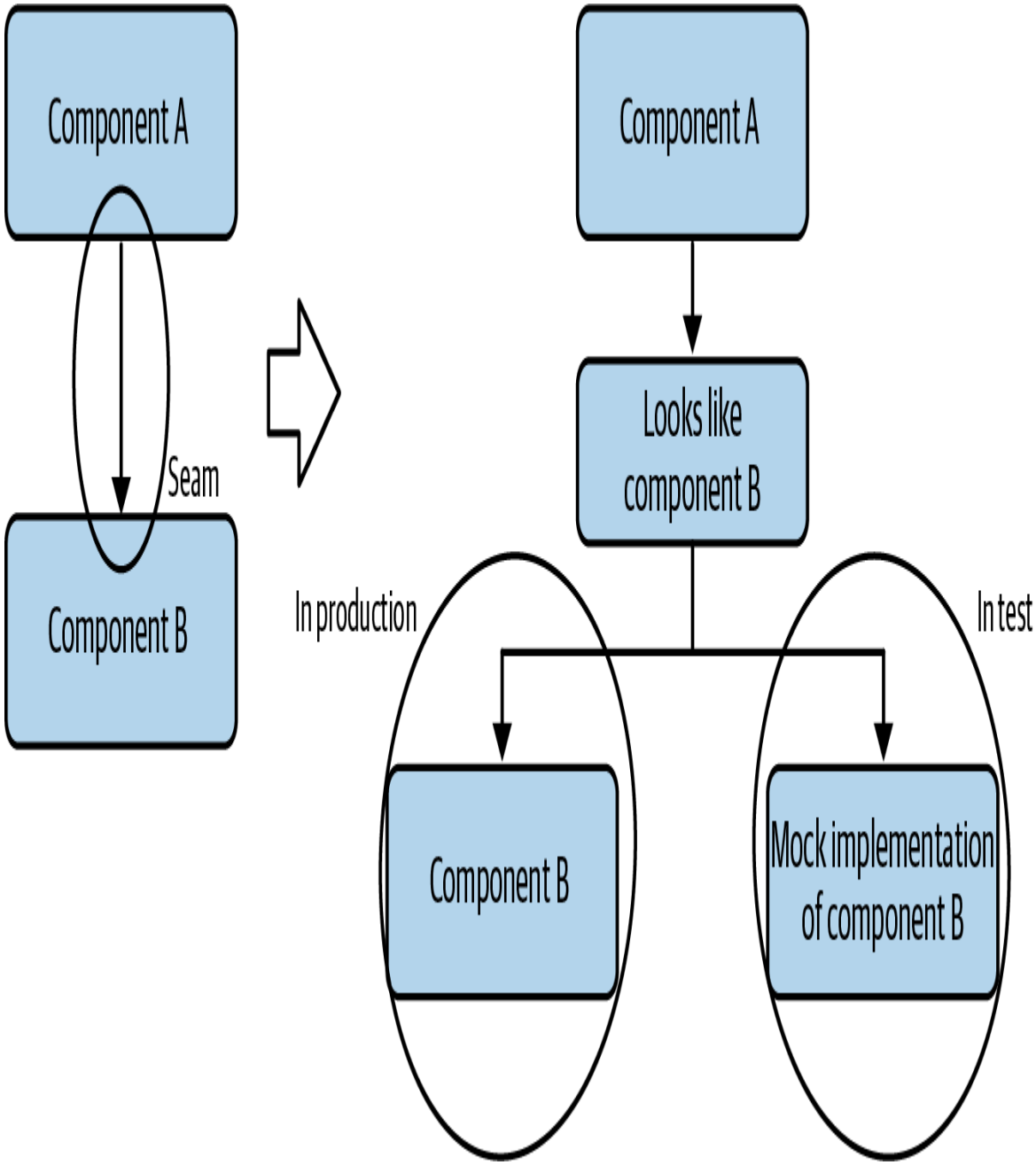


Figure 7-2. Seam: a place in software that we can cut open.

There are different types of seams; the most common is the *object seam*. Feathers describes the following possible attack vectors:

- An *object seam* exploits the *polymorphism* of object-oriented languages. Wherever a method `m()` is called on a variable `v` of type `T`, `v` can be replaced by a variable of a subtype `S`. In `S` we can override `m()` and implement the behavior we want.

- To the *preprocessor* in languages like C and C++ (*preprocessing seam*) we can pass different macros than originally defined.
- The *linker* (*link seam*); in Java, for example, the implementation of a class can be exchanged by "sneaking in" a separate implementation (class with the same fully qualified name) via the classpath. (Note: we only want to do this at test time. In production the "normal" class would be used.)

For each seam, it must be determined which counterpart will be called. For this purpose, Feathers defines the *enabling point*.

DEFINITION: ENABLING POINT

"Every Seam has an enabling point, a place where you can make the decision to use one behavior or another" [Feathers 2004, p. 36].

With object seams, the enabling point is often the configuration of the dependency injection; with link seams, on the other hand, it is the definition of the libraries to be loaded, such as the classpath in Java.

Seams provide us with the ability to get code under test where it previously seemed impossible. If we have a piece of code with automatic tests, then we can change the code without fear that it will break right away. This makes the next step possible: improve the internal structure.

Improving Internal Structure Using Metrics

> *In poorly structured code, the move from figuring things out to making changes feels like jumping off a cliff to avoid a tiger*
>
> —Michael Feathers [Feathers 2004, p. 8]

Another tool that many teams are using today to track down the parts in their system where they can start improving the implementation is dashboards such as SonarQube.

Such dashboards summarize bad smells from the source code and calculate metrics on them. Make sure you only use metrics that are relevant to your technologies and frameworks. Nevertheless, dashboards often report a very large number of problems in the source code. Sometimes they even calculate how many working days it will take to fix the technical debt determined by the dashboard using calculations based on the metrics. Don't be discouraged by such predictions, but start prioritizing.

Our advice for working with dashboards and improving the implementation of the system is:

- Take care of the *size metrics* first and foremost, such as LOC at all levels and cyclomatic complexity. Focus on the large and complex entities (methods, classes, packages, directories, projects, modules) in your software system.

  Units that stand out due to their size or cyclomatic complexity are always a sign of poor implementation or design. Test and reshape these large units through refactorings so that you can work with smaller units in the future.

- Also take care of the *coupling metrics*, such as cycles, FanIn, and FanOut. Concentrate on the units in your software system that are used by many other units, or use an above-average number of other units.

  Units that stand out because of many relationships are also always a sign of poor implementation or poor design. Test and refactor these strongly coupled units to get more independent units.

Observe how the metric values change over time in different areas of the source code. Notable changes, for example, when the number of relationships between classes in one area of the system increases, can also indicate slowly developing problems. The sooner you take action here, the better.

CAROLA'S FIELD STORY: IN-HOUSING OF A LEGACY SOFTWARE

I have a customer who asked us to help them with their legacy system. Five years ago, this customer had started planning a new development to replace a legacy system that was currently maintained by a service provider. When it became clear that the new development would take close to 10 years and the complaints from the business about the legacy system and the service provider became louder and louder, this customer decided three years ago to bring the maintenance and further development of the legacy software from the service provider back to their own development department.

At this point, the system was in a miserable state: it crashed frequently and some of the entries were stored incorrectly in the database. Trust in the service provider had been completely destroyed, both among users and in the IT department, and a solution had to be found more quickly than waiting 10 years for a new development. So the development team started writing unit tests for the known bugs and fixing the errors. At the same time, planning for the new development continued apace.

When the call came to us, the legacy system had been very stable and working perfectly for a year due to the successful repair work done by the development department. So the in-housing had been a complete success. The reason they called us, however, was not that they wanted support with the new development, but that they wanted us to help them take the next steps in improving the quality of the old system. In the meantime, the users were so satisfied with the largely error-free legacy system that a new development was no longer an option from a business perspective.

As a next step, we recommended that this team implement a dashboard so they could more easily identify and prioritize the biggest issues in the implementation.

How the metrics change over time can tell a lot about the code. Adam Tornhill presents an approach in *Your Code as a Crime Scene* to use the history of a project and the evolution of the code, utilizing the version control system to find improvements (see [Tornhill 2015]).

Improving Error Robustness

Frequently, we have to deal with systems where the users and the development team have to cope with crashes due to errors. In the extreme form, we have seen administrators (or, if DevOps is practiced, members of the development team) regularly restarting servers late in the evening or early in the morning to avoid the number of crashes during the next day.

To stabilize our system against this, we look here at possible techniques to make it more robust against errors.

Build Robustness Against Null

Tony Hoare invented null references originally for ALGOL W back in 1965. Virtually all practically relevant subsequent programming languages have adopted the principle. Hoare calls his invention today his "billion-dollar mistake" [Hoare 2009]. From our practice, we can only agree with this because so many errors are caused by unassigned references. For example, every Java or C# programmer knows the notorious `NullPointerException` or `NullReferenceException` being thrown again and again.

Modern languages such as Kotlin therefore exclude null by default, and modernized languages such as C# since version 8.0 offer the possibility to exclude it by configuration. Reference variables may then not be null by default. If null is to be allowed for a variable despite this, it must be explicitly permitted.

To stabilize legacy code, we recommend that you take the following steps to eliminate these `NullPointerExceptions`:

- If the programming language allows retroactive prohibition of null, turn on this mechanism. Example: nullable annotation context in C# 8 and later.
- Use annotations expressing nullability/nonnullability and have them checked. Examples: in Java different variants, like `@NotNull` from JetBrains or `@NonNull` from Lombok or the newer project JSpecify with its `@NullMarked`.
- If the above measures are not possible, check by hand at places where parameters are passed or variables are assigned in the source code. Assertions can be used for this; more about this in "Introduce Design by Contract".

This is a lot of work, but it increases the robustness of the system immensely.

Improve the Use of Exceptions

From our point of view and also from the point of view of other authors in our discipline (see [Fowler 2014], [Spolsky 2003], and [Hunt and Thomas 2019]), the use of exceptions should be treated with caution. Joel Spolsky even says:

> *The reasoning is that I consider exceptions to be no better than "goto's," considered harmful since the 1960s, in that they create an abrupt jump from one point of code to another. In fact they are significantly worse than goto's.*

The biggest problem caused by exceptions is that they create too many possible exit points from a method (again see [Spolsky 2003]). When we program, we need to think about every possible code path in our source code. We need to keep in mind the various exit points from a method and think through the response to those exit points in the method calling them as well. We also need to keep in mind what happens to the data that has been processed in the lines that have already passed through a method, and ask ourselves if we have run into an inconsistent state. This is hard for software

developers to keep track of, and leads to misunderstandings, fear of change, and disimprovement.

> **TIP**
>
> We recommend that exceptions should generally only be used for unexpected behavior and only for technical, not business problems.[3] For example, when validating input data, the occurrence of an error is normal because users make mistakes or interfaces provide erroneous data. So the error is expected behavior and should not be handled by an exception (see [Fowler 2014]).

In their book *The Pragmatic Programmer* (see [Hunt and Thomas 2019]) Hunt and Thomas say:

> *We believe that exceptions should rarely be used as part of a program's normal flow: exceptions should be reserved for unexpected events. Assume that an uncaught exception will terminate your program and ask yourself, "Will this code still run if I remove all the exception handlers?" If the answer is no, then maybe exceptions are being used in nonexceptional circumstances.*

But what is unexpected behavior? In principle, we can distinguish between two types of unexpected behavior:

External environment errors

> Errors that are beyond the control of the development team, such as an error opening a file or exceeding array bounds in C++. In principle, these exceptions are predictable. Therefore, developers of programming languages, libraries, and runtime systems have already taken precautions to deal with these errors. We normally do not need to program or throw the exceptions for environment errors ourselves, but only need to catch them in our programs with the try-and-catch mechanism and handle them if necessary, so that the program either terminates in a consistent state or can be used further after a suitable error message to the user.

Internal programming errors

> Programming errors are made during software development, and we cannot incorporate an anticipatory method of dealing with all possible programming errors in our software because they are unpredictable.
>
> However, we can take several measures to make our software as robust as possible against programming errors: pair programming, code reviews, automated testing (see "Increase Test Coverage When Fixing Bugs"), and the introduction of *Design by Contract*. All of these techniques are designed to find programming errors as early as possible and to prevent accidentally introducing new errors when changing or extending the system. The occurrence of errors contained in the code despite all precautions cannot be prevented by these techniques. An attempt to predict, catch, and handle such errors is a futile endeavor.

So let's take a look at how Design by Contract can help us reduce internal programming errors alongside pair programming, code reviews, and automated testing.

Introduce Design by Contract

With his *Design by Contract*, Bertrand Meyer (see [Meyer 1997]) has laid a good foundation to make classes more robust against internal programming errors. Meyer's basic concept consists of the idea that classes and methods offer contracts, which guarantee a certain behavior to the calling code, if the calling code fulfills certain conditions—exactly like with a legal contract.

In the source code, contracts in the classes are formulated as assertions. Assertions are Boolean expressions that can be used as *preconditions* and *postconditions* of methods, and as *invariants*. Preconditions refer to the state before a method is executed, postconditions refer to the state after a method is executed, and invariants refer to any valid state of an object. Meyer implemented this concept in his programming language Eiffel as part of the language. In other programming languages we have to make do with simpler programming constructs, e.g., in Java with the `assert` statements

built in there, in C# with the methods `Debug.Assert()` or `Trace.Assert()`, or in PHP the info function `assert()`.

Eric Evans recommends the use of Design by Contract in his “blue book” (see [Evans 2004]) in a separate section entitled “Assertions.” For stabilization, we recommend introducing Design by Contract into legacy code retroactively as well. In our cinema example, let’s take a look at how we can provide the seating plan with preconditions and postconditions.

IN THE CINEMA: DESIGN BY CONTRACT

For the art house cinema, we had developed a small implementation with a domain-oriented method `markAsSold()` in the basics chapter on Domain-Driven Design in Listing 3-1, which we repeat in Listing 7-1.

Listing 7-1. A domain model for the seating plan (repetition)

```
public class SeatingPlan {
    public void markAsSold(Seat soldSeat)
    //...
}
```

If we want to protect this method with preconditions against improper use, then we must at least check—as described in "Build Robustness Against Null"—that the passed reference to the seat is not null. We introduce a precondition for this right at the beginning of the `markAsSold()` method.

The preconditions are implemented with `assert` and are the first calls after the method header (see Listing 7-2).

Listing 7-2. A domain model for the seating plan following Design by Contract

```
public class SeatingPlan {
    public void markAsSold(Seat soldSeat) {
        assert soldSeat != null;
        //...
    }
    //...
}
```

What do we achieve with such preconditions? We prevent *internal programming errors* (see above) from creeping in unnoticed and having to be corrected later, when they are found through user testing or perhaps only when the software is already in production. As soon as a software developer

calls the method `markAsSold()` with `null` as parameter for the seat, the assertion fails, an `AssertError` is generated, and the program is aborted. `AssertErrors` are unchecked exceptions, which is why they do not need to be declared in the method definitions and do not need to be caught by the calling methods with `try` and `catch`. This way, the software developers learn as early as possible that they have made an error and can fix it early.

Here we used Design by Contract for protection against `NullPointerExceptions`; another reason is protection against empty lists. These are technical reasons to use Design by Contract; in Chapter 8 we show the business reasons as well.

> **TIP**
>
> We recommend to all teams that we work with to introduce Design by Contract into their source code according to the scout rule. At least preconditions should be part of the usual toolkit of every software developer, because they make it clear at the interface of every method under which conditions a method can be called. Postconditions and invariants are clearly more difficult to implement, because one must fall back often to old conditions to describe the effect of a method.

In our experiments with AI code assistants, they suggested useful preconditions that allowed us to at least maintain the status quo of the functionality contained in the legacy source code.

If you like Design by Contract and you want to go a step further with preventing invalid states, in many situations you can push it into the type system and "make illegal state unrepresentable." How this is done for domain models is the topic of *Domain Modeling Made Functional* [Wlaschin 2018].

Summary

In this chapter, you learned about different ways to technically stabilize legacy software: build and deployment were automated, and language,

libraries, and frameworks were updated. Test coverage was increased in parallel with bug fixing, and refactorings were performed to break dependencies and improve internal structure based on metrics. Finally, we looked at error robustness by preventing null, exceptions, and Design by Contract. If you are looking for more information, especially on the topics of cycles and the proper use of interfaces, we recommend Carola's book on sustainable software architecture (see [Lilienthal 2019]).

With the support of AI tools, the stabilization phase could be shortened, leaving more time for the more interesting steps of the transformation, such as tactical and strategic transformation.

With these initial technical improvements behind us, we can now turn to the topic of tactical transformation: strengthening the source code with domain knowledge, in the next chapter.

Where Do You Stand?—Made Easy

Check your level in this area of quality using the statements from Table 7-1. How firmly do these apply in each case? You can enter your own assessment on a scale of 1 to 5, between (1) = "I do not agree at all" and (5) = "I agree completely."

Table 7-1. Assessment sheet on the topic of technical quality of the system

No.	Statement	1	2	3
1	The system is easily expandable and maintainable.			
2	If you know one part of the system, it is easier to understand other parts.			
3	The code is monitored with SonarQube or similar tools.			
4	Problematic parts of the code are regularly refactored.			

No.	Statement	1	2	3
5	The system is developed with continuous integration and continuous delivery.			
6	There are enough automated unit tests so that changes are easy and predictable.			
7	There are enough integration tests so that changes are easy and predictable.			

No.	Statement	1	2	3
8	There are enough user interface tests so that changes are easy and predictable.			
9	The system is a big ball of mud.			
10	The system has places that are better left unchanged.			
11	The system has areas that nobody understands anymore.			
12	It is difficult to remove parts from the system or to test them.			

For the evaluation of your results, see Table II-1 and Table II-2.

[1] The name is derived from the multiplication which holds: $factor \times factor = product$. If you refactor, then the factors (= structure) change, but the product (= external behavior) remains the same.

[2] Kent Beck put this wonderful line in a tweet in 2012: *https://x.com/KentBeck/status/250733358307500032*.

[3] For a different opinion on this and the introduction of *domain exceptions* into the code, see [Krilic 2024].

Chapter 8. Strengthening the Source Code with Domain Knowledge

Once the system has been technically stabilized, we can turn to tactical Domain-Driven Transformation. This involves strengthening our legacy software with domain knowledge. Of course, there is domain knowledge in all legacy software, but it is often hidden under a lot of technology and is inconsistent.

This chapter shows different measures, how to free domain knowledge from technology and inconsistency, and how to make it more explicit, and thus bring it to light. We also discuss which domain-driven refactorings (which are for easier reference described in the appendices) can support these measures.

In Chapter 5 we discussed two basic principles of modular architecture: *cohesion* and *coupling*. With the help of these two criteria, the measures for strengthening the source code with domain knowledge can be well located. Together with two higher-level measures, this results in the following classification:

- Separate business and technical source code
- Increase cohesion
  - – Enrich modeling with domain knowledge
  - – Utilize value objects
  - – Introduce domain-oriented preconditions
  - – Make domain identity of entities explicit

- Reduce coupling
    - Introduce ID references at aggregate boundaries
    - Reduce inheritance in business source code
    - Use domain events for decoupling
- Document and check architecture in code

All these measures of strengthening the source code with domain knowledge lead us to refactor the source code. In the last chapter, we introduced refactoring techniques, pointed out the need for unit tests to validate refactorings, and motivated the use of AI coding assistants in this context. These things form the basis for this chapter and should be considered in each of the following measures.

Let's look at each of the topics in this chapter in detail and in context.

Separating Business Source Code and Technical Source Code

In Chapter 3 we saw that DDD in large parts of its methodology "only" or perhaps better "especially" takes care of the part of the source code that is based on the domain. We call this the application core or the business source code in contrast to the technically motivated source code, which should be around the application core (see Figure 8-1).

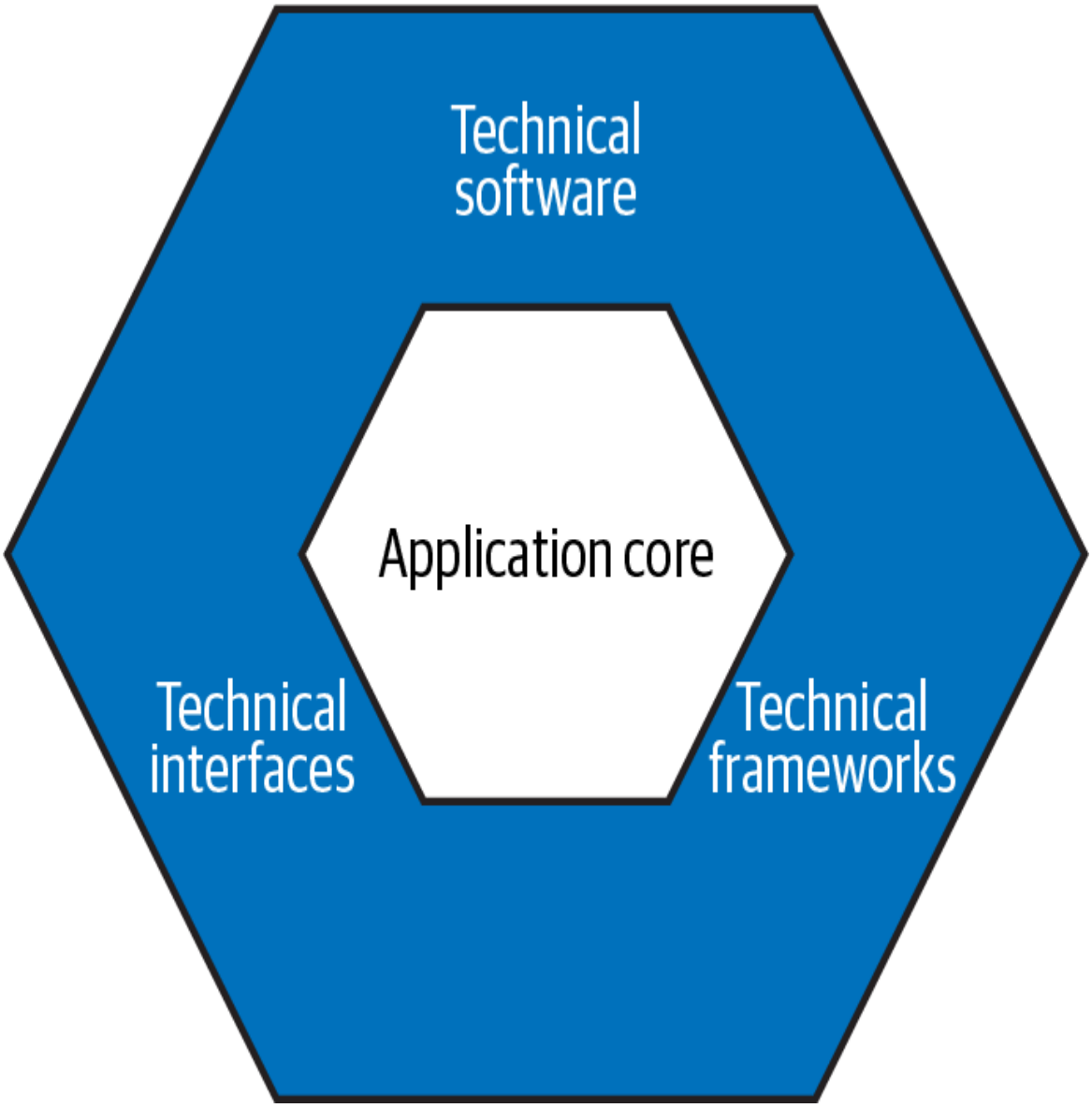


Figure 8-1. The application core embedded in technology.

Moreover, we saw in Chapter 5, "Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More", that many different architectural concepts published in the last decades follow this same principle and offer a set of design patterns to enable loosely coupled collaboration between the application core and technical source code.

For a legacy system, this means that as part of the tactical Domain-Driven Transformation we have to check the extent to which the technical source code and the source code of the application core are intermingled, and

which refactorings should be carried out to reduce this dependency. Where can we start here?

Database queries in business logic

> Are there database queries embedded in the source code of the application core? If this is the case at various points, the development team should start refactorings that separate the database access from the source code of the application core via repositories. (See "Tactical Design" for the repository pattern.)
>
> Refactorings that can help here are: *Split Active Record into Aggregate and Repository* and *Split Repository into Interface and Implementation*. In very old legacy systems, there are sometimes database queries even in the user interface code. A lot of work is needed here to separate the technical source code from the source code of the application core.

Business logic in the database

> Are there stored procedures and triggers in the database that not only exist for technical reasons but also contain business logic? If so, it often makes sense to extract that business logic from the database and bring it into the domain layer. If that is not possible, try at least to separate the business logic and database into different stored procedures.

User interface access in business logic

> Are there business logic classes that display their content themselves on the UI or convert it into a format that is needed for the UI? Or, conversely, is there logic within the UI code that actually belongs in the domain model?
>
> Refactorings that can help here are: *Extract Entity from Smart UI* and *Extract Service from Smart UI*.

Inheritance Relationships

> Are there classes from the application core that inherit from technical classes? Part of this is about implementation inheritance, for which we

can see the right way to handle this in "Reduce Inheritance in Business Source Code".

Another possibility is to refactor the code such that the real domain class is used by a new third class that inherits from the technical one and uses the domain class. However, invasive frameworks sometimes require that domain classes inherit from certain technical classes of the framework. These places are very difficult to remove, because this is often only possible with a removal or update of the deployed framework.

Henning investigated and described how to do it anyway in his master's thesisfootnote:[Perhaps the procedure described there may help another (German-speaking) reader; see [Schwentner 2008].

Increasing Cohesion

The very goal of Domain-Driven Design and Domain-Driven Transformation is to strengthen the elements of a system at all levels of the architecture with domain knowledge. The best way to achieve this is to increase cohesion in these elements. In this section, we will see how to increase cohesion at the class level and how to combine different types of classes as a DDD building block to strengthen cohesion across class boundaries.

Enrich Modeling with Domain Knowledge

In Chapter 3, the foundational chapter on Domain-Driven Design, we saw how to use the ubiquitous language to develop a behavior-rich domain model where the methods are domain motivated.

In practice, however, we often encounter the opposite, the *anemic domain model*, whose classes have (almost) only getters and setters to offer as methods. Martin Fowler first described this anemic domain model in 2003 (appropriately enough after a conversation with Eric Evans) as an antipattern (see [Fowler 2003b]).

WARNING

The nasty thing about an anemic domain model is that it contains nothing but classes with names coming from the domain language, which leads the interested observer to believe at first glance that this model has captured the domain. On a closer look, however, it becomes clear that these domain classes model data from the domain, but do not have any domain-oriented behavior. Instead of domain-oriented methods, they offer only setters and getters at their interface. This undermines a fundamental principle of object orientation—encapsulation.

CASE STUDY “BANK”

IN THE BANK: ANEMIC DOMAIN MODEL

Let's turn here to the bank example we learned about in Chapter 4. We came across the concept of an "account" in the EventStorming and had started to develop a domain model (see Figure 4-18). But what does the implementation look like in the legacy system we find in the bank? It has an admittedly very simple implementation, which can be seen in Figure 8-2 and Listing 8-1. This account simply has an instance variable `_balance` with `double` as its type, which can be set to any value via a setter method. The getter method can be used to get the present balance.

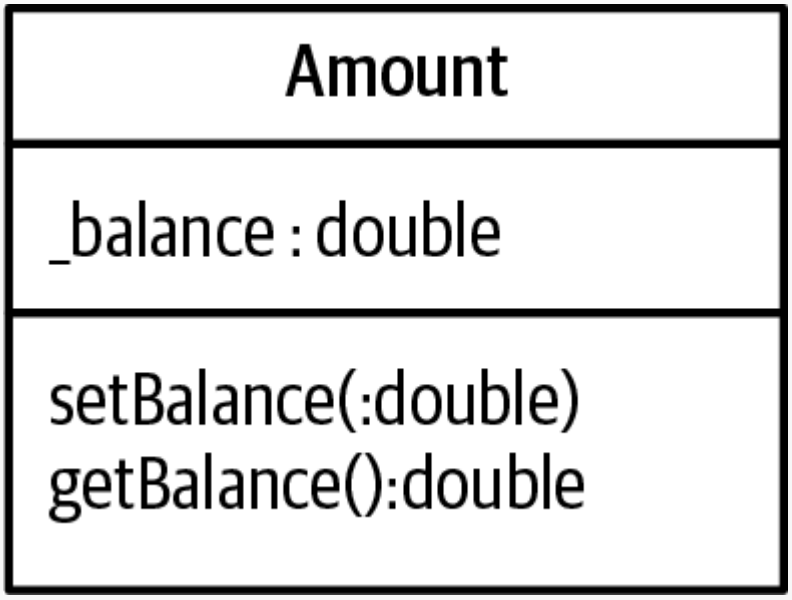


Figure 8-2. Bad: account—anemic.

Listing 8-1. Bad: account—anemic

```
public class Account {
    private double _balance;

    public Account() {
        _balance = 0.0;
    }

    public void setBalance(double amount) {
        _balance = amount;
    }
```

```
    public double getBalance() {
        return _balance;
    }
}
```

This implementation of the account is clearly an anemic domain model. We derive a more domain-driven implementation from the understanding gained through EventStorming which we saw back in Figure 4-18. There are two methods to manipulate the balance: the command (blue sticky note) "deposit amount" becomes the method `deposit()` with a parameter `amount`, and the command "withdraw amount" becomes the method `withdraw()` also with a parameter `amount`. These two methods replace the setter method `setBalance()` (see Figure 8-3 and Listing 8-2).

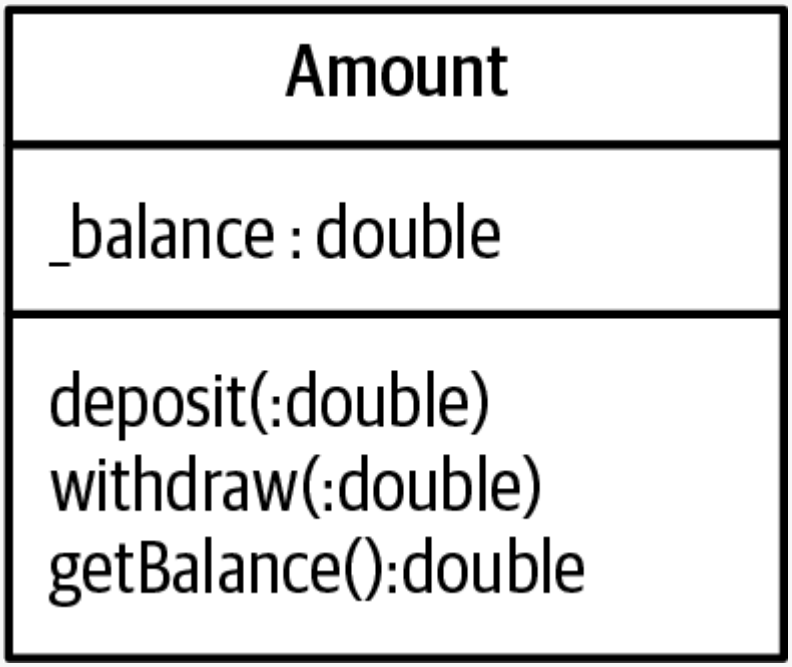


Figure 8-3. Better: account—with rich domain behavior.

Listing 8-2. Better: account—with rich domain behavior

```
public class Account {
    private double _balance;

    public Account() {
        _balance = 0.0;
    }
```

```
    public void deposit(double amount) {
        _balance = _balance + amount;
    }

    public void withdraw(double amount) {
        _balance = _balance - amount;
    }

    public double getBalance() {
      return _balance;
    }
}
```

What are the fundamental problems of an anemic as opposed to a behavior-rich implementation?

Loss of domain lingo

> Bankers don't talk about "setting the balance," but about "a customer depositing an amount." In the anemic implementation, you can't look at the method name to see what the class exists for. It lacks what DDD calls an *intention-revealing interface*.

Error-prone usage

> If someone uses the anemic implementation to deposit or withdraw money from the account, they have to use the same `setBalance()` method for both actions, even though the two actions semantically mean very different things. This is much more error-prone than the two methods `deposit()` and `withdraw()` in the domain-oriented correct variant. Here the semantics of the methods is unambiguous.

The impossible seems mistakenly possible

> In the anemic variant, the balance of the account can simply be set to an arbitrarily high value. Alas, the bank accounts of the authors cannot be filled so easily.

Consistency violation made easy

With the anemic variant, the calculation of the new balance is moved to the chunk of source code that uses it. In this chunk of source code, with the anemic variant, the current balance must be requested with `getBalance()`, then the amount to be deposited must be added to this value, and finally the result must be transferred to the `Account` class with `setBalance()`. Thus, the anemic variant cannot ensure that the account is only used in a manner meaningful from the domain perspective. Furthermore, the anemic implementation offers the possibility to convert the account into illegal states. The inner state of the account is published with this interface, and the consistency of the account can be violated when it is in use. Basically, the anemic variant allows arbitrary setting of the state of the class `Account`—which does not result in a meaningful sequence of states and consistency.

In the behavior-rich variant, the state of the account is encapsulated. The account class itself takes care of its consistency inside. When `deposit()` or `withdraw()` are called, then the balance in the account itself is calculated against the amount deposited or withdrawn, without the consistency being violated from the outside. The domain-oriented variant of the account ensures its consistency and a reasonable sequence of states by the implementation itself.

As you can see in the account example, anemic domain models lose the real domain-oriented handling or are reduced to setting fields with setters and reading fields with getters. The anemic domain model is significantly less expressive from a domain-oriented point of view. It can also be said that it has significantly less semantics than a domain model that is domain driven. In addition, an anemic model has its interior turned inside out and its data can be changed without regard to consistency constraints that would make sense from a domain-oriented point of view. Such anemic domain models are often created as a direct mapping of database tables.

If you find such domain models in your legacy system, you can (as we described in Chapter 6) perform the tactical transformation either before or

after the strategic transformation steps described in Part III. Three good initial refactorings are:

Strengthen the use of domain language in code

> The identifiers of variables, methods, classes, database tables, etc., should come from the language of the domain. If this is not the case so far, they should be renamed. The refactoring *Enforce Ubiquitous Language* helps with this. Sometimes this can mean that we then use local language domain terms instead of English domain terms to be more precise and use words that are understandable for the domain experts (see Chapter 3, "English or the local language?").

Check setter methods

> Check to see if they can be reworked into a useful domain-oriented method—as in the example from the account above. The getter methods are usually kept because the data needs to be displayed in the user interface. This is where the refactoring *Heal Entity Anemia* and its smaller parts, *Replace Setter* and *Remove Setter*, help.

Introduce value objects

> Introduce value objects throughout the business logic (see next section) using *Replace Primitive with Value Object*.

Now let's look next at how we can take advantage of value objects to enrich our domain model and make it more expressive.

Utilize Value Objects

A building block from the tactical design of DDD (see Chapter 3) can be introduced comparatively simply and step-by-step: the *value objects*. The goal here is to squeeze all primitive data types out of the business logic and replace them with domain-driven types. In this way, we use the type system of the programming language to achieve both more expressivity and type safety.

CASE STUDY “BANK”

IN THE BANK: INTRODUCING VALUE OBJECTS

The account from Listing 8-2 represents the balance as a primitive type `double`. This has two problems:

- Any number can be added to the balance, i.e., not only other monetary amounts, but also a length, a weight, etc.
- It is unclear in which currency the balance is given.

We therefore build a value object class `Amount` in order to be able to use our own type. This class holds the actual value as a `double` and has a second field for the currency (making it a *whole value*). If you think further at this point, it makes sense to introduce a separate value object class for the currency as well. Value objects can also contain other value objects (see Chapter 3). For simplicity, we implement `Currency` here first as an enum with only three possible values. Altogether, we get Listing 8-3.

Listing 8-3. Good: value object for amount

```
public enum Currency { EUR, GBP, USD }

public class Amount {
    private final double _amount;
    private final Currency _currency;

    public Amount(double amount, Currency currency) {
        _amount = amount;
        _currency = currency;
    }

    public Amount add(Amount otherAmount) {
        return new Amount(_amount + otherAmount.getAmount(),
_currency);
    }

    public Amount subtract(Amount otherAmount) {
        return new Amount(_amount - otherAmount.getAmount(),
_currency);
    }

    public double getAmount() {
```

```
        return _amount;
    }

    @Override
    public boolean equals(Object other) {
        var otherAmount = (Amount) other;
        return _amount == otherAmount._amount
          && _currency == otherAmount._currency;
    }
}
```

In addition, the `Amount` class in Listing 8-3 offers two methods to add and subtract amounts. All in all, the class `Amount` in this form is, of course, not yet usable in a real system:

- In the logic of both methods, the currency of the first amount is taken in each case. We will see how to deal with this when two amounts of different currencies are added in "Introduce Design by Contract".
- The `equals()` method is not null safe.
- The `hashcode()` method, which must always be overridden when overriding `equals()`, is missing.
- A monetary amount has exactly two decimal digits; `double` is a floating-point type. Maybe it would be better to store the cents internally as a `long` or `BigDecimal`.

For the sake of clarity, we omit these extensions necessary for a real system here. For a deeper introduction to the topic of how to model money amounts, see the pattern "Money" in *Patterns of Enterprise Application Architecture* [Fowler 2003a, p. 488].

If you have already updated your legacy system to Java 14 (or C# 9) or higher, you can create value objects as record types, which reduces the implementation of the always same but necessary parts—the boilerplate code (constructor, `equals()`, `hashcode()`, `toString()`, and accessor methods) to a minimum. In the bank this is done in Listing 8-4.

Listing 8-4. Good in Java 14+: value object for amount—with record types

```
public enum Currency { EUR, GBP, USD }

public record Amount(double amount, Currency currency) {

    public Amount add(Amount summand) {
        return new Amount(
            this.amount + summand.amount,
            this.currency);
    }

    public Amount subtract(Amount subtrahend) {
        return new Amount(
            this.amount - subtrahend.amount,
            this.currency);
    }
}
```

Let's build this value object into the account now (see Figure 8-4 and Listing 8-5). When creating an account instance, the currency in which it is managed must now be specified, because the opening balance should always be 0.0.

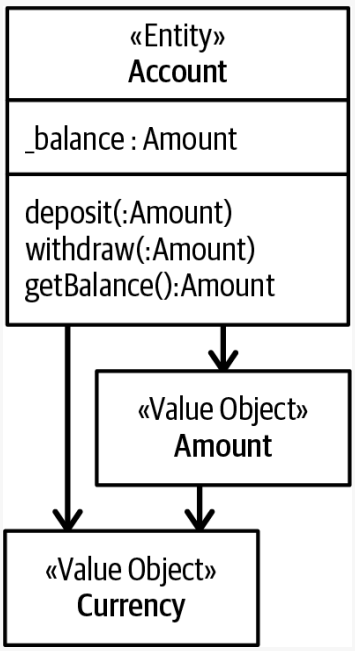


Figure 8-4. Even better: account—with domain behavior and value objects.

Listing 8-5. Even better: account—with domain behavior and value objects.

```
public class Account {
    private Amount _balance;

    public Account(Currency currency) {
        _balance = new Amount(0.0, currency);
    }

    public void deposit(Amount amount) {
        _balance = _balance.add(amount);
    }

    public void withdraw(Amount amount) {
        _balance = _balance.subtract(amount);
    }

    public Amount getBalance () {
        return _balance;
    }
}
```

How has the class `Account` changed? By using the type `Amount` instead of `double`, the signatures of the methods of `Account` have become more expressive. With this modification, in addition to the names of the methods and the names of the parameters, the type of variables and the return type now also give a clear indication of the proper domain-oriented use of the class. That means the class now has what DDD calls an *intention-revealing interface*.

In particular, incorrect use of `double` can no longer happen because it is always necessary to pass an object of type `Amount`. So we can no longer accidentally pass an arbitrary `double` instead of an amount; the compiler would detect a type violation.

Wherever you use primitive types in your system—such as in our example language Java `int`, `long`, `float`, `double`, `char`, and `String`—while something domain-oriented is processed or calculated, you can use value objects and thus strengthen the domain knowledge in your classes and on their interface. You will see that not only is it easier to understand the

source code, but also the error-proneness in the use of the classes will decrease.

We will look at how we can further improve the `Account` class so that it can also better maintain its consistency constraints in the next section.

Introduce Design by Contract

In Chapter 7 we learned about the concept of *Design by Contract* by Bertrand Meyer and used it to prevent technical problems like `NullPointerExceptions`. Here we go one step further and introduce domain-driven preconditions with the help of the refactoring *Introduce Contract*. We omit null checks, postconditions, and invariants in the next code examples to keep the example as small as possible.

CASE STUDY “BANK”

IN THE BANK: DESIGN BY CONTRACT

Earlier we had already noticed that the implementation of the value object `Amount` allows amounts of different currencies to be added, but then a wrong result comes out, e.g., 100 euros plus 100 dollars results in 200 euros. To protect ourselves from such nonsense, we extend the methods of the class `Amount` with domain-driven preconditions and arrive at Listing 8-6.

Listing 8-6. Very good: value object for amount—with preconditions

```
public enum Currency { EUR, GBP, USD }

public record Amount(double amount, Currency currency) {

    public Amount add(Amount summand) {
        assert currency.equals(summand.currency);

        return new Amount(
            amount + summand.amount,
            currency);
    }

    public Amount subtract(Amount subtrahend) {
        assert currency.equals(subtrahend.currency);

        return new Amount(
            amount - subtrahend.amount,
            currency);
    }
}
```

With this implementation in mind, we can imagine other methods on the value object, such as `lessThanOrEqualTo()`, `isNegative()`, etc. Unfortunately, in our example language Java, such methods have to be implemented a bit more extensively than in programming languages that allow operator overloading. For the sake of simplicity, we omit these further implementations here.

Let's look at preconditions now in Listing 8-7 for our account class as well. Of particular interest are the two methods `deposit()` and

`withdraw()`, which is why we do not list the constructor and `getBalance()` again here.

Listing 8-7. Good: account—with domain behavior, value objects, and preconditions

```
public class Account {
    private Amount _balance;

    // ...

    public void deposit(Amount amount) {
        assert amount.currency().equals(this.currency());
        assert !amount.isNegative();

        _balance = _balance.add(amount);
    }

    public void withdraw(Amount amount) {
        assert amount.currency().equals(this.currency());
        assert !amount.isNegative();
        assert amount.lessThanOrEqualTo(_balance);

        _balance = _balance.subtract(amount);
    }

    // ...
}
```

The first two preconditions in both `deposit()` and `withdraw()` check that the amount passed has the correct currency and is not negative. In the `withdraw()` method there is a third precondition that ensures that the amount is covered by the account. So you cannot overdraw an object of the class `Account`.

To fully implement Meyer's Design by Contract, one small thing is still missing: to be able to reasonably check when calling an object of the class `Account` that we are not accidentally overdrawing, the preconditions to be checked must also be offered at the interface as Boolean methods. For our class `Account`, this means that we introduce the method `isAmountCovered()` in Listing 8-8.

Listing 8-8. Very good: account—with domain behavior, value objects, and Design by Contract

```
public class Account {
    private Amount _balance;

    // ...

    public boolean isAmountCovered(Amount amount) {
        return amount.lessThanOrEqualTo(_balance);
    }

    public void withdraw(Amount amount) {
        assert amount.currency().equals(this.currency());
        assert !amount.isNegative();
        assert isAmountCovered(amount);

        _balance = _balance.subtract(amount);
  }

  // ...
}
```

As the first method in Listing 8-8, you will find the method `isAmountCovered()`, and in the following method, `withdraw(Amount)`, it is directly used as a precondition.

Why are we doing all this? The goal of the tactical transformation measures is always the same: we want to improve our legacy software and have the source code express the domain knowledge as clearly as possible. This is valuable on its own and can also pave the way for a subsequent strategic transformation that might be hard to achieve without lifting the inner quality of the legacy system.

This can be achieved on a purely syntactic level if the names of methods and parameters are chosen in a domain-driven way. However, it is even better if the semantics of the class are expressed in the implementation and not just “encoded” in the name or described in a comment. We achieve this with the use of domain-driven preconditions.

Make the Identity of Entities Explicit

In Chapter 3 we learned that each entity, unlike a value object, needs an immutable identity. Now you might say, “Our business objects, entities, all have an identity; it comes as a technical key from the database!” This technical identity is not what is meant by identity in DDD. In DDD, we only speak of the business source code and, to that extent, of a domain-oriented identity.

In Collaborative Modeling at the fine-granular level, we often discover this domain identity. Such a domain identity can usually also be found in legacy systems in the entity-like classes. In the case of `Account`, for example, this could be the `IBAN`, i.e., the international form of the `AccountNumber`. Or with a class `Customer`, as we will introduce in a moment, a `CustomerNumber` is often assigned.

This unique identity is realized in DDD systems, like all values, as a value object. The identity of an entity is immutable. Therefore, in Java we make the corresponding field `final`, in C# `readonly`. Entity methods that compare two entities of this type are based on this immutable value. The identity is initialized in the constructor of the entity, in which the two methods `equals()` and `hashcode()` must also be appropriately overridden.

IN THE BANK: THE IDENTITY

Let's have a look at the class `Customer` with its `CustomerNumber` as a domain identity. To make sure that the `Customer` class is not too bare, it also gets an address as a field (see Listing 8-9). In a real system, the `Customer` class will have considerably more fields and, most importantly, more behavior in the form of domain-oriented methods.

Listing 8-9. Bank customer—with explicit identity

```
public record CustomerNumber(int cnr) {}

public class Customer {
    private final CustomerNumber _cnr;
    private Address _address;

    public Customer(CustomerNumber cnr, Address address) {
        _cnr = cnr;
        _address = address;
    }

    @Override
    public final boolean equals(Object other) {
        return other != null
            && other instanceof Customer
            && _cnr.equals(((Customer) other)._cnr);
    }

    @Override
    public final int hashCode() {
        return _cnr.hashCode();
    }

    // ...
}
```

Look at the classes of the domain model in your legacy system while searching for domain-oriented identities, and replace the primitive types with the appropriate value object.

We will use this improvement for ID references at aggregate boundaries in "Introduce ID References at Aggregate Boundaries" to decouple large domain models, building one improvement on top of another.

Reduce Coupling

Decoupling is an important issue in strengthening the source code with domain knowledge, because by decoupling, we enable higher cohesion in the decoupled parts. They are no longer dependent on each other and can thus condense the domain knowledge they contain. Strictly speaking, strategic Domain-Driven Transformation is about precisely this point: we want to decompose a legacy system so that its bounded contexts are decoupled and can achieve higher cohesion.

But the transformation is also worthwhile at the tactical level. Here you can break up cycles, introduce ID references, and reduce inheritance hierarchies.

Break Cycles and Dependencies

When cleaning up systems with an MMI (see Chapter 5) below 8, special attention should be paid to the cycles and dependencies between classes and packages. Time and again we encounter constructions in the domain model where all domain classes have direct cyclic relationships to each other because they are a direct reflection of the entity-relationship model in the database. Such a large cycle in the domain model must be reduced to allow decomposition into contexts.

Even if our programming languages allow us to have cyclic relationships, to strengthen cohesion and reduce coupling, we should make a decision in which direction the relationship between two domain classes should go, avoiding direct cycles between domain classes.

At the package level, it is often necessary to reorder the classes between packages so that those classes are in a package that actually work together.

Thus, you increase the cohesion and reduce the coupling on the package level.

Methods to deal with entangled class and package cycles can be found in Carola's book *Sustainable Software Architecture* (see [Lilienthal 2019]).

Introduce ID References at Aggregate Boundaries

The DDD building block aggregate exists to draw boundaries at the tactical level and thus loosen coupling. This helps with the following problem. In many legacy systems, the classes of the domain model are connected by real object references and thus form either a large hierarchical tree or—in the worst case—a large, entangled class cycle.

If you are lucky enough to find "only" a large hierarchical tree or a class cycle with cycles of two that mirror the database structure, you should replace the real references with ID references as a measure of tactical Domain-Driven Transformation along aggregate boundaries.

CASE STUDY “BANK”

IN THE BANK: ID REFERENCES

At the bank, each account has a holder who is a bank customer. This customer may have more than one account and, in addition to accounts, may have securities portfolios, loan agreements, etc. All these domain terms describe self-contained business concepts, each of which should be modeled as a separate aggregate. In the legacy system of our bank example, the modeling is shown in Figure 8-5, where the class `Account` directly knows the class `Customer`.

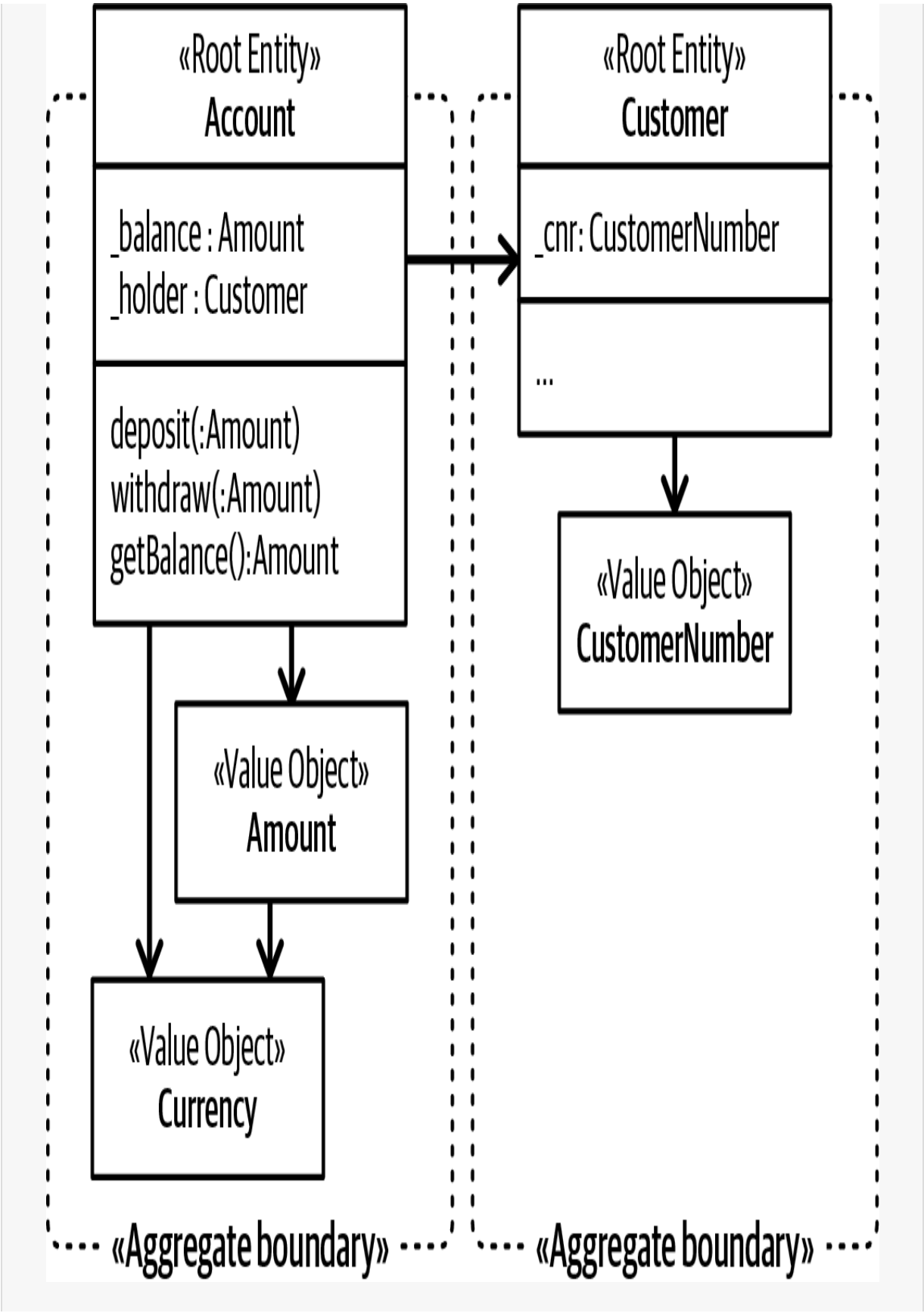

«Root Entity»
Account
_balance : Amount
_holder : Customer
deposit(:Amount)
withdraw(:Amount)
getBalance():Amount
«Root Entity»
Customer
_cnr: CustomerNumber
...
«Value Object»
CustomerNumber
«Value Object»
Amount
«Value Object»
Currency
«Aggregate boundary»
«Aggregate boundary»

Figure 8-5. Bad: aggregate `Account` references other aggregate `Customer` directly.

A better approach here is to follow one of Vaughn Vernon's rules of thumb for aggregates: "Reference Other Aggregates by Identity" (see [Vernon 2013], p. 359). This leads to a design shown in Figure 8-6.

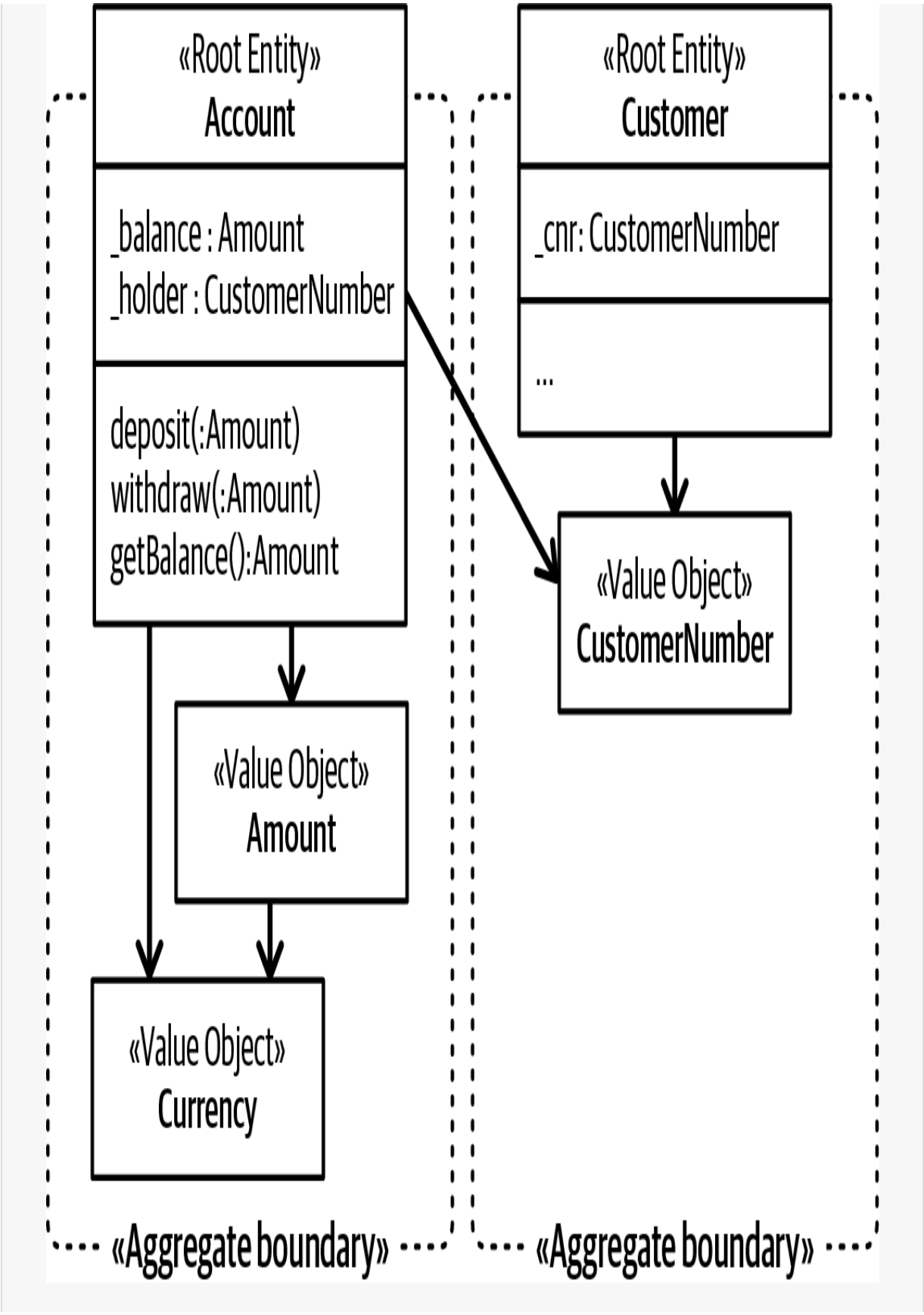

«Root Entity»
Account
_balance : Amount
_holder : CustomerNumber
deposit(:Amount)
withdraw(:Amount)
getBalance():Amount
«Root Entity»
Customer
_cnr: CustomerNumber
...
«Value Object»
CustomerNumber
«Value Object»
Amount
«Value Object»
Currency
«Aggregate boundary»
«Aggregate boundary»

Figure 8-6. Better: aggregate `Account` references other aggregate `Customer` indirectly via its identity.

Now you can work on the account alone, without always automatically having the whole customer object on your hands as well.

Drawing aggregate boundaries clearly helps with partitioning on a small scale and thus as preparation for partitioning on a large scale.

In the next section, we will see that we should not only improve individual classes, but also take a close look at the use of inheritance in our business source code.

Reduce Inheritance in Business Source Code

When object-oriented programming with C++, Java, and other programming languages became more and more popular from the 1990s onward, not only had programmers gained the class concept over procedural programming and could encapsulate the implementation, but they could also let classes inherit from each other. Unfortunately, this inheritance was often misunderstood and abused.

How often have we discussed with teams about their legacy code where three to four levels of inheritance have been introduced in the business source code? The motivation for these deep inheritance hierarchies was the desire to have to implement common functionality in the superclasses only once.

As a rule, these inheritance hierarchies prove to be a real impediment to further development over time. If you build class hierarchies, you abstract, which means that you omit or simplify details. However, the further you go into a domain and incorporate new subdomains into your system, the clearer it usually becomes that simplifications do not apply after all, and then the conversion becomes very difficult, because inheritance hierarchies are the most extreme form of coupling.

When we work with domain experts to retroactively build a ubiquitous language for their legacy system, in the initial discussions the teams tend to rarely tell us the term hierarchies, i.e., the "inheritance" between terms. Usually, we are talking about synonyms and similarities between terms. To express such similarities between terms in the class model, programming languages provide the simple device of delegation. Classes delegate an aspect of their tasks to another class, which can also be used by other classes for that aspect of their tasks. By delegation you get class structures, which get along without a strong coupling in contrast to class hierarchies.

Therefore, take a look at your legacy system and check whether you have worked with implementation inheritance in the business source code. Wherever possible, you should try to dissolve this inheritance and replace it with delegation. This step will make it much easier for you to decompose your legacy system later on, because you will increase the number of loosely coupled domain classes.

The Liskov Substitution Principle

From a software engineering point of view, such inheritance hierarchies based solely on implementation inheritance are a misunderstanding anyway. Inheritance should really only be used from a software engineering point of view if the subclass has an *is-a* relationship to the superclass. So a `CurrentAccount` is an `Account`, whereas the sentence "a savings account is a current account" is not correct.

This connection was first formulated by Barbara Liskov and has since been called the Liskov Substitution Principle (LSP, see [Liskov and Wing 1994]). This principle expresses how inheritance should be used exclusively: "Subclasses should be substitutable for their base classes." This means that if an object of the subclass is passed to a variable of the type of the superclass, then everything should work when the variable is used, although certainly with slightly different results than if the object was of the superclass. Robert C. Martin has included the Liskov Substitution Principle in his book *Agile Software Development, Principles, Patterns, and*

Practices (see [Martin 2013]) as the third point in his SOLID principles and has thus once again given it the deserved importance.

To check this concretely, you have to look at the methods of the subclass. Also at this point you can use Design by Contract by Bertrand Meyer from "Introduce Design by Contract" to help: subclasses must satisfy all contracts of their superclass if they are in an *is-a* relationship. An overridden method in the subclass must be callable in all contexts in which the original method of the superclass was callable, if there is an *is-a* relationship.

Subclasses that fulfill all the contracts of their superclasses must not require stronger preconditions in methods they override, i.e., at most weaker preconditions, and must not set weaker postconditions, i.e., at most stronger postconditions. In short, subclasses should expect no more and deliver no less than their superclasses.

CASE STUDY “BANK”

IN THE BANK: INHERITANCE

Let's look at a positive example of inheritance as an *is-a* relationship with the superclass `Account` from the last section and a subclass `CurrentAccount` (see Figure 8-7).

We extend the subclass `CurrentAccount` in Listing 8-10 compared to the superclass `Account` by an overdraft credit—so this account can be overdrawn. The methods `deposit()` and `getBalance()` are inherited as they are from `Account`. They are therefore not seen again here. The only method that `CurrentAccount` must override from its superclass `Account` is the `isAmountCovered()` method.

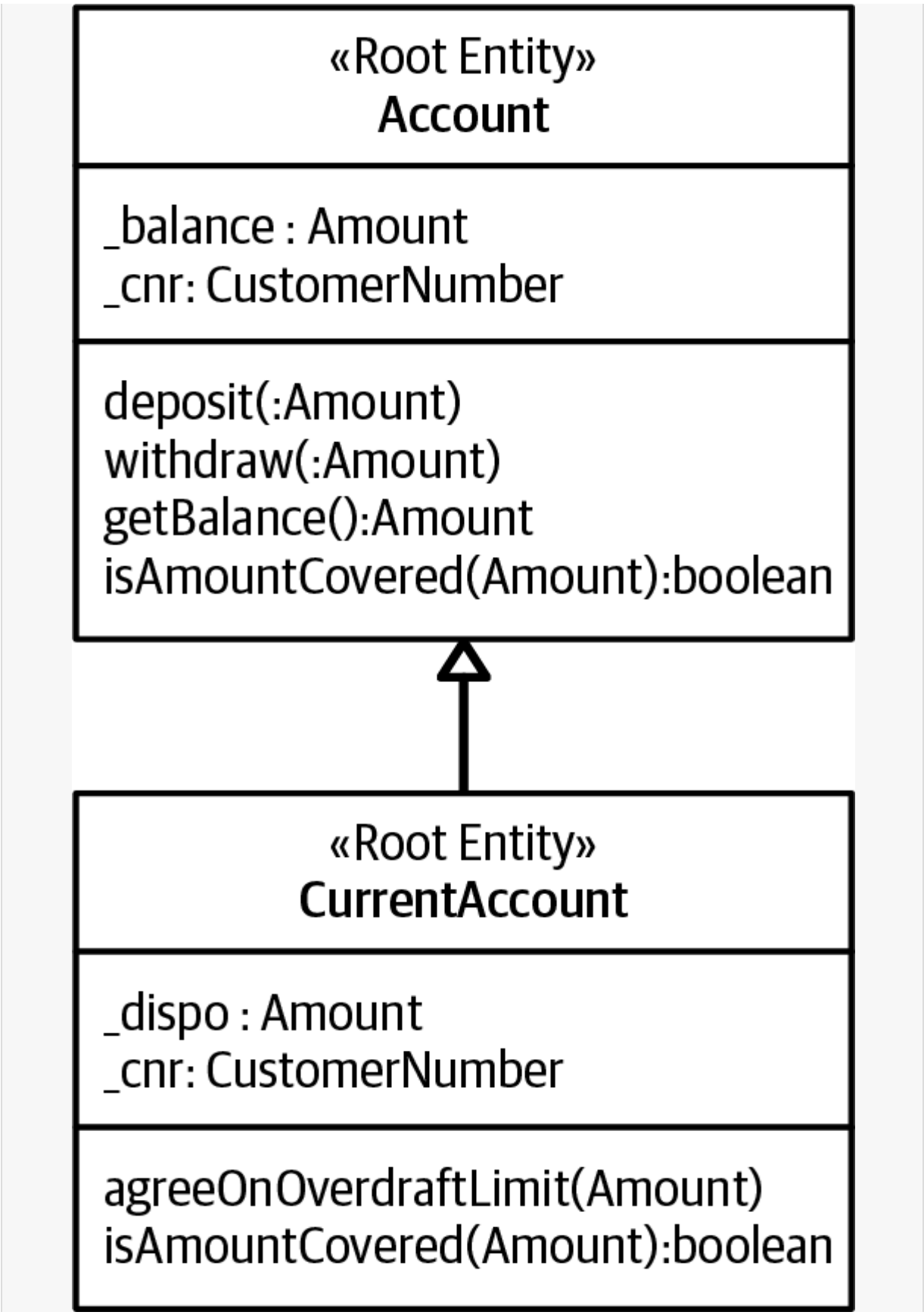

«Root Entity»
Account
_balance : Amount
_cnr: CustomerNumber
deposit(:Amount)
withdraw(:Amount)
getBalance():Amount
isAmountCovered(Amount):boolean
«Root Entity»
CurrentAccount
_dispo : Amount
_cnr: CustomerNumber
agreeOnOverdraftLimit(Amount)
isAmountCovered(Amount):boolean

Figure 8-7. Inheritance OK: `CurrentAccount` is a subtype of `Account` and satisfies the Liskov Substitution Principle.

Listing 8-10. Subclass `CurrentAccount`

```
public class CurrentAccount extends Account {
    private Amount _overdraftLimit;

    public CurrentAccount(Currency currency) {
        super(currency);
        _overdraftLimit = new Amount(0.0, currency);
    }

    public void agreeOnOverdraftLimit(Amount amount) {
        assert amount.currency().equals(this.currency());
        assert !amount.isNegative();
        _overdraftLimit = amount;
    }

    public boolean isAmountCovered(Amount amount) {
        return
amount.lessThanOrEqualTo(getBalance().add(_overdraftLimit));
    }
}
```

The subclass `CurrentAccount` extends its superclass with another instance variable, `_overdraftLimit`, by which the account can be overdrawn. The implementation of `isAmountCovered()` takes the overdraft credit into account, so that this account can be overdrawn up to the amount of the overdraft credit.

This modification of `isAmountCovered()` weakens the precondition in the `withdraw()` method in `CurrentAccount` with respect to the account. Or again, the `withdraw()` method in `CurrentAccount` expects less than the one it is overriding in its superclass `Account`. Just as the Liskov Substitution Principle requires for a superclass-subclass relationship.

Let's look at this in the source code (see Listing 8-11). Now let's imagine that we assign two variables of compile-time type `Account` each, one object of runtime type `Account` and one of runtime type

`CurrentAccount`. In addition, we create a variable of the type `Amount` of 100 dollars.

Listing 8-11. Use of superclass and subclass

```
var account1 = new CurrentAccount(Currency.USD);
account1.agreeOnOverdraftLimit(new Amount(1000.0,
Currency.USD));

var account2 = new Account(Currency.USD);

var usd100 = new Amount(100.0, Currency.USD);

account1.deposit(usd100); // account1.getBalance() == USD 100.00
account2.deposit(usd100); // account2.getBalance() == USD 100.00

account1.withdraw(usd100); // Amount covered,
account1.getBalance() == USD 0.00
account1.withdraw(usd100); // Amount covered,
account1.getBalance() == USD -100.00

account2.withdraw(usd100); // Amount covered,
account2.getBalance() == USD 0.00
account2.withdraw(usd100); // Amount NOT covered,
AssertionException
```

In Listing 8-11 you can observe that the current account behind `account1` will not have a problem with either a 100 or 200 dollar withdrawal, because the current account has been created with an overdraft limit of 1,000 dollars. In contrast, the account behind `account2` can only make the first withdrawal of 100 dollars. The second one for another 100 dollars will not work because the amount is no longer covered.

So the `CurrentAccount` behaves exactly like an object of type `Account` and fulfills all its contracts. It will allow at least as high amount as that in the account when withdrawing. However, even higher withdrawals are possible due to the overdraft feature. So the precondition is weakened: quite in the sense of the Liskov Substitution Principle.

Inheritance relationships that can be traced back to an *is-a* relationship, like the inheritance between an account and current account, we can leave in place in our legacy system. We should resolve inheritance relationships that do not satisfy this criterion by delegation where possible.

Documenting Architecture in Code

In the class diagrams of this book, we have and will use UML stereotypes to express “this class is an entity” and “that class is a value object.” It is also possible—in the spirit of “living documentation” (see [Martraire 2019])—to make the building blocks explicit in the code. We like to use the xMolecules libraries for this (see *https://www.xmolecules.org*).

IN THE BANK: JMOLECULES

For the bank, the Java variant of the xMolecules is used, namely jMolecules, and the code is annotated with the appropriate annotations, as seen in Listing 8-12. The root entities are marked here with the jMolecules annotation `@AggregateRoot`.

Listing 8-12. Domain model of the bank with architecture documentation in xMolecules

```
import org.jmolecules.ddd.annotation.AggregateRoot;
import org.jmolecules.ddd.annotation.Identity;
import org.jmolecules.ddd.annotation.ValueObject;

@ValueObject
public record IBAN(/*...*/) {/*...*/}

@ValueObject
public enum Currency {/*...*/}

@ValueObject
public record Amount(/*...*/) {/*...*/}

@AggregateRoot
public class Account {
    @Identity
    private final IBAN _iban;
    /*...*/
}

@AggregateRoot
public class Customer {
    @Identity
    private final CustomerNumber _cnr;
    /*...*/
}

@AggregateRoot
public class CurrentAccount extends Account {/*...*/}
```

Why should xMolecules be used? There are several reasons (see [Drotbohm 2022]):

- The code is easier to read because it expresses directly and by itself which DDD building block the programmer was thinking of when implementing it.
- In DDD, there are clear rules about which DDD building blocks may call which others. For example, a service may use both entities and value objects. Value objects and entities, on the other hand, are not allowed to use services. If we have annotated classes of our domain model with xMolecules, we can automatically check these architecture rules with ArchUnit (see [Gafert 2023]) or similar test frameworks.
- If the annotations of xMolecules are used, libraries are available through which we can reduce the lines of boilerplate code that would otherwise be necessary. In the case of Java, this applies to Spring, JPA, and Jackson, for example.

All three points are important for tactical transformation because they make the domain architecture explicit and clearly separate it from technical parts of the system. The first point is particularly important for preparing for strategic transformation because the different building blocks are handled differently. Value objects, for example, tend to be duplicated or moved to a shared kernel, and entities tend to be split up.

We have used the patterns of tactical DDD here. If you want to go deeper in the context of a tactical transformation, you should also look at the patterns from *Analysis Patterns* (see [Fowler 1997]). In the sense of that book, `Amount` is a *quantity* with the *unit* `Currency`, and `Account`, of course, is an *account*.

Does the AI have Knowledge About My Domain?

Coding assistants like GitHub Copilot or JetBrains AI Assistant can help a lot with tactical refactorings. The LLMs they use know about concepts like the tactical patterns of DDD, Design by Contract, and xMolecules. When you open a legacy project with a modern IDE with AI support, simple prompts like the following can help a good deal:

- "Which tactical refactorings would you advise?"
- "Refactor class `Contract` to utilize value objects"
- "Can `Currency` be an enum?"
- "Introduce Design by Contract!"
- "Can you use jMolecules in the code?"

Of course, the results have to be taken with a grain of salt. This is especially true for the domain knowledge that this chapter is about. The LLM will have generic knowledge about banking as described in the case study in this chapter, but it will usually not know the subtleties of the particular bank that the to-be-transformed legacy software was built for.

Summary

All of the options for tactical Domain-Driven Transformation presented in this chapter serve to strengthen the source code with domain knowledge. They can also help prepare your legacy system for strategic Domain-Driven Transformation.

Where Do You Stand?—Made Easy

Check your level in this area of quality using the statements from Table 8-1. How strongly do each of these apply? You can enter your own assessment on a scale of 1 to 5, between (1) = "I do not agree at all" and (5) = "I agree completely."

Table 8-1. Assessment sheet on the subject of domain knowledge in the syste

No.	Statement	1	2	3
1	Our domain classes are free from technical implementation details.			
2	The DDD building blocks are used consistently in large parts of the system.			
3	The DDD building blocks and other design patterns make the purpose of each class clear.			
4	The interfaces of the domain classes are safeguarded with preconditions.			

No.	Statement	1	2	3
5	The inheritance structures in the domain classes are based on “is-a” relationships.			
6	The DDD building blocks are used in such a way that the class structure is hierarchical.			
7	The domain classes have behavior-rich methods at their interfaces.			
8	There are classes that behave like value objects.			

No.	Statement	1	2	3
9	Our source code has no domain-oriented structure at the package or module level, but only a technical sorting.			
10	We have an anemic domain model.			
11	There are many set and get methods at the interfaces of the domain classes.			
12	The domain logic can be found above the domain classes in services and other classes.			

For the evaluation of your results, see Tables II-1 and II-2.

Chapter 9. Improving Team Organization

In order to bring a legacy system into a better state, more needs to be taken into account than "just" the technical dimension and the domain-oriented dimension of Domain-Driven Transformation. It is often worthwhile to rethink the team structures and the approach as well. In many organizations, Agile approaches are already well established, which always delights us. If this is not yet the case for you, now is the time to think about it, because Agile transition and Domain-Driven Transformation often go hand in hand.

In this chapter, we provide you with concepts and arguments for how you can lead discussions in your organization about the existing team structures and initiate the approach that makes sense for you to transform your legacy system.

Software as a Sociotechnical System

Not only does the architecture of a system play an important role when it comes to keeping the system usable, maintainable, and expandable over many years, but the organizational and communication structures of the teams that build and develop the system are also crucial for success. The relationship between team structures and architecture was first formulated by Mel Conway:

> *[…] organizations which design systems […] are constrained to produce designs which are copies of the communication structures of these organizations. [Conway 1968]*

Conway has thus described what can be observed in many organizations: the boundaries of software systems often coincide with the (communication) boundaries between teams, whereas the team boundaries

that are observed in real life often do not match the boundaries specified in the org chart. Because this relationship is so important, software nowadays is no longer viewed as a technical system alone, but as a *sociotechnical system*. Software does not exist independently of the people programming and using it, so software architecture and team organization are inextricably connected.

The transformation of legacy software into domain-oriented bounded contexts can therefore only succeed if teams are also structured according to domain-oriented criteria.

Let's make this clear with a simple example of an architecture that consists of three layers (see Figure 9-1): frontend, backend, and database. In each of these layers there are different components into which the system is organized. For simplicity, we have named them here with letters.

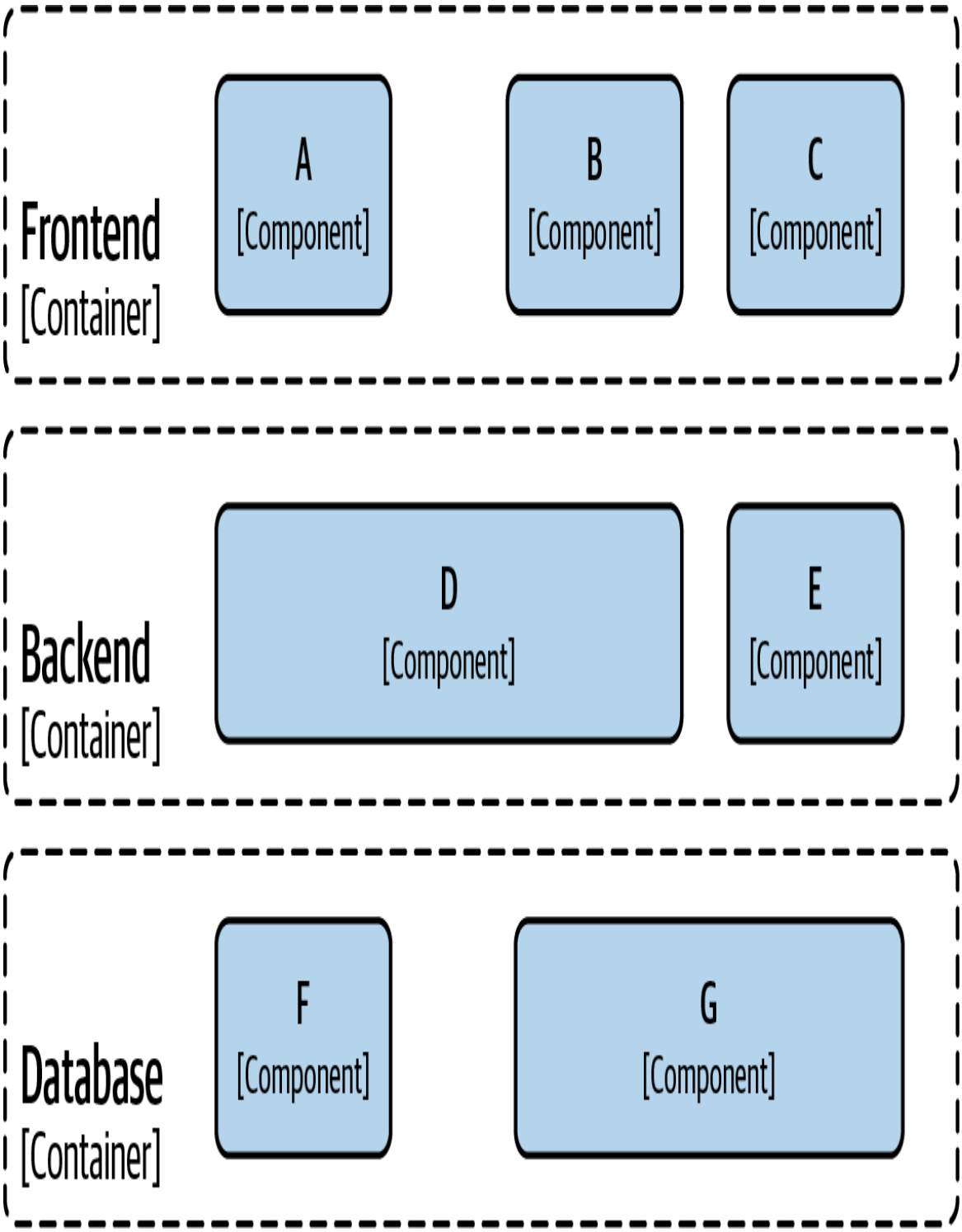


Figure 9-1. Layers with components (as a C4 Model components diagram; for the notation see Chapter 5).

Horizontal Architecture/Team Organization

If this architecture is divided into teams according to technical criteria, the result is *layer teams*. The (hoped-for) advantage is that all experts for a topic are gathered in one team. Here in our example this leads to a *frontend*, *backend*, and *database* team, respectively (see Figure 9-2).

In such a technical team division, the frontend team is dependent on the backend team adding needed features to the frontend interface. The backend team also has no freedom and must in turn wait for adjustments by the database team. In such a team structure, new features almost always affect several teams, which are dependent on each other in terms of time during implementation, and have to communicate a lot with each other to ensure that the interfaces are correct.

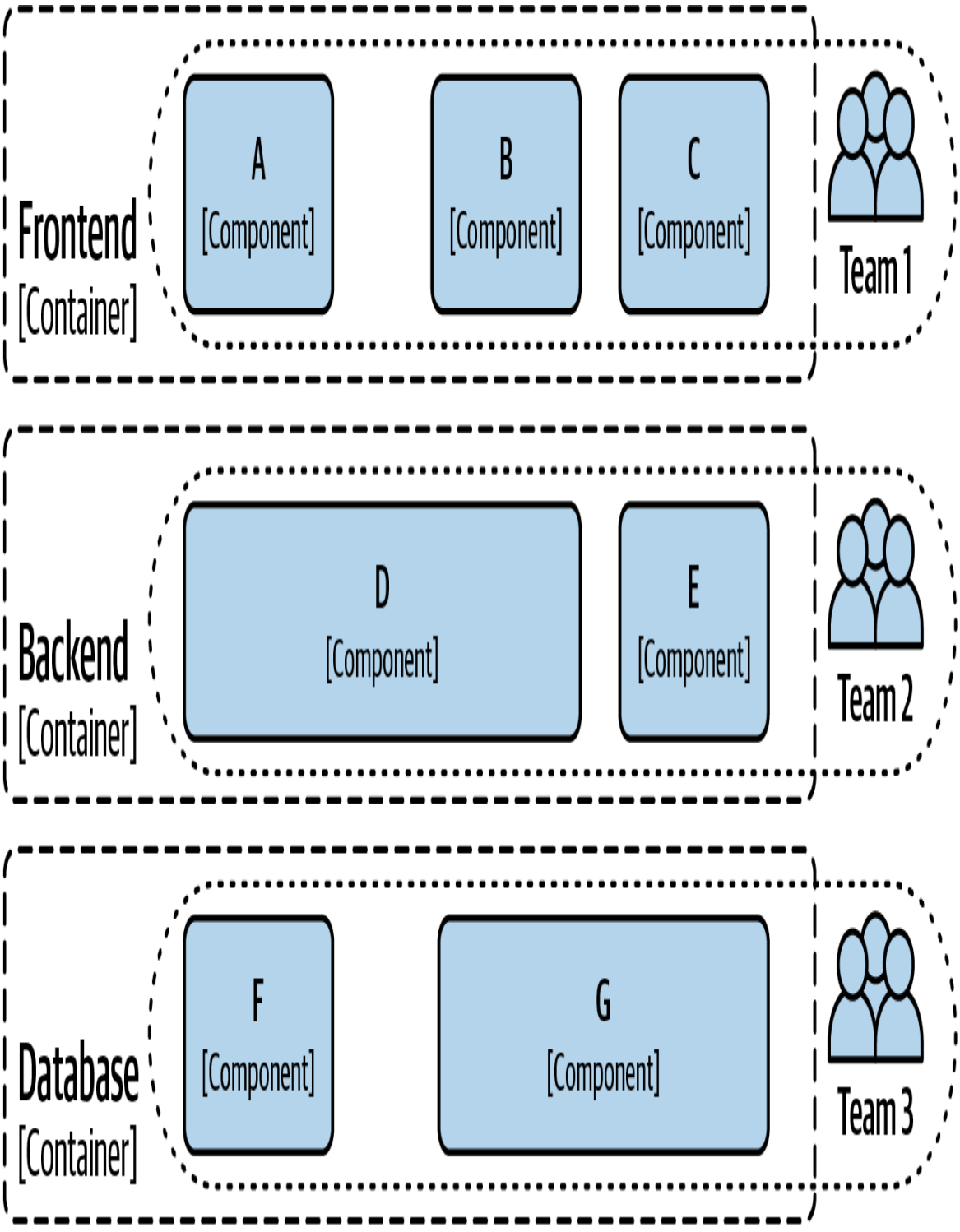


Figure 9-2. Lead to dependency: layers with horizontal teams.

Special case framework team

This type of team organization becomes even more difficult in large companies, where there are often not only frontend, backend, and database teams, but also one or more framework teams that provide foundation features for several software developments. If an organization has chosen such a team division, then the frontend, backend, and database teams that build an application together are dependent not only on each other, but also on the framework team.

If the framework team builds only basic technical services, the hindrance to the other teams from waiting for new technical functionality may not matter as much. The frontend, backend, and database teams can simply implement more features for users based on the old technical services, despite delays in basic technical services. The driver for technical changes is usually the framework team itself, which wants to introduce new technologies or has to implement security or performance requirements.

However, if there are framework teams in an organization that provide domain-oriented base classes and services, e.g., the universally used `Customer` class, then the delays and content misunderstandings directly impact the teams that build new features for the users. Unlike technical innovations, business innovations typically grow out of feature requests from users. This means that the frontend, backend, and database teams approach the framework team for basic business services with a request for business enhancements and changes. Here, on the one hand, the business wishes have to be communicated and, on the other hand, the frontend, backend, and database team have to cope with the time schedules of the framework team. In contrast to the extension of the technical services where the framework team is the driver, in this case the teams that build the features for the users are the drivers, i.e., the frontend, backend, and database team. Simply building on top of the existing domain-oriented base services without waiting for the innovations is often not possible. This exacerbates the dependency on the framework team even more.

Problems with the horizontal decomposition

With such a horizontal division, on the one hand, you have teams that are constantly waiting for deliveries from other teams on which they depend. On the other hand, following Conway's Law, many teams have a tendency to unify and merge the components they are working on.

In Figure 9-3 you can get an idea of what many teams do:

- They find commonalities in the components they work on and put them together completely, like $B + C$ as well as $F + G$.
- They also create common frameworks and libraries in their components that further connect and couple the individual parts, such as the connection between A and $B + C$, and between D and E.

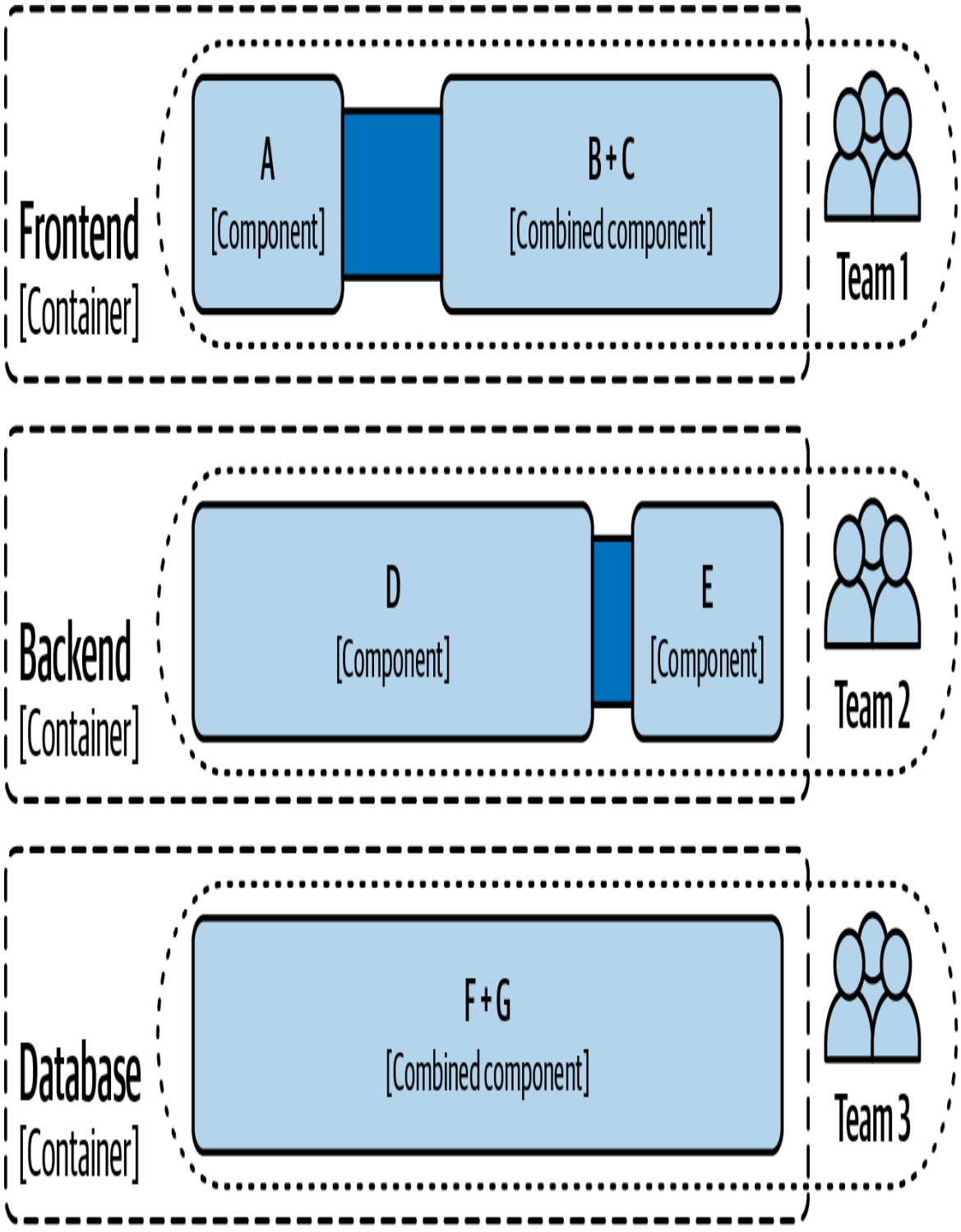


Figure 9-3. Problematic: layers with merging components in horizontally organized teams.

There is another problem worth mentioning that appears with horizontal decomposition: any team tends to optimize its local performance. This means that a team will shift dependencies further down the pipeline, isolate

the work they can do locally, and prioritize it accordingly. In a horizontal organization, a team's end product is heavily mock-based and is tested without or with outdated dependencies. A product delivered by a horizontal team is significantly less complete and cannot be delivered to the customer because it first has to undergo integration testing with the products of the other horizontal teams.

If you want to avoid these effects, you have to organize your teams vertically.

Vertical Architecture/Team Organization

In Chapter 3, the fundamentals chapter on Domain-Driven Design, we learned that the boundary of a context should be drawn vertically—that is, drawn across the technical layering. If we follow Conway's logic that architecture reflects team structure, then it follows that the boundaries between teams that maintain the bounded contexts must also be drawn vertically.

Then we need members in each team with expertise from all layers and not "just" backend developers or frontend developers, for example. In Agile terminology, we say that the teams should be composed *cross-functionally*.

A vertical division of teams according to domain-oriented criteria makes it possible for a team to be responsible for a bounded context in the software that runs from top to bottom through all technical layers, from the user interface to the database and the frameworks (see Figure 9-4).

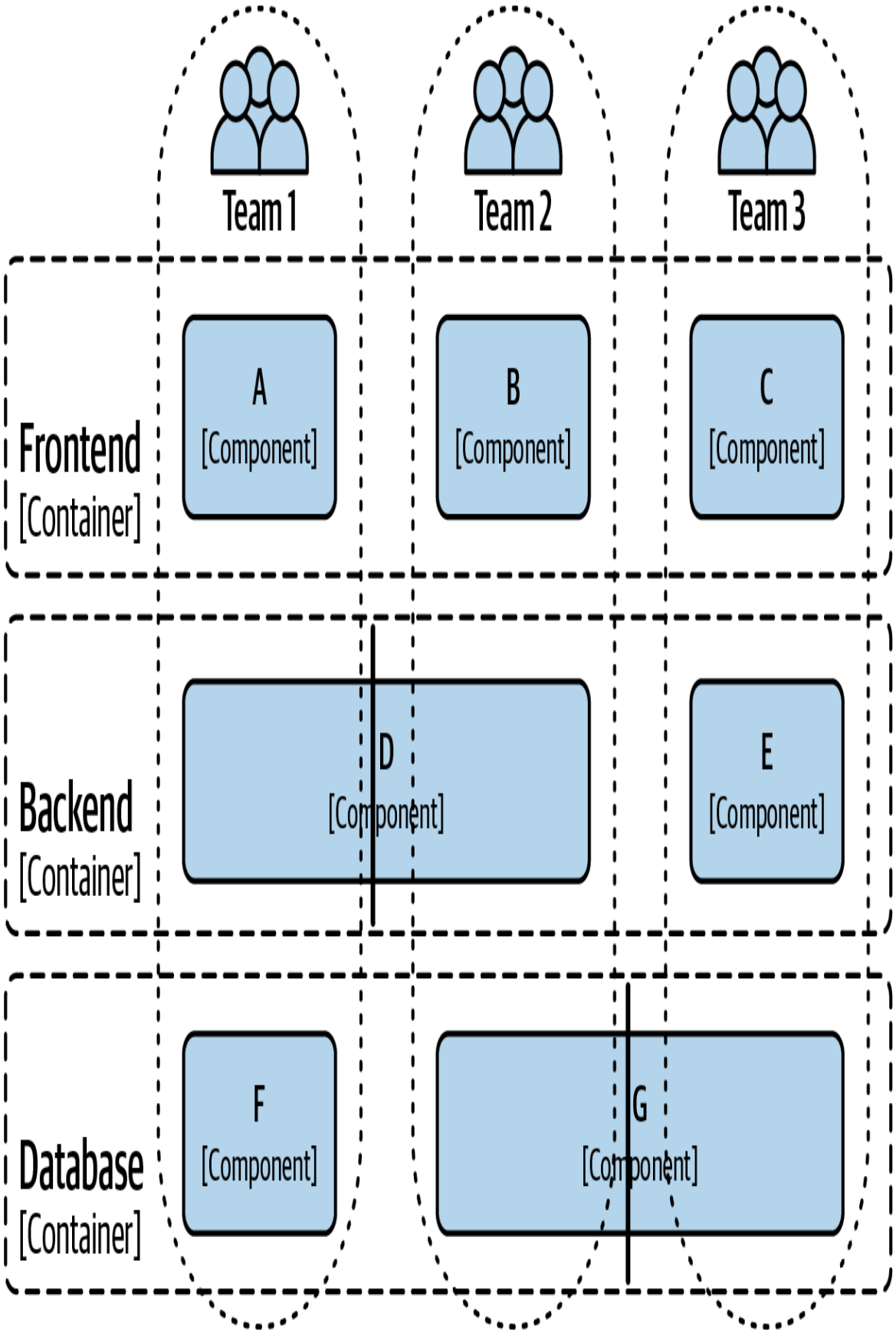

Team 1
Team 2
Team 3
Frontend
[Container]
A
[Component]
B
[Component]
C
[Component]
Backend
[Container]
D
[Component]
E
[Component]
Database
[Container]
F
[Component]
G
[Component]

Figure 9-4. Makes independence possible: layers with cross-functional teams.

For team 1 and team 2 to truly develop separately, component *D* in our architecture must be split into two separate components in the backend layer. In the database, too, the data tables that make up *G* must be split. We will learn how to do this in the following chapters on strategic Domain-Driven Transformation.

How to Perform This Team Reorganization?

Organizational Domain-Driven Transformation often means evolving horizontally organized teams (see Figure 9-2) into vertically organized teams (see Figure 9-4). To give teams an indication of how the planned changes will proceed, we have described the following sociotechnical refactorings:[1]

- *Form Cross-Functional Team out of Layer-Team Members*
- *Form Second Cross-Functional Team out of Partly Layer-Team and First-Team Members*
- *Form Second Team out of Only Layer-Team Members*

Organizational Domain-Driven Transformation often goes hand in hand with strategic Domain-Driven Transformation. For an example, see Chapter 11.

In most organizations, this simple view of architecture and team organization is a good starting point or initial explanatory model to kick off discussions about the organization. However, to achieve sustainable change in team structures, more is needed. The book by Matthew Skelton and Manuel Pais titled *Team Topologies* (see [Skelton and Pais 2019]) has made a widely respected contribution in this area. In the next section, we introduce the approach and link it to DDD so that you can use both approaches in combination for change in your organization.

Team Topologies

According to Skelton and Pais, the goal of an organization should be to have impactful and efficient teams in order to generate as much added value for the organization as possible. Teams are impactful and efficient when they get into the *flow* of software development, quickly moving new ideas and features into implementation and production. Exactly what every user wants: a new idea is taken up by the team and can be found in the product in a short time.

Working in the Flow

Skelton and Pais name two important prerequisites for working in the flow:

1. You have to keep the *cognitive load* of team members under control so that they can focus on their task and get into the flow. The goal is for everyone to be able to devote as much of their mental energy as possible to creating value for customers—and not waste their energy on side issues, such as coordination with other departments or accidental complexity (see Chapter 2) of all kinds.
2. It is important to help team members achieve the greatest possible *autonomy* in their work by reducing external dependencies and enabling local decision-making. With a high level of autonomy, they can carry out their tasks at their own pace and when they have time, without being slowed down by dependencies.

To handle the cognitive load and create autonomy, the following basic requirements must be met in the Team Topologies approach:

- A team should not exceed five to nine people so that the need for communication between team members does not become overwhelming. In large teams, the cognitive load is increased by more communication and by the fact that they work on larger pieces of software together.

- Teams should work together *long-term* to develop and optimize efficient workflows. If a team works together for the long term, it reduces the cognitive load because people know each other well and don't have to go through the storming-forming-norming-performing cycle[2] over and over again—and hopefully also regularly sort out the pain points in retrospectives.
- Each team should have clear *responsibilities* and *boundaries* for their work so that the tasks and the goal remain understandable and actionable for all team members. This creates autonomy and reduces the cognitive load because there is less need to think about unclear constraints, and decisions can be made more quickly.

Types of Topologies

Building on these foundations, Skelton and Pais describe four fundamental topologies:

Stream-aligned team

> These teams create value for customers along the value stream. They are cross-functional so that they can deliver significant increments fast with this mix of skills. However, fast is only possible if these teams can work without having to wait for other teams.
>
> This topology should be the most common in an enterprise. In the context of DDD, such a team would work exclusively on one (or several) bounded context(s), as this allows the responsibilities and boundaries to be very clearly defined.

CASE STUDY “CINEMA”

IN THE CINEMA: STREAM-ALIGNED TEAMS

At Clintwood Cinemas, we had developed a context map back in Figure 3-12.

Our target picture for working on the different contexts is to have stream-aligned teams for the contexts, as suggested by *Team Topologies*. The software development department building the system Cine2001 has so far a layer team organization with frontend, backend, and database teams.

In a cross-team workshop, we discuss with the developers and architects new team structures with cross-functional teams. We can form four cross-functional teams and ask the teams to give themselves names. Finally, we discuss which team best fits which bounded context. Figure 9-5 shows the result.

We had identified the two contexts "ticket sales" and "program planning" as core domains back in Chapter 3, and there will be a lot of work to do in these two contexts when rebuilding Cine2001. Therefore, we envision a separate team for each of these two contexts. The supporting subdomain "concessions" and the generic subdomain "accounting" will be handed over to the third team. We also give a name to the external team for "film distribution," even though these developers do not belong to our organization.

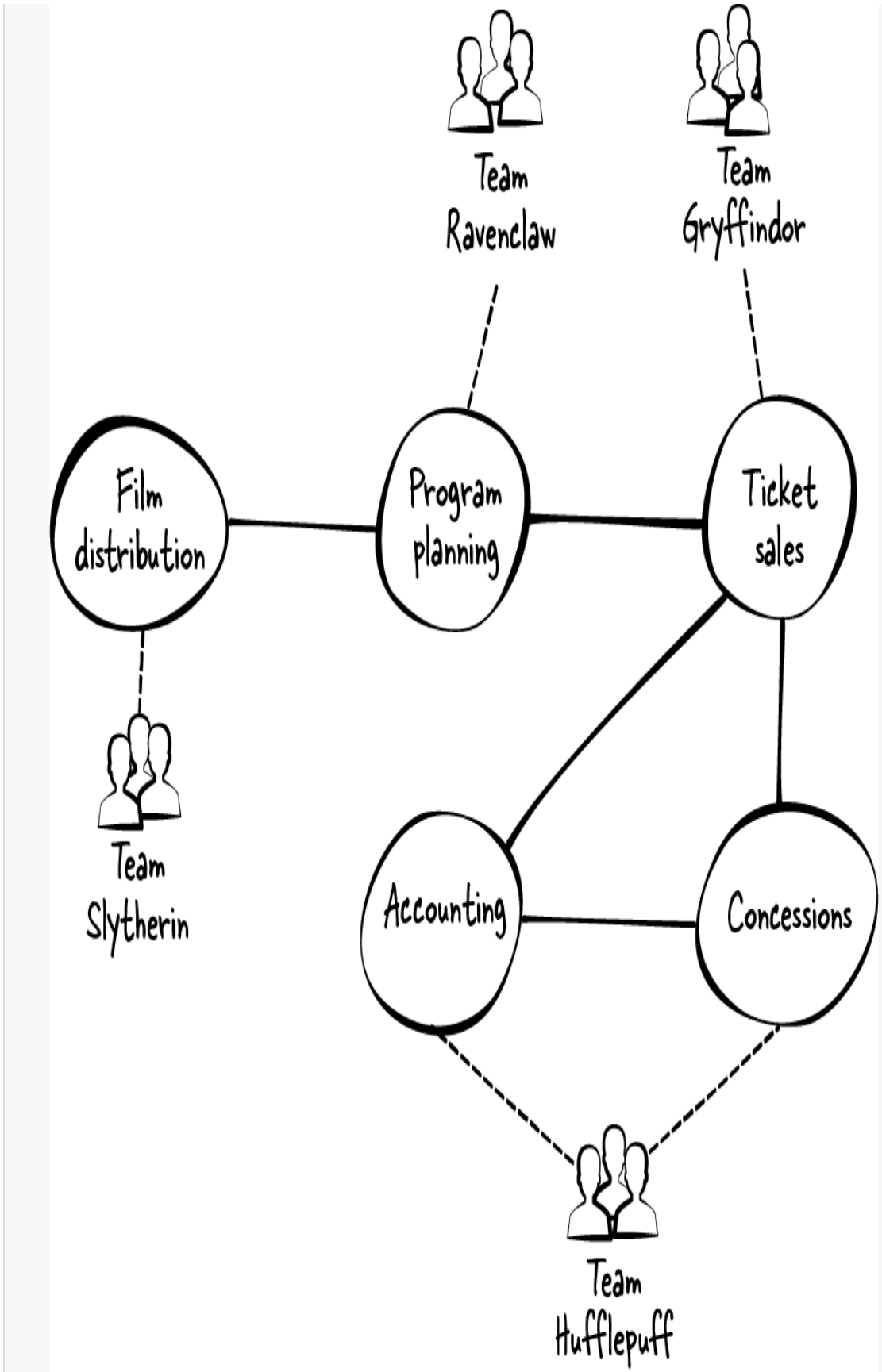

Team
Ravenclaw
Team
Gryffindor
Film
distribution
Program
planning
Ticket
sales
Team
Slytherin
Accounting
Concessions
Team
Hufflepuff

Figure 9-5. Stream-aligned teams in the context map for the cinema.

Platform team

Such a team provides an underlying platform to support the stream-aligned teams. The key here is that the platform provides services that simplify the complexity of the underlying technology or domain, reducing the cognitive load on the stream-aligned teams to enable them to work quickly.

The goal of a platform team should be to create a platform that is "just good enough." It also should be the "thinnest viable platform" (TVP), that is, not just a thin platform, but the thinnest platform possible.

CASE STUDY "CINEMA"

IN THE CINEMA: PLATFORM TEAM

Cine2001 is currently still a monolith and we don't have an underlying platform that we can have a separate team working on. In "Architecture Modernization Enabling Teams" we will see that only by decomposing Cine2001 will a platform team emerge.

Complicated subsystem team

These teams have a specific mission for a subsystem that is too complicated to be handled by a normal stream-aligned team or platform team.

It is important to note that these teams are formed when needed and broken up after a short period of time, because again, the goal is to reduce the cognitive load on the stream-aligned teams. These teams work on a part of a bounded context that is later integrated into the area of responsibility of the stream-aligned team that is responsible for the bounded context.

CASE STUDY “BANK”

IN THE BANK: COMPLICATED SUBSYSTEM TEAM

In the bank, an AI module is introduced for assessing customer credit risk when concluding loan agreements. To set up this AI module, it makes sense to pull AI experts together in a team for a limited period of time to solve this task.

Enabling team

These teams support the stream-aligned teams as well as the teams from the other topologies in learning new techniques and technologies. The goal here is for the stream-aligned teams to maintain focus on their mission while increasing their effectiveness.

The enabling teams consist of experts in a field or technology. They have time to try out new topics important to the stream-aligned teams and to research questions that have arisen there.

CASE STUDY “CINEMA”

IN THE CINEMA: ENABLING TEAM

For the planned transformation of Cine2001 into a microservices architecture, our development teams will need a lot of know-how that they currently do not have. Our development teams will learn modern technologies, such as containers and messaging systems, from corresponding enabling teams.

From the description of the different Team Topologies, the following becomes clear: the entire system is focused on the stream-aligned teams. The three other kinds of teams act with the goal of supporting the stream-

aligned teams in such a way that they experience more autonomy and less cognitive load. The bottom line is that these three teams are service providers for the stream-aligned teams, and at least the complicated subsystem teams are teams of short duration.

The topologies are visualized using a custom notation shown in Figure 9-6.

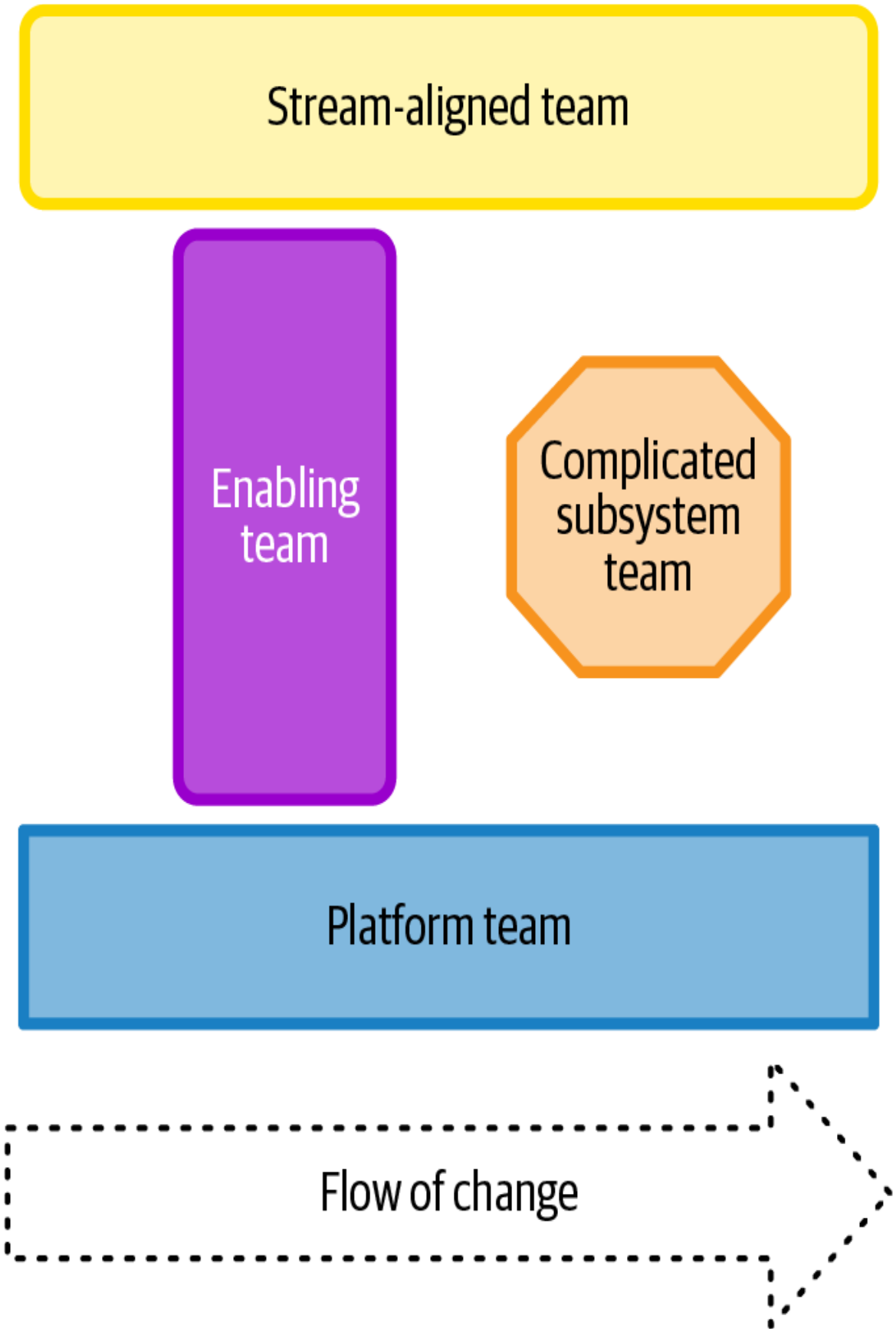

Stream-aligned team
Enabling team
Complicated subsystem team
Platform team
Flow of change

Figure 9-6. Graphical representation of the different topologies.

Team Interaction Modes

The second point that the Team Topologies approach contains is how these topologies should work together. In the description of the individual teams, it became clear that the teams must work together; otherwise the enabling team would not be able to support the stream-aligned teams, and the complicated subsystem and platform teams would not be able to do any work for the stream-aligned teams. The Team Topologies approach describes three basic types of working together (called *team interaction modes*):

Collaboration

Two teams work closely together to quickly communicate and share ideas for a defined purpose. This type of working together leads to a lot of interaction between the teams, sometimes even blurring the boundaries between them.

The effect is that the cognitive load increases and the autonomy of the teams decreases. Therefore, collaboration should be treated with caution and should only take place in the short term.

This form of team interaction has similarities to what is called partnership in DDD (see "Context Mapping"). In a partnership, two teams can only be successful together and that can lead to their boundaries becoming blurred sooner or later.

X as a service (XaaS)

A team provides a library, component, or the like as a service to one or more other teams. The service should be usable by the other team without much interaction or input.

DDD also has similarities to this kind of team interaction. In "Context Mapping", we met the open host service and upstream-downstream relationships. In context mapping, the former provides a service to

many possible consumers, while the latter describes a power relationship where the upstream team has the power to define the interface. The downstream team consumes that interface. XaaS team interaction, which aims to help the stream-aligned team work faster, corresponds to the customer-supplier version of upstream-downstream team interaction.

DDD also has a term for the technical implementation of XaaS interaction: the open host service, possibly with published language.

Facilitating

Finally, there is one type of team interaction that is not described in DDD's context mapping: facilitating. This team interaction mode is short-term and occurs when one team supports and coaches another. Typically, enabling teams collaborate with other teams in this way.

The interaction modes are visualized as shown in Figure 9-7.

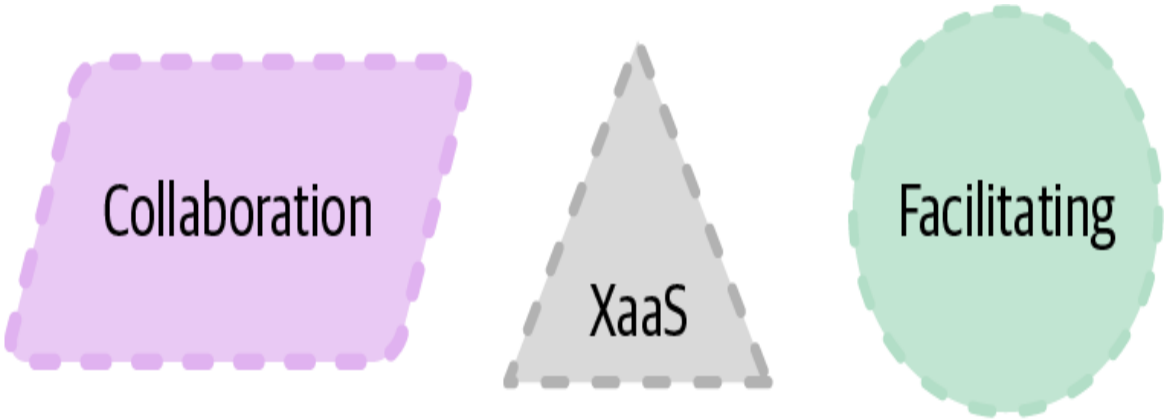


Figure 9-7. Graphical representation of the team interaction modes.

CASE STUDY “CINEMA”

IN THE CINEMA: TEAMS AND THEIR TOPOLOGIES AND INTERACTION MODES

What do the topologies of the teams look like for our program cinema example with its core domains “ticket sales” and “program planning,” its supporting subdomain “concessions,” and the generic subdomain “accounting”?

We have the context map (see Figure 9-5) in place, which shows how we want to divide the teams in the future, and we have acquired knowledge about how software development currently works:

- Our planned stream-aligned teams are currently working as one team on our monolith Cine2001, which we plan to break up.
- There is an external team Slytherin for film distribution that we will not include in our considerations.
- Our future stream-aligned teams should learn modern technologies from an enabling team.
- Standard software should be purchased for the generic subdomain “accounting,” and that standard software should exchange data with Cine2001 via an anticorruption layer.

From this we develop the topologies in Figure 9-8.

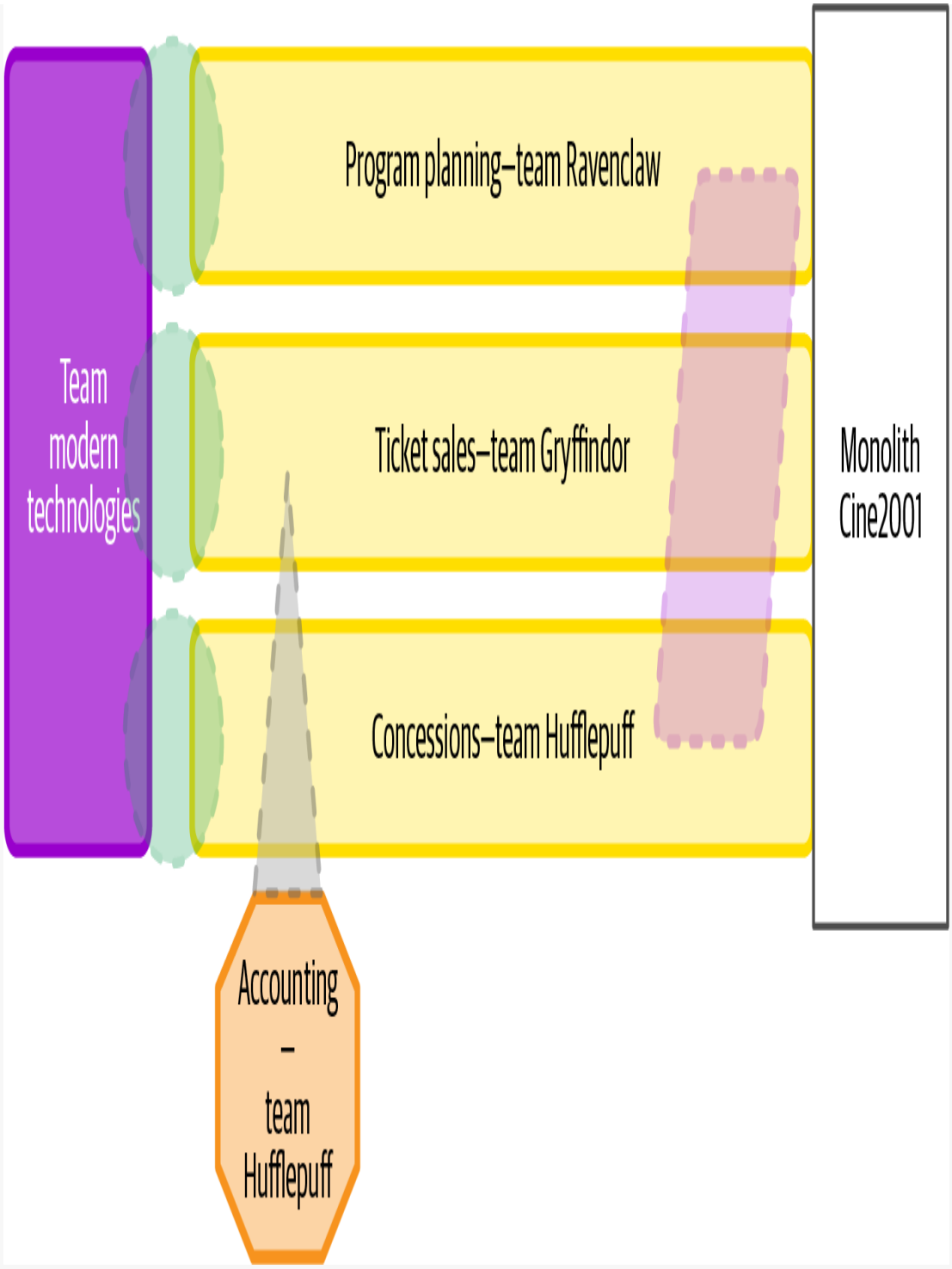


Figure 9-8. Topologies of the teams in the cinema (without time factor).

Since program planning, ticket sales, and concessions are still entangled in the monolith, our three teams working on these contexts will have to

collaborate with each other. The team interaction will be close, and they will constantly get in each other's way.

The work on the accounting system, i.e., its integration via an interface with the anticorruption layer with Cine2001, is done by a part of the Hufflepuff team as a complicated subsystem team. This will create an X as a service interaction with the Gryffindor team because ticket sales will want to send messages to accounting. Concessions will also need an X as a service interaction with accounting to book the sales of snacks and drinks.

Team Evolution Over Time

In Figure 9-8 we looked at the topologies at a particular point in time. But you can also use the Team Topologies approach to visualize interaction and the types of teams over time. In Figure 9-9, one platform team, one complicated subsystem team, two enabling teams, three stream-aligned teams, and their interactions are mapped as examples. Time or the "flow of change" runs from left to right.

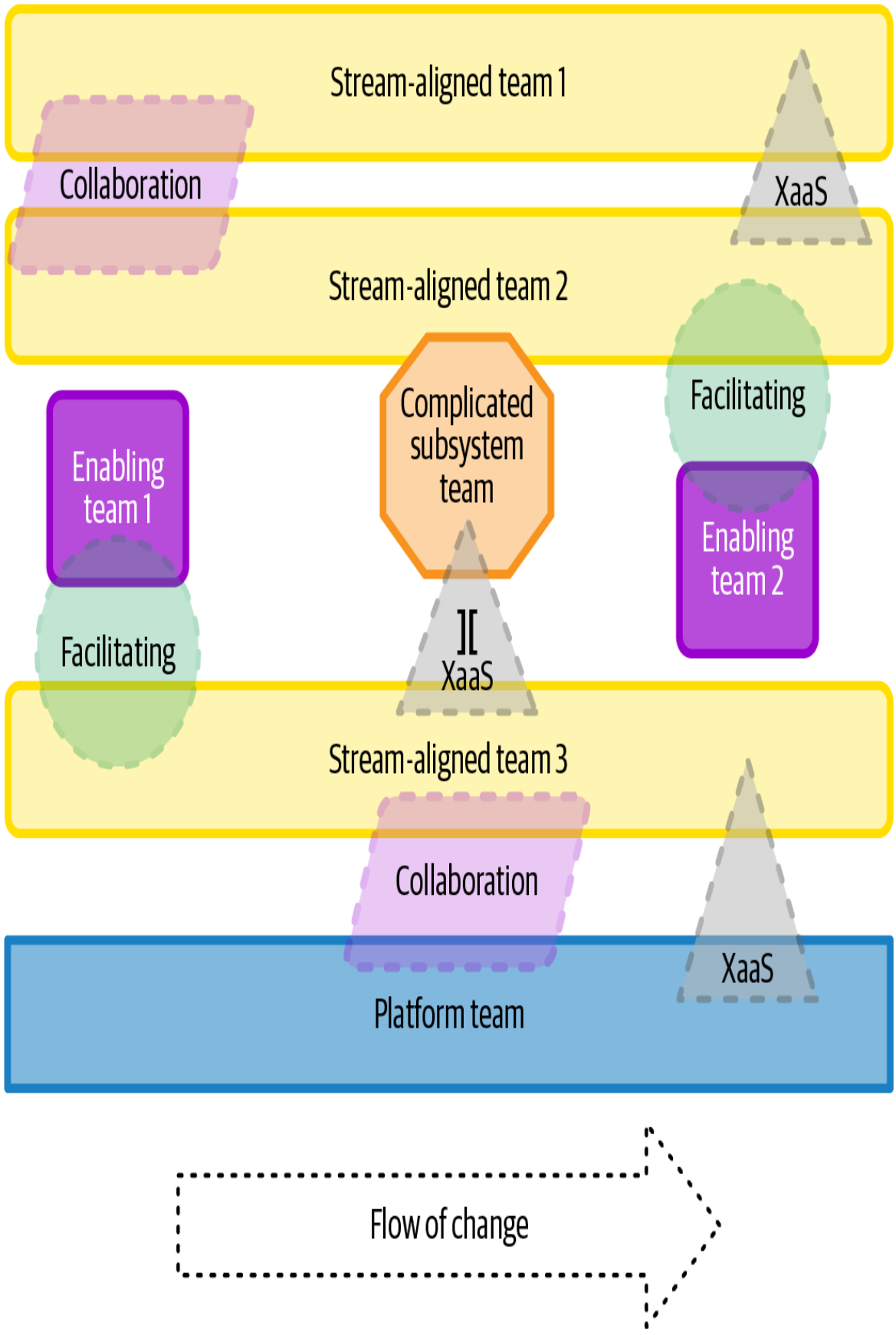

Stream-aligned team 1
Collaboration
XaaS
Stream-aligned team 2
Facilitating
Complicated subsystem team
Enabling team 1
Enabling team 2
Facilitating
XaaS
Stream-aligned team 3
Collaboration
XaaS
Platform team
Flow of change

Figure 9-9. Topologies of the teams in time.

Enabling team 1 assisted stream-aligned team 3 at the beginning, while enabling team 2 did not assist stream-aligned team 2 until the end of the time period shown here. Stream-aligned team 1 and stream-aligned team 2 worked closely together for some time—they collaborated. Stream-aligned team 3 was closely associated with the platform team for some time. In both cases, there was an XaaS interface between the two teams afterwards. They may have been working on these interfaces during the collaboration phase. Finally, in the middle, you can see a complicated subsystem team that provided an XaaS interface for stream-aligned team 3.

This temporal representation is suitable for organizations where the monolith is already decomposed and there are actually stream-aligned teams working separately. Our cinema is not there yet. In the next section, we will see how organizational development can be visualized together with the decomposition of the monolith in time.

Architecture Modernization Enabling Teams

To be look at the transformation of a monolith over time, Eduardo da Silva and Nick Tune have developed the approach *Architecture Modernization Enabling Teams* (see [da Silva and Tune 2023]), AMET for short. AMET targets the same problem as Domain-Driven Transformation: an enterprise has a large monolith that, because it is so entangled, is difficult and slow to change for the teams working on it. When the enterprise needs faster innovation, the knotty architecture becomes a bottleneck.

In their article, the authors recommend using a temporary architecture improvement team to drive modernization (or transformation, as we say in this book). Four broad steps are suggested, which we replicate here with the cinema example, echoing the graphics from [da Silva and Tune 2023]:

1. Business problem

In the initial state, the architecture is a blocker to the business strategy (see Figure 9-10). We have three teams that need to collaborate because

they are working on the same piece of software.

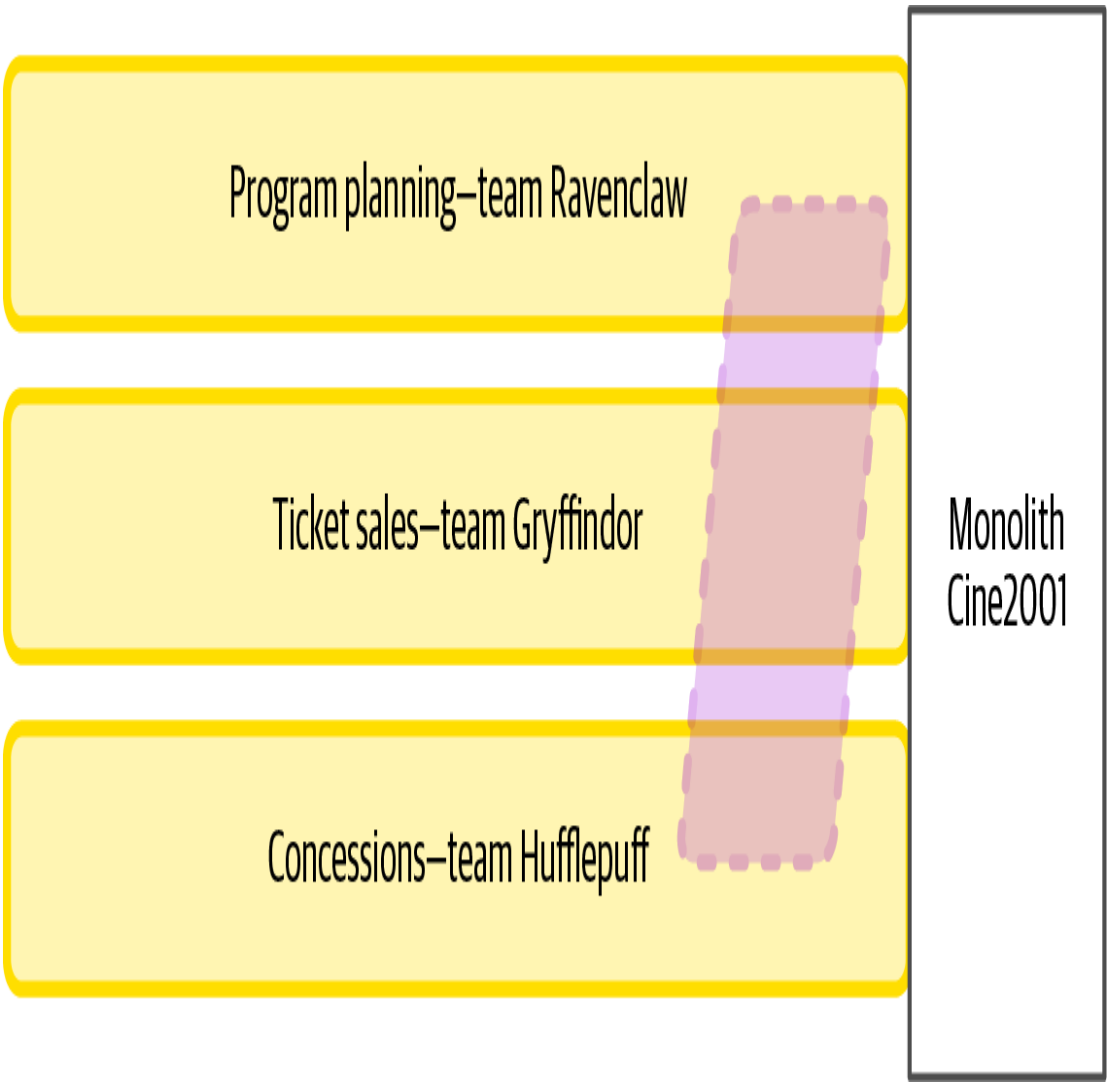


Figure 9-10. Initial state with monolith.

2. Kick-off for modernization

An AMET is formed to start the architecture improvement initiative. This brings together the appropriate people and establishes a first idea on how to proceed (see Figure 9-11).

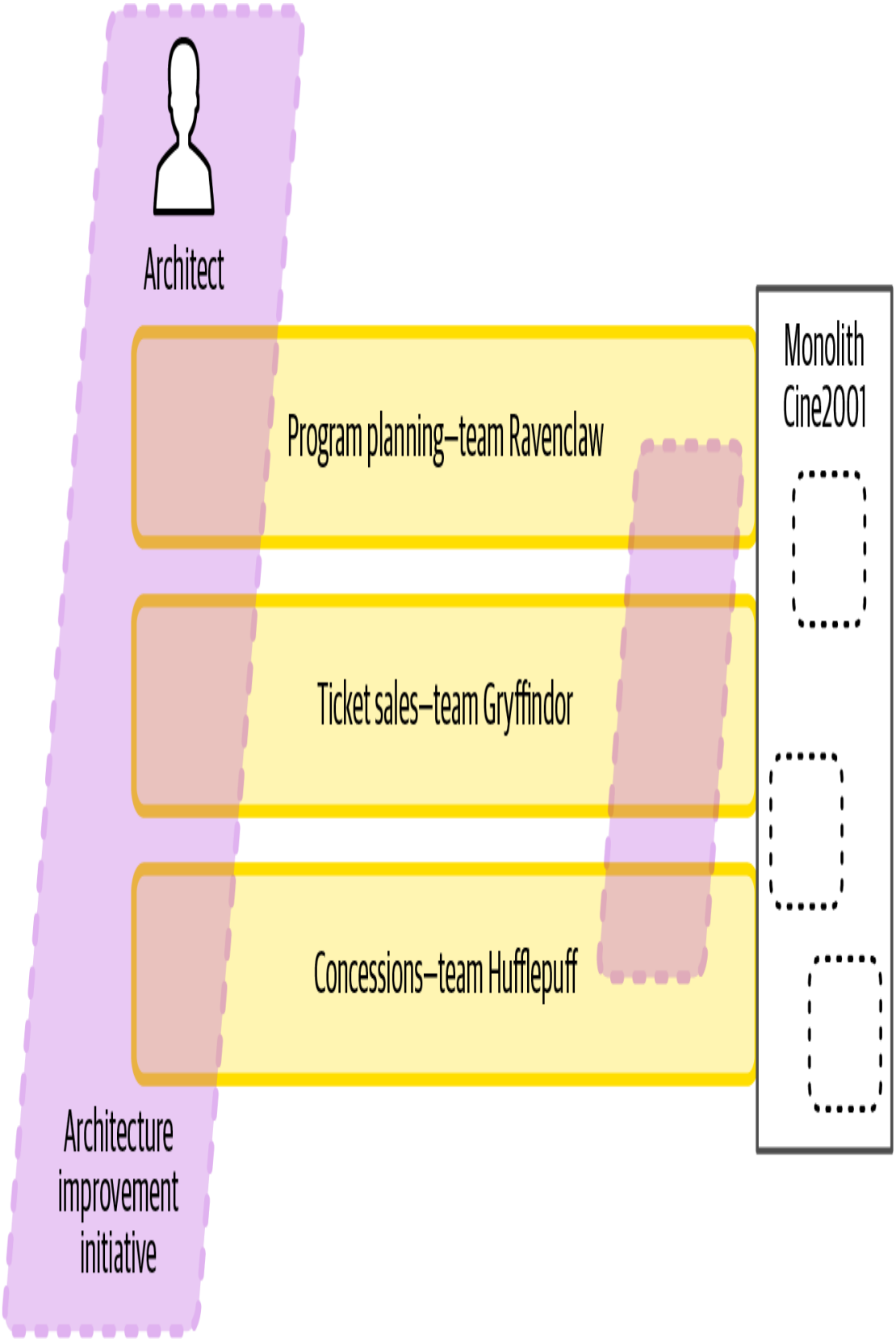


Figure 9-11. Architecture modernization initiative—kick-off.

3. Executing modernization

The AMET helps teams maintain the momentum of architecture improvement and design a better architecture (see Figure 9-12).

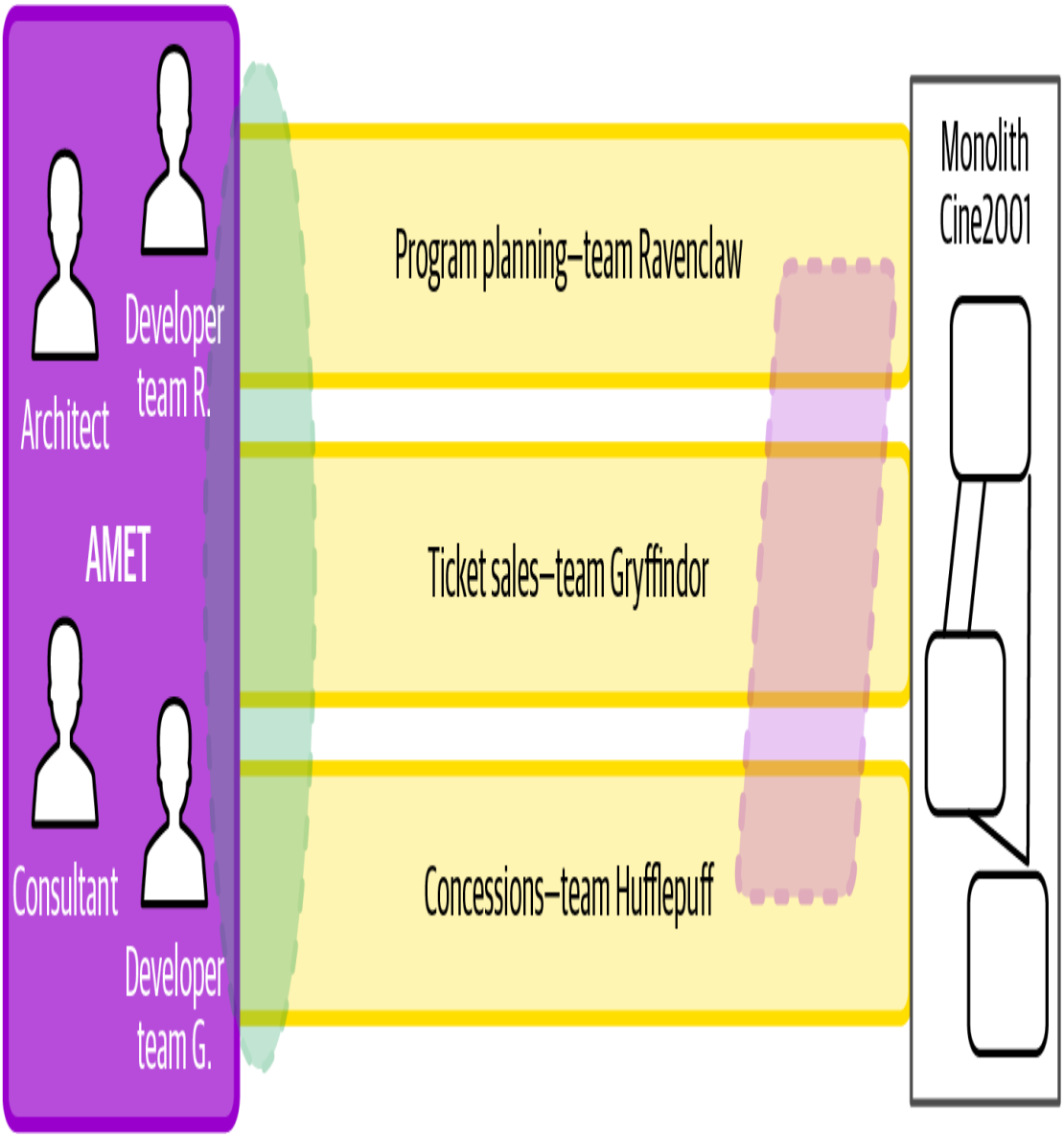


Figure 9-12. Architecture modernization initiative—execution.

4. End of modernization

Architecture is no longer a blocker to business strategy. The design has been improved for the long term and the remainder of the monolith are

maintained by a platform team. The AMET is broken up (see Figure 9-13).

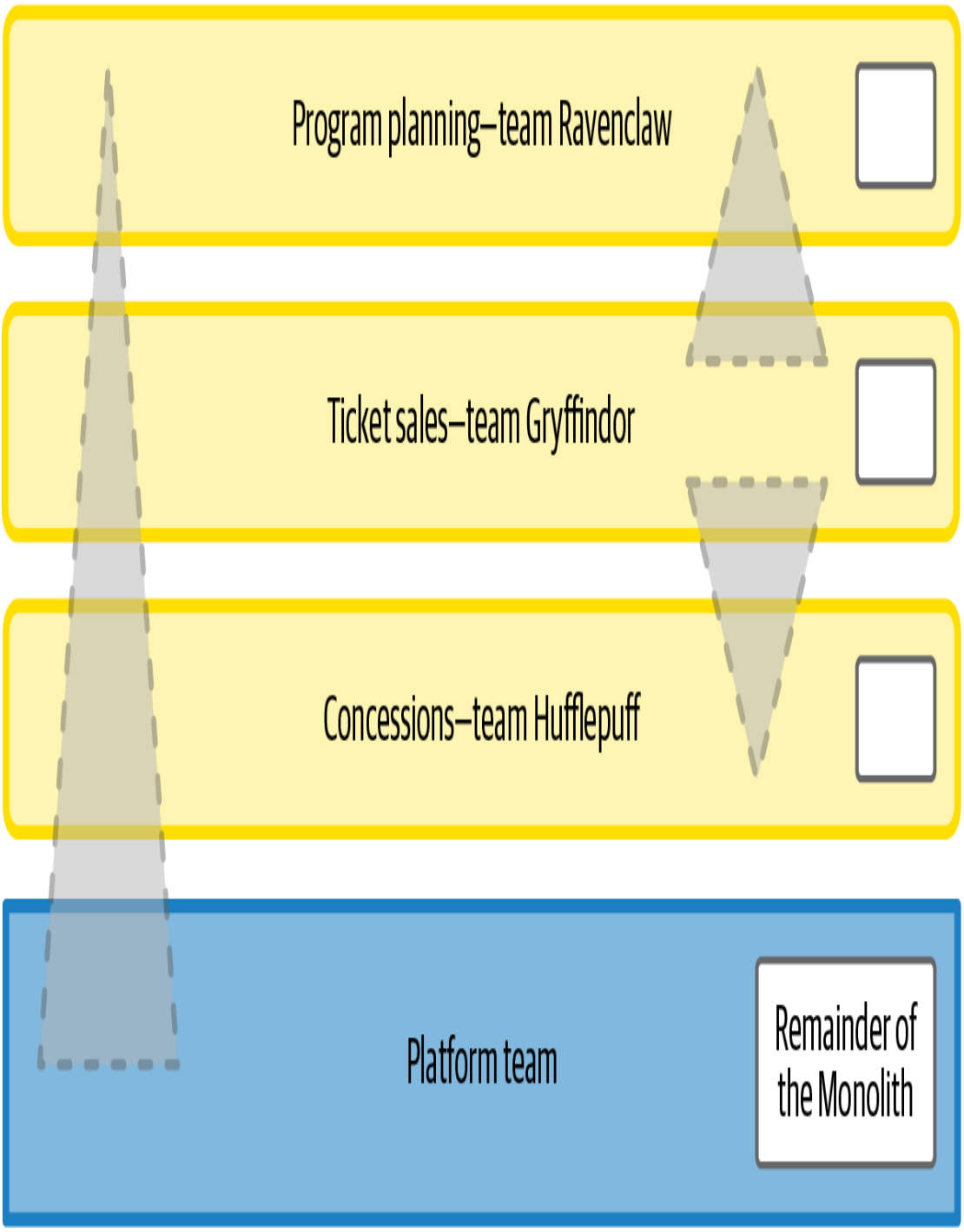


Figure 9-13. Architecture modernization initiative—finish.

In this sequence of steps, Tune and da Silva have provided us with a great template for Domain-Driven Transformation. A deeper presentation of the AMET idea can be found in Chapter 15 of Nick Tune's awesome *Architecture Modernization* [Tune and Perrin 2024]. Domain-Driven Transformation, as we presented it in this Part II and will develop further in Part III, explains what you need to do in detail in Step 3, i.e., the execution of modernization in order for the modernization described by da Silva and Tune to succeed.

Summary

In this chapter, we have provided you with concepts and arguments on how to initiate discussions in your organization about the existing team structures. In the process, you have learned about the connection between software architecture and team structure as well as Team Topologies.

The next part of this book will now focus on strategic Domain-Driven Transformation. The question that always comes up is: how can we use Domain-Driven Transformation to decompose our legacy system so that it becomes manageable again for our development teams and at the same time better supports the users?

Where Do You Stand?—Made Easy

Now we have learned some things about possible team organizations and how to improve them. Check your level in the area of team organization using the statements in Table 9-1. How strongly do each of these apply? You can enter your own assessment on a scale of 1 to 5, between (1) = "I do not agree at all" and (5) = "I agree completely."

Table 9-1. Assessment sheet on the subject of team organization

No.	Statement	1	2	3
1	Our teams are assembled with cross-functional members.			
2	Our organization has an idea of cognitive load and autonomy.			
3	Our teams consist of five to nine people and are designed for long-term collaboration.			
4	We have stream-aligned teams that implement parts of the domain, such as a bounded context.			

No.	Statement	1	2	3
5	We have teams that tend to be enabling, complicated subsystem, or platform teams.			
6	Our teams know which communication relationship they have with each other.			
7	Our enabling and complicated subsystem teams have already experienced that their support for the other teams does not have to be permanent at all.			

No.	Statement	1	2	3
8	Our platform teams have an understanding that their interfaces should be as simple as possible for the stream-aligned teams that build on them.			
9	Our teams are organized by UI/client, business logic/backend, and database.			
10	Our teams have to coordinate a lot to implement enhancements.			
11	We have teams that consist of more than 10 members.			

No.	Statement	1	2	3
12	The team composition is constantly changing.			

For the evaluation of your results, see Tables II-1 and II-2.

---

[1] We use the term *refactoring* here in a very broad sense. Alternative terms would be *reorganization* or *sociotechnical transformation*.

[2] Also called “Tuckman’s stages of group development” and first described in [Tuckman 1965].

Part III. Strategic Domain-Driven Transformation

In Part II, we looked at Domain-Driven Transformation both in the small (technically and tactically) and in terms of team organization. Now in this third part of the book, we will look at Domain-Driven Transformation in the larger context, i.e., at the decomposition of a legacy system. To do this, we want to understand the domain behind it so that we can then decompose it on the basis of the subdomains found.

In a sense, we have the opposite problem compared to building a new system on a greenfield site.

If you want to develop new software on a greenfield basis for an application area for which no specialized software yet exists, you have to find out which steps in the users' business processes—with paper and pencil, or with Word, Excel, and email—are really relevant and worth preserving. At this point, you will usually encounter process steps that can already be greatly simplified or even made superfluous without software, but even more so through the development of new software.

Software Development in the Brownfield

If there is already legacy software that has been used and developed over the years, then you are more likely to see this phenomenon: business processes have shifted over the years because of one or more of the following reasons:

- The organization of work has evolved.
- New services and products have been developed.

- Legal changes have occurred.

In legacy software, however, these shifts have not been fully replicated. Reasons for this are:

- Lack of budget and time
- Lack of understanding of changes and their appropriate implementation in the legacy software

Users have become accustomed to workarounds, and move parts of their work results to Word, Excel, or other external formats. Over the years, shadow IT developed around the legacy software, without which the current business processes cannot be understood.

Therefore, to decompose legacy software in a domain-driven way, we need to grasp and think through the current business processes with all their workarounds.

The Key: Domain (Re-)Discovery/(Re-)Understanding the Domain

It is therefore not enough to look at the solution alone—the existing source code. Rather, you have to start with the problem—in the domain—in order to unravel and simplify a software system that has been entangled over the years through additions and modifications. You should:

1. Mentally "put aside" the monolith and the structure existing in it for the time being.
2. Take another fundamental look at the subject matter, or more precisely at the division of the domain into subdomains.

Interestingly, at least analytically, we start with a greenfield, even though we already have a system. On the basis of this fresh understanding of the

domain, we are then able to identify the superfluous or misunderstood parts in the software solution and expand and modify them.

Therefore, the following applies to both the development of new software systems and the decomposition of monoliths:

- The critical activity is not just writing code, but first and foremost learning and understanding among all stakeholders.
- Developers need to learn what exactly the domain experts do in their daily work.
- The domain experts need to understand what the technical reasons are for constraints and prioritization in the solutions.

Overview of Strategic Transformation

If everyone is ready to focus on the domain without thinking about the software, then we can start with the requirements analysis! We usually go through the following steps with our customers and in our own projects:

- Step 1: Domain rediscovery (see Chapter 10)
- Step 2: Modeling the target architecture (see Chapter 11)
- Step 3: Alignment of the actual with the target architecture (see Chapter 12)
- Step 4: Prioritization and execution of the transformation measures (see Chapter 13)

The representation of the steps as a domain story is shown in Figure III-1.

The entire process is led by a *domain modeler*[1] and an *architecture analyst*.[2] They form the Architecture Modernization Enabling Team that we introduced in Chapter 9. In Figure III-1, two actors are shown for this purpose. However, these two roles can indeed be performed by one person

who has expertise in both areas. In addition, in Figure III-1 we see the domain experts as well as the development team as further actors.

The first step in the strategic Domain-Driven Transformation is that the domain is rediscovered by the domain modeler with the help of the domain experts and the development team (step 1). In this process, the domain understanding is renewed and the slices in the domain are discussed so that subdomains become apparent.

In the middle of the diagram, the target architecture is modeled in step 2. In step 2, the domain-oriented slices from step 1 serve as the basis for creating a target context map. In this step, both the domain modeler and the architecture analyst are involved with their respective expertise.

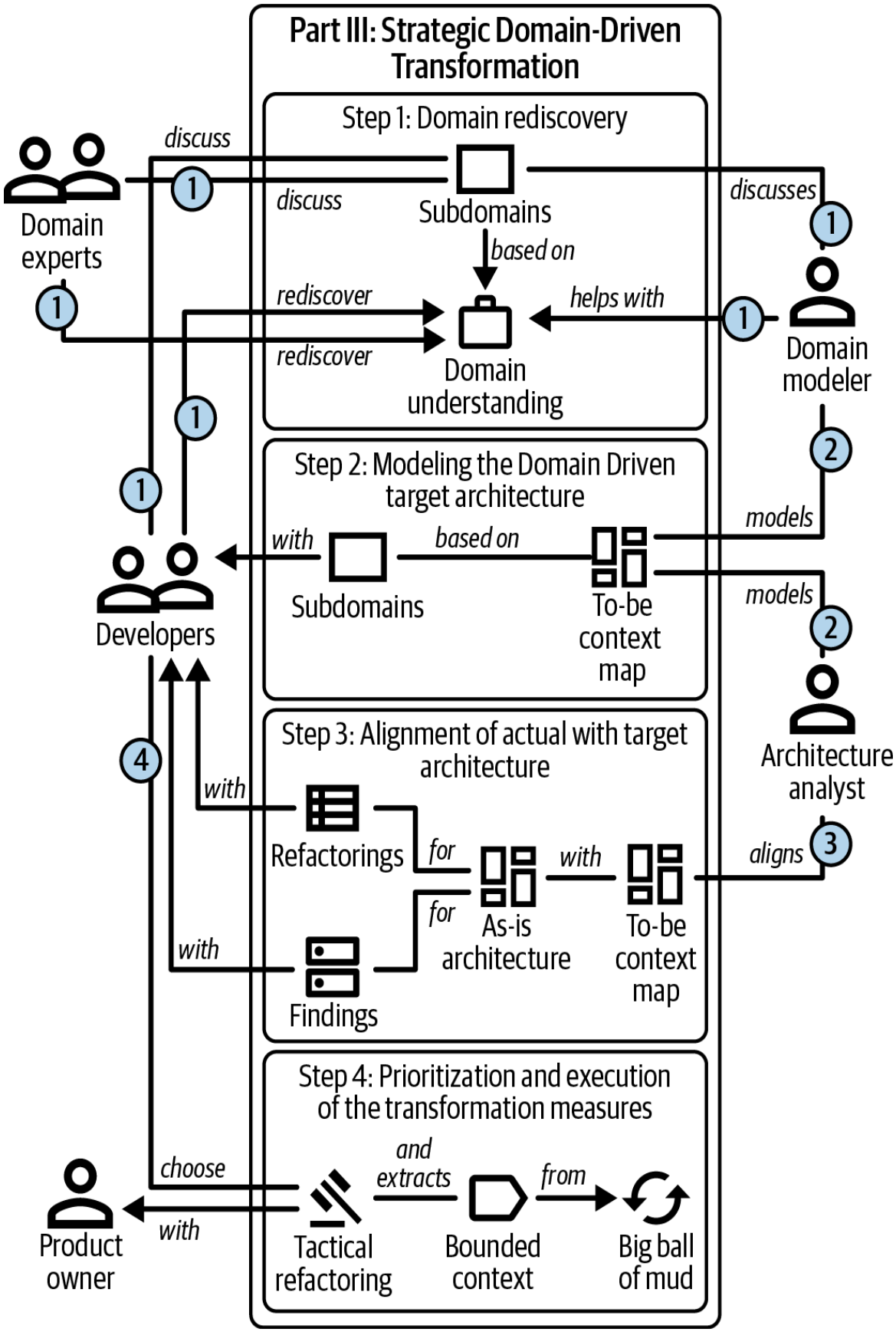

Part III: Strategic Domain-Driven Transformation
Step 1: Domain rediscovery
Step 2: Modeling the Domain Driven target architecture
Step 3: Alignment of actual with target architecture
Step 4: Prioritization and execution of the transformation measures
Domain experts
discuss
discuss
Subdomains
based on
discusses
rediscover
rediscover
helps with
Domain understanding
Domain modeler
models
Developers
with
Subdomains
based on
To-be context map
models
Architecture analyst
aligns
with
Refactorings
for
for
As-is architecture
with
To-be context map
with
Findings
choose
with
Product owner
and extracts
from
Tactical refactoring
Bounded context
Big ball of mud

Figure III-1. Procedure for decomposition with DDD.

In step 3, the target context map is overlain on the actual architecture, and a list of refactorings is derived. At the same time, the architecture analyst discusses with the development team the prioritization of bounded contexts based on the context map.

Finally, step 4 follows, in which the development team prioritizes the refactorings together with the product owner and extracts the bounded contexts from the source code using the prioritized refactoring list.

Of course, these steps are repeated iteratively, just as we discussed in "The Method". We are not presenting a waterfall procedure here!

The Structure in This Part of the Book

In the following chapters, we look at the individual steps based on the following questions:

- Which *actors* are involved in this step?
- Which *substeps* does it consist of?
- Which *tools* do we use?
- Which *results* do we generate?
- What *goal* are we pursuing in this step or *for what reason* are we taking this step?
- What *challenges* do we experience in implementing this step in our practice?

Where it makes sense, we refer to the appropriate *domain-driven refactorings* from our catalog in the appendices.

Examples are particularly important for understanding, so in addition to the cinema example and the bank example, we introduce another example: the legacy software of Alphorn Auto Leasing, Inc.

In the following chapters of Part III, we look at the individual steps in detail. We begin with the reunderstanding of the domain.

---

1 This role is often filled by an external DDD consultant, business analyst, requirements engineer, or domain architect.

2 This role is often filled by an external architecture consultant or solution architect.

Chapter 10. Step 1—Domain Rediscovery

In this chapter we will cover the first step of strategic Domain-Driven Transformation: domain rediscovery. To rediscover the domain, we need to recapture and understand the business processes in the domain. The goal is to uncover how the business really works together with domain experts, developers, and domain modelers—for the time being independently of existing software.

We have shown the coarse process of strategic transformation in Figure III-1. In Figure 10-1 we go one level deeper and see the substeps that make up step 1.

Domain modelers, domain experts, and the development team come together in a workshop and use methods from Collaborative Modeling (see Chapter 4) to rediscover the business domain. The goal of this workshop is to clarify:

- *How* the subject matter experts work (business processes)
- *What* they work with (work items/work objects and domain concepts)
- *What* it is exactly (domain language) that they work with
- *Why* the domain experts do or need these things

Here it makes sense not to dive into every detail, but to stay on a coarse- to medium-grained level, otherwise the discussion will get out of hand. So you will hear the domain modeler say time and again: “This is getting into too much detail now; let’s get an overview first!”

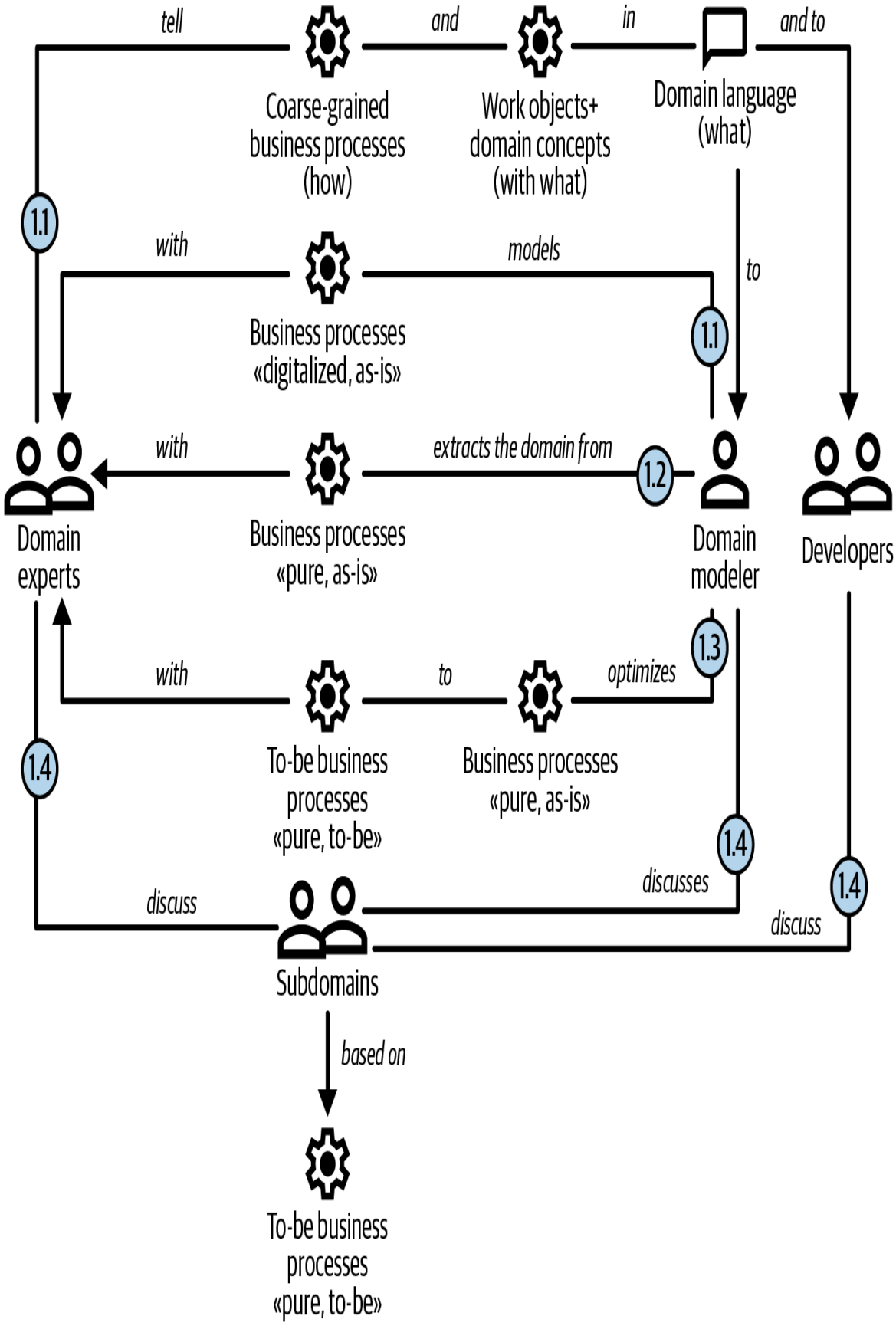

tell
and
in
and to
Coarse-grained business processes (how)
Work objects+ domain concepts (with what)
Domain language (what)
1.1
with
models
to
Business processes «digitalized, as-is»
1.1
with
extracts the domain from
1.2
Domain experts
Business processes «pure, as-is»
Domain modeler
Developers
1.3
with
to
optimizes
1.4
To-be business processes «pure, to-be»
Business processes «pure, as-is»
1.4
discusses
1.4
discuss
discuss
Subdomains
based on
To-be business processes «pure, to-be»

Figure 10-1. Strategic transformation—step 1 in its substeps.

In Figure 10-1 you can see the substeps that are necessary to sharpen the understanding of the business processes more and more. First (substep 1.1), the actual business processes are recorded in a digitalized version. Then the domain is extracted (substep 1.2), so that pure as-is processes are created. Then, the as-is processes are optimized into to-be processes (substep 1.3). Finally, the domain modeler, domain experts, and the development team begin to discuss possible subdomains on the basis of the to-be business processes (substep 1.4).

Schematic Description of Step 1

Step 1 includes the following elements:

Actors

- Domain experts
- Development team
- Domain modeler

Substeps

1. Getting to know the as-is situation (substep 1.1)
2. Extracting the domain (substep 1.2)
3. Optimizing the business processes to to-be processes (substep 1.3)
4. Finding subdomains in the to-be business processes (substep 1.4)

Tools

- Facilitation materials (e.g., EventStorming set)
- Collaboration tools (e.g., Miro)
- Process modeling tools (e.g., Egon.io)

Results

- As-is and to-be business process
- Subdomains

Goals/reason

- Bring domain knowledge back to the foreground
- Clear away “solution dust”

Challenges

- Gathering the right domain experts
- Leading domain experts back to the actual domain
- Making sure not to decompose by work items
- Finding the right size

TIP

In this chapter, and in the following chapters on the other three steps, you will find alternating challenges and substeps at the same level of the chapter structure. In this chapter, there are four substeps (1.1–1.4) and four challenges. We have placed the challenges in each case before, between, or after the substeps where they tie in with the content.

CASE STUDY “CAR LEASING”

A THIRD EXAMPLE DOMAIN: CAR LEASING

What is the business of the Alphorn Auto Leasing, Inc.? Those who do not have enough liquid funds available (or want to take advantage of accounting benefits) do not have to buy a car, but can instead lease it from Alphorn. For this purpose, Alphorn then buys the car from the manufacturer and leases it to the customer (called *lessee*) for a monthly installment.

Alphorn has branches in various cities and an IT system in use that has grown over the last 25 years: *Monolease*. Recently, Alphorn started to use the C4 Model (see Chapter 5) and created a context diagram for Monolease (see Figure 10-2).

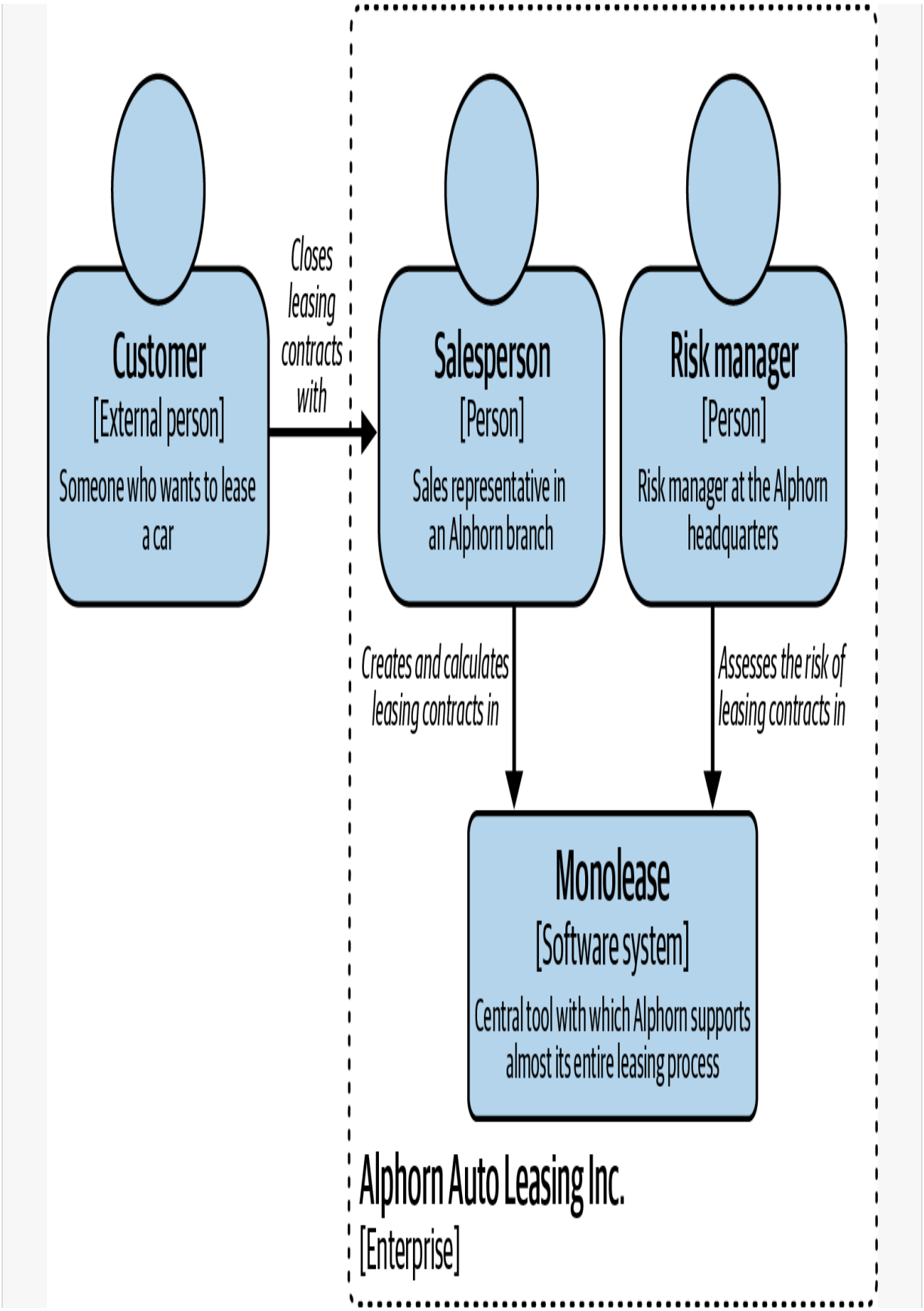

Customer
[External person]
Someone who wants to lease a car
Closes leasing contracts with
Salesperson
[Person]
Sales representative in an Alphorn branch
Risk manager
[Person]
Risk manager at the Alphorn headquarters
Creates and calculates leasing contracts in
Assesses the risk of leasing contracts in
Monolease
[Software system]
Central tool with which Alphorn supports almost its entire leasing process
Alphorn Auto Leasing Inc.
[Enterprise]

Figure 10-2. Monolease in context (in the sense of the C4 Model).

Monolease has served faithfully over the years (or decades) and has been expanded again and again. Over a long period of time, a lot of technical debt has built up and the system now suffers from various problems that no longer fit into the new times:

- Further development has become slow and error-prone.
- Initially, there was only one development team, now there are several. These teams often "get in each other's way."
- Each branch of Alphorn has its own installation of Monolease; new versions have to be rolled out individually. As a result, installations regularly have different versions.
- The new IT board therefore wants to move the system to the cloud, and all branches should then be able to access this central instance. The present monolithic structure makes it difficult to take advantage of the cloud.[1]

To eliminate these problems, or at least mitigate them, the monolith is to be modernized and modularized. To do this, Alphorn is bringing in a team of experts to help: us—Carola and Henning.

Challenge: Gathering the Right Domain Experts

To some of our customers, a further analysis of the domain appears to be an additional and seemingly superfluous step. A typical statement is: "We want to improve the software, don't we? Why do you need the domain experts? They have better things to do than to explain again how they use the software! Besides, there's documentation and a bunch of BPMN diagrams describing the processes."

But that kind of thinking is too short-sighted! By (re)entering via the subject domain and without thinking about the existing software, we deliberately focus on the problem space—for which the software is supposed to be a solution. For this, we absolutely need the domain experts, because they can explain to us why they perform certain activities in their business processes.

Challenge: Leading Domain Experts Back to the Actual Domain

In Chapter 2 we learned about the *problem*-inherent complexity in the problem space and the *solution*-dependent complexity in the solution space, and that there is an essential and an accidental part in each case. In both the problem space and the solution space, we want to bring our software back to the essence. An essence that typically lies buried under a thick coat of solution "dust" in both areas (see Chapter 8). So we need to free ourselves from the dust of time in two areas:

- In the problem space, the *essential* domain knowledge (see Chapter 2) is waiting to be rediscovered (*domain rediscovery*).
- In the solution space, the *implemented essential* domain knowledge (see Chapter 8) is dormant in the source code, dusted with a lot of accidental complexity.

Especially in areas where software has been used for a long time, it is difficult to see why a certain activity is undertaken. Do we do it because it is rooted in the actual domain, or because "the system wants it that way"? This "because the system wants it that way" is the solution dust that we have to clear away in order to get to the actual domain knowledge.

IN THE CAR LEASING EXAMPLE: DOMAIN REDISCOVERY

At Alphorn, a new risk manager is hired. During the onboarding process, a colleague tells her that he is filling in a "field" in the "E37 mask." The new risk manager doesn't understand what that means, but doesn't want to look uninformed and nods with a friendly smile. A few weeks later, the risk manager becomes bolder and asks the colleague, who clarifies: "Oh, what I mean by that is when I've calculated the resale value, I'll note it in the credit rate assessment."

Separating the two areas of problem space and solution space is not always easy. It is difficult for many users who use the software in their daily work as well as for many developers who (further) develop the software. That's why teams often have more success if they can involve a domain modeler. This helps them rediscover the core of the domain and clear up the misunderstandings about the domain that have arisen over the years among all stakeholders.

Substep 1.1: Getting to Know the As-Is Situation

As just discussed, the domain experts are often strongly fixated on their IT system at the beginning of the analysis. This is because they know their work in an IT-supported fashion and can also only imagine it with precisely this IT system. In this situation, it is advisable to model the first processes with the IT system—in other words, the digitalized as-is. In this case, the experts are asked: "Tell me how you are working now with the help of the IT system."

The very first substep we take is to do an as-is modeling, and we work with the domain experts and the development team to come up with a common

idea of how the process works with IT systems today. During the as-is modeling, we have to ensure intensive communication between the development team and the experts, because the development team often does not have a clear idea of what the domain experts actually do with their system. The domain stories developed in this process have the scope (coarse-grained, digitalized, as-is; see Chapter 4), and through them the work on a glossary of the ubiquitous language begins (see Chapter 3).

Since developers are trained to find a solution as quickly as possible when working on a problem, the domain modeler must pay particular attention to ensuring that the developers focus on the as-is situation and are not already thinking about the future. After all, if they hear how cumbersome it is for the experts working with the existing system, the developers will quickly go into the solution space.

Substep 1.2: Extracting the Domain Knowledge (Purification)

Our experience is that for decomposing legacy systems, a focus on the domain without software as actor(s) works best. This is because to find subdomains, we need the "untainted" domain-centric process. Untainted (i.e., pure) domain stories are especially helpful when transforming or replacing legacy software systems. They allow us to understand a domain without having to deal with the complexity added by existing software. The domain stories developed in this process have the scope (coarse-grained, pure, as-is).

To do this, we start with the digitalized as-is processes modeled in substep 1.1 and take out the digital aspects. What are these "digital aspects"? They are defined as:

- Software systems
- Steps in the process that are taken only to satisfy the systems, but have no basis in the domain

Various techniques can help with taking out the digital aspects of the processes, for example, the following thought experiment.

Technique: Thought Experiment “Paper Process”

The idea here is to imagine the process running completely without IT support. In other words, only “with paper,” hence “paper process.” We examine the process step-by-step, i.e., in every sentence of domain stories, and in every command/event in EventStorming. For each step—especially for those that run in the system or are supported by it—it is important to ask:

- How was this done in the past when there was no IT?
- Do you know of any companies that still do the process manually? How do they work there?
- How would the step be done paper-based without IT support?
- If we work paper-based, does this step still need to be done?

At first, this can feel a little strange. A typical dialog might go like this:

> *Domain modeler*: “How would that be done on paper?”

> *Developer (rolls eyes)*: “There’s an email being sent, it can’t be done on paper!”

> *Domain modeler*: “OK, why is an email being sent? What is the content of this email?”

> *Domain expert*: “There’s an order confirmation in there.”

> *Domain modeler*: “Aha, then instead of previously modeling ‘system sends email,’ I now model ‘system sends order confirmation.’”

The key is to be persistent. After a while, workshop participants will be able to understand the idea and see the value.

Technique: Thought Experiment “Virtual Actors”

To make the process entirely explicit, it sometimes helps to introduce actors who don’t even exist in real life. A typical dialog might go as follows:

> *Domain modeler*: “Well, ‘system sends order confirmation’ still has the system in it. Who sent the order confirmation in the past when there was no system?”

> *Domain expert*: “That was not done then. That would have been much too time-consuming and is only possible because we have computer support.”

> *Domain modeler*: “OK, if we imagine time and effort don’t matter —who would do it, what would the role be called?”

Domain expert: “Hmm, that might be the ‘confirmer.’”

Domain modeler: “I then model another virtual actor ‘confirmer,’ and instead of ‘system sends order confirmation’ now write ‘confirmer sends order confirmation.’”

Virtual actors are a first indication in the direction of a possible subdomain in the business process. Often, a virtual actor later becomes a subdomain of its own.

The two lastly described techniques help to purify the process. This is not always necessary.

HENNING'S FIELD STORY: FACILITATING VIRTUAL ACTORS TO MODERNIZE A COMPILER

A team that builds safety compilers for machine automation approached me to do some Domain Storytelling sessions with them. To be honest, I was skeptical that this modeling method would be the right one for such a technical domain. After all, Domain Storytelling had been conceived to model the behavior of people, not of compilers, right? Anyway, they convinced me to try it and we started out.

A couple of very successful modeling sessions later, I was proven wrong. We had managed to get a very deep understanding of the process of both the people that ensure the safety of the machines and the safety compiler they use to do so.

Besides people actors like "plant engineer" and "automation technician," we had found virtual actors like "parser," "lexer," and "OTA provider" (the last one handled the over-the-air deployment of the compiled code to the controllers in the machine). The virtual actors helped us to understand the process and also to find the boundaries in the domain that later were carved out as bounded context in the modernized system.

Alternative: Modeling Pure in the First Place

If you are lucky, the workshop participants are able to distinguish between problem space and solution space. This can be the case, for example, if they already have experience with Collaborative Modeling. Then you can skip the modeling of the digitalized as-is and start right away with the pure version. We often additionally use the question techniques of the domain modeler from "Substep 1.2: Extracting the Domain Knowledge (Purification)".

Let's take a look at this using the car leasing example.

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: DOMAIN STORY

Alphorn had already modeled its processes in BPMN before we arrived for optimization, and subsequently had a good understanding of the digitalized as-is processes. We now also analyze the business side with several coarse-grained pure domain stories.[2] The coarse process of the happy path looks as shown in Figure 10-3.

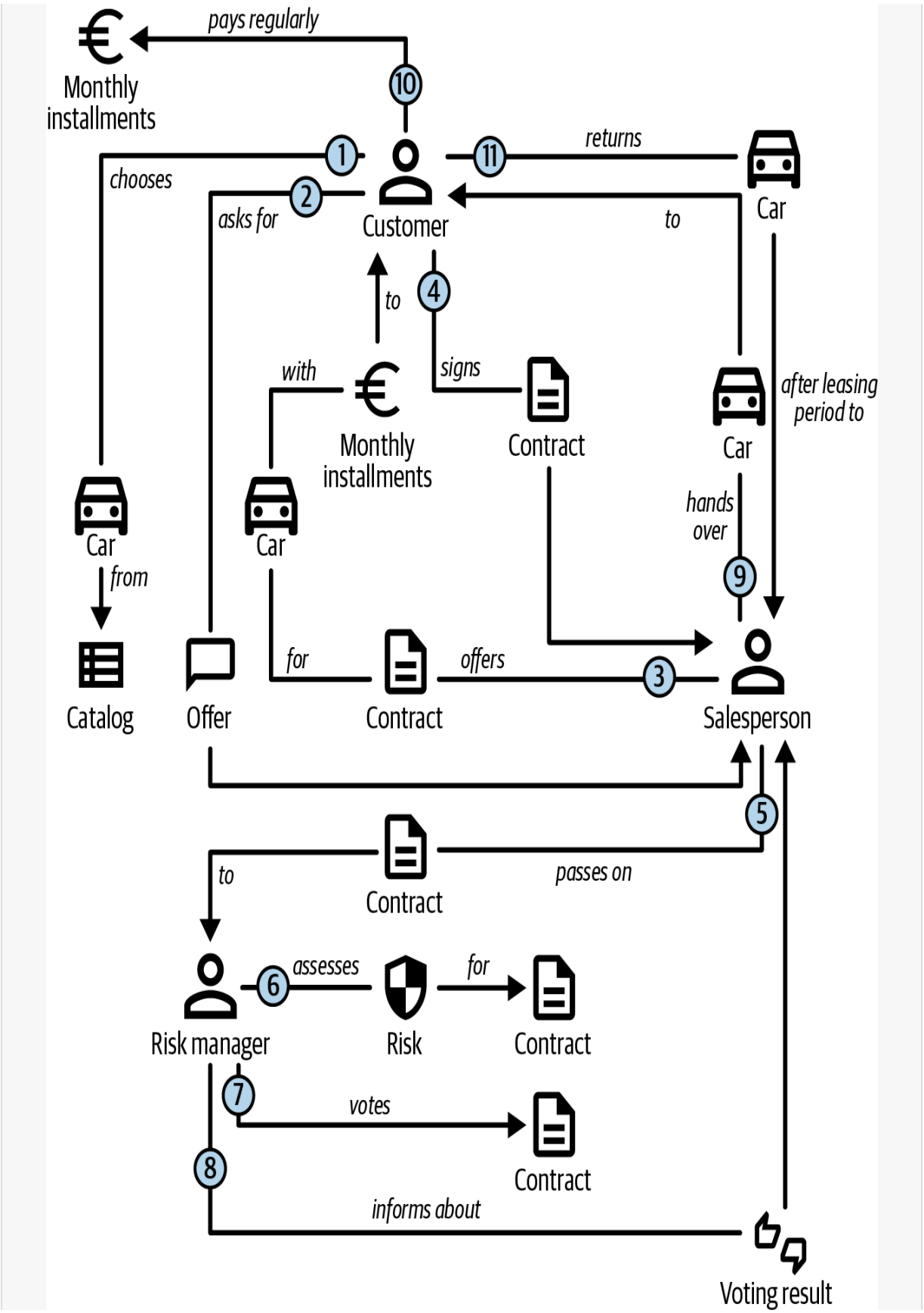

pays regularly
Monthly installments
10
1
chooses
2
asks for
Customer
11
returns
Car
to
4
to
with
Monthly installments
signs
Contract
Car
after leasing period to
Car
from
Car
hands over
9
Catalog
Offer
for
Contract
offers
3
Salesperson
5
to
Contract
passes on
Risk manager
6
assesses
Risk
for
Contract
7
votes
Contract
8
informs about
Voting result

Figure 10-3. The coarse-grained process at Alphorn Auto Leasing, Inc.

Distilling Ubiquitous Language

Parallel to extracting the domain knowledge in the business processes, we dedicate ourselves to the ubiquitous language and pay particular attention to the following questions (see "Ubiquitous Language"):

- Where is the *same term used differently*?
- Where do *synonyms* of terms actually mean the same thing?
- Where is domain language polluted by *technical terms*?

This clarity in language will help us significantly in substep 1.4 when searching for subdomains.

Substep 1.3: Optimizing the Business Processes

After we freed the as-is process(es) from technology in the last section, the optimization toward to-be processes now follows. The domain stories developed in this substep have the scope (coarse-grained, pure, to-be).

Usually, clues have already surfaced in the previous substeps where the as-is processes can be improved. Typical clues and questions are:

- Are there redundant steps in the as-is process?
- Are all these roles needed for the work?
- Is there a better solution that eliminates steps in the process?
- How can the shadow IT discovered during modeling be reduced or replaced with artifacts that should be in the system or could be added to it?

In the discussion among the domain modeler, domain experts, and development team, these questions are talked about and the to-be processes are modeled.

CASE STUDY "CAR LEASING"

IN THE CAR LEASING EXAMPLE: OPTIMIZING THE PROCESS

At Alphorn, optimization had already taken place in the course of the BPMN modeling. On this basis, it is decided that Figure 10-3 is good enough as a basis for the next steps.

Substep 1.4: Finding Subdomains in the Business Processes

Having started with the as-is processes (substep 1.1 in Figure 10-1), brought them back to the domain-oriented core (substep 1.2), and optimized them into to-be processes (substep 1.3), we can now identify the subdomains in the to-be processes and thus decompose the domain (substep 1.4).

The crucial questions here are:

- What belongs together in the business process to form a unit?
- What separates sections in the business process from each other?

In *Domain Storytelling* we ask ourselves: which activities belong together from the point of view of the actors? Which activities start a new subprocess? Which activities can be executed in parallel with others independently?

In *EventStorming*, these questions translate into: which events form a related sequence? Which events initiate a new stream of events? Which events happen independently of those before and after them? These events are also called pivotal events (see [Baas-Schwegler and Heimeshoff 2021]).

In *Scenario Casting*, the question asked is: at which points in the orientation scenario is a mission accomplished (and thus a subprocess permanently completed)?

Indicators for Subdomain Boundaries

Attention should be paid to the following indicators, which from our experience signify that it is possible to derive the subdomains in a domain from the modeled business process:

Completed work results

> In one part of the overall process, a work object is processed to such an extent that it is passed on to another subprocess as the finished result of this subprocess. There it can be used, but also further processed. This further processing can also continue in a series of subsequent subdomains.

Passing-on work results one-way

> A finished work result is passed on from one subprocess to another without there being any way back. This means that there is no collaboration, but rather successive processing.

Groups of domain experts

> There are groups of domain experts, each of whom is well acquainted with one subprocess, but only knows the other subprocesses from the outside and knows nothing about the details of the work steps there.

Definition of domain terms

> Domain terms from the ubiquitous language (see "Distilling Ubiquitous Language") have different meanings in different subprocesses; i.e., domain experts from two different subprocesses each associate something (maybe only slightly) different with the same domain term.

Handling of work objects

In different subprocesses, different actions are performed with domain objects or terms that have the same name.

Rhythms and triggers

Subprocesses of the overall process are carried out in different rhythms and start based on their own triggers in each case.

Let’s run through these indicators in two of our examples.

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: THE INDICATORS

At Alphorn, we decompose the domain story from Figure 10-3 and draw boundaries between the subdomains (see Figure 10-4). If you are interested in how this works in detail, you can read about it in [Hofer and Schwentner 2022, chap. 10].

The indicators help us in several ways:

Completed work results and passing-on work results one-way

> The contract is negotiated between the salesperson and the customer, and finally the contract is concluded by a signature. In step 5 in Figure 10-4, the salesperson passes the signed contract to the risk manager so that they can evaluate the risk. The contract is further processed by the risk manager by voting it and informing the salesperson about it.

Groups of domain experts

> The salesperson and the risk manager come from two different groups of experts. The salesperson has the expertise to bring products to customers, for which they have received the appropriate training. Whereas the risk manager may be a mathematician. Such different groups of domain experts are often found in separate departments in large companies.

Definition of domain terms and handling of work objects

> Salespeople and risk managers work on the same contract, even the same piece of paper. But they have different ideas about it and also take different actions on it. For the salesperson, the contract is something where they tweak the many parameters until the customer agrees. The goal is to get a signature on the paper. The risk manager, on the other hand, sees the contract and its parameters as unchangeable. For them, the only question to answer is: “approve or reject?”

Rhythms and triggers

> Finally, the last indicator is also easy to find. The monthly installment is obviously paid exactly once a month. Contracts are offered whenever a customer asks for one, so typically several times a month. The risk assessment takes place only for a signed contract and not for all the offered contracts that were never signed. So this is all done at different rhythms.

Figure 10-4 shows only the happy path for the time being. To capture the domain well, other domain stories with other detailed flows need to be considered. For example, what happens in the following cases:

Voting result negative (steps 7 and 8)

> Then a guarantor is requested in the subdomain "offering" to reduce the risk.

Payment not made (step 10)

> Then the customer is reminded; in this we discover another subdomain, "dunning."

For reasons of space, we do not depict these stories here.

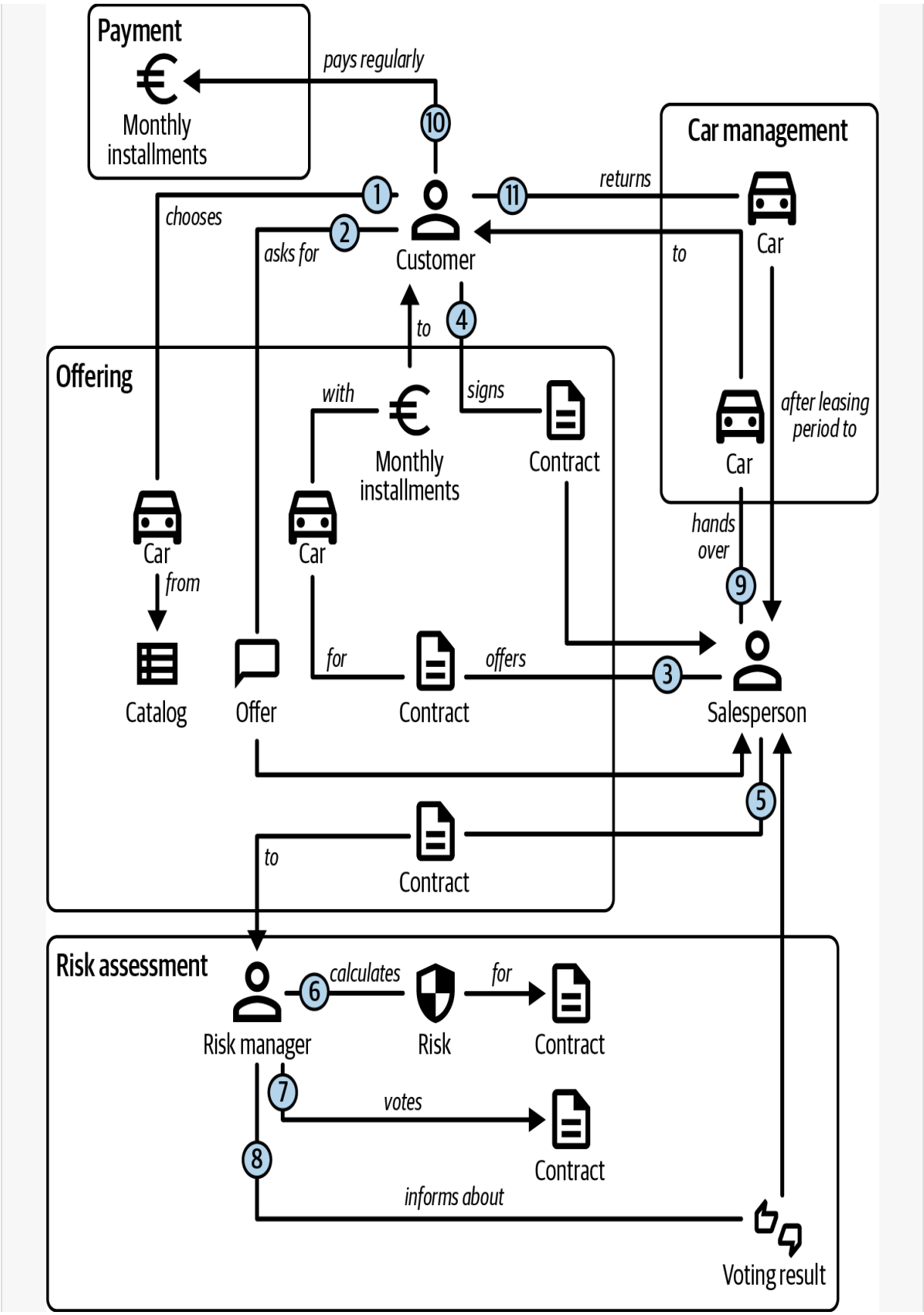

Payment
Monthly installments
pays regularly
10
Car management
returns
11
1
chooses
2
asks for
Customer
to
Car
to
4
Offering
with
Monthly installments
signs
Contract
Car
after leasing period to
Car
Car
hands over
9
from
Catalog
Offer
for
Contract
offers
3
Salesperson
5
to
Contract
Risk assessment
Risk manager
6
calculates
Risk
for
Contract
7
votes
Contract
8
informs about
Voting result

Figure 10-4. The coarse-grained process at Alphorn Auto Leasing, Inc. with subdomains.

What is the impact of the indicators in the cinema domain?

CASE STUDY “CINEMA”

IN THE CINEMA EXAMPLES: THE INDICATORS

In the cinema, we had found subdomains in Chapter 3 and drawn them into the domain story in Figure 4-8 (shown again in Figure 10-5). All the mentioned indicators can be found in our coarse-grained process for the cinema:

Completed work results and passing-on work results one-way

> The weekly schedule is the work item that the cinema manager works on during program planning. In step 4, the manager passes it on to the other subdomain "ticket sales." Just like in car leasing, there is no way back. Unlike in the car leasing example, the weekly schedule is not processed further in the "ticket sales" subdomain, but forms the basis of the work in this subdomain.

Groups of domain experts

> At the small art house cinema, Metropolis, there are two experts: the cashier at the box office and the cinema manager, each as an individual. However, in the cinema chain, Clintwood, there are a lot of cashiers and a separate cinema manager for each cinema. So there are groups of domain experts who know their respective subprocess very well, but only have a rough knowledge of the other subprocesses.

Definition of domain terms and handling of work objects

> The weekly schedule is also a good example of the different meanings of domain terms or work items and the use of them as an indication of subdomains. In the subdomain "program planning," the weekly schedule is the central work item, which, in addition to the planned shows, contains a lot of other information, such as the advertising planned before the film, the deployment of the cleaning staff and ice cream vendors, and certainly the expected number of visitors. Thus, the cinema manager has a rich handling of the weekly schedule. In the "ticket sales" subdomain, the weekly

schedule is also important, but it only contains the shows. If the weekly schedule here would also show the advertising blocks, the moviegoers would probably only come to the beginning of the movie and, as a consequence, the advertising revenues of the cinema would be reduced. In the subdomain “ticket sales” we therefore only have a simplified weekly schedule, which serves as an information medium for the moviegoers, allows much less handling, and has a completely different meaning for the cashier and the moviegoers than for the cinema manager.

Rhythms and triggers

Also in the cinema, there are different rhythms and triggers: the weekly schedule is planned once per cinema week by the cinema manager, while the ticket sales subprocess is run whenever a customer arrives. For the program planning subdomain, the respective trigger is step 1 in Figure 10-5, when the advertising agency sends the booking schedule for the advertising. For ticket sales, the trigger is step 6 when the moviegoer comes to the theater and studies the weekly schedule.

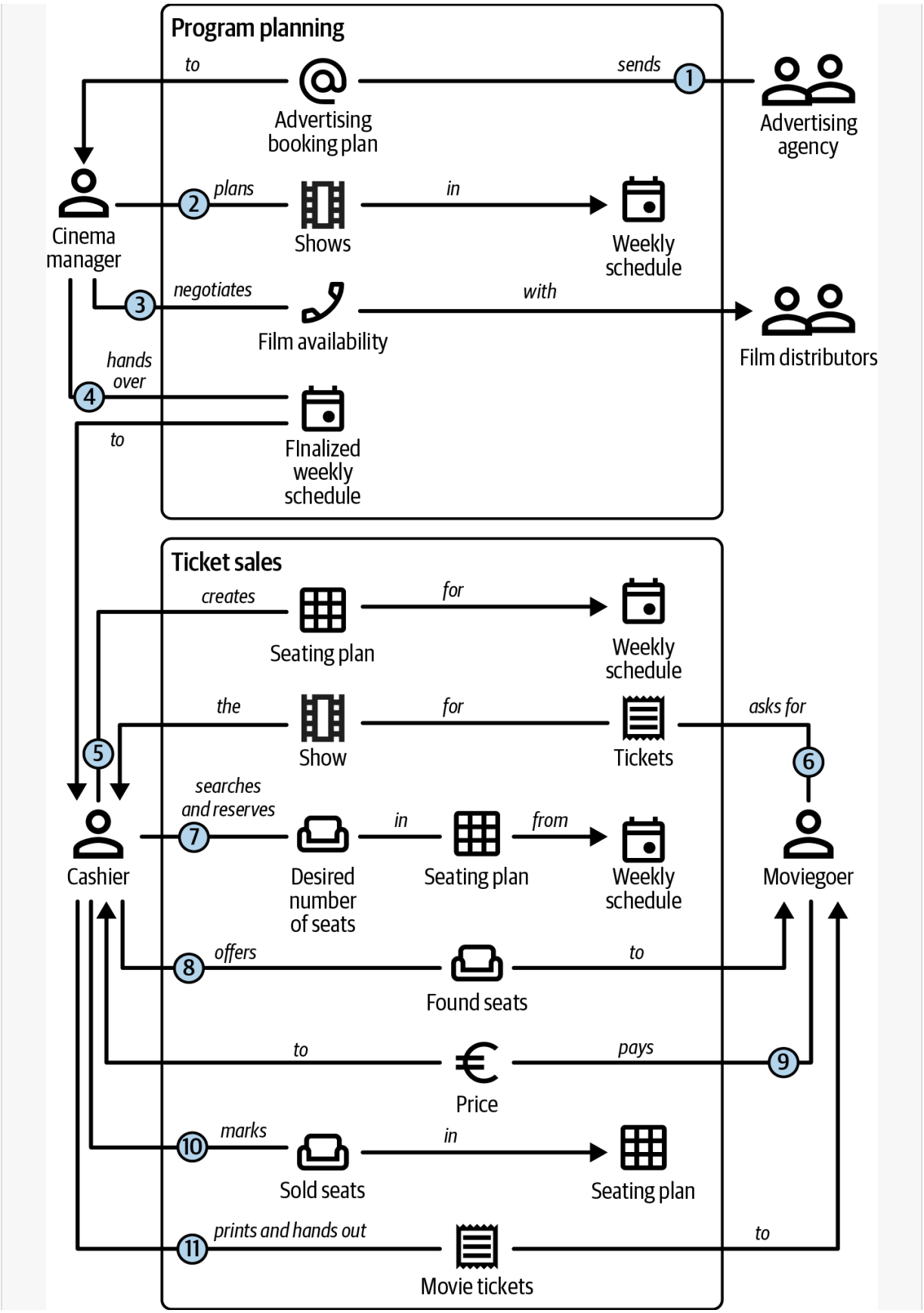

Program planning
to
Advertising booking plan
sends
1
Advertising agency
Cinema manager
2
plans
Shows
in
Weekly schedule
3
negotiates
Film availability
with
Film distributors
hands over
4
to
FInalized weekly schedule
Ticket sales
creates
Seating plan
for
Weekly schedule
the
Show
for
Tickets
asks for
6
5
searches and reserves
7
Cashier
Desired number of seats
in
Seating plan
from
Weekly schedule
Moviegoer
offers
8
Found seats
to
to
pays
9
Price
marks
10
Sold seats
in
Seating plan
prints and hands out
11
Movie tickets
to

Figure 10-5. Once again: overall process in the art house cinema with subdomains.

Challenge: Do Not Decompose by Work Items

In the last section, we learned about various indicators that point to places in the processes where a boundary should be drawn around a subdomain in the business process.

One contraindicator we encounter frequently is one that you should avoid at all costs with your teams: make sure you don’t decompose your processes by work items or work concepts or, in technical terms, entities or aggregates.

Especially when a legacy system is in place, teams repeatedly yield to the temptation to draw their subdomain boundaries based on the entities, or aggregates implemented in the system.

CASE STUDY “CINEMA”

IN THE CINEMA EXAMPLE: POOR DECOMPOSITION

For our cinema, this would mean that we would have a “weekly schedule” subdomain and a “seating plan” subdomain, where all the work with one or the other would happen respectively. If we group the domain story from Figure 4-7 accordingly, it would then look like the one shown in Figure 10-6.

In the upper subdomain, the cinema manager works on their weekly schedule. But if the cashier wants to open the seating plan from the weekly schedule, then the other subdomain must be used. This becomes visible by a bad smell in domain stories: namely that step 5 does not remain within a subdomain, but an arrow crosses the subdomain boundaries.

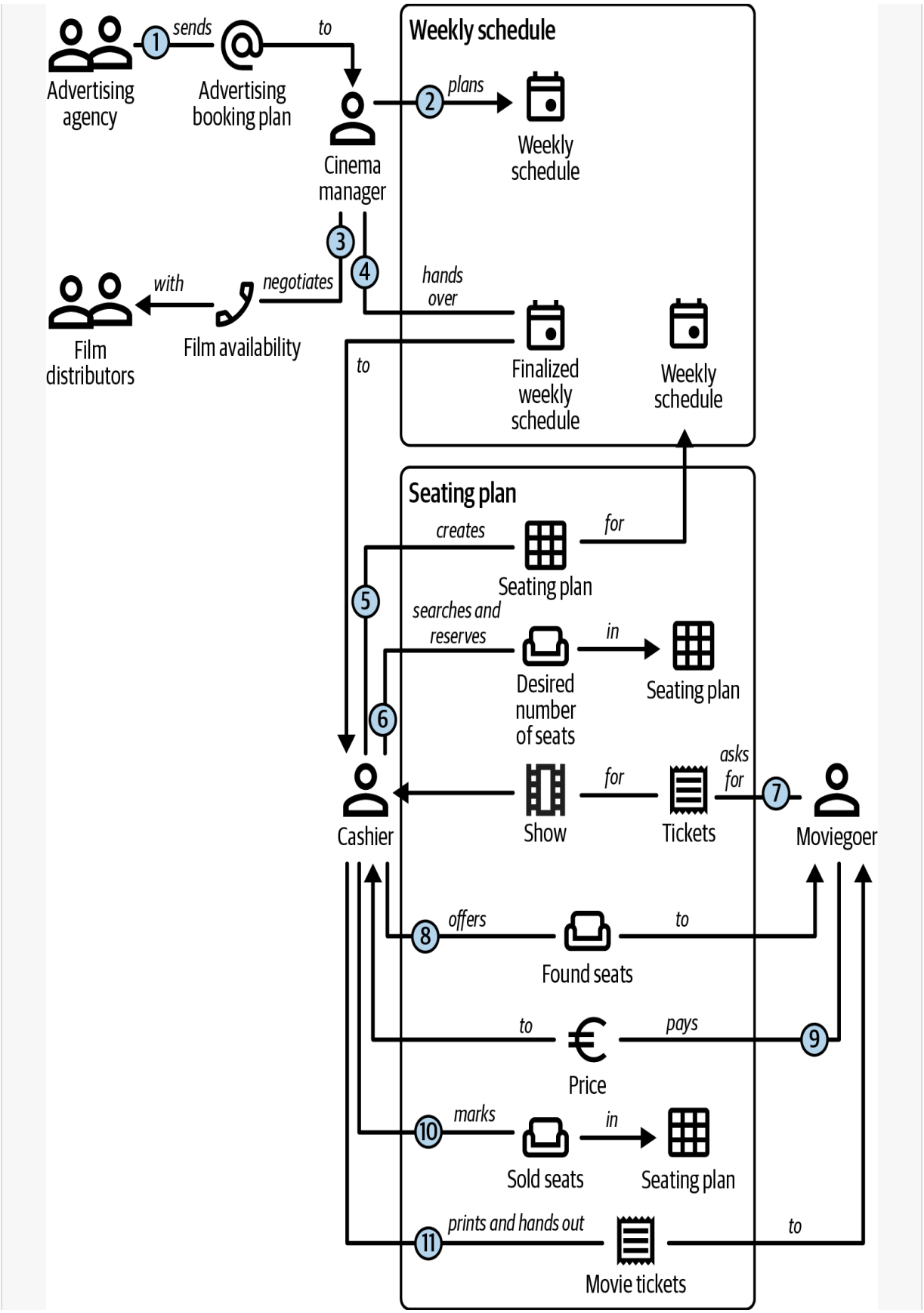

Weekly schedule
Advertising agency
1
sends
Advertising booking plan
to
Cinema manager
2
plans
Weekly schedule
3
4
with
negotiates
Film distributors
Film availability
hands over
to
Finalized weekly schedule
Weekly schedule
Seating plan
creates
for
Seating plan
5
searches and reserves
in
Desired number of seats
Seating plan
6
Cashier
for
Show
Tickets
asks for
7
Moviegoer
8
offers
to
Found seats
to
pays
9
Price
10
marks
in
Sold seats
Seating plan
11
prints and hands out
Movie tickets
to

Figure 10-6. Not recommended: division of context according to work items or entities in the coarse-grained cinema process.

The picture looks even more dramatic if you imagine the split in a software system. In Figure 10-7 you can see schematically how a mapping of the subdomains as "aggregate contexts" would look in the cinema. There are the two contexts "weekly schedule" and "seating plan," each with its aggregate, and a service that operates on the aggregate. Here is what is difficult about this design:

- The UIs for program planning and ticket sales are outside the contexts, because at least the ticket sales UI needs both services to be able to display the weekly plan to the cashier.
- In addition, the `SeatingPlanService` calls the `WeeklyScheduleService` in order to get the finished weekly schedule for the creation of the seating plans.

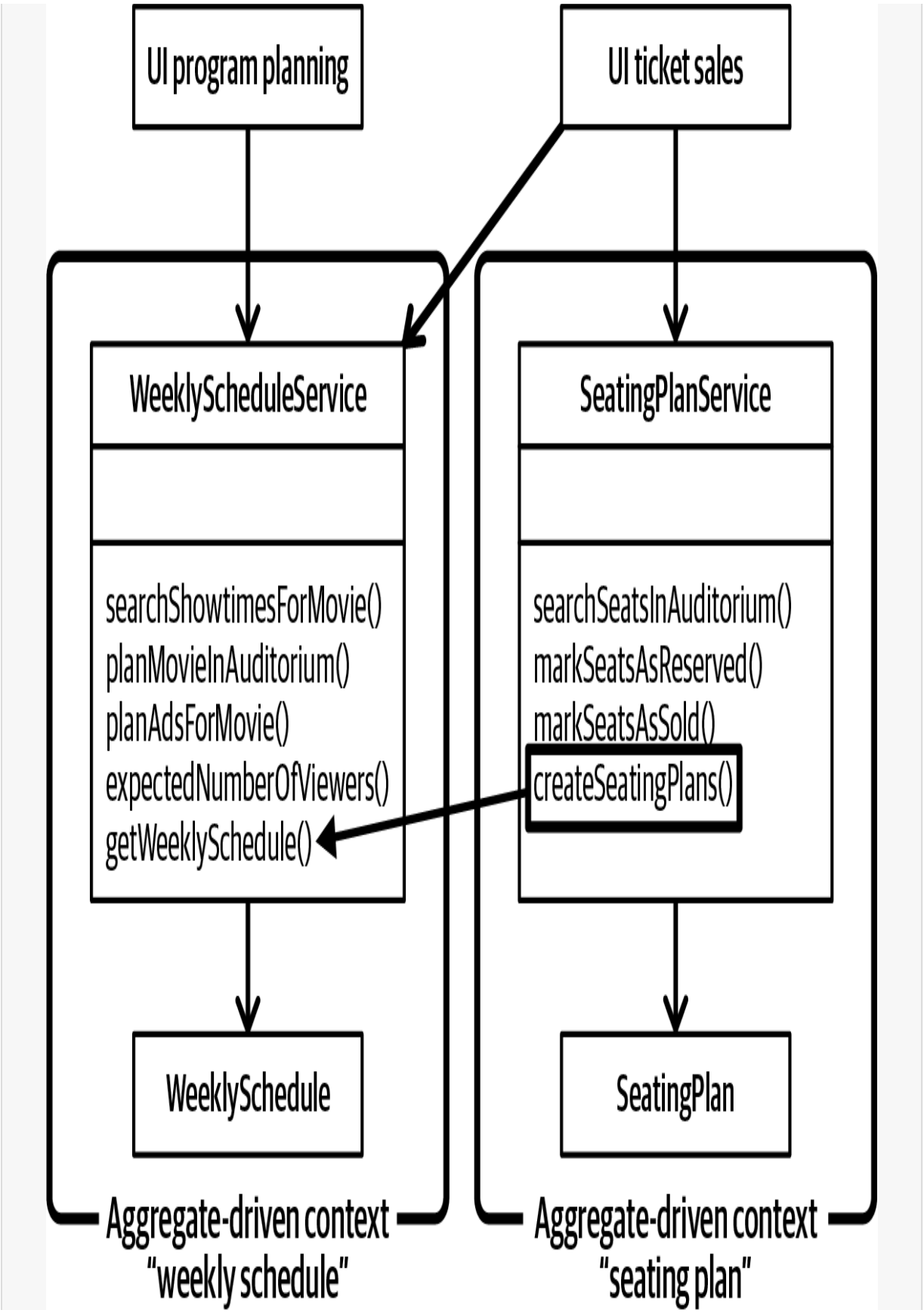

UI program planning
UI ticket sales
WeeklyScheduleService
searchShowtimesForMovie()
planMovieInAuditorium()
planAdsForMovie()
expectedNumberOfViewers()
getWeeklySchedule()
SeatingPlanService
searchSeatsInAuditorium()
markSeatsAsReserved()
markSeatsAsSold()
createSeatingPlans()
WeeklySchedule
SeatingPlan
Aggregate-driven context
"weekly schedule"
Aggregate-driven context
"seating plan"

Figure 10-7. Not recommended: division of contexts according to aggregates.

This solution in no way leads to a vertically loosely coupled architecture of contexts. If you instead split the subdomains based on the original indicators, the two contexts, “program planning” and “ticket sales” (see Figure 10-8), are created, which can work independently of each other.

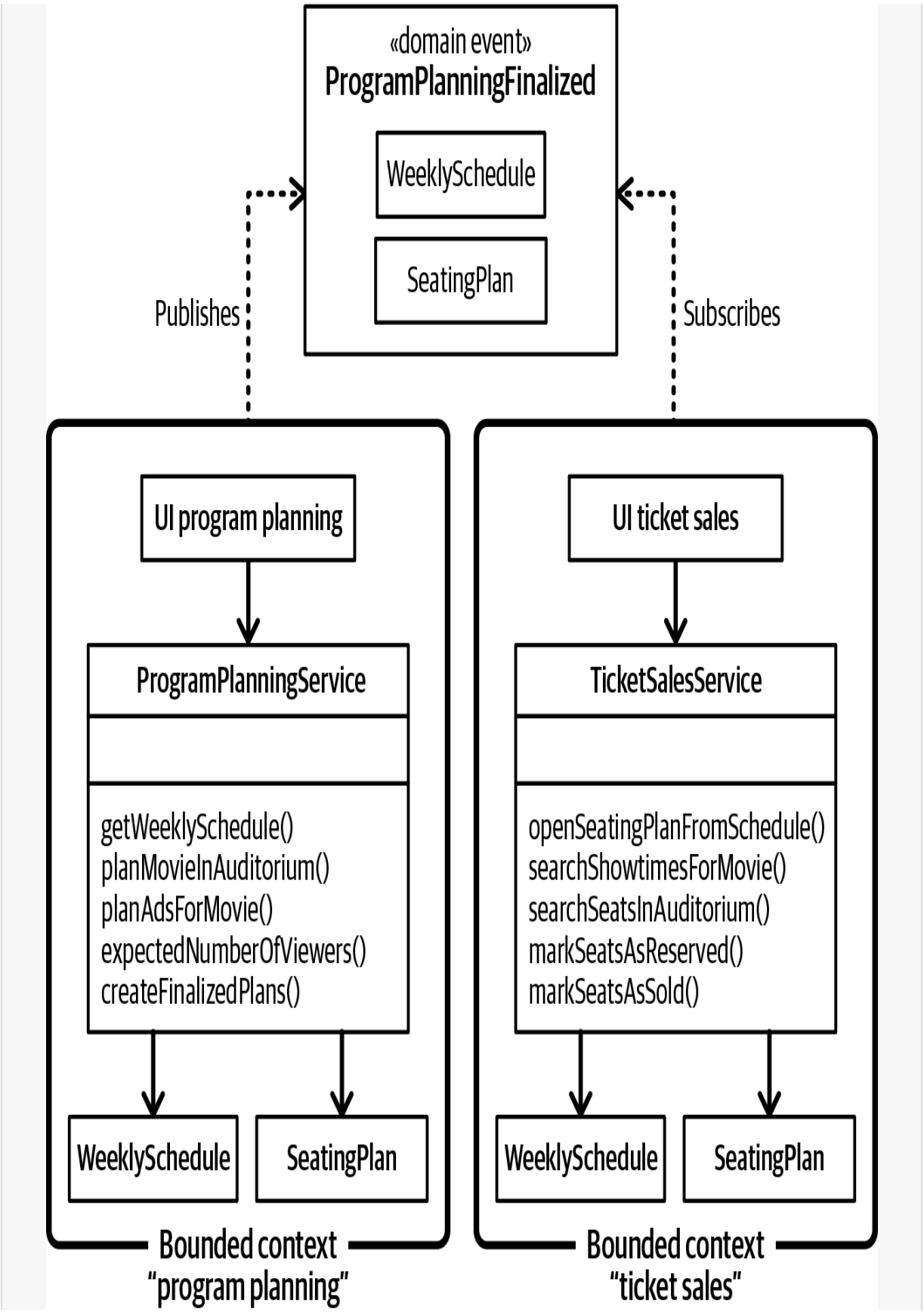

«domain event»
ProgramPlanningFinalized
WeeklySchedule
SeatingPlan
Publishes
Subscribes
UI program planning
ProgramPlanningService
getWeeklySchedule()
planMovieInAuditorium()
planAdsForMovie()
expectedNumberOfViewers()
createFinalizedPlans()
WeeklySchedule
SeatingPlan
Bounded context
"program planning"
UI ticket sales
TicketSalesService
openSeatingPlanFromSchedule()
searchShowtimesForMovie()
searchSeatsInAuditorium()
markSeatsAsReserved()
markSeatsAsSold()
WeeklySchedule
SeatingPlan
Bounded context
"ticket sales"

Figure 10-8. Recommended: division of bounded contexts by subdomain.

Only when the program planning is completed, a domain event (see Chapter 3) with the finalized weekly schedule and the completed seating plans is forwarded to the context "ticket sales" via an asynchronous interface.

The design from Figure 10-7 is not recommended because it does not break down the big ball of mud, but rather intensifies it. It also does not correspond to the implementation recommended in strategic design (see Chapter 3), but rather to a service-oriented architecture (see Chapter 5), and is only shown here as a cautionary example and to avoid misunderstandings.

Challenge: Finding the Right Size

> **WARNING**
>
> Before you move on to decomposing your domain into many small subdomains based on the target process, we would like to point out a misunderstanding. One problem we encounter time and again when cutting processes is the tendency to aim for far too fine a division into subdomains.

Each subdomain should comprise a largely independent process part whose software support is implemented by a team (we remind you of the explanations of *subdomain* from strategic design, Chapter 3, and of *cognitive load* from Team Topologies, Chapter 9). Thus, an architecture of independent bounded contexts emerges.

Goals of the Right Size

The pragmatic goals of such an architecture are:

Independent stream-aligned teams

A common goal of decomposition is to have teams of efficient size work as independently as possible by being responsible for bounded contexts that are decoupled from each other. This is associated with the notion that one team is solely responsible for each bounded context.

As we saw in Chapter 9, a team can be responsible for more than one bounded context. Ideally, there would actually be one team for exactly one bounded context, but since not all bounded contexts are equally important and there is not always work in all bounded contexts, two to three bounded contexts per team make sense. From a work organization and cognitive load standpoint, we do not recommend going beyond this number. More bounded contexts per team create noticeable development and runtime overhead.

Independent releases

Another important goal of decomposition is the ability to release bounded contexts independently. This allows incremental changes to be made to the system (instead of publishing releases as a whole) and thus teams can work decoupled from each other.

When decomposing into bounded contexts, you should check which reasons (e.g., technical, legal, or business triggers) could lead to a need for change in bounded contexts. If such considerations show that certain groups of bounded contexts practically always have to be changed and published together, there is at least an implicit coupling; in this case, the granularity is probably selected too finely and the bounded contexts concerned should be combined, or at least assigned to the same team.

Indicators for the Right Size

When decomposing our target processes into subdomains, we should therefore ensure that the resulting units:

- Are of a size that allows the software development teams to work *efficiently*

- Be grouped into larger units if it is expected or observed that groups of them will each be subject to *common changes* on a regular basis

If you follow this advice, the planned subdomains later turn out to be too large for individual teams. However, we recommend that you go this way, especially in the beginning, i.e., choose potentially very large subdomains first, which can be further divided later, rather than starting with too small subdomains.

Some questions that help finding the right size are: is a cross-functional/stream-aligned team able to:

- Deliver high-quality software?
- Evolve the software/architecture/business?
- Fully understand the business complexity?

Since any refactoring involves effort, we always take this path of careful decomposition into subdomains. Further decompositions can follow in a second iteration of Domain-Driven Transformation. Domain-Driven Transformation is thus a permanent process in which decomposition will occur again and again throughout the transformation of the legacy system.

Discussing Decomposition with AI

When the processes are stored in a machine-readable format (like domain stories saved as Egon.io files or even photographs from the wall of an EventStorming), they can be stored together with the source code. Now, the AI assistant in the IDE can be a discussion partner for finding boundaries. Our experiments have produced mixed results. Sometimes, the LLMs step into the “decomposition by work item” trap.

Summary

In step 1, we worked together with the domain experts and the development team to rediscover the domain, significantly deepened our understanding through discussions about the as-is situation and domain purity, and thus first developed to-be processes, and then used these to identify the subdomains.

In the next chapter, we will look at how to transfer the subdomains into bounded contexts, gather these into a context map, and then categorize them.

---

1 Technically, most monoliths can also be brought into the cloud by "simply" moving them to a virtual server. But because the monolith is a large coherent unit, it can only be scaled as a whole, can only be fast or highly available (or not) as a whole, and so on. The hoped-for cost savings and flexibility can be achieved much better with a fine-grained system.

2 If you are interested in more details about the emergence of these domain stories, you can find more in [Hofer and Schwentner 2022, chap. 8].

Chapter 11. Step 2—Modeling the Domain-Driven Target Architecture

The previous chapter discussed rediscovering domain knowledge; this chapter covers how that domain knowledge helps define the target architecture.

To model the domain-driven target architecture, we must derive the target context map from the subdomains modeled in step 1 (see Figure 11-1). This is the second step of the strategic Domain-Driven Transformation, the coarse-grained process that we saw in Figure III-1. In Figure 11-1, we see the substeps that make up step 2.

In Figure 11-1 the architecture analyst works together with the domain modeler on the following substeps: First, the target context map is derived from the subdomains and the to-be business processes (substep 2.1). Then, the bounded contexts in the context map are categorized as core, supporting, and generic subdomains (substep 2.2). In substep 2.3, the teams are assigned to the individual bounded contexts, and finally, in substep 2.4, the interfaces between the bounded contexts are defined.

Step 2, like step 1, takes place in an imaginary greenfield. Not until step 3 do we turn to the existing software system and leave the greenfield. This second step is about finding the ideal context map (ideal from today's point of view). A target definition, in other words, that asks the question: "Where actually do we want to go?" At the same time, such a context map is never something final, but a snapshot. Better understanding of the domain and changes in it are also reflected in changes to the context map.

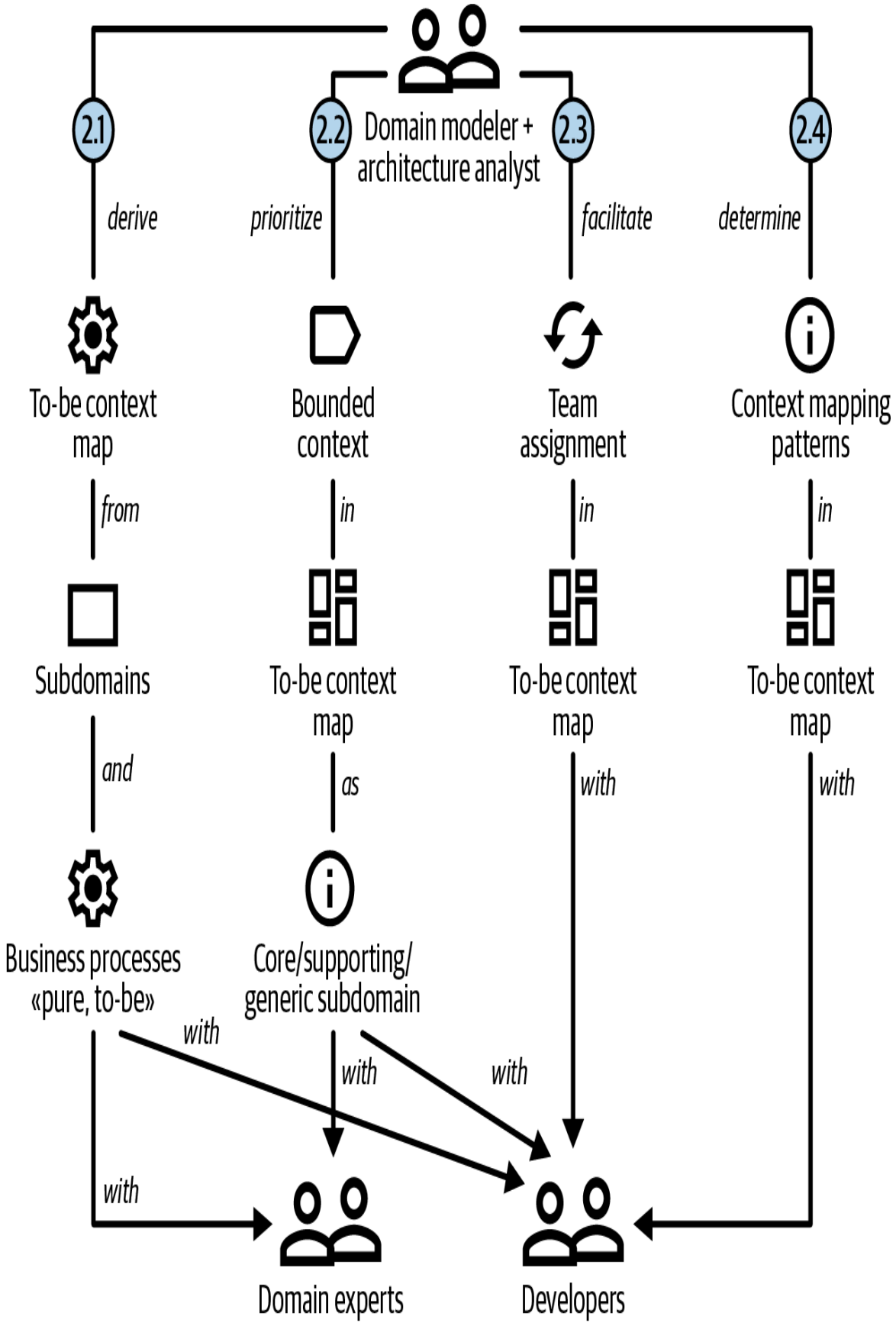

Domain modeler + architecture analyst
2.1
derive
To-be context map
from
Subdomains
and
Business processes «pure, to-be»
2.2
prioritize
Bounded context
in
To-be context map
as
Core/supporting/ generic subdomain
2.3
facilitate
Team assignment
in
To-be context map
with
2.4
determine
Context mapping patterns
in
To-be context map
with
with
with
with
with
Domain experts
Developers

Figure 11-1. Strategic transformation—step 2 and its substeps.

Schematic Description of Step 2

Step 2 includes the following elements:

Actors

- Domain modeler
- Architecture analyst
- Development team
- Domain experts

Substeps

1. Creating a context map (substep 2.1)
2. Categorizing subdomains (substep 2.2)
3. Assigning bounded contexts to teams (substep 2.3)
4. Determining relationships between the bounded contexts (substep 2.4)

Tools

- Facilitation materials
- Collaboration tools (e.g., Miro)

Results

- Target context map

Goals/reason

- Derive bounded contexts that make sense from a business point of view
- Determine teams for bounded contexts
- Make interfaces between bounded contexts explicit

Challenges

- Solution-space induced bounded contexts
- Assigning teams often means shaping teams

Just like in the last chapter, you will find alternating challenges and substeps at the same level of the chapter structure. In this chapter, there are four substeps (2.1–2.4) and two challenges. We have placed the challenges before, between, or after the substeps where they tie in with the content.

Substep 2.1: Creating a Context Map

In step 1, we found subdomains using our to-be business processes. Now we define how the software system will look in the future. To do this, we create a new artifact—the context map. We start by turning each found subdomain into a bounded context in the context map. This context map represents a target vision; this is how the system should be partitioned in the future.

Most often, the bounded contexts for a map come from different sources: different scenarios modeled as different domain stories, multiple EventStorming sessions, etc. The context map is a more coarse overall view and typically applies to the whole domain.

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: CONTEXT MAP

At Alphorn, the development team meets again with the key domain experts and us, Carola and Henning. Together we look at the domain stories created in Chapter 10. To make it easier for the reader, we show the story with the happy path again in Figure 11-2.

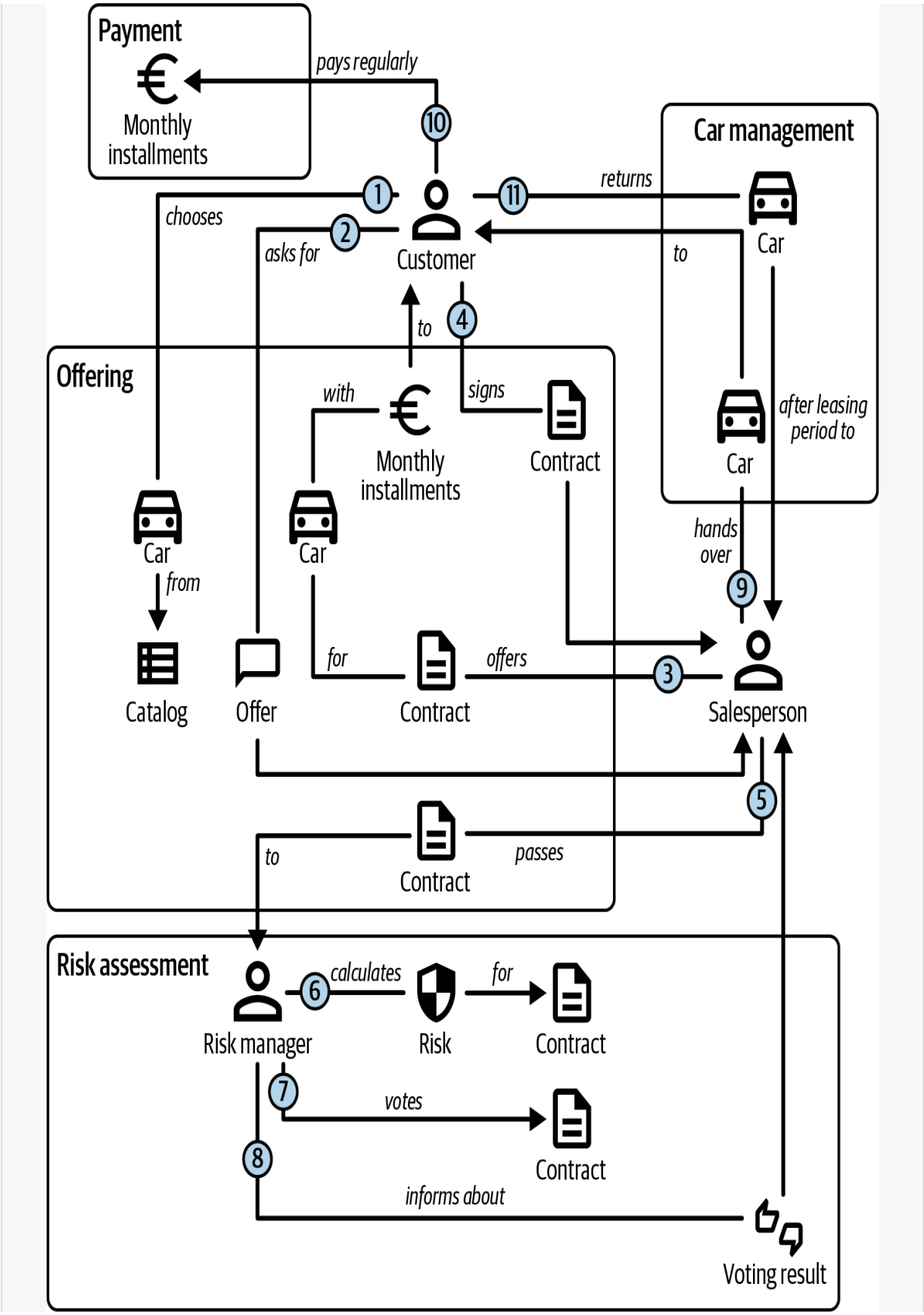

Payment
Monthly installments
pays regularly
10
Car management
returns
11
1
chooses
2
asks for
Customer
to
Car
to
4
Offering
with
Monthly installments
signs
Contract
Car
after leasing period to
Car
Car
from
hands over
9
Catalog
Offer
for
Contract
offers
3
Salesperson
5
to
Contract
passes
Risk assessment
Risk manager
6
calculates
Risk
for
Contract
7
votes
Contract
8
informs about
Voting result

Figure 11-2. Once again, the coarse-grained process at Alphorn Auto Leasing, Inc. with subdomains.

> *Domain modeler*: "So far, we've taken a process view; now we're also looking at it a little more structurally. I take the groups (the 'frames') from the story and transfer them to a new sheet on the flip chart."

We have drawn the subdomains in this story and transfer them as bounded contexts into the context map. This results in a first draft for a context map (see Figure 11-3).

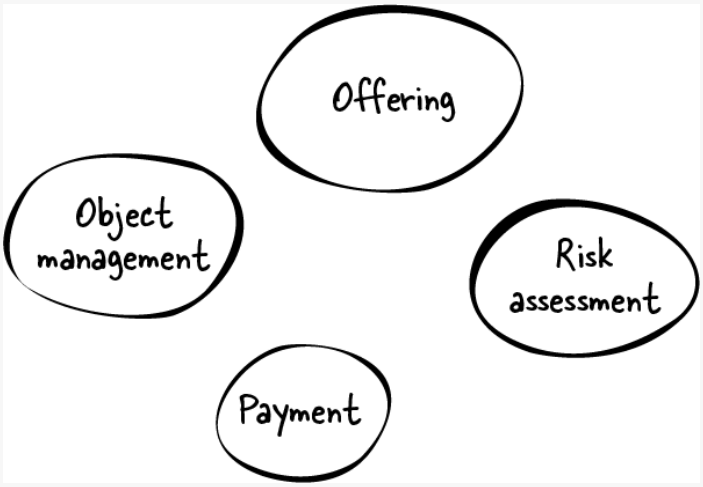


Figure 11-3. First draft for the context map of Alphorn Auto Leasing, Inc.

In the further domain stories from Chapter 10, we additionally discovered the subdomain "dunning," which we also add to the context map as a bounded context (see Figure 11-4).

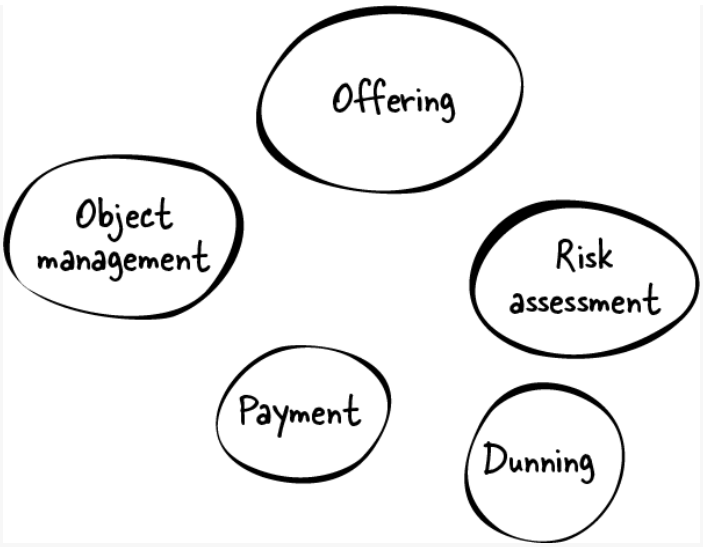


Figure 11-4. The context map of Alphorn Auto Leasing, Inc. with dunning.

> *Domain expert*: "Very nice, that's a good breakdown of what we do in car leasing!"

Challenge: Solution-Space Induced Contexts

It is important to start with bounded contexts that arise one-to-one from subdomains—genuine domain-driven bounded contexts. (This is how to represent the essential complexity of the domain in the system.) Only then do we look at the system parts that are needed because software is used. (Thus, adding complexity in the solution space.) Almost every system also contains entire bounded contexts that are solely solution induced.

A typical example is a bounded context for user administration. The concept of "user" does not exist in the domain in the cinema, or in the bank, or in car leasing. However, the fact that an IT system is used makes it essential that a login is provided to restrict the use of the system to a user group. Roles and rights are also required in most software systems. These roles and rights usually come from the domain.

IN THE CAR LEASING EXAMPLE: ROLES AND RIGHTS

For example, at our car leasing company we have the following roles:

- Junior risk manager
- Senior risk manager

The following rules apply to these roles and voting. A car may be voted with a list price:

- Up to $50,000 by one junior risk manager alone
- Up to $80,000 by two junior risk managers jointly or one senior risk manager alone
- Up to $100,000 by one junior risk manager and one senior risk manager jointly
- Above that by two senior risk managers jointly

At Alphorn, these concepts are required in both the "user administration" and "risk assessment" contexts.

A "user administration" usually has its very own language: "user," "login," "password," "to lock," etc. These words do not apply specifically to the "car leasing" domain, but are more general. This is an indication that "user administration" is a solution-induced bounded context belonging to a generic subdomain (more on this in the next section).

To not overload our context map (and also because solution-induced bounded contexts are usually less interesting for decomposition), we leave out "user administration" in the book.

HENNING'S FIELD STORY: SOLUTION-SPACE INDUCED BOUNDED CONTEXT IN A FLEET MANAGEMENT SYSTEM

I was consulting for a big truck manufacturer. Their main customers are haulage companies that operate more than one truck. To support the number of trucks, the manufacturer offers not only the vehicles but also a fleet management system. This fleet management system was built as a monolith that ran in the manufacturer's data center. I was hired as a consultant to help move the system into the cloud and convert it into a microservices architecture. The first thing I did was run an EventStorming workshop to learn about the domain and find its boundaries. We found subdomains like "fleet control," "tour planning," and more that were designated to be turned into bounded contexts. Each of these bounded contexts was planned to be developed and operated by a different team, and for maximum independency they would be implemented as different microservices.

The fleet management system was able to manage all trucks that were equipped with a so-called "device," i.e., an on-board computer. Those devices measure different data like geolocation, tire pressure, and fuel level, and then send this data into the central instance of the fleet management system. The newer devices were updated by flashing their firmware. For several older generations of these "devices" that were also still in use, the firmware updates were not possible. With the modernization effort, this became a problem since the older devices sent their data in a message format that was old-fashioned, incomprehensible, and prone to errors.

At this point of the modularization, a decision had to be made: where should we put the knowledge about the data message format of the old devices?

1. The monolith had some kind of a translation layer used by the rich client of the system. This "device connectivity" contained most of the knowledge about the old data message format.

2. Now, the first instinct was to say: “We want the microservices to run independently from each other. When ‘tour planning’ is down, this should not influence ‘fleet control,’ so they should both be able to connect to the devices on their own.” While everybody agreed that the independent deployability was a main goal, the disadvantage of having the arcane knowledge of dated devices (plus about the new ones) as a must in every team seemed significant.
3. Eventually, the “device connectivity” was formed into a bounded context in its own right that was operated as a microservice by a dedicated team. Now the contexts “fleet control” and “tour planning” were independent from each other and they could also be operational at least for a short time when the “device connectivity” microservice was down. (For the business it is OK when fleet control and tour planning don’t receive updates from the trucks for the few minutes that it takes to restart the device connectivity microservice.) From a Team Topologies perspective, the device connectivity team operates as a platform team.

The “device connectivity” bounded context is an example of a solution-space induced bounded context. It is not driven by the domain fleet management; it is nothing an employee of a haulage company would think of. But to make the overall system work, it is an integral part. In this particular case it serves the task of an anticorruption layer; that is why the team described it as an “anticorruption bounded context.”

It is also a typical example for the trade-offs that have to be made in legacy modernization[1]—in this case, independent deployability versus knowledge distribution.

Substep 2.2: Categorizing Subdomains

Once the bounded contexts are gathered, it is interesting to ask which ones are more important than others. This is where strategic design helps us with its categorization of subdomains (see Chapter 3) into core domain, supporting, and generic subdomain.

In most organizations, this cannot be answered by domain experts alone; rather, the strategic level of the organization must be involved. So in substep 2.2 from Figure 11-1, domain experts means not only the people who do the work, but also their management.

The discussion held in substep 2.2 aims to identify the following:

- Which subdomains, and therefore which bounded contexts, belong to the core of the business (*core domain*).
- Which subdomains, in contrast, are specific to our business but should not be part of the core (*supporting subdomain*).
- Where the purchase of standard software and thus the replacement of the existing source code would be worthwhile (*generic subdomain*).

Typical questions that we use to initiate discussions among domain experts, the development team, and the strategic management level are:

- What do we earn the (most) money with?
- What are we known and popular for in the market?
- What do we want to be better at than the competition, and where is it enough for us to be average?
- What is our unique selling point or what makes our company special?
- In which subdomains are there particularly many different players? Here the work has become very differentiated because it is carried out so often and is correspondingly important.

- Which subdomains have particularly demanding activities that lead to important results?
- Which subdomains are at the center of the business processes and are supplied with information by many other subdomains?

It makes sense to classify all bounded contexts with the help of the subdomains. If a bounded context does not yet have a priority, this means that a decision is still pending here, not that no decision is to be made.

Sometimes mixed forms arise, such as supporting-generic, to indicate that the importance of a subdomain lies between supporting and generic.

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: CATEGORIZATION

The team discusses the categories of identified bounded contexts.

> *Domain modeler*: "We extracted four bounded contexts from the subdomains and drew them on our context map; which of these subdomains is the most important?"

> *Domain expert*: "It's all important! We need to keep supporting all subdomains. Otherwise we can't use the system!"

> *Domain modeler*: "OK, OK, and what is the core? Where is our company different from its competitors? Where do we want to be better than others?"

> *Domain expert*: "Well, our customers love us first and foremost for the fact that it's so easy to lease a car from us, and they quickly have the contract signed and sealed."

> *Domain modeler*: "I see, then I mark the bounded context 'offering,' in which this takes place, as core. For the 'payment' context, it's enough if we don't stand out against our competition and use a standard solution, right?"

Domain expert: “Yes, that’s right. If the standard solution can do what we need!”

Domain modeler: “Alright, I’ll mark the context as generic.”

The team continues to discuss until all subdomains are categorized, and we see the next version of the context map, as shown in Figure 11-5.

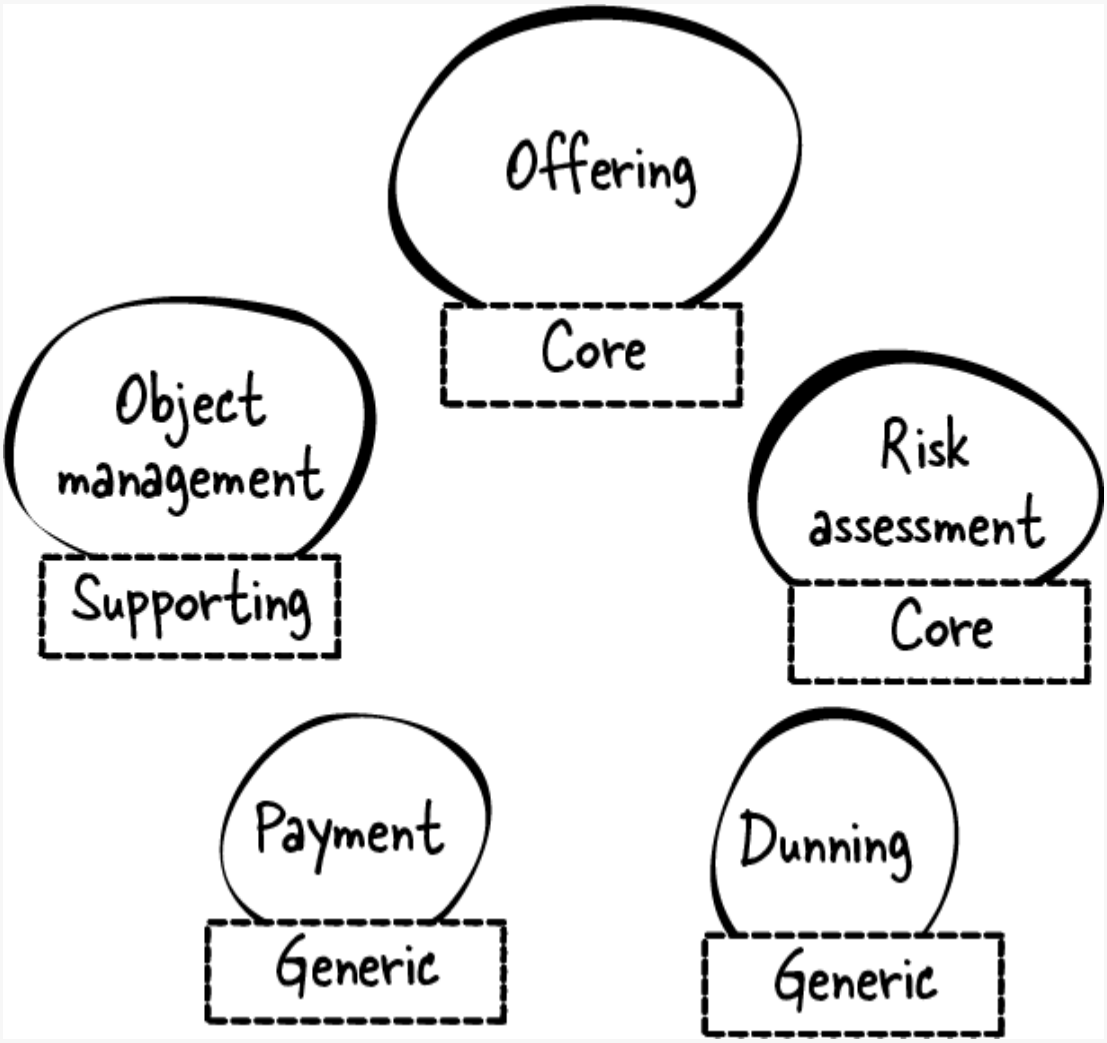


Figure 11-5. The context map of Alphorn Auto Leasing, Inc.—core versus supporting.

The categorization may change over the lifetime of the software as the focus of the organization shifts. Core domain charts can help to identify and visualize this evolution (see "Core domain charts").

Often, during the 20 or 30 years that a legacy system has existed, standard software solutions have been developed for parts that had to be custom-built when the legacy system was conceived. For the subdomains that are not core anymore, it is worth considering if nowadays a standard (i.e., COTS, SaaS, or open source) solution can replace a custom-built part.

Substep 2.3: Assigning Bounded Contexts to Teams

Context mapping and team assignment are closely related, as we saw in Chapter 3, "Context Mapping". But categorization by core domain, supporting, and generic subdomain also play a role. For example, some organizations impose the following rules for assignment:

- "Each core domain should have its own team." The idea here is similar to what the Team Topologies approach recommends (see Chapter 9): the cognitive load of the individual team should not become too high.
- "Each team should have a core domain." The idea behind this is that every team should have something interesting and important to do and not feel useless or second-rate.
- "The core domains are only something for the experienced teams, and the newbie teams get a supporting subdomain to learn first."
- "The core domains are only for the excellent teams. The mediocre teams get supporting subdomains where excellence is not required."

- “The core domains are only for the in-house teams. The external teams from contractors get the less important subdomains where loss of knowledge drain is not as severe as in the core domain.”

As you can see, these rules come from different philosophies and can contradict each other. For example, we personally are not big fans of considering a team, and thus the people who work on it, as “mediocre.”

We typically discuss these rules with the development team and the developers’ managers to come up with an assignment that is acceptable to the organization.

Challenge: Assigning Teams Often Means Shaping Teams

Usually, the existing team structure in an organization fits the big ball of mud, but not the desired new organization in bounded contexts. Therefore, in most cases, in addition to splitting the software, splitting the teams must also take place. This is where the knowledge from Team Topologies and the sociotechnical refactorings help (see Chapter 9).

CASE STUDY “CAR LEASING”

ASSIGNING TEAMS IN THE CAR LEASING EXAMPLE

At Alphorn, teams were previously separated by layer. In the future, there will be three cross-functional teams. After a discussion in the team and with the managers, the assignment of teams to bounded contexts shown in Figure 11-6 is chosen.

The three teams are all stream-aligned in the sense of Team Topologies (see Chapter 9).

It is decided that team Chewbacca should be shaped first. For this, the sociotechnical refactoring *Form Cross-Functional Team out of Layer-Team Members* is scheduled. The knowledge gained will also be passed on to the next team being shaped, R2D2, so *Form Second Cross-Functional Team out of Partly Layer-Team and First-Team Members* is planned for team R2D2 at some point in the future.

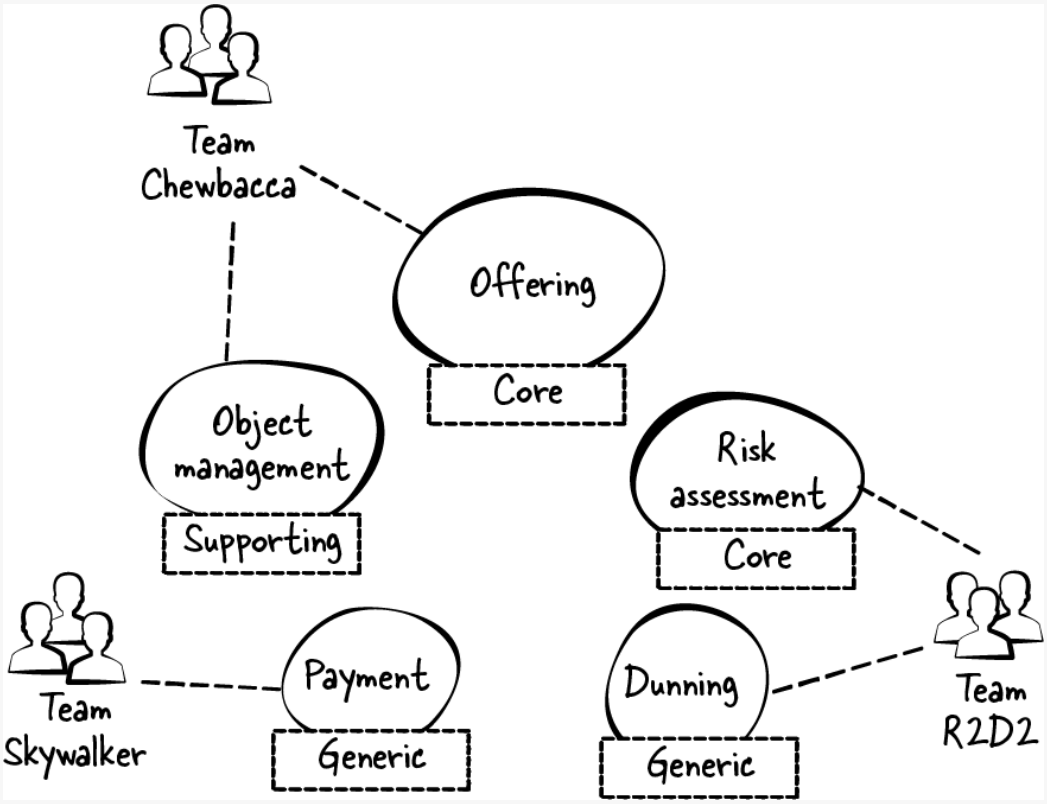


Figure 11-6. The context map of Alphorn Auto Leasing, Inc.—with teams.

Substep 2.4: Determining Relationships

As the last substep in modeling the domain-oriented target architecture, we determine the relationships between the contexts with the teams (substep 2.4).

Typical questions that we discuss with the development team are:

1. Where are the interfaces between the bounded contexts?
2. Which bounded contexts should not have an interface between them at all?
3. Which patterns from context mapping (see Chapter 3) do these interfaces follow?
4. What is the relationship between the teams implementing the bounded contexts and communicating via the interfaces?

For the first two questions about the existence of interfaces, we find answers in the decomposed to-be business processes that we developed in step 1 with Domain Storytelling or EventStorming (see Chapter 10).

To answer the third and fourth questions, we rely on the patterns from context mapping (see Chapter 3) and the ideas from Team Topologies (see Chapter 9).

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: RELATIONSHIPS

For the discussion of the interfaces between the bounded contexts, we simplify the context map by omitting the prioritization and the teams, and look only at the bounded contexts.

In the domain story in Figure 11-2, we noted that in sentence 5 the contract is passed from the salesperson to the risk manager. In the system, this will be reflected as an interface between the "offering" context and the "risk assessment" context. We draw a line between the two bounded contexts to make this visible in the context map (see Figure 11-7).

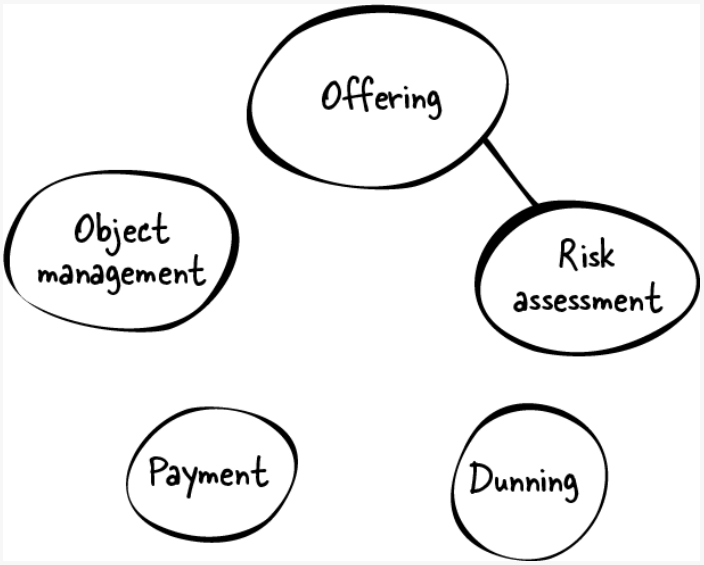


Figure 11-7. The context map of Alphorn Auto Leasing, Inc.—with a first relationship.

The further discussion among architecture analyst, developer, and domain expert about interfaces proceeds as follows:

> *Architecture analyst*: "In sentence 8 of Figure 11-2, the risk manager informs the salesperson of the voting result; is this also an interface in the other direction between offering and risk management?"

> *Developer*: “No, the story continues in object management afterwards, so I understand this to be an interface between risk assessment and object management, correct?”

> *Domain expert*: “Yes, exactly.”

The conversation continues until we have discussed and visualized all the relationships. The result is shown in Figure 11-8.

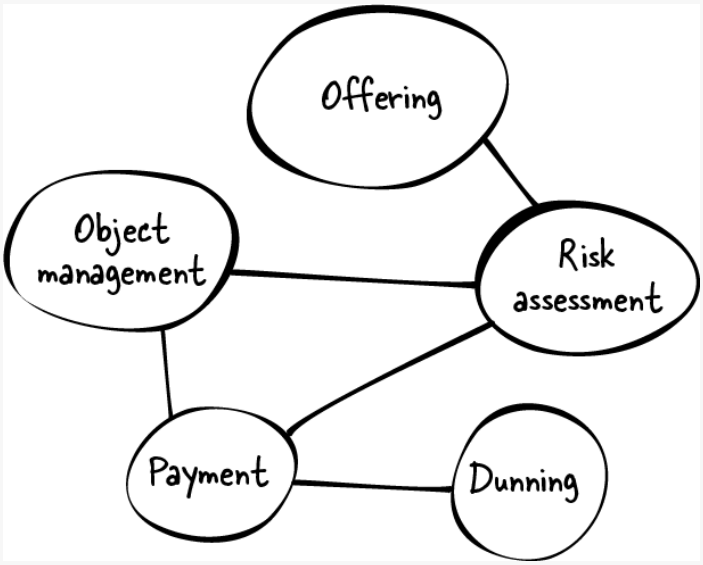


Figure 11-8. The context map of Alphorn Auto Leasing, Inc.—with relationships.

After a break, when all participants have replenished with caffeine, we finally add the types of relationships:

> *Architectural analyst*: “Risk assessment and dunning both have an interface to payment. If I understand correctly, the point of both is that the payment bounded context informs when an installment has not been paid as agreed.”

Domain expert: “That is correct. Both get the same information.”

Architectural analyst: “Aha, so this is the same interface. I’m putting that as an open host service.”

Developer: “Payment is, after all, a standard software whose interface does not correspond one hundred percent to our ideas. For the generic subdomain dunning, this is probably OK, but the risk assessment will protect against this with an anticorruption layer.”

Architecture analyst: “I see. I’ll draw this in the context map.”

Step-by-step, we go through all the relationships and categorize them; finally, the context map looks like Figure 11-9.

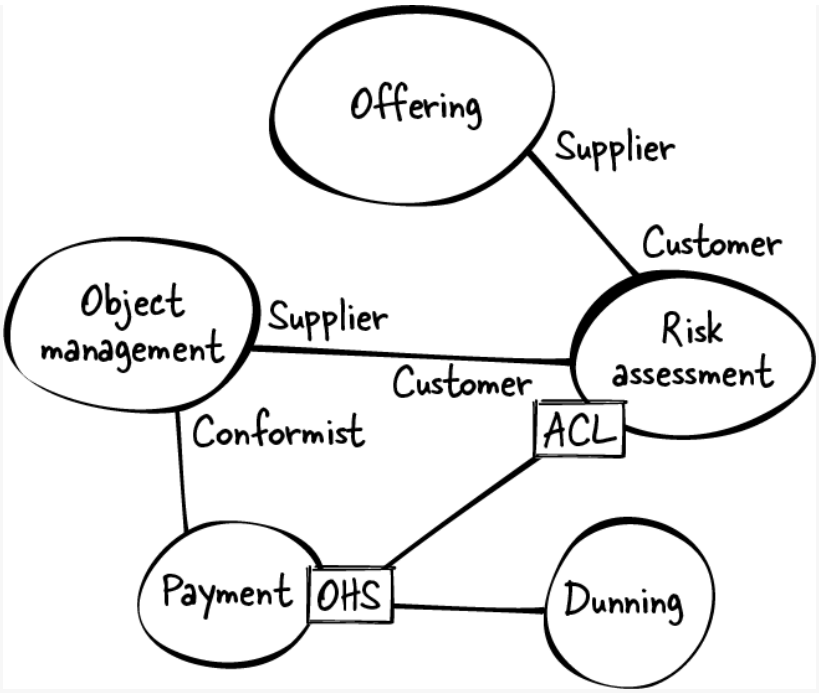


Figure 11-9. The context map of Alphorn Auto Leasing, Inc.—with patterns. Recall from Chapter 3 that OHS stands for open host service and ACL for anticorruption layer.

> *Developer*: "Wow, this is what the system should look like. It's a pity that it will be a lot of work to get there."

Putting It All Together

It can be useful to display all the details we have discussed in this chapter together in the context map. However, then there is a lot of information to digest at once. That's why many teams prefer to use the simplified maps shown in the previous examples.

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: FINAL CONTEXT MAP

For Alphorn, we put all the information together in Figure 11-10.

This version of the context map is redrawn on a large scale and hung in the lobby where most colleagues eat lunch and where cross-team all-hands meetings are held.

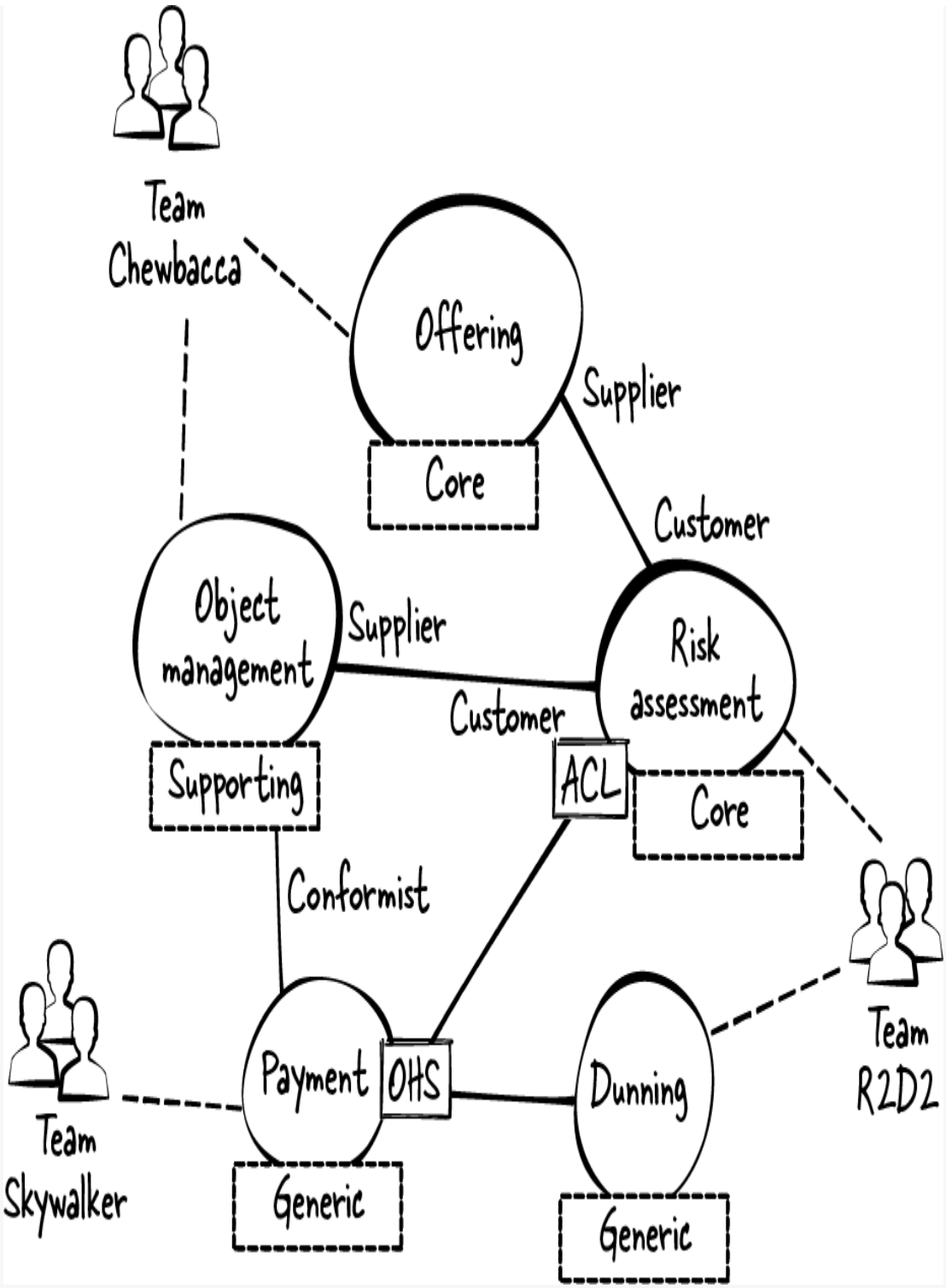


Figure 11-10. The context map of Alphorn Auto Leasing, Inc.—with everything.

Summary

In step 2, we derived the target context map from the subdomains of the business processes. In the target context map, we find the categorized bounded contexts with their interfaces and the context mapping patterns used in the process.

In the next chapter, we analyze the actual architecture of the real source code in step 3 and compare it with the target context map. The insights gained in this process will be incorporated into a refactoring list, which we can then use in step 4 to begin the reconstruction of the legacy system.

---

[1] A direct application of the First Law of Software Architecture: “Everything in software architecture is a trade-off” (see [Richards and Ford 2020, p. 19]).

Chapter 12. Step 3—Alignment of Actual with Target Architecture

How the target architecture can be found was the topic of the previous chapter. This chapter will discuss how to align the actual architecture with that target architecture. It will also cover how to define and collect the steps that must be taken to get from the actual to the target architecture.

From the sheer length of this chapter compared to the preceding chapters, it can be seen that step 3 takes more effort than steps 1 and 2. The next chapter is even longer, as step 4 takes even more effort.

If we used the target context map in step 2 to determine which bounded contexts we want to implement in our software, the next step is to figure out where we are starting from. So the next question is: “Where are we actually?” Once we have found the answer to that question and know where we are (step 3) and where we want to go (target context map from step 2), then we can deduce which direction we need to go to get from the as-is to the target.

In Figure 12-1 you can see how the comparison between the actual architecture and the target architecture works. First, in substep 3.1, the architecture analyst uses an analysis tool (such as the ones listed in the next section) to determine the actual architecture that actually exists in the source code. In substep 3.2, the architecture analyst places the target context map onto the source code. (Again, this can be done with an analysis tool.) In doing so, they take into account the findings from the creation of the actual architecture in substep 3.1, and discuss with the development team where the target context map and the source code fit together well and where they do not. In the course of this alignment, the development team and the architecture analyst develop and discuss refactorings and findings in

substep 3.3. Refactorings are concrete steps that will be necessary to implement the planned bounded contexts in the big ball of mud source code. Findings are defects where the exact implementation still needs to be worked out.

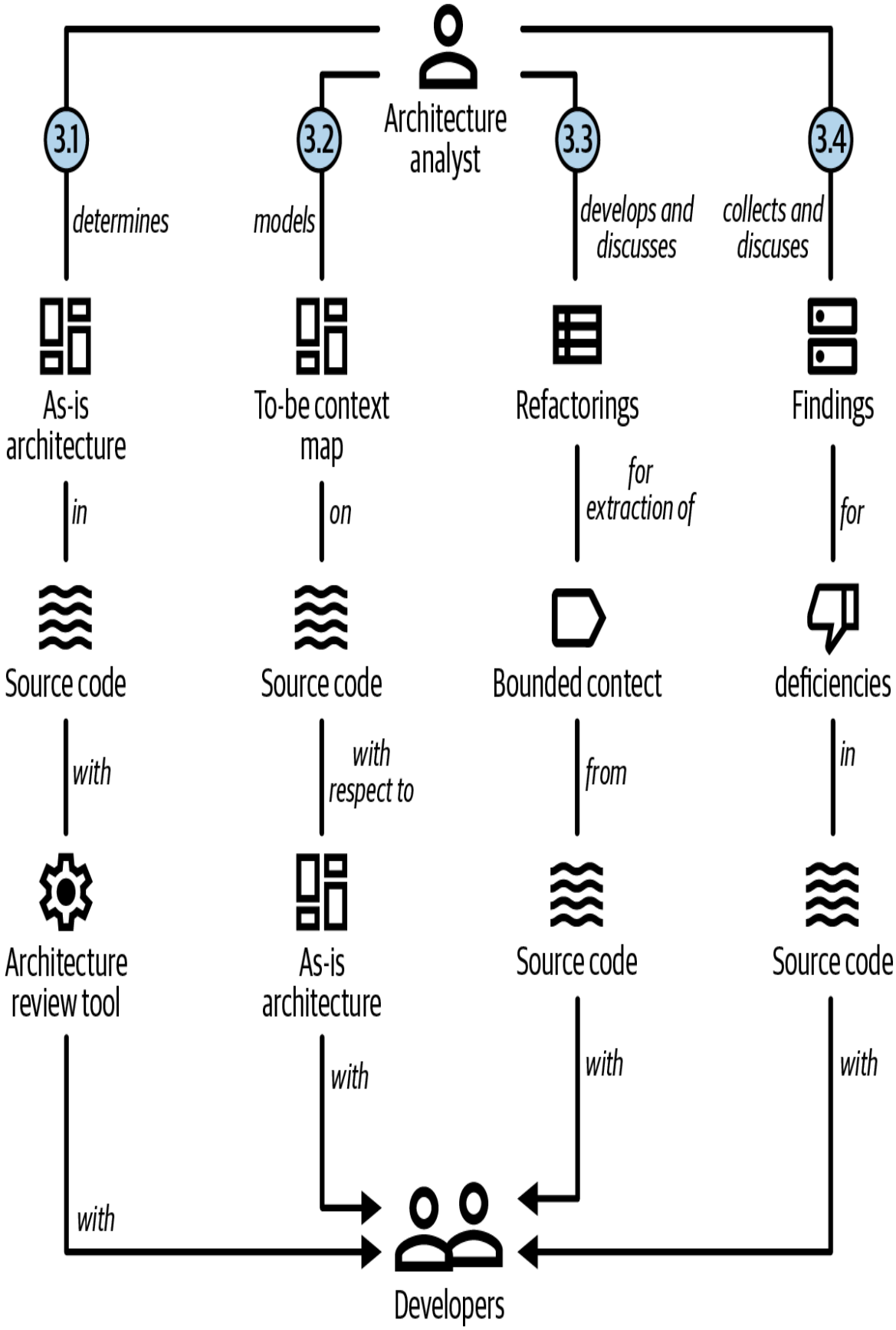

Architecture analyst
3.1
determines
As-is architecture
in
Source code
with
Architecture review tool
with
3.2
models
To-be context map
on
Source code
with respect to
As-is architecture
with
3.3
develops and discusses
Refactorings
for extraction of
Bounded contect
from
Source code
with
3.4
collects and discuses
Findings
for
deficiencies
in
Source code
with
Developers

Figure 12-1. Strategic transformation—step 3 in its substeps.

> **WARNING**
>
> A warning is in order here: often you will not want to achieve the goal from the target context map one hundred percent, because the findings in particular are often too costly. You will often achieve the goal of having software that is easier to change even with a slightly less modular architecture. As always with the topic of architecture and architecture improvement, it is not about perfection, but about improvements that subsequently let us develop faster.

In addition, as moving from the current, highly entangled source code to a well-structured modular system, it will be discovered that the context map that originally seemed ideal was not so ideal after all. As we improve, we will gain insights that make a different decomposition seem more appropriate in some places. That means some context boundaries may have to be realigned. And even if we had developed the ideal context map for the future at one point in time, the peripheral conditions often change while we are carrying out the conversion, and we have to replan.

Therefore, we proceed step-by-step or iterative-incremental (Agile) in Domain-Driven Transformation (see Chapter 6), and check after each partial reconstruction whether we have to adapt our way.

Schematic Description of Step 3

Step 3 includes the following elements:

Actors

- Architecture analyst
- Development team

Substeps

1. Identifying the actual architecture (substep 3.1)

2. Comparing the target context map and the source code (substep 3.2)

3. Collecting refactorings and findings (substep 3.3)

Tools

- Facilitation materials
- Collaboration tools (e.g., Miro)
- Architecture review tools (e.g., Sonargraph, Structure101, TeamScale, Lattix, JQAssistant)
- C4 Model

Results

- Actual architecture
- Difference between actual and target architecture
- Strategic and tactical refactorings
- List of findings

Goals/reason

- Determination of the legacy software condition
- Outline extent of the reconstruction

Challenges

- Unbounded models
- Unbounded domain model
- Unbounded anemic domain model
- Unbounded data model
- Decomposing the user interface

Just like in the last two chapters, you will find alternating challenges and substeps at the same level of the chapter structure. In this chapter, there are three substeps (3.1 to 3.3) and five challenges. We have placed the challenges before, between, or after the substeps where they tie in with the content.

Substep 3.1: Identifying the Actual Architecture

The fastest way to determine the actual architecture is to use an architecture review tool. In [Lilienthal 2019] you can read about the possibilities that the common architecture review tools offer today. Most of these tools are designed to analyze monoliths. However, nowadays it's also possible to analyze microservices architectures with multiple source code repositories and to include the relationships between the microservices in the review.

To start the architecture review, the architecture analyst discusses the structures found in the source code with the development teams. In the discussion, they jointly work out the actual architecture from these structures. Most of the systems we deal with today have a layered architecture (see Chapter 5), as it came into fashion in the early 2000s. At the end of the 1990s, the most common problem software architects had to

deal with was that the source code was happily mixed with the business logic, the user interface, and the database access.

Such a layered architecture first of all leads to the separation of the business source code (application and domain layer) from the technical source code (user interface and database access). Thus, a technical order is created in the system. That was a great improvement at the time! In our reviews, we therefore always first appreciate the layer architectures that we find in the systems.

For systems that can be handled by a capable development team alone, a layered architecture may be sufficient. As already discussed in Chapter 9, such a team has between five and nine people. However, most systems that have been evolving and expanding since the early 2000s have long since exceeded this size, so technical layering is not sufficient. Above the technical layering, the system needs a domain-oriented structure consisting of bounded contexts. In our experience, dividing a large layered architecture into bounded contexts with narrow interfaces is a laborious undertaking—it would certainly have been better to think in terms of bounded contexts right from the start of development. Now this work has to be made up.

Sometimes we are lucky and find architecture documentation for a system. If we do, we usually find documentation according to arc42 (see [Starke 2022a]) or the C4 Model (see Chapter 5 and [Brown, forthcoming]). If there is no documentation, we let the development team explain the planned structures to us.

Actual Architecture and AI

Every now and then, we encounter teams where nobody really knows the architecture of the legacy code that the team is responsible for anymore. The team works on the source code, but no one knows what architecture was originally planned for the system or how this architecture might be reflected in the source code and its structure.

In these cases, we have to engage in software archaeology. We have to use the existing source code to figure out what the plan might have been. AI

tools can save us valuable time here by examining the source code for us. In this case, the task of the AI tools is to create an architecture description in a format of our choice for the existing source base and to tell us which architectural style (or architectural styles) are used in the source base.

Typical questions we ask an AI tool are:

- Can you find a technical layering? Where is this layering violated?
- Can you find bounded contexts? Are there violations?
- Can you find any types of interfaces?
- Can you find architectural patterns and design patterns?
- If there are patterns, how do they work together or which relationships do they have? Do they follow the rules?
- Can you find special or local patterns (i.e. patterns that only appear in this system and are not described in the literature or used in frameworks)?

In 2025, we are only at the beginning of using AI tools in our reviews. We look forward to seeing how this list of questions will grow in the coming years.

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: ANALYSIS OF THE ACTUAL ARCHITECTURE

At Alphorn, the first thing the architecture analyst asks is what kind of architecture the system has. The quite typical answer is, “Monolease has a layered architecture!” Then the team shows the architecture analyst the C4 Model container view (see Figure 12-2) that the development team prepared for the as-is review.

Using the C4 container view, the team explains to the architecture analyst that the system consists of a frontend, backend, and database, so it has a classic layered architecture.

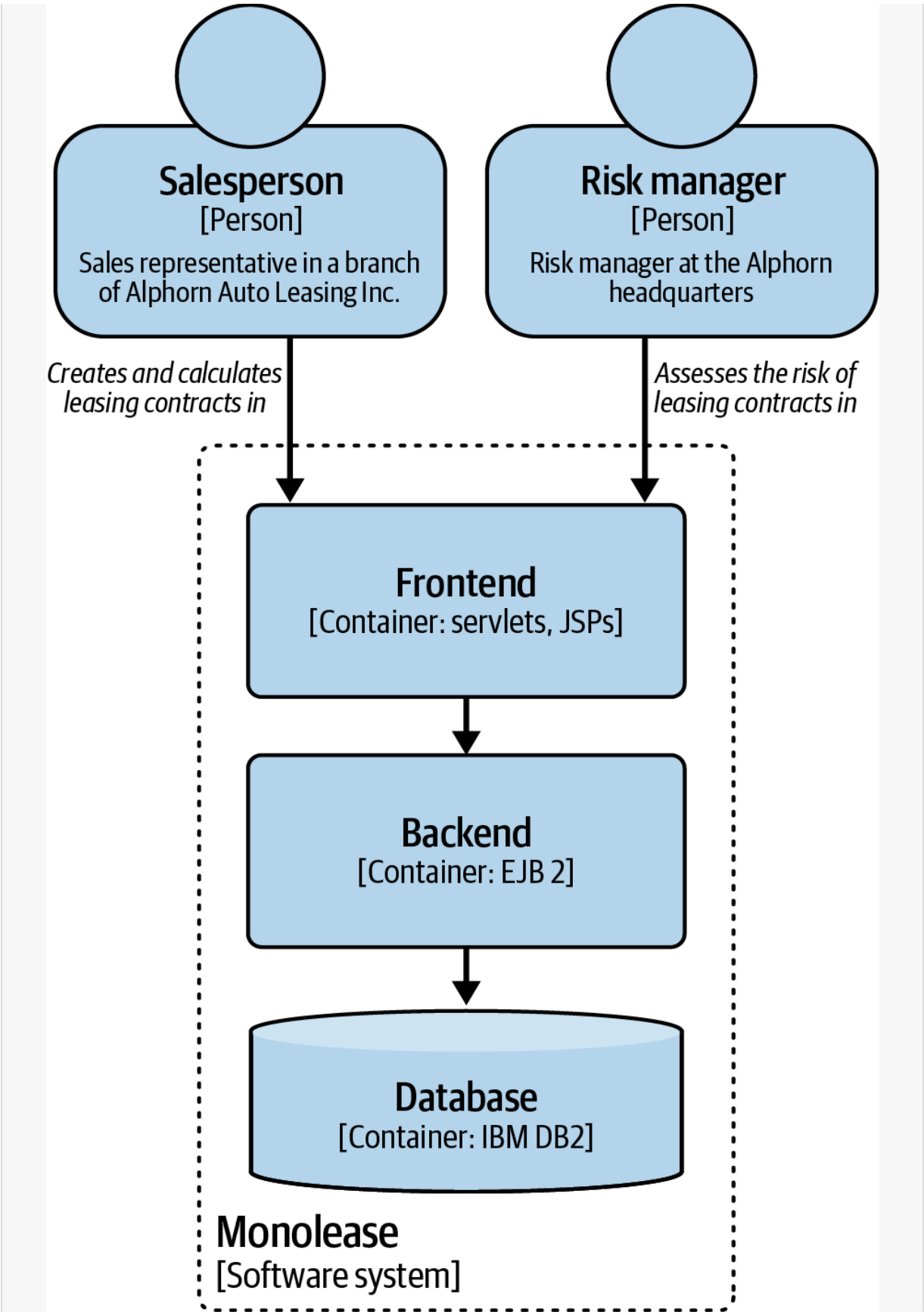

Salesperson
[Person]
Sales representative in a branch of Alphorn Auto Leasing Inc.
Risk manager
[Person]
Risk manager at the Alphorn headquarters
Creates and calculates leasing contracts in
Assesses the risk of leasing contracts in
Frontend
[Container: servlets, JSPs]
Backend
[Container: EJB 2]
Database
[Container: IBM DB2]
Monolease
[Software system]

Figure 12-2. The actual architecture of Monolease (C4 container view).

Next, the architecture analyst reads the source code into their analysis tool and, together with the team, begins to drill deeper into the three layers found.

The first look into the source code reveals that the source code of Monolease contains at least some ubiquitous language. For example, the domain term “contract” is called `Contract` in the code. Also interesting is that the concept “customer” appears in the code only as a generalized class `Partner`.

To store the domain concepts, there are domain-named database tables such as `CONTRACTS` and `PARTNERS`. The convention of naming the tables in capital letters shows that this system is already a bit older.

In the analysis tool, you can see that Monolease has layering, but no bounded contexts. Central classes, like `Contract` and `Partner`, are used by `ContractService`. In Figure 12-3, these structures are shown as a class diagram.

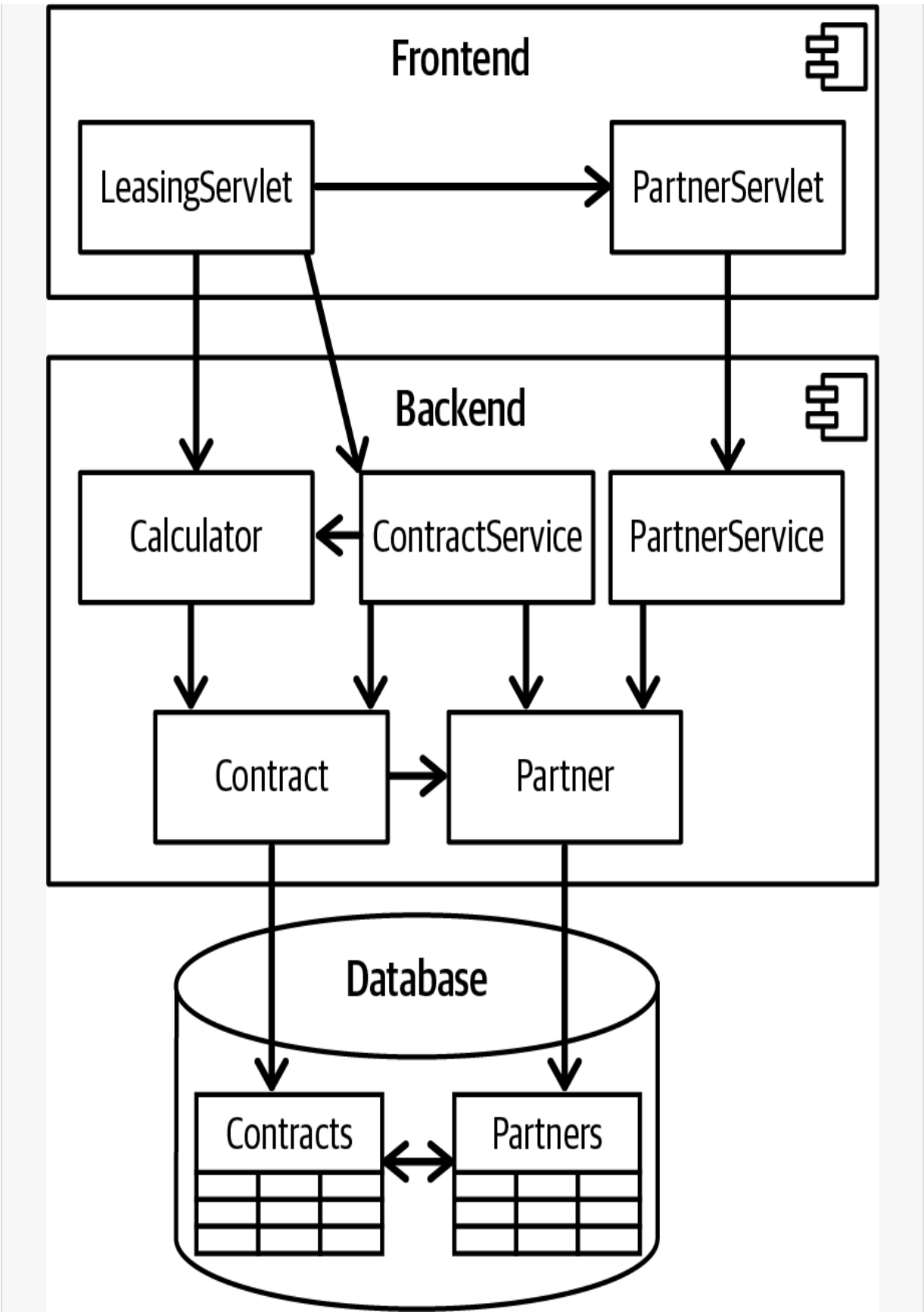

Frontend
LeasingServlet
PartnerServlet
Backend
Calculator
ContractService
PartnerService
Contract
Partner
Database
Contracts
Partners

Figure 12-3. The actual architecture of Monolease.

When we look at the source code of the central domain classes, we also notice that there are only setter and getter methods everywhere. So there is a lot of work waiting for us!

Substep 3.2: Comparing the Target Context Map and the Source Code

Now comes the thrilling moment: how will the target context map fit to the source code?

Sometimes you are lucky and find larger units at the module or package/package tree level in the actual architecture that match the bounded contexts planned in the target context map. In this case, you can assign entire package trees or even modules to the corresponding bounded contexts. Or you can find classes with the bounded context in their names, so that you can assign them all together to the bounded context.

In most cases, however, the situation is different: the context boundaries run right through packages and classes. At these points, we have to start splitting up packages and classes. Depending on how much domain-oriented structure already exists in a system, this mapping work is more or less time-consuming. In [Lilienthal 2019], you can find all variants of structuring that Carola and her team had the opportunity to analyze in the past years.

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: TARGET CONTEXT MAP

The team has another look at the contexts found in Figure 11-4, shown again in Figure 12-4.

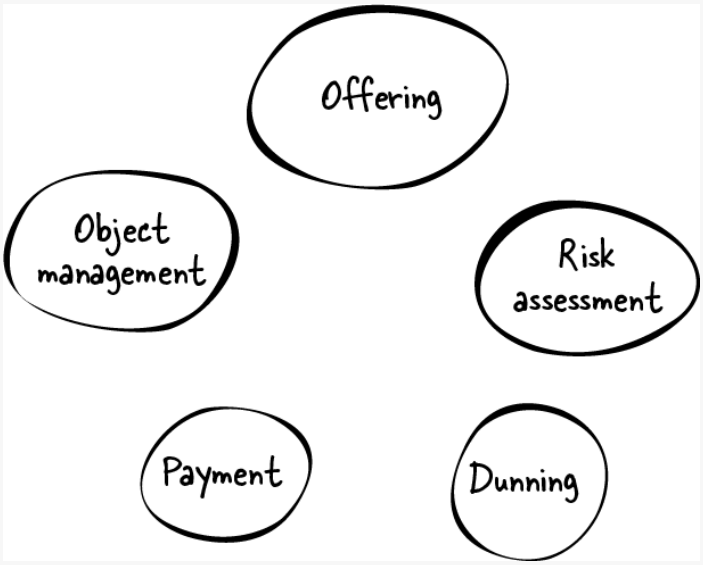


Figure 12-4. The context map of Alphorn Auto Leasing, Inc. with dunning—once again.

Even though in Figure 12-3 none of the contexts from Figure 12-4 can be seen in the source code, we still try to map the existing classes to the planned contexts. We start with the backend and draw in the contexts "offering" (dashed line) and "risk assessment" (dotted line, see Figure 12-5).

Phew, that looks pretty confusing. In addition, the lines cross; there is no sign of a clear demarcation. Although the `Calculator` class can be clearly assigned to the "offering" context and the `PartnerService` to the "risk assessment" context, the other classes are used in both contexts. A clear case of an unbounded model!

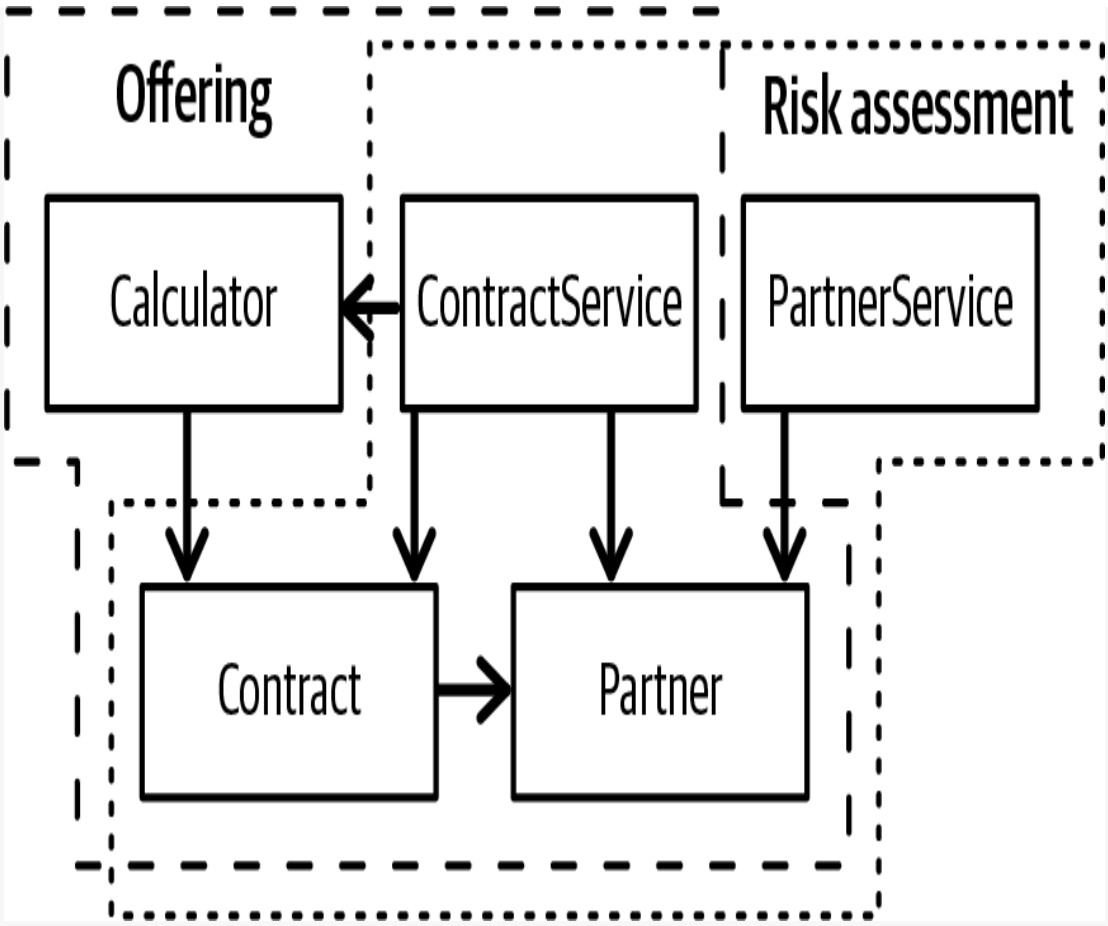


Figure 12-5. Actual and target architecture in the backend: allocation of classes to contexts.

Challenge: Unbounded Models

We refer to an *unbounded model* as a model that does not exist neatly in one bounded context, but crosses boundaries. Unboundedness occurs in domain models and data models as well as in the interface and at the service level. In each layer of the system, we find its own variant of an unbounded model:

- In business logic, there is an *unbounded domain model* with one large representation for each business concept.
- The large domain classes are managed by *unbounded services*.
- The system is based on an *unbounded data model* in one database analogous to the domain model.

- The interface also incorporates an *overall view* of these domain concepts.

Often these unbounded models occur in combination because they are mutually dependent: the domain model forms the core of the application. If the domain model crosses context boundaries, then the data model, the services, and the interface are usually also affected.

Decomposition from Inside to Outside

During the transformation, we therefore start by decomposing the unbounded domain model and then work our way out. In some cases, we approach the domain model and services together. We can subsequently decompose the data model and the interface more easily with the knowledge gained from the new small domain models per context. For more on all that, see the subsequent sections that begin with “Challenge.”

In the next field story, we will show you the impact of an unbounded domain model in an architecture review.

CAROLA'S FIELD STORY: AN UNBOUNDED DOMAIN MODEL

Some time ago, my team and I were called by a company that had a system of one million lines of code (LOC) in Java and would like to decompose it into microservices for better scalability. When we asked, "What kind of architecture does the system currently have?" we got a surprising answer that gave us hope that this decomposition might be possible: "We have an architecture by use cases." A system with a domain-oriented architecture! Full of anticipation, we set to work on the review.

In Figure 12-6 you can see the first result of a mapping of the package structure to architectural elements. Each rectangle contains a portion of the package tree and thus all the classes that are in these packages. If you press the + before one of the rectangles, the rectangle expands to reveal the package tree and the classes it contains. From top to bottom, you can see the names of the use cases noted on the rectangles. Here we have anonymized them—`mailing`, etc., are not the real names of the use cases—to keep the client of this review anonymous.

On the right side of the diagram, you can see the relationships represented as arcs between the classes located in the respective use case rectangles. The arcs on the left side of the axis, which appear green in the ebook, represent relationships from top to bottom. The arcs on the right side of the axis, which are red in the ebook, are relationships that go from bottom to top. There are relatively few relationships at the top of the diagram, and you could imagine that the first three use cases, `mailing`, `importexport`, and `remove`, belong together—after all, they have relationships to each other. However, they have nothing to do directly with the next use case, `startup`.

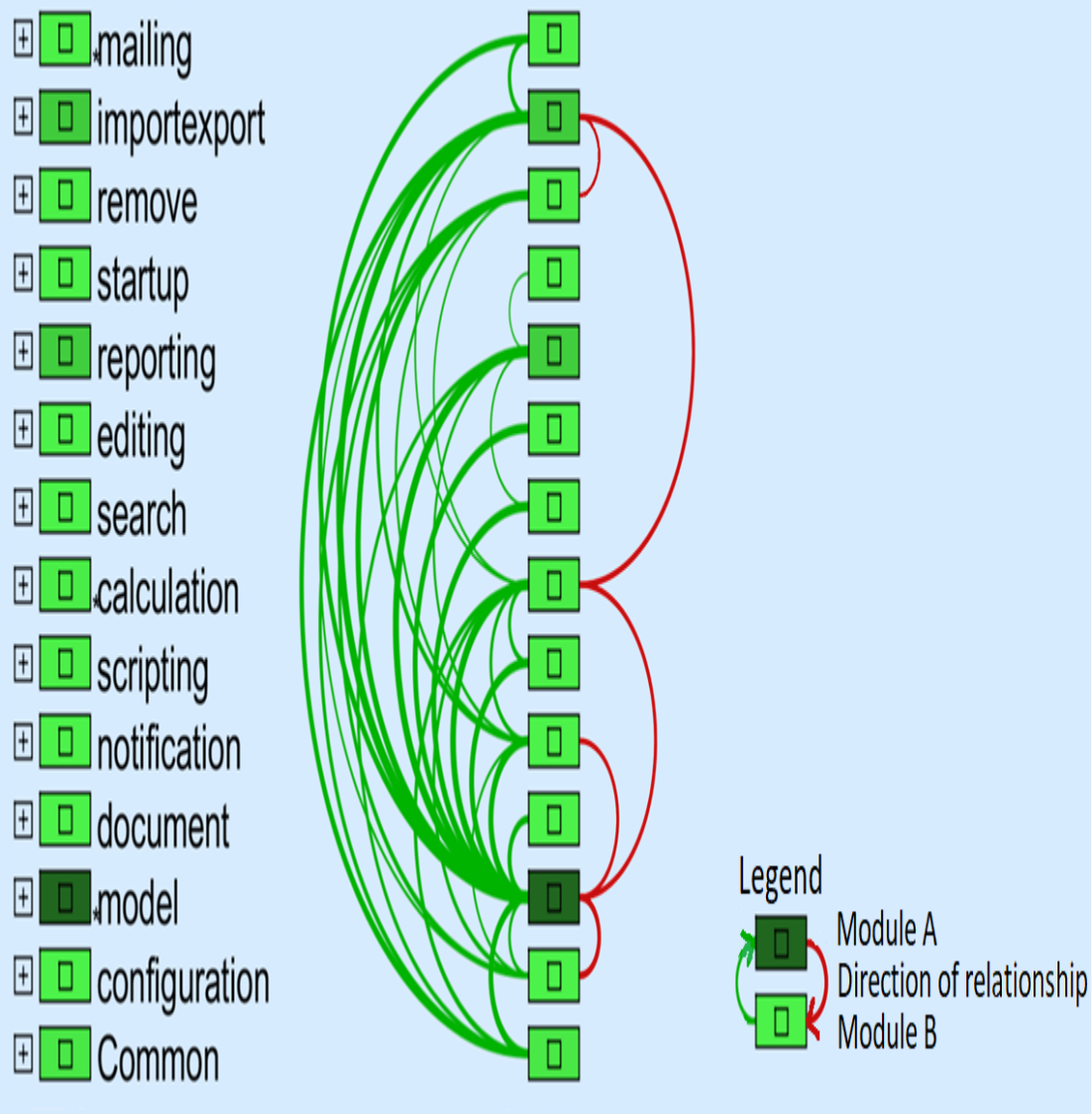


Figure 12-6. Screenshot of the architecture review tool: architecture according to use cases.

But the further down you go, the more relationships there are, and finally you end up with `model`. When we asked what that use case was, the architect of the system told us, "That's the domain model!" Oh dear! It all started so hopefully with an architecture based on use cases. In order to move forward at this point, we asked the architect whether it would be possible to assign the classes from `model` to the respective use case to which they belong from a domain perspective. The architect was happy to give it a try, and step-by-step we started moving classes from `model` into the use case architecture elements. At about the `calculation` use case, we stopped because we could see in the tool

the entangled structure created by the domain model. In Figure 12-7 you can see the structure created by the move. A great many relationships on both sides of the axis have been added.

If you zoom further into these relationships, you can see that these many additional relationships have two causes:

- The *classes in the unbounded domain model itself* that use and need each other, and whose relationships only become visible when they are distributed among the use cases
- The *relationships of the use case classes* to the classes of the unbounded domain model, because the use cases do need more than just the domain model classes that we have assigned to the respective use case

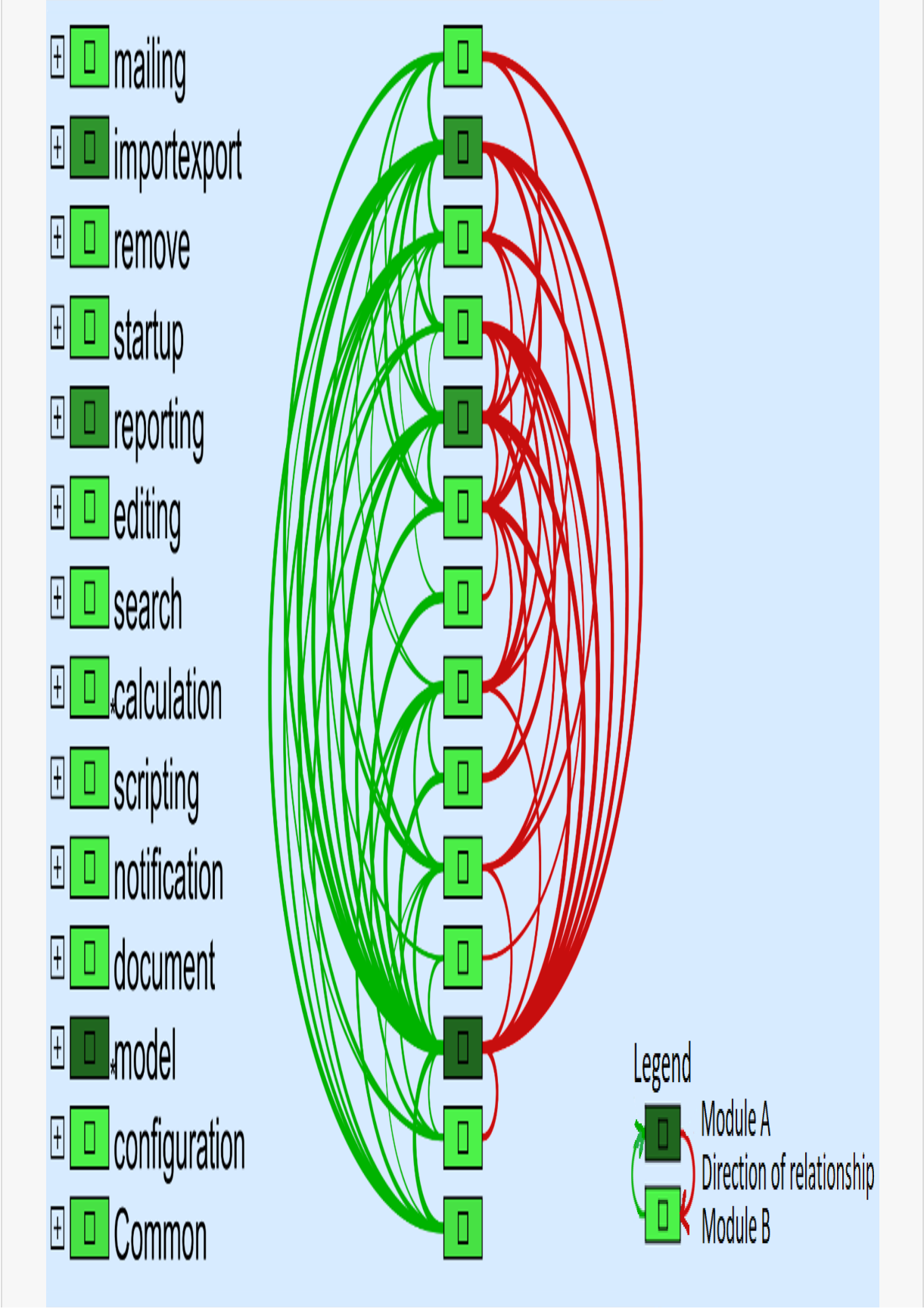

mailing
importexport
remove
startup
reporting
editing
search
calculation
scripting
notification
document
model
configuration
Common
Legend
Module A
Direction of relationship
Module B

Figure 12-7. Screenshot of the architecture review tool: architecture according to use cases with split domain model.

Such structures entangled via the domain model can be found in any system with an unbounded domain model.

Challenge: The Unbounded Domain Model

From the developers' point of view, such an unbounded model came about quite naturally and represents the domain as detailed as possible in all aspects. As soon as new features were added to the existing system, it seemed quite normal to take the existing classes and database tables from the domain model and extend them if necessary. So the central database tables got more and more columns, and the central classes got more and more methods and attributes.

In many legacy systems, you will therefore find very large classes for the central concepts of the domain model. Typical representatives are: `Product`, `Contract`, `Customer`, or whatever is central in your application domain. If there is already a class `Product` in the source code repository and functionality is developed that the `Product` needs, then the one class `Product` is used and extended to fit. The idea is to reuse the existing class and only have to look in one place if an error occurs with the `Product`. This way the central classes got one or two new methods and attributes with each new functionality.

Problem Cause: Reusability Misunderstood

It is only a pity that these new methods are not needed at all in the rest of the system, but were built in only for the new functionality. From the viewpoint of the reuse of existing classes, this is a logical consequence, and thus these classes grow ever larger and larger. The annoying thing is that the architecture is thereby very strongly coupled on the level of the domain models.

In addition, the unbounded domain models with their large core classes and tables eventually become so complex that the architecture as a whole is no longer understandable and therefore becomes more error-prone with each change. Unfortunately, many teams create this problem for themselves. Believing in reuse, they develop a software system that contains several entangled domain models without clear boundaries. Different concepts, which are actually not related to each other, are thus distributed over many modules and connected by contradictory elements. As multiple teams work simultaneously on the large domain model, they interfere with each other. Speed suffers and the independence of the teams is no longer a given.

Let’s look at the banking example to see how such an unbounded domain model can be decomposed.

CASE STUDY “BANK”

IN THE BANKING EXAMPLE: REFACTORING OF THE UNBOUNDED DOMAIN MODEL

In Chapter 8, we learned about the bank's domain model and enriched it in a domain-driven way. Nevertheless, our bank still has a monolith and wants a decomposition.

We try to apply the bounded contexts found in the EventStorming in Chapter 4 to the source code of the bank and get Figure 12-8. We see an unbounded domain model that is shared by the domain-driven modules built on top of it. Fortunately, the domain-driven modules `monetary transactions`, `lending`, and `stock trading` fit well with the bounded contexts we found in the EventStorming (see Figure 4-14).

So our task now is to split up the domain model. The class `Customer`, for example, contains functionality for the creditworthiness of the customer, which is only needed in the area of lending—but is of no interest in monetary transactions or stock trading.

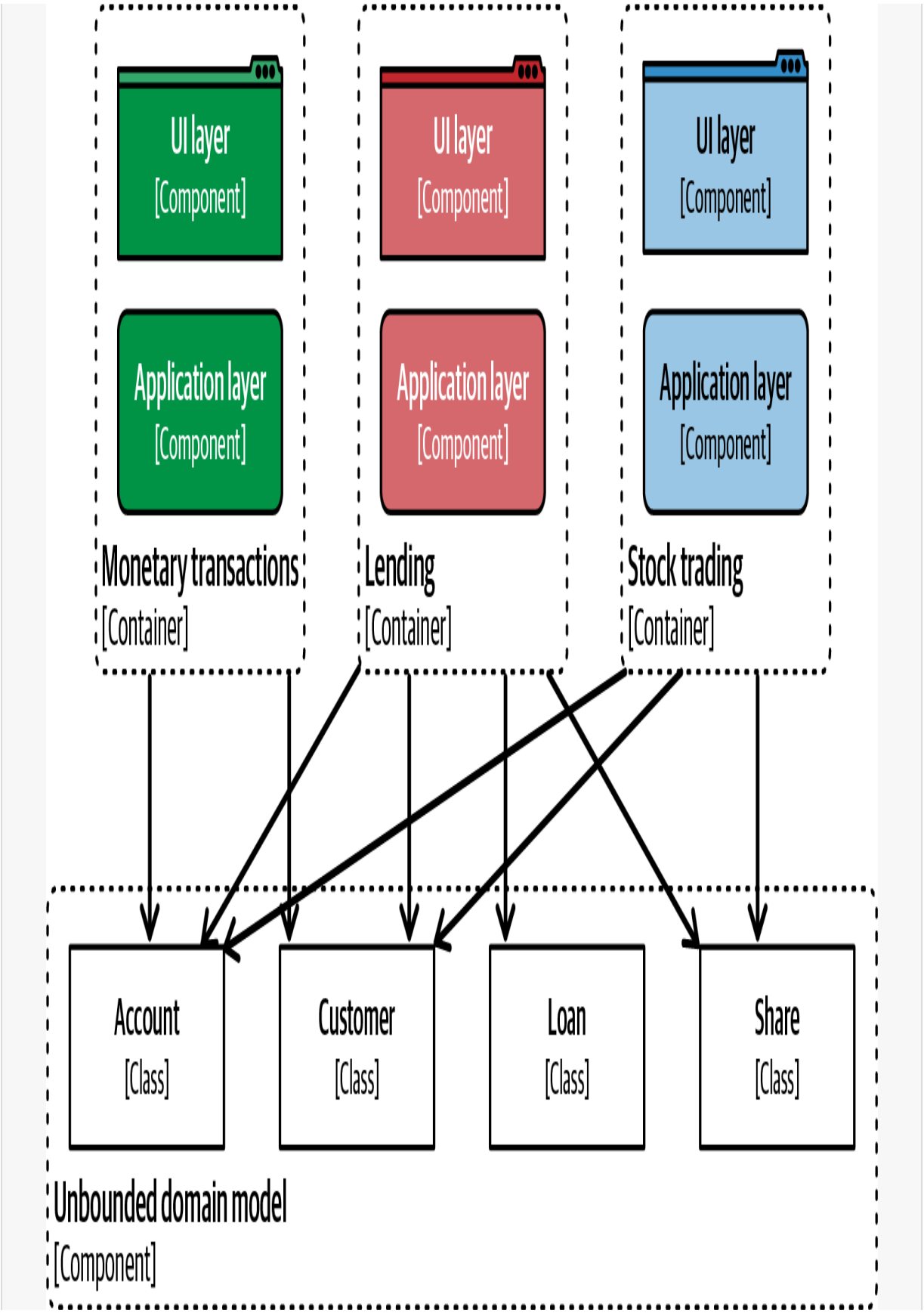

UI layer
[Component]
Application layer
[Component]
Monetary transactions
[Container]
UI layer
[Component]
Application layer
[Component]
Lending
[Container]
UI layer
[Component]
Application layer
[Component]
Stock trading
[Container]
Account
[Class]
Customer
[Class]
Loan
[Class]
Share
[Class]
Unbounded domain model
[Component]

Figure 12-8. Banking system with one large domain model.

Our goal is for each individual bounded context to have its own small domain model. Figure 12-9 shows schematically what our target picture looks like. The respective context-specific concepts of the local domain models are specialized according to the needs of each bounded context.

For “customer,” there are different models with the same name in each case. The term “account” has been sharpened into the terms “current account,” “credit account,” and “securities account.”

The step from an unbounded domain model (see Figure 12-8) to small context-specific domain models (see Figure 12-9) is a Herculean task in most large systems. The parts of the system based on the domain model are too entangled with the classes of the domain model.

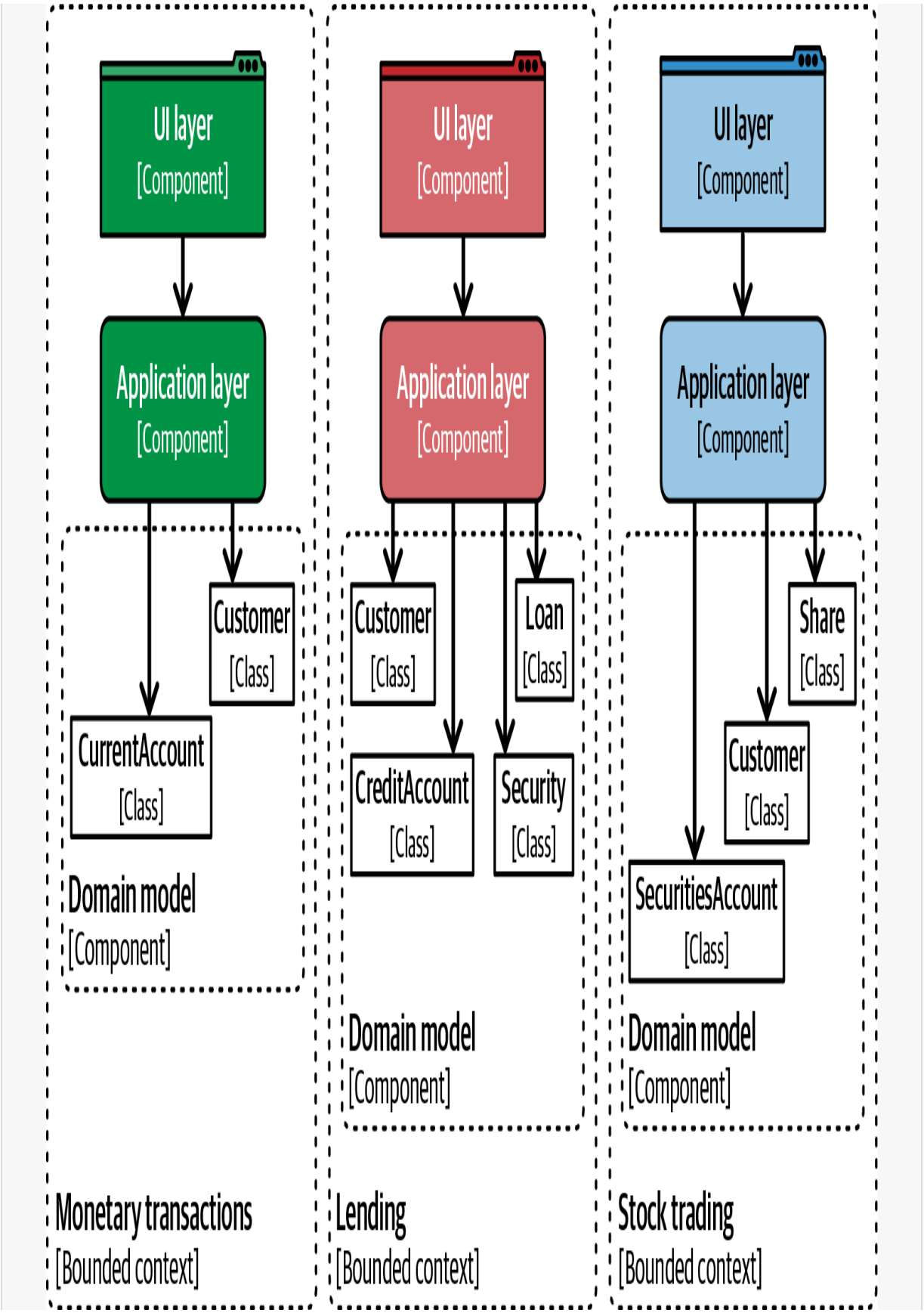

UI layer
[Component]
Application layer
[Component]
Customer
[Class]
CurrentAccount
[Class]
Domain model
[Component]
Monetary transactions
[Bounded context]
UI layer
[Component]
Application layer
[Component]
Customer
[Class]
Loan
[Class]
CreditAccount
[Class]
Security
[Class]
Domain model
[Component]
Lending
[Bounded context]
UI layer
[Component]
Application layer
[Component]
Share
[Class]
Customer
[Class]
SecuritiesAccount
[Class]
Domain model
[Component]
Stock trading
[Bounded context]

Figure 12-9. Banking system with specific domain models.

Challenge: The Unbounded Anemic Domain Model

If you have to deal with an anemic domain model, the situation is usually even worse. In addition to the many large classes that are used everywhere, all fields are also accessible via “setter” and “getter” methods. The classes have no methods with domain behavior, but are pure data classes.

If you look for the domain logic, you will usually find it above the domain model in the services or even in the user interface classes. In the next example, we therefore decompose the unbounded anemic domain model together with the associated service classes into two bounded contexts.

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: REFACTORING OF THE UNBOUNDED ANEMIC DOMAIN MODEL, INCLUDING SERVICES

The last time we put our context map over the source code, it looked like the one shown in Figure 12-10.

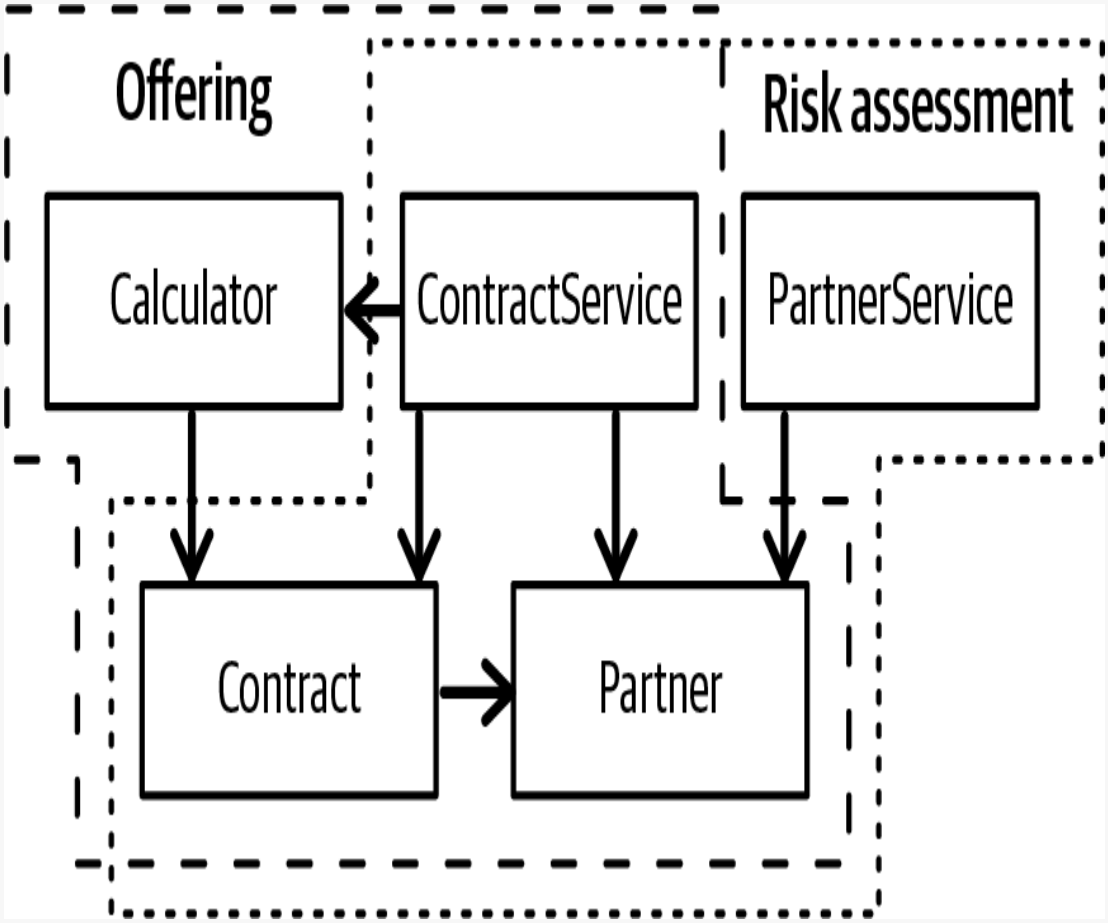


Figure 12-10. As-is and target architecture in the backend: allocation of classes to contexts (again).

To bring more clarity into this picture, we duplicate the classes needed by both contexts and get Figure 12-11.

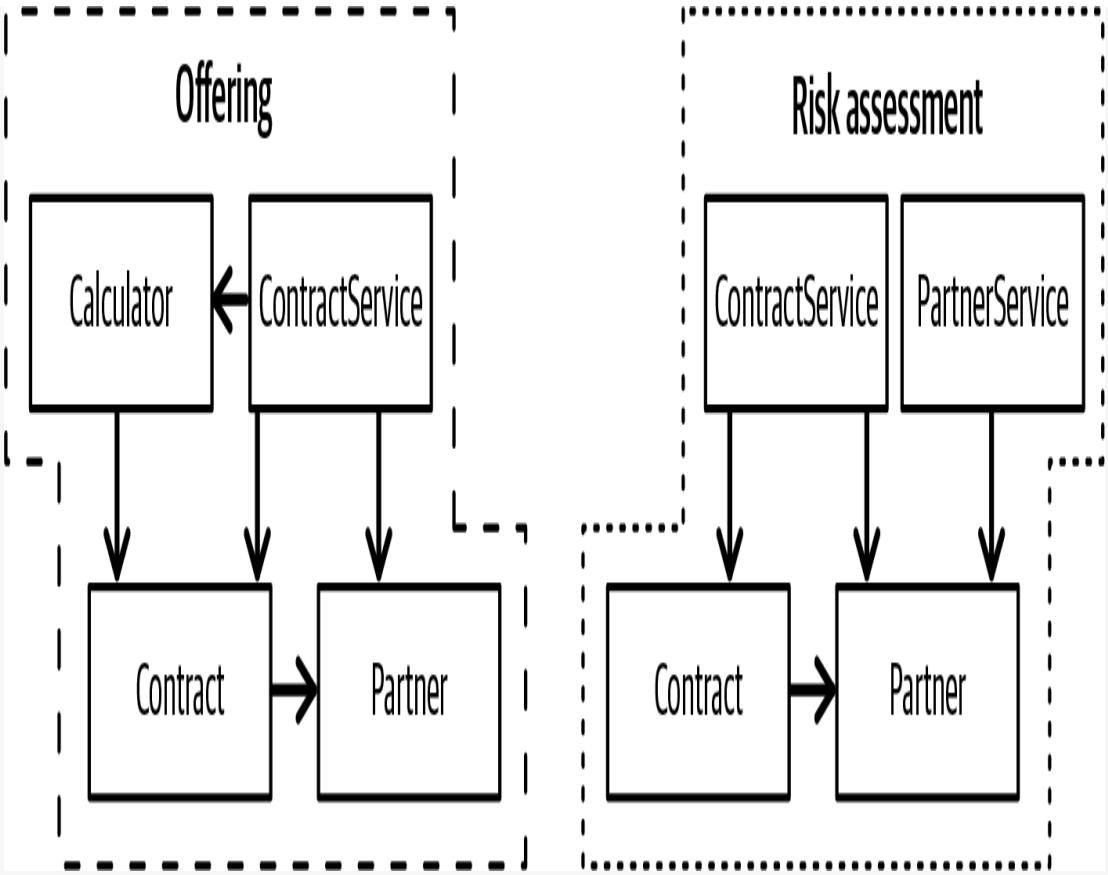


Figure 12-11. The contexts with duplicated classes.

A proper UML diagram would look like Figure 12-12.

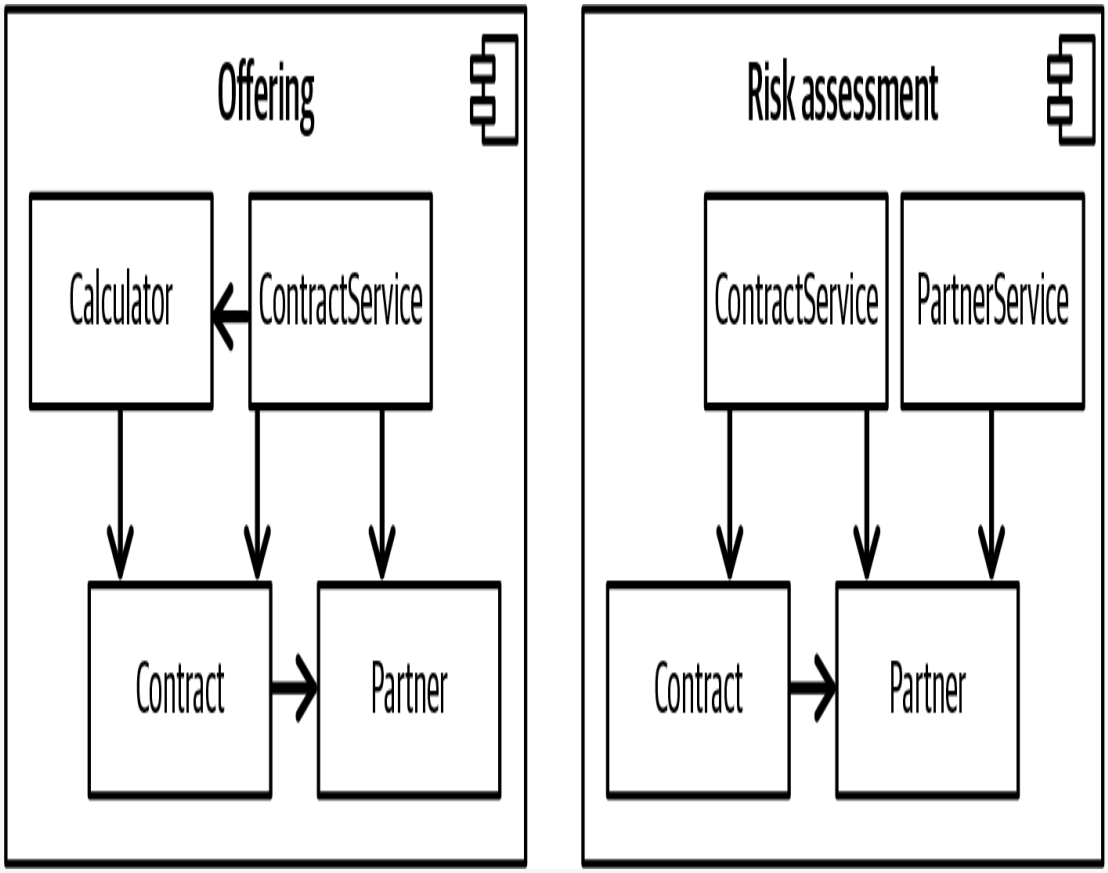


Figure 12-12. The contexts with duplicated classes—tidy.

We start with the classes `ContractService` and `Contract`, and zoom into them to see their methods and fields (see Figure 12-13).

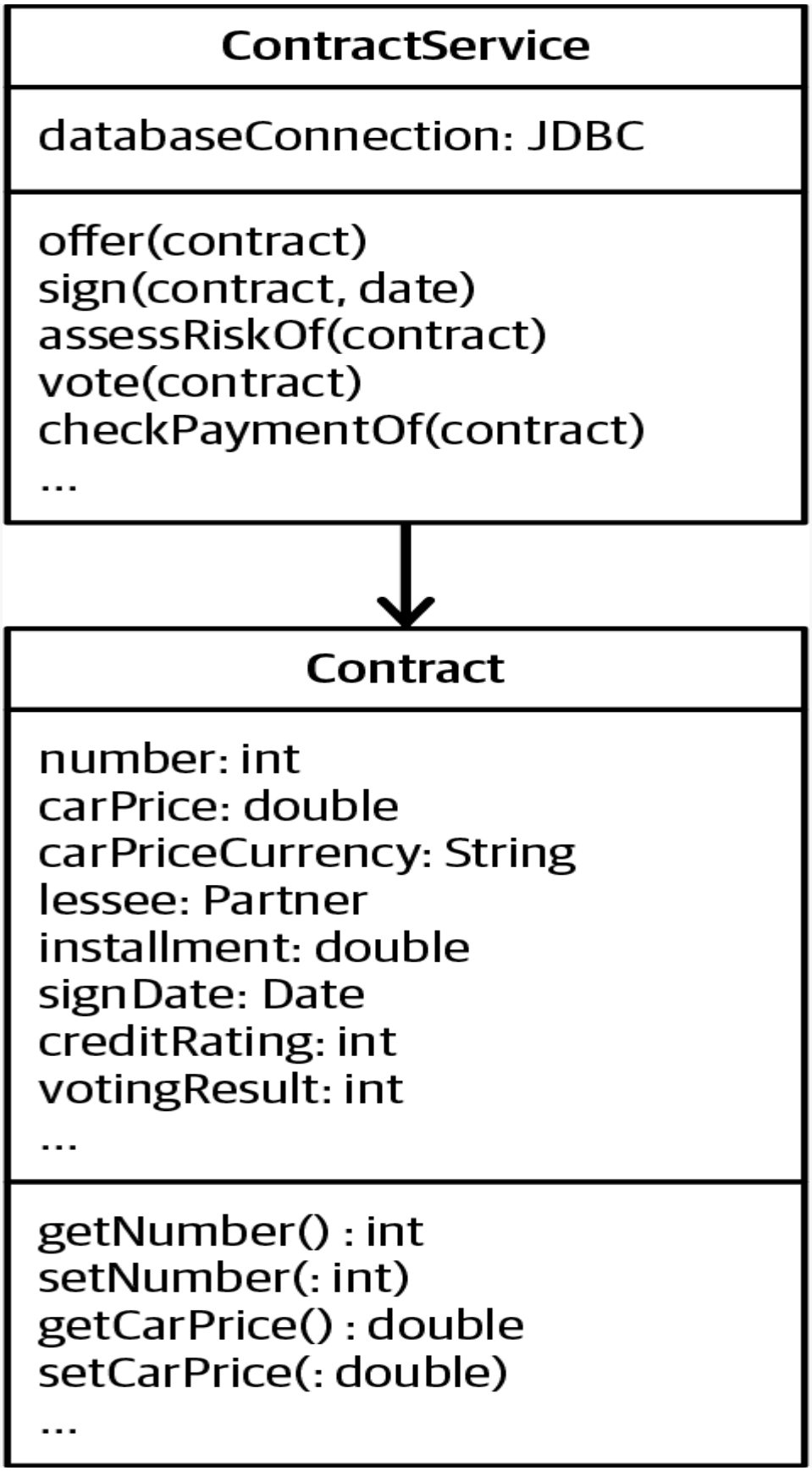

ContractService
databaseConnection: JDBC
offer(contract)
sign(contract, date)
assessRiskOf(contract)
vote(contract)
checkPaymentOf(contract)
...
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
creditRating: int
votingResult: int
...
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
...

Figure 12-13. The actual architecture at the method and field level.

Then we map which methods and fields are needed in the bounded contexts “offering” and “risk assessment,” and arrive at Figure 12-14.

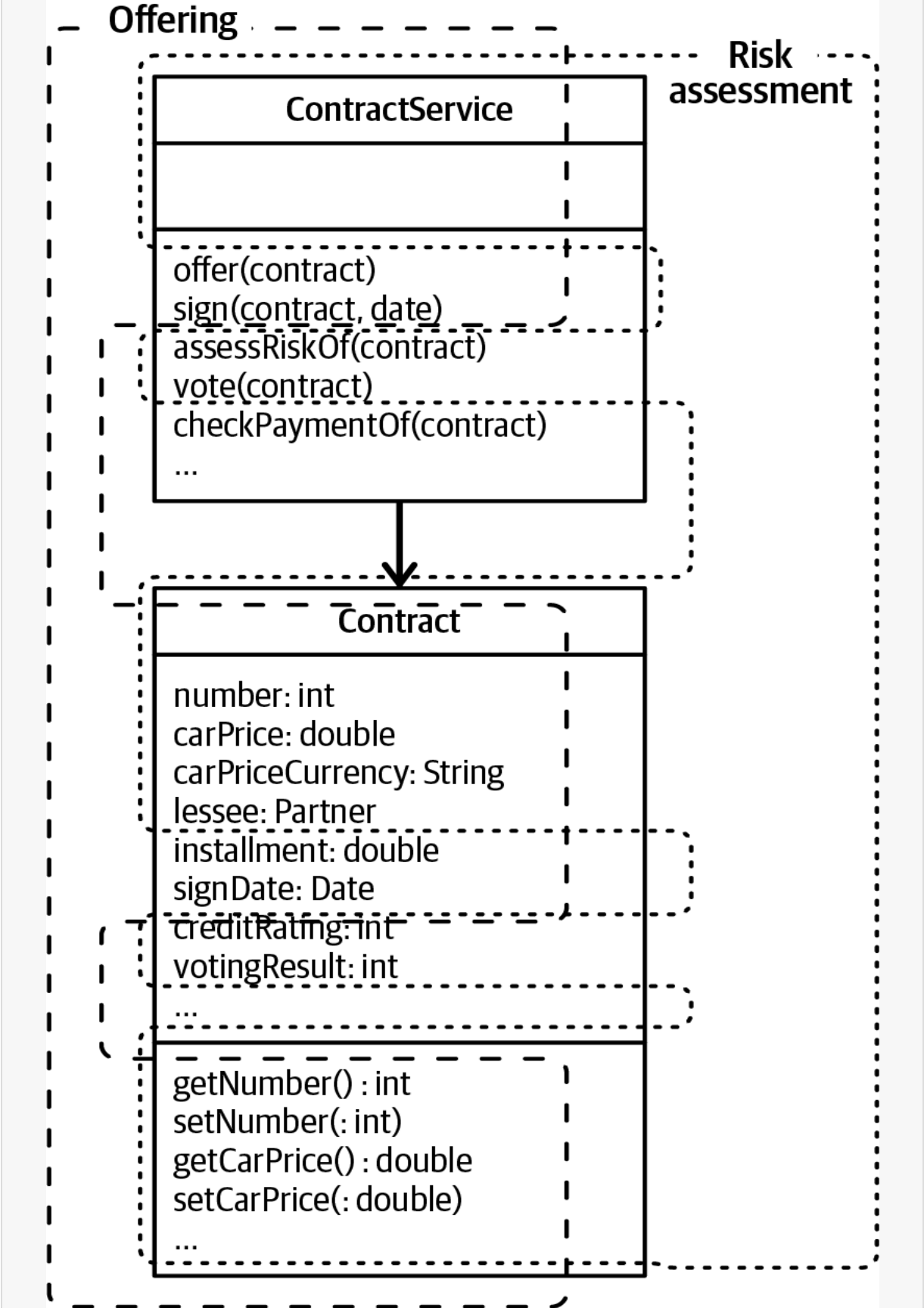

Offering
Risk assessment
ContractService
offer(contract)
sign(contract, date)
assessRiskOf(contract)
vote(contract)
checkPaymentOf(contract)
...
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
creditRating: int
votingResult: int
...
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
...

Figure 12-14. Mapping of actual architecture and target architecture on method and field level.

Aha, again quite a mess. Let's pull this apart by first completely doubling the classes (see Figure 12-15).

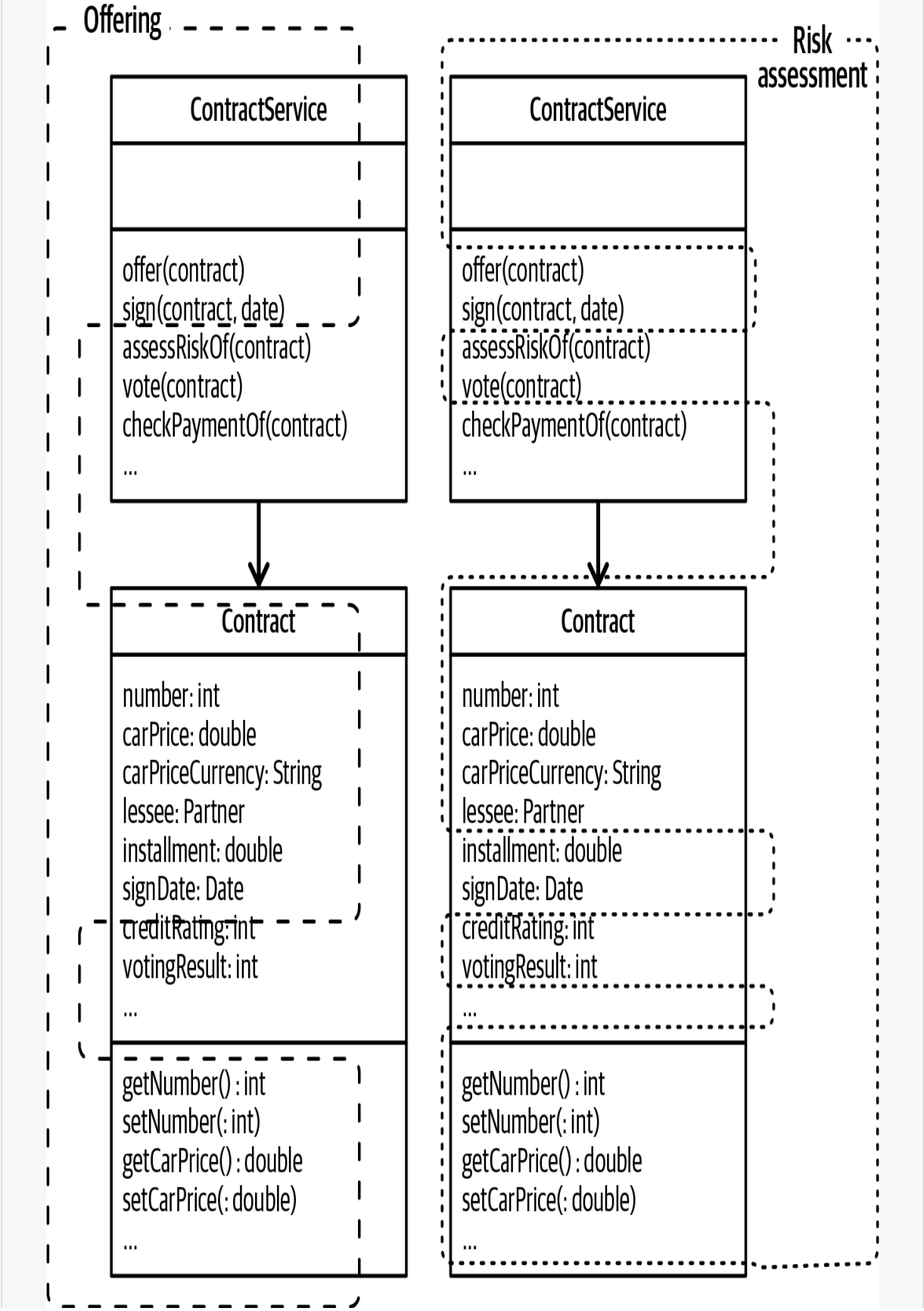

Offering
Risk assessment
ContractService
offer(contract)
sign(contract, date)
assessRiskOf(contract)
vote(contract)
checkPaymentOf(contract)
...
ContractService
offer(contract)
sign(contract, date)
assessRiskOf(contract)
vote(contract)
checkPaymentOf(contract)
...
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
creditRating: int
votingResult: int
...
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
...
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
creditRating: int
votingResult: int
...
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
...

Figure 12-15. Mapping of actual architecture and target architecture on method and field level —pulled apart.

We take a closer look and see that methods do not need to be duplicated at all, and fields only partially. So we throw out the fields and methods that are outside the boundaries (see Figure 12-16).

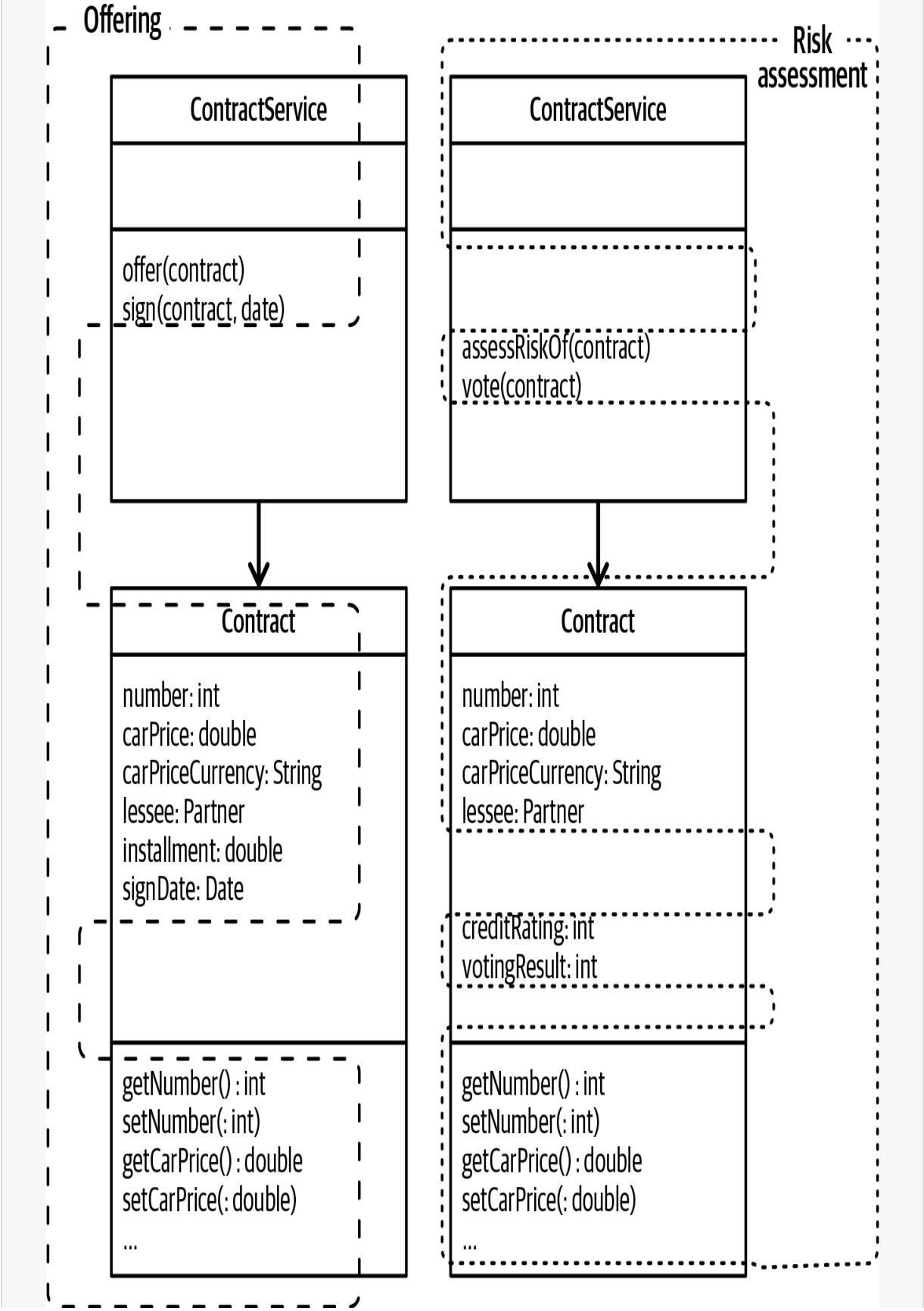

Offering
Risk assessment
ContractService
offer(contract)
sign(contract, date)
ContractService
assessRiskOf(contract)
vote(contract)
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
...
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
creditRating: int
votingResult: int
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
...

Figure 12-16. Mapping of actual architecture and target architecture on method and field level —superfluous parts deleted.

Again, at this level we make a proper UML diagram out of this and get Figure 12-17.

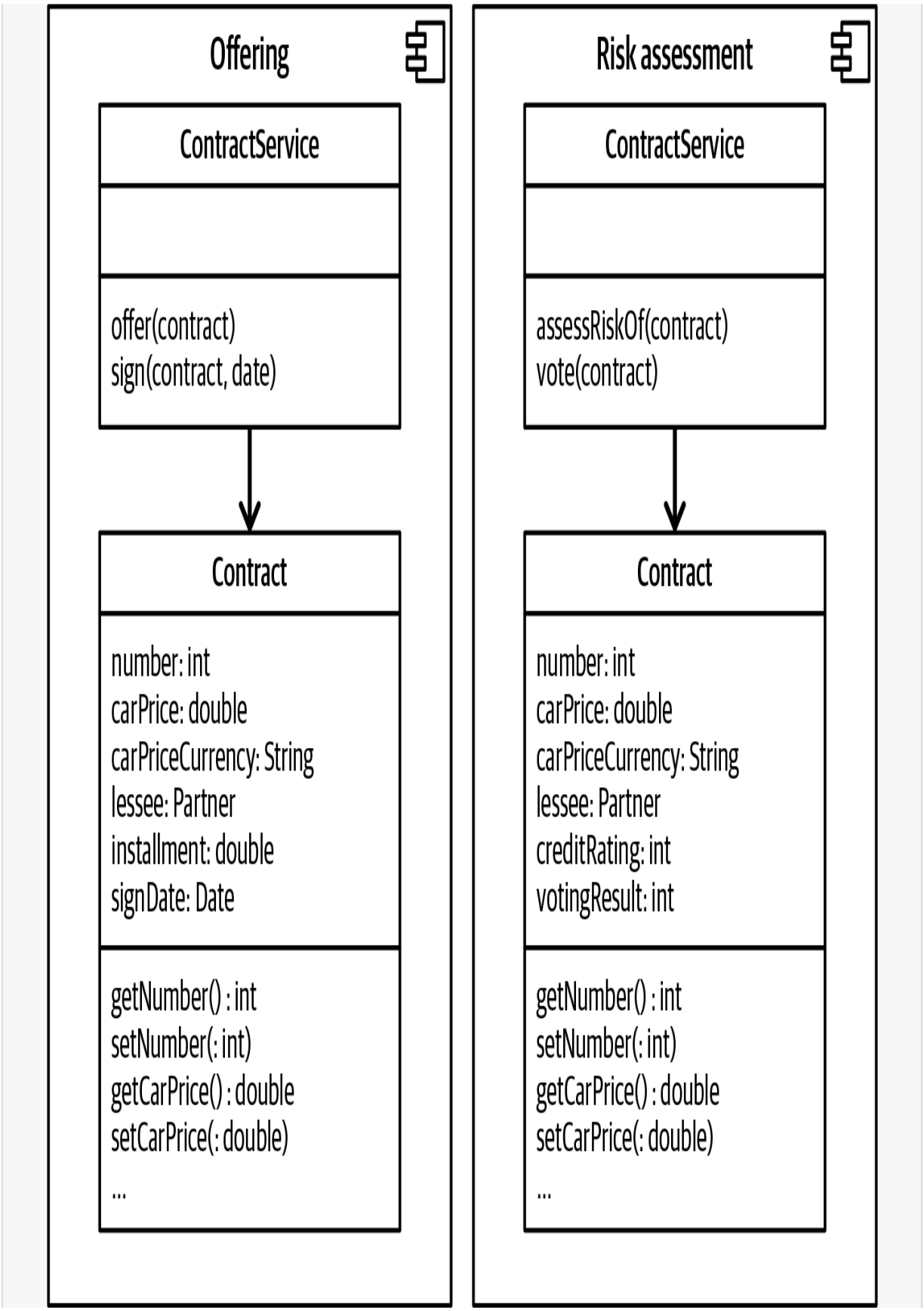

Offering
ContractService
offer(contract)
sign(contract, date)
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
...
Risk assessment
ContractService
assessRiskOf(contract)
vote(contract)
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
creditRating: int
votingResult: int
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
...

Figure 12-17. Mapping of actual architecture and target architecture on method and field level —tidy.

Using these classes and bounded contexts as examples, the team works on the other classes and bounded contexts in a similar way.

Overall, this approach duplicates a large portion of the domain and service classes in each bounded context. We then rebuild the domain and service classes for their respective bounded contexts so that there are no longer any duplicates, at least of the domain classes. This is because the duplicates become context-specific domain classes that can be extended and customized by their respective teams independently of the rest of the system.

Certain domain class properties, such as the ID and name, must be kept the same across all bounded contexts in which a variant of a domain class occurs. For example, we need to ensure that the product name for a product ID is the same in all bounded contexts. Changes must be transported between bounded contexts via domain events. In "Challenge: Introducing Domain Events Between Services", we will see how this is done.

In addition, new objects of a domain class must be made known to all bounded contexts that use them when a new object is created in a bounded context. This information is also reported via domain events from the creating bounded context to all other bounded contexts that work with the respective object.

Once the unbounded (possibly anemic) domain model has been decomposed, the natural next step is to adapt the database tables to the new situation. In the anemic domain model, the database table and class for a domain term often have the same structure. Therefore, it is often even possible to perform the decomposition of the two in parallel. For the sake of simplicity, we will do it one after the other here.

Challenge: The Unbounded Data Model

An unbounded data model usually sits below an unbounded domain model. These central tables are used everywhere in the system and are correspondingly difficult to change. In addition to the sheer size, these central tables usually contain columns that are no longer needed. The typical way of dealing with large central tables is that only new columns are added, but existing columns are rarely changed and certainly not deleted.

Refactorings of the database are particularly hard, because in each case three reconstruction measures must be taken, as described in the seminal work on database refactoring [Ambler and Sadalage 2006]:

- Adjustment of the *database schema*
- Adaptation of the *code*, which accesses the database schema
- Migration of the already *existing data*

Let's take a look at this refactoring in our leasing example. More helpful approaches to database decomposition can be found in [Newman 2020].

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: REFACTORING THE DATABASE

The disentanglement team at Alphorn is also taking a chance on the database. Looking across the layer boundary, it has designed an architecture shown in Figure 12-18 with the planned refactoring of the anemic domain model.

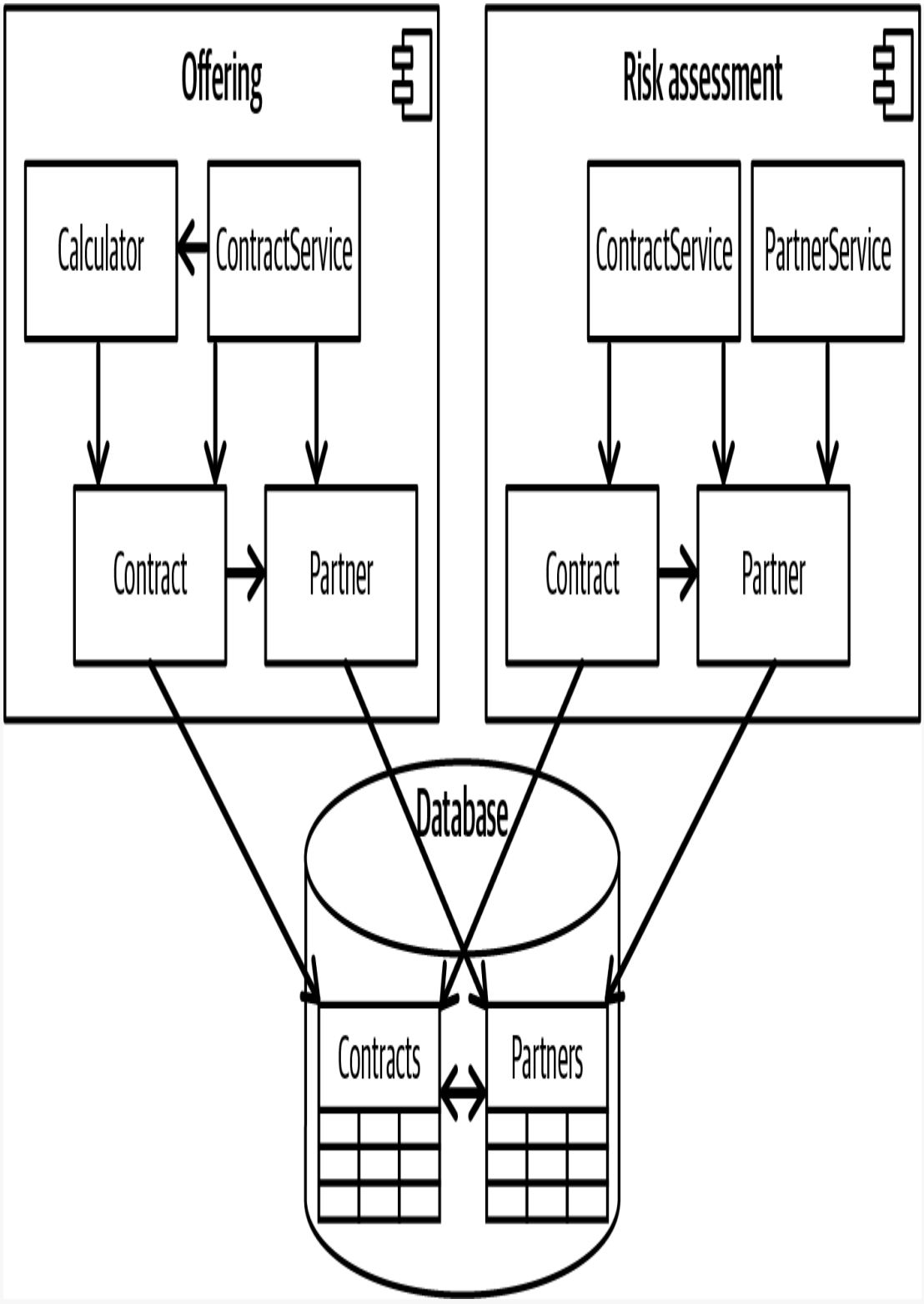


Figure 12-18. Unbounded data model at Alphorn.

Now the team plans the following: just as separate classes for `Contract` and `Partners` will be created for each of the contexts

“offering” and “risk assessment,” separate database schemas `OFFERING` and `RISK_ASSESSMENT` will be created, each with its own database tables `CONTRACTS` and `PARTNERS`. This can be seen in Figure 12-19. So we follow the domain model refactoring in the database by applying *Extract Specialized Table* several times.

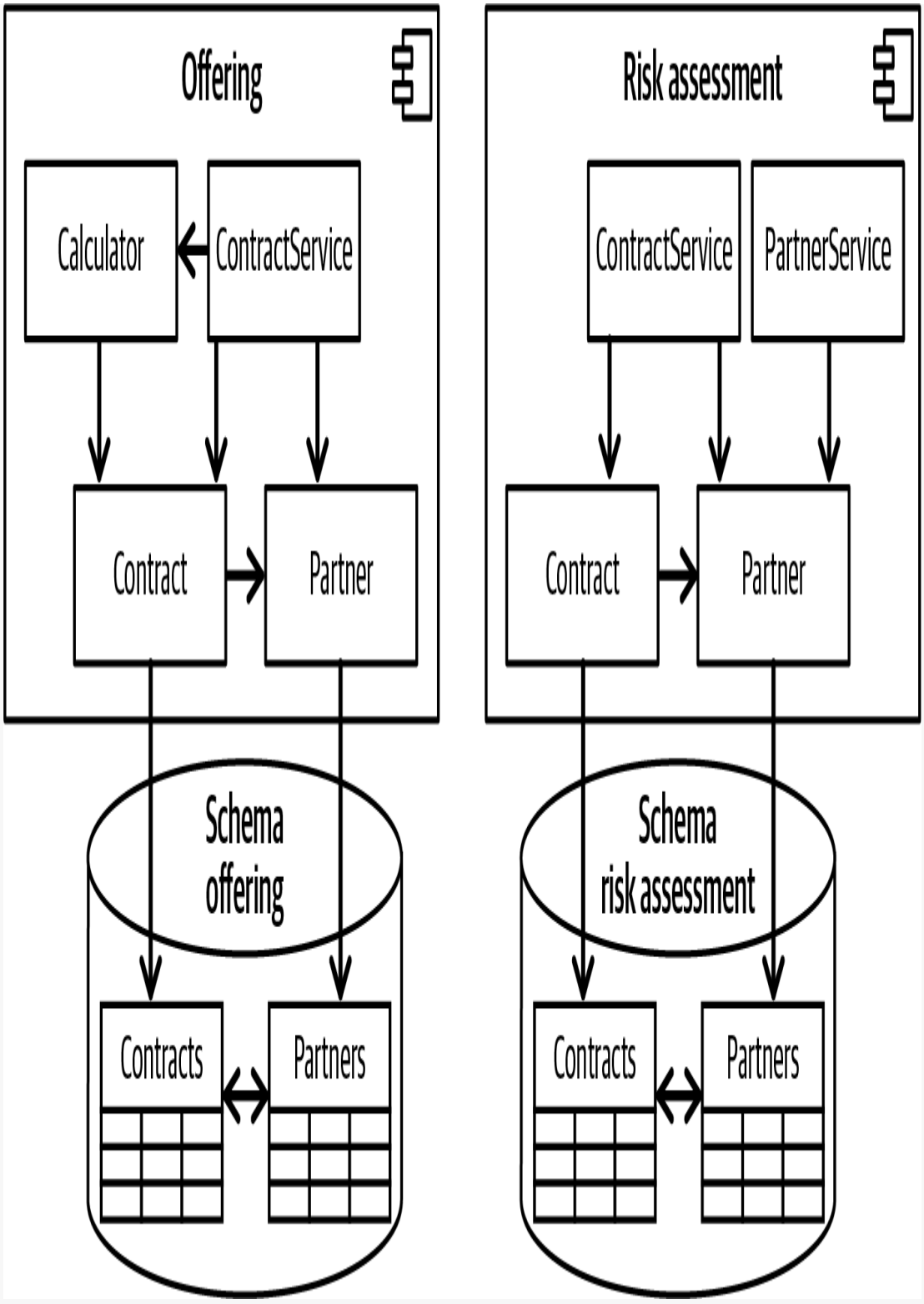


Figure 12-19. Alphorn data models decomposed by context.

Now that we have taken the database apart, we turn to the user interface.

Challenge: Decomposing the User Interface

A difficult issue in decomposing legacy systems into bounded contexts is the question of how to decompose the user interface and then implement it in an architecture built from bounded contexts. This question can be dealt with differently depending on the goal.

In the next two subsections, we show the fundamental difference between one large frontend, which sits separately on top of the bounded contexts, and microfrontends, each of which is developed in the bounded context and integrated in a lightweight manner.

Monolithic UI

In some architectures, the bounded contexts do not contain a UI. The user interface is layered over all bounded contexts as a monolithic component (see Figure 12-20).

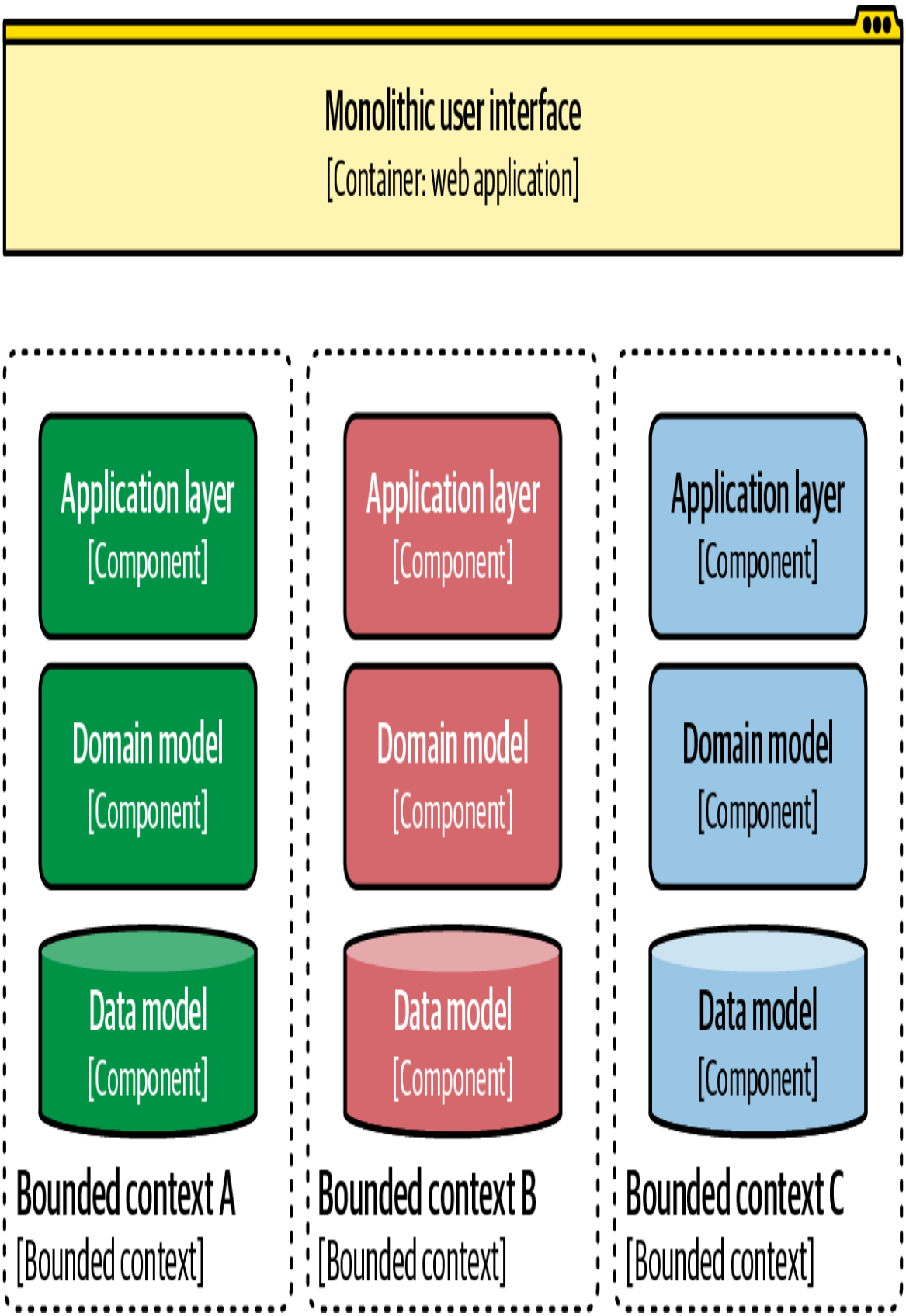


Figure 12-20. Monolithic UI over contexts.

The *benefit*: the user interface can be designed and styled from "a single mold." The *drawback*: the team responsible for the user interface is dependent on all underlying bounded contexts and therefore often has to re-release their components as a whole (when one of the contexts changes). In addition, features can usually no longer be implemented within a team by all layers because the user interface is separated. In Chapter 9 we have listed further drawbacks of layer teams.

UI in Context: Microfrontends

In other architectures, each bounded context contains its own user interface. At *martinfowler.com* you can find a comprehensive article on microfrontends by Sam Jackson (see [Jackson 2019]), which lists all currently common variants. We present two of them here to make the problem clear.

The interfaces of the bounded contexts either stand side by side and are essentially integrated only via hyperlinks (see Figure 12-21), almost comparable to completely separate applications.

Alternatively, the user interfaces of the bounded contexts can also be technically integrated via a common framework and dynamic inclusion (see Figure 12-22).

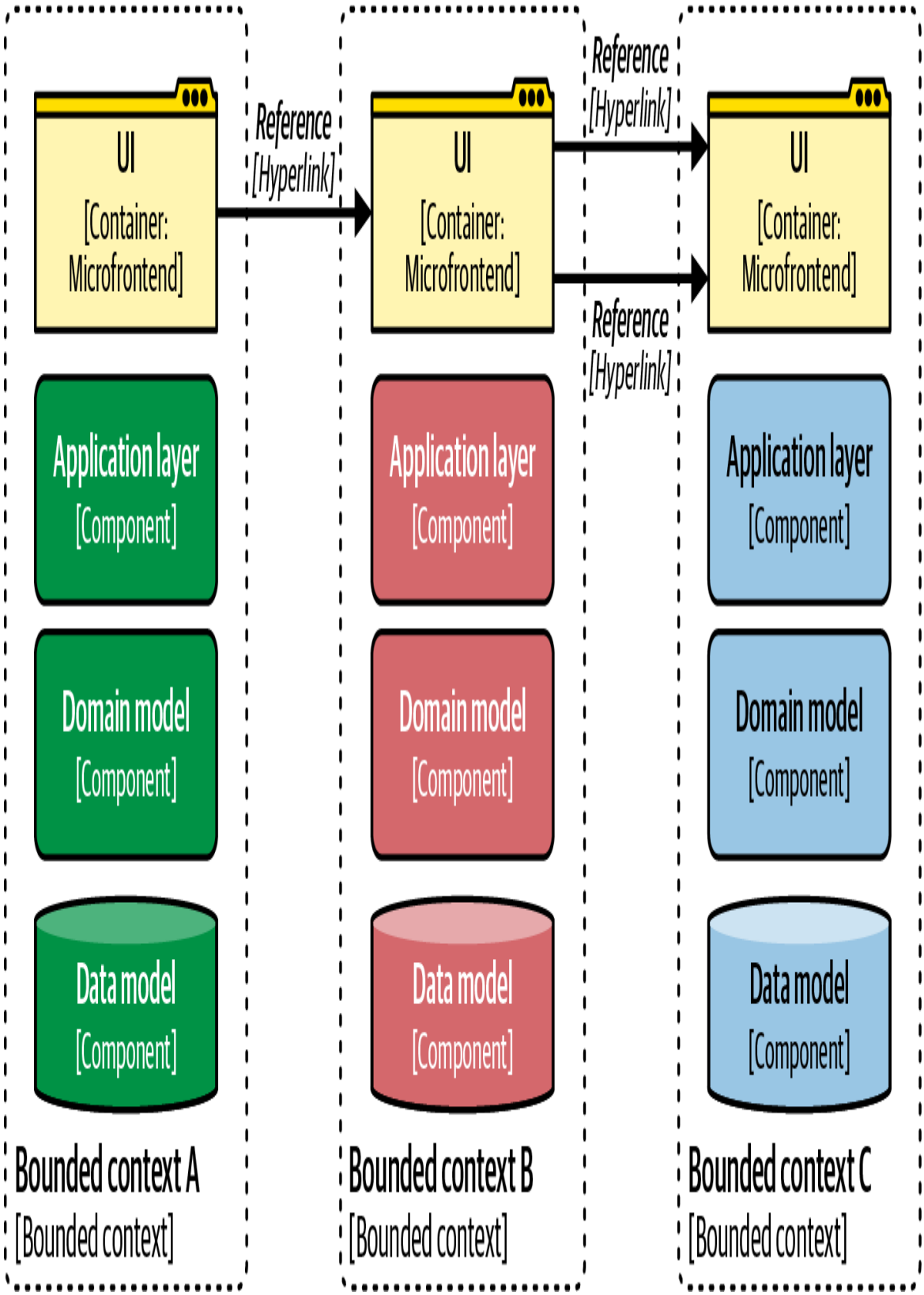


Figure 12-21. UI in every context.

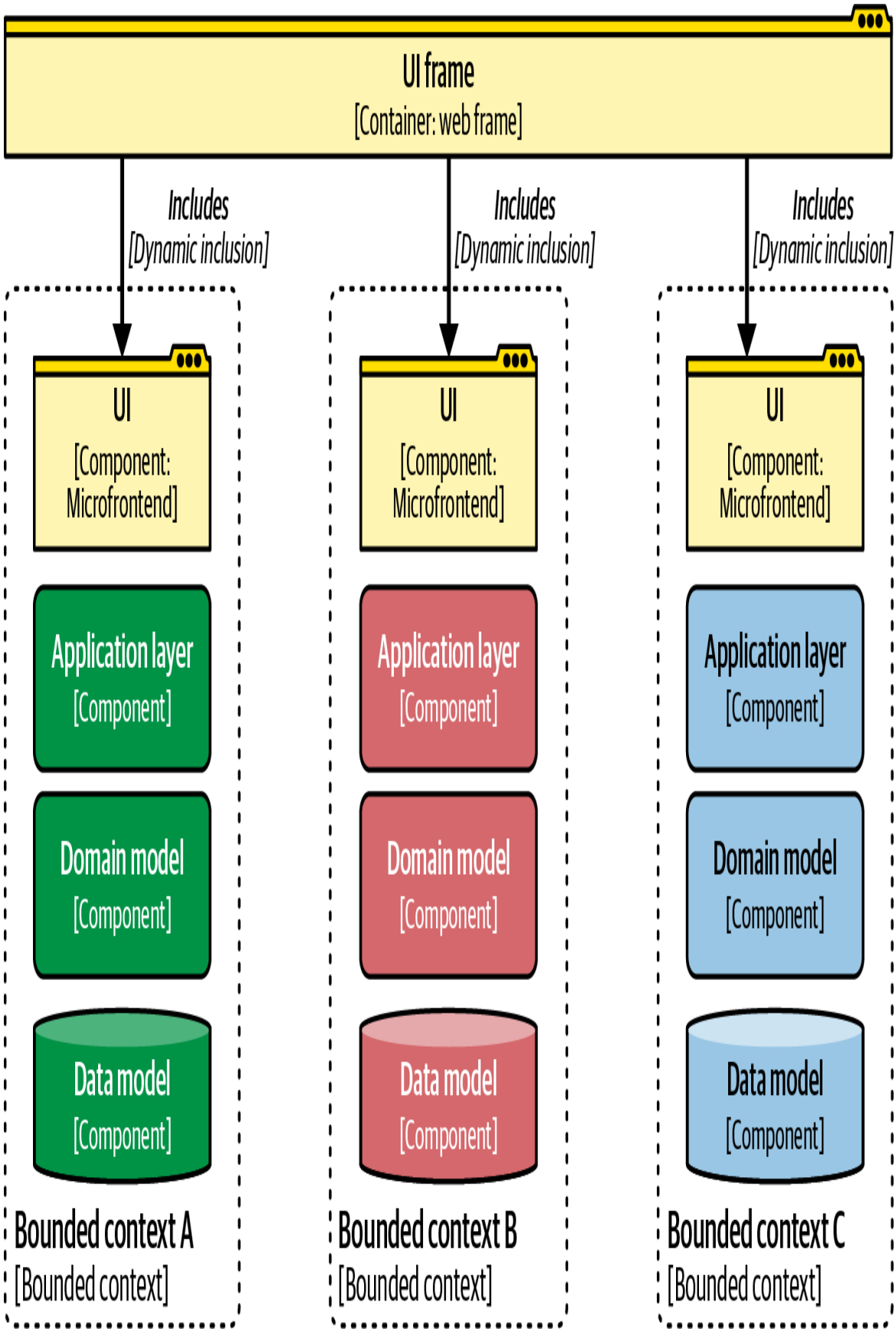

UI frame
[Container: web frame]
Includes
[Dynamic inclusion]
Includes
[Dynamic inclusion]
Includes
[Dynamic inclusion]
UI
[Component: Microfrontend]
UI
[Component: Microfrontend]
UI
[Component: Microfrontend]
Application layer
[Component]
Application layer
[Component]
Application layer
[Component]
Domain model
[Component]
Domain model
[Component]
Domain model
[Component]
Data model
[Component]
Data model
[Component]
Data model
[Component]
Bounded context A
[Bounded context]
Bounded context B
[Bounded context]
Bounded context C
[Bounded context]

Figure 12-22. UI in every context with integration.

Whatever type of integration is allowed, the *benefit* of bounded contexts with their own UI is that teams can implement features independently of others across all layers. The *drawback*: making the user interface coherent across all bounded contexts involved is challenging; maintaining a consistent look and feel is also difficult.

For strategic design, it is of great advantage to clarify in advance how the user interface is to be constructed. This is because the way the UI is constructed has consequences at the strategic level:

- Which domain events (see Figure 12-23) couple which bounded contexts together?
- Which runtime dependencies exist?
- Which functionalities are implemented redundantly, which are not?

If, for example, it is possible to integrate functionalities of other bounded contexts at the level of the user interface, domain events can possibly be omitted in return, and the integration at the logic level can be reduced.

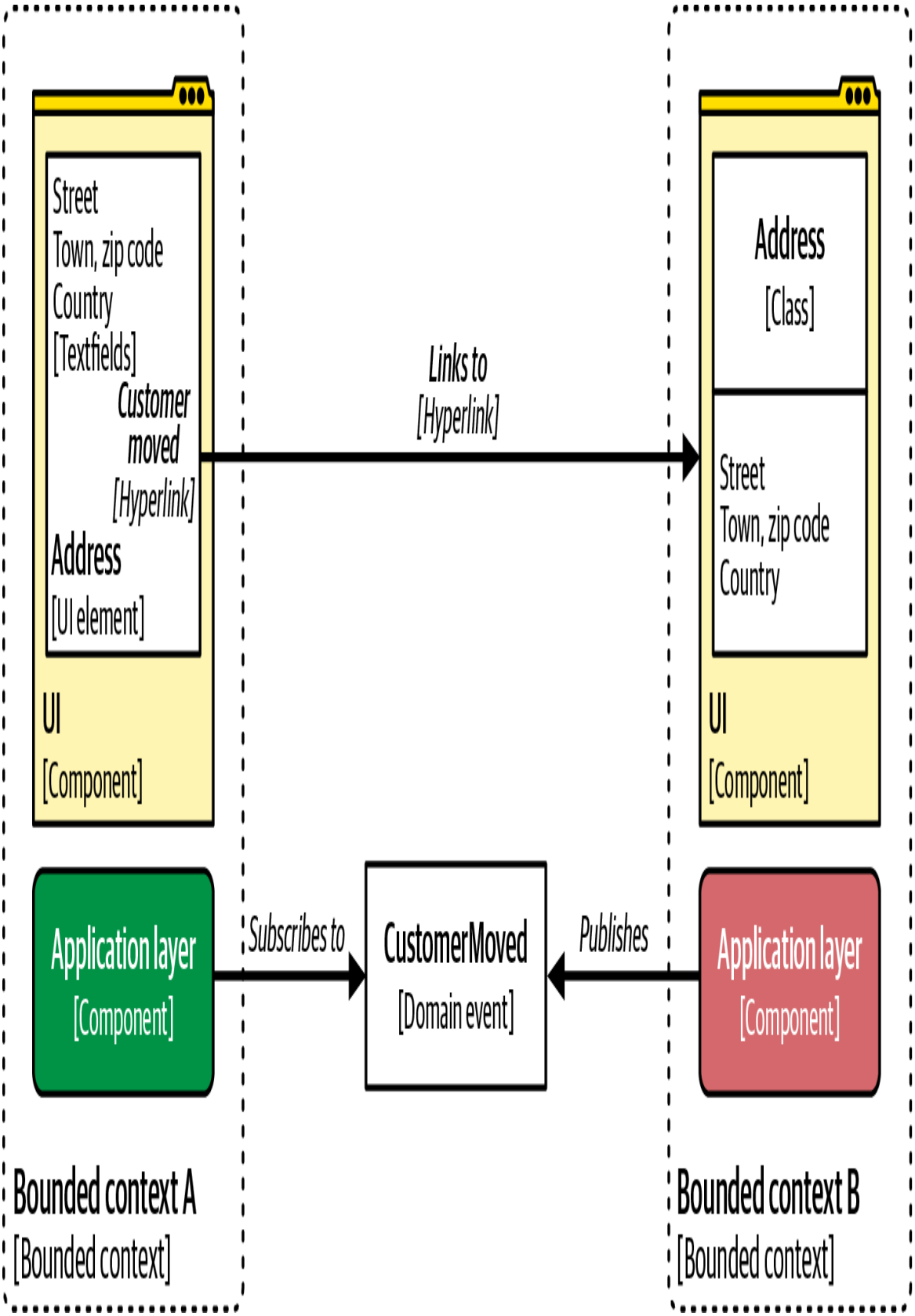


Figure 12-23. UI with integration and events.

We have observed on several occasions that a lack of understanding of the relationship between the bounded contexts and the user interface makes it difficult, especially for domain experts, to correctly assess the impact of their decisions and to successfully engage in strategic design.

This difficulty exists especially for the domain experts because they know the software exclusively through the user interface. As a prerequisite for collaboration, we have to explain to them how the UI and bounded contexts are interrelated.

For this reason, we strongly recommend clarifying this question before the strategic design begins, i.e., before step 2, and explaining the specifications in this regard to the contributors to the strategic design at an early stage. Then the concrete consequences of domain-driven decomposition and context mapping will be clearer to all participants, and the strategic design will deliver a more valid result.

Substep 3.3: Compiling Refactorings and Findings

As a final substep, we compile the refactorings and findings that we developed in the architecture review workshops with the architecture analyst and the development team.

Normally, we first create these lists simply with sticky notes on a whiteboard, on the Miro board, or in an Excel spreadsheet without any special tool support. Later, they are transferred to the tool that the respective team also uses to manage its backlogs, such as Jira.

In our workshops we distinguish between:

Findings

> Defects that we see, but for which we cannot think through an implementation in the workshop.

Refactorings

Where we have thought through an implementation in individual steps. Here we distinguish between:

- A superordinate *strategic refactoring*, e.g., “the extraction of a bounded context”

- The associated *tactical refactorings*, with which the strategic refactoring is implemented step-by-step

For the sake of clarity, we combine the tactical refactorings with the superordinate strategic refactoring by numbering them. The strategic refactoring gets the number X.0, and the respective tactical refactorings get the numbers X.1, X.2 etc. In this way, the goal of the strategic refactoring remains visible for each tactical refactoring and can be better kept in view by the development team.

In the end, the boundaries between findings and strategic refactorings are fluid; thus, it is a question of the company’s strategy if and when a finding should be solved by one or more strategic refactorings.

Artificial intelligence can also help find refactorings. We hear from the community that the results are quite promising when you ask an AI tool to identify refactorings that could bring the source code from the actual to the target architecture. Of course, you have to make the descriptions of the actual and target architectures readable for the AI tool.

AI can also help structure and break down strategic refactorings into tactical sub-refactorings. It is possible to teach the AI how small a refactoring must be so that the team can implement it in a sprint and then ask for a corresponding decomposition.

Once you have reached this point, however, you can also ask the AI about the implementation of the refactorings, and the team only needs to review the proposed refactorings in the sprint. Currently (in 2025), it is still unclear whether the proposed refactorings and their review will actually make development teams faster. Or whether understanding, thinking through,

reviewing, and, if necessary, rejecting the refactorings developed by AI will not lead to more work after all.

Let’s take a look at the car leasing example.

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: COMPILING THE FINDING LIST AND REFACTORING LIST

During the workshop, the development team and the architecture analyst discussed that a move from the outdated servlet technology to a more modern web frontend could provide a significant security benefit for the company's customers. However, this conversion would be so extensive that this topic only is recorded as a finding on the designated list for the time being.

Another point in the workshop, which is noted as a finding, is whether the part of Monolease responsible for monitoring customers' monthly lease payments could not easily be replaced by a purchased solution. Finally, the bounded context "payment" is a generic subdomain. Again, this issue cannot be addressed immediately, but needs to be discussed in appropriate strategic meetings with management (see Table 12-1).

Table 12-1. Findings modernizing Monolease

No.	Finding
A	Replace servlet technology with modern web frontend
B	Replace bounded context "payment" with SaaS solution

The development team and architecture analyst decide that to improve the architecture of Monolease, it is critical to separate the "offering" bounded context from the monolith. The high-level strategic refactoring and several supporting tactical refactorings are written up together accordingly (see Table 12-2).

Table 12-2. Refactorings to carve the bounded context "offering" out of Monolease

No.	Refactoring	In the catalog
1.0	Extract bounded context "offering"	*Extract Bounded Context*
1.1	Separate anemic domain class `Contract` from the monolith and decompose it	*Extract Specialized Anemic Entity*
1.2	Separate table `CONTRACTS` from the monolithic database	*Extract Specialized Table*
1.3	Separate service `ContractService` from the monolith and decompose it	*Extract Specialized Service*

Similarly, the development team and the architecture analyst are preparing the other refactorings and now have a good first draft of the unbundling plan.

Summary

In this chapter, we analyzed the actual architecture in the real source code and compared it with the target context map. A whole set of challenges and meaningful refactorings and future findings to be analyzed more deeply have been found by the architecture analyst and the development team.

Finally, we have listed strategic and tactical refactorings together in the refactoring list and are now ready for the next step. Based on this, we can start the actual refactoring of the legacy system in the next chapter (step 4).

To decompose in the solution space, a number of authors have written books and articles that have inspired and helped us along the way. Perhaps these will be of interest to you as well: *Refactoring* [Fowler 2019], *Working Effectively with Legacy Code* [Feathers 2004], *Monolith to Microservices* [Newman 2020], and "Strangler Fig Application" [Fowler 2004b].

Chapter 13. Step 4—Prioritization and Implementation of the Transformation Measures

This chapter concludes Part III and covers planning and conducting the transformation measures from the previous chapter. First, refactorings are broken down into manageable steps, then they are applied to the system under transformation.

From the sheer length of this chapter compared to the preceding chapters, it can be seen that step 4 takes more effort than steps 1 and 2, and even more effort than step 3.

After the idealized target context map was created in step 2, and the lists of findings and refactorings were developed in step 3, step 4 now deals with the prioritization and implementation of the refactorings.

In contrast to the refactorings, the findings cannot be implemented directly. At the end of this chapter, we will see how to deal with this challenge.

With the refactorings, the development team can finally get their hands on the code and implement their improvement ideas. But wait! Before that, there needs to be discussion with the development team, product owner, and architecture analyst. They need to choose the bounded context that should be carved out first.

In Figure 13-1 you can see the substeps that are now due. Product owner, architecture analyst, domain modeler, and developers choose the strategic refactoring from the refactoring list to extract the first bounded context (substep 4.1). Once this decision is made and possibly the next candidates are determined, the developers add the tactical refactorings that belong to

the strategic refactoring to the product backlog (or improvement backlog) together with the product owner (substep 4.2). At the beginning of a sprint, the developers and the product owner enter a manageable amount of refactorings into the sprint backlog and schedule them (substep 4.3). Finally, the first bounded context is extracted (substep 4.4).

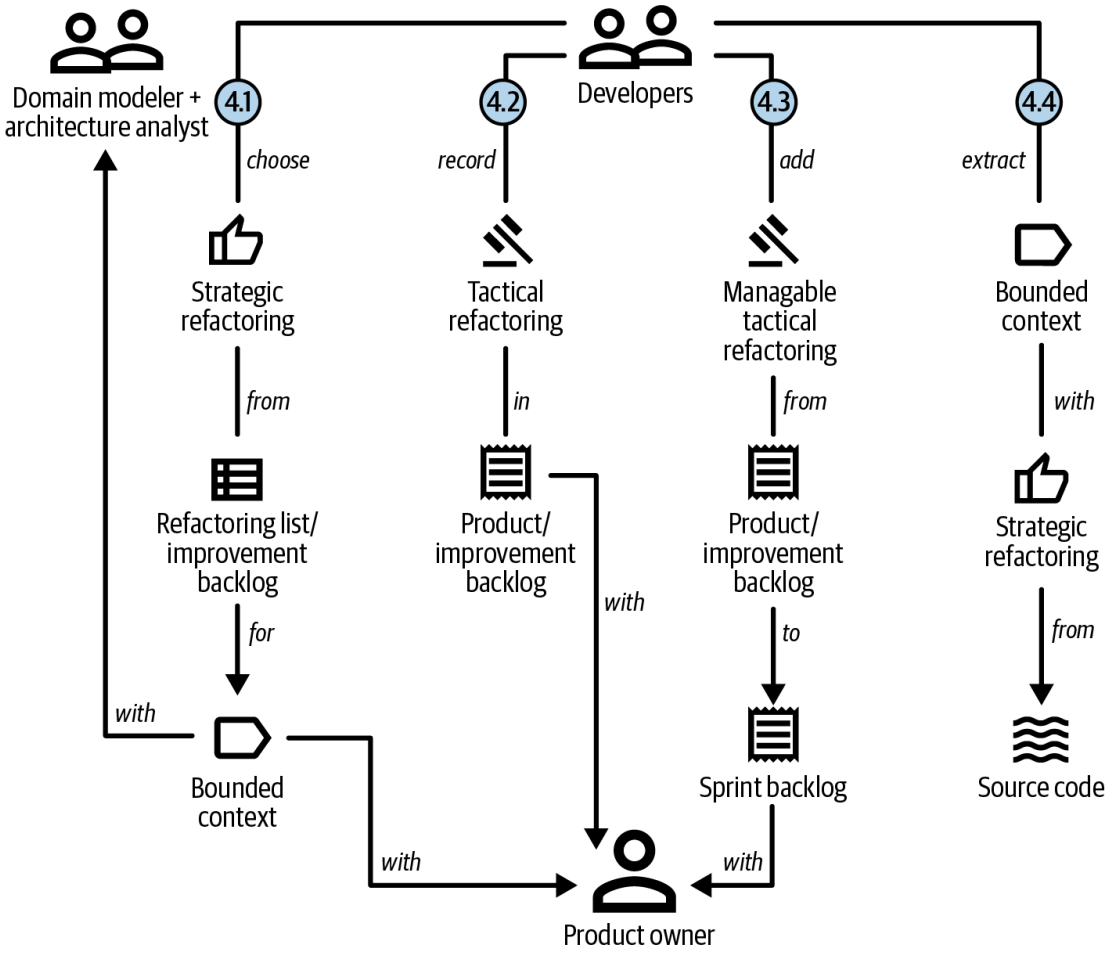


Figure 13-1. Strategic transformation—step 4 in its substeps.

Schematic Description of Step 4

Step 4 includes the following elements:

Actors

- Architecture analyst

- Domain modeler
- Development team
- Product owner

Substeps

1. Selecting the first strategic refactoring (substep 4.1)
2. Adding tactical refactorings to the product backlog, or improvement backlog (substep 4.2)
3. Planning achievable refactorings into the sprint backlog (substep 4.3)
4. Extracting context (substep 4.4)

Tools

- IDE with refactoring support
- Project planning tool, e.g., Jira

Results

- Decision on what to start with
- Architecture blueprint for the transformation
- Prioritized refactoring list

Goals/reason

- Minimize risk of the reconstruction by selecting a *supporting* or *generic subdomain* to start with
- Gain experience with the reconstruction

Challenges

- Disentangling business logic in services
- Moving local data, setters, and getters into a context
- Duplicating data, setters, and getters into multiple contexts
- Introducing domain events between services
- Moving domain logic out of services and into the anemic domain model
- Selecting findings at the appropriate moment
- Keeping the stamina

Just as in the last three chapters, you will find alternating challenges and substeps at the same level of the chapter structure. In this chapter, there are four substeps (4.1 to 4.4) and seven challenges. In each case, we have placed the challenges before, between, or after the substeps where they tie into the content.

Substep 4.1: Selecting the First Strategic Refactoring

In the previous step, decisions were made on what to do with each bounded context (strategic refactorings), and appropriate tactical refactorings were outlined. In addition, the team has classified the bounded contexts as *core*, *supporting*, and *generic* using the means of DDD.

Now the question is which bounded context (or contexts) to start with. The spontaneous answer to this question is often, "It's obvious! We have to start with the most important one, a core!" We regularly advise teams that have no experience with monolith transformation against this first step and recommend a different start.

Pattern: Start with a Supporting Subdomain

Because every project and every monolith is different, it is especially important at the beginning of a strategic transformation to find out a lot about the challenges ahead. Extracting the first bounded context is a great opportunity to learn what pitfalls await the development team. The first time, you will make a lot of mistakes, from which, as we all know, you can learn particularly well and a lot.

From the quality point of view, the first extraction will certainly not be a masterpiece, but only the second or third time will the team be able to deliver a high-quality result right away.

That's why we recommend starting with a not-so-important part of the domain (and thus of the system): a supporting or generic subdomain. This way, the nooks and crannies that arise during the first extraction do not pollute the most important parts of the system in the long run: the subdomains to which we have given the label *core*.

From the set of supporting and generic subdomains in question, it is also advisable to select a bounded context that is relatively independent of the rest of the system, i.e., has little coupling. The key question here is: "From what can the team learn the most quickly?" If you make a good choice here, the development team will quickly succeed and be able to harvest a *low-hanging fruit* (see [Manns and Rising 2015]).

Pattern: Proceed with a Core Domain

Once the first subdomain has been carved out of the big ball of mud, the situation should be reevaluated. Has the team learned enough to tackle a core subdomain? Or is it better to give it a second try with another supporting subdomain? Maybe pick a bounded context that is a bit more entangled with the rest than the first carved-out bounded context?

No matter what the decision, our goal is for the team to evolve from a learning state to achieving the greatest possible benefit for the organization. Since we're dealing with technical debt, we want to identify code with a high interest rate (see [Tornhill 2018, chap. 2]). The greatest benefit comes from (and therefore should be the stated goal of all preparations) cutting out the core parts from the monolith.

If the team manages to carve out the most important parts piece by piece, a real improvement for maintenance and enhancement will become noticeable and visible. Often, our experience is that "the rest" is not so terrible once the most important parts have been disentangled.

Substep 4.2: Adding Tactical Refactorings to the Product Backlog (or Improvement Backlog)

Decomposing a monolith is a major operation (see Chapter 6), i.e., a *large refactoring*. With large refactorings the reconstruction work takes so long that most development organizations cannot afford to do without any further development during this time (see [Roock and Lippert 2006]).

Therefore, it is important to divide even the small tactical refactorings into still smaller refactorings. It is crucial that the system under refactoring is in an operational state after each small refactoring. From Chapter 7 we know that a refactoring may change in principle only the structure, but not the functionality of the system—otherwise it would not be refactoring, but a new development or a bug fix (see [Fowler 2019]). Michael Feathers

impressively describes how to also implement refactorings in legacy systems behavior-neutrally in his book *Working Effectively with Legacy Code* (see [Feathers 2004]).

The larger the system is, the more careful we must be with the decomposition and the smaller the steps must be in which we proceed. In Extreme Programming (XP) this technique is called *baby steps* (see [Beck and Andres 2005]).

If we compare the division into strategic and tactical refactorings in terms of granularity with the usual planning elements in the Agile process, then the strategic refactoring corresponds to an epic, and the tactical refactorings are at the level of user stories. Just like user stories, tactical refactorings can be broken down into tasks to achieve even finer-granular work items.

The tactical refactorings for the first strategic refactoring are entered into the product backlog by the development team and are now ready for the next sprint planning. Some teams also decide to keep the refactorings and findings in a separate backlog—the *improvement backlog* (see also [Starke 2022b]).

IN THE CAR LEASING EXAMPLE: PRIORITIZING REFACTORINGS

Our team Chewbacca at Alphorn Auto Leasing, Inc. (see Figure 11-6), which was formed in Chapter 11, has already gained experience with the first strategic refactoring on the supporting subdomain "object management" and now has the confidence to tackle the core domain "offering."

The team and its product owner decide to spend 20% of their effort per sprint on disentanglement for the upcoming period. So the respective sprint backlogs will contain the following composition of backlog items:

- User stories brought in by the users (80%)
- Tactical refactorings (20%)

After three sprints, the development team, together with the product owner, want to review in retrospect whether this share is the right one and if the progress is fast enough.

The refactorings from Table 12-2 are put into a meaningful (i.e., building on each other) order by team Chewbacca and are first inserted into the improvement backlog.

Substep 4.3: Planning Achievable Refactorings into the Sprint Backlog

In the sprint planning, the product owner and the team discuss which user stories and tactical refactorings will be transferred to the sprint backlog. There are basically two approaches to this:

One-sided sprints

A sprint is either completely dedicated to further development; i.e., only user stories are scheduled into the sprint. Or a sprint is completely dedicated to transformation; i.e., only tactical refactorings are included in the sprint.

Mixed sprints

In each sprint, a certain amount of effort is reserved for transformation, and the rest is used for further development. For example, a team with a velocity of 10 story points might use 3 of them for refactoring.

At the beginning of a transformation, we favor the second variant, the mixed sprints, because we want to avoid users not getting any new requirements implemented for a whole sprint, but we still have to test the system and approve (unnoticeable) changes.

Especially at the beginning of a transformation project, such confidence building measures are absolutely recommendable. If the team has succeeded in gaining the trust of the users and the product owner through a lengthy, high-quality refactoring, you can think about dedicating an entire sprint to the refactoring work.

Involving the entire team in the transformation work usually leads to significant productivity gains, because everyone can focus on the transformation together.

Let the Refactoring Pause Only in Emergencies

If you have decided to reserve a fixed part of the velocity, then you should make sure that you do not deviate too often from this allocation and use the refactoring time for other purposes.

In our experience, only in real emergencies should you use the workforce reserved for refactoring for further development or just for emergency measures. The rule of thumb here is that real emergencies are very rare (≤ 1 time per year) and do not last longer than one sprint. Anything that occurs

more frequently or lasts longer should not be an emergency that needs to be resolved by an emergency action.

If the development teams working on the monolith are constantly dealing with emergencies, then there is a systemic problem: either stabilizing actions from Chapter 7 need to be taken or the organization needs a change as described in Chapter 9. In such a case, the rebuilding of the monolith should be put on hold and the systemic problems should be addressed first.

Estimating Refactorings

In order to be able to schedule tactical refactorings in the sprint planning, they must first be estimated.

Much like user stories, estimating strategic refactorings is a matter of practice. In the beginning, the team will miss the mark when it comes to estimating. Over time, a learning effect will become apparent and the estimates will improve.

Nevertheless, estimates remain a fundamental problem in software development, because many difficulties are only found in the implementation.

A complete analysis for a precise estimate is only possible if you have already done the implementation—so it is a nonsensical endeavor. Aim42 (see [Starke 2022b]) offers a pragmatic approach for estimating all kinds of improvement activities, called “improvements,” which also include refactorings.

Domain-Driven Transformation with Kanban

Teams that use Kanban instead of Scrum should reserve part of the WIP limit for refactorings. The refactoring is best managed in a separate swim lane on the Kanban board, so that you can separate the refactorings from the further development.

If you work with Kanban, then you are in the pleasant situation that you can do without estimating, because in Kanban where there are no sprints; it is

“only” about limiting the tasks to be processed simultaneously by means of the WIP limit.

If you have the option to work with Kanban, then you can significantly reduce the cognitive load (see Chapter 9, “Team Topologies”), for your teams by eliminating estimating and managing sprints.

Now let’s look at how the team handles refactorings in our car leasing example.

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: PLAN REFACTORINGS INTO THE SPRINT BACKLOG

From the carving out of the supporting subdomain "object management," the development team Chewbacca has learned painfully that it is better to start with carving out the services (refactoring 1.3 from Table 12-2) before changing the domain model (refactoring 1.1) or the database (refactoring 1.2).

The product owner, in consultation with the team, therefore schedules number 1.3 (separate the service `ContractService` from the monolith and deconstruct it) from Table 12-2 into the next sprint as the first tactical refactoring. With this, the team starts a first part of the carving out, which is laid out in strategic refactoring 1.0. The further tactical refactorings that are part of this strategic refactoring will be implemented in the next sprint or the sprint after that.

The initial situation for the development team before refactoring 1.3 is shown in Figure 13-2. From further up, the method `monolease.ContractService.offer()` is called, and this in turn calls the getters and setters of `carPrice` and `installment` in the `Contract` class.

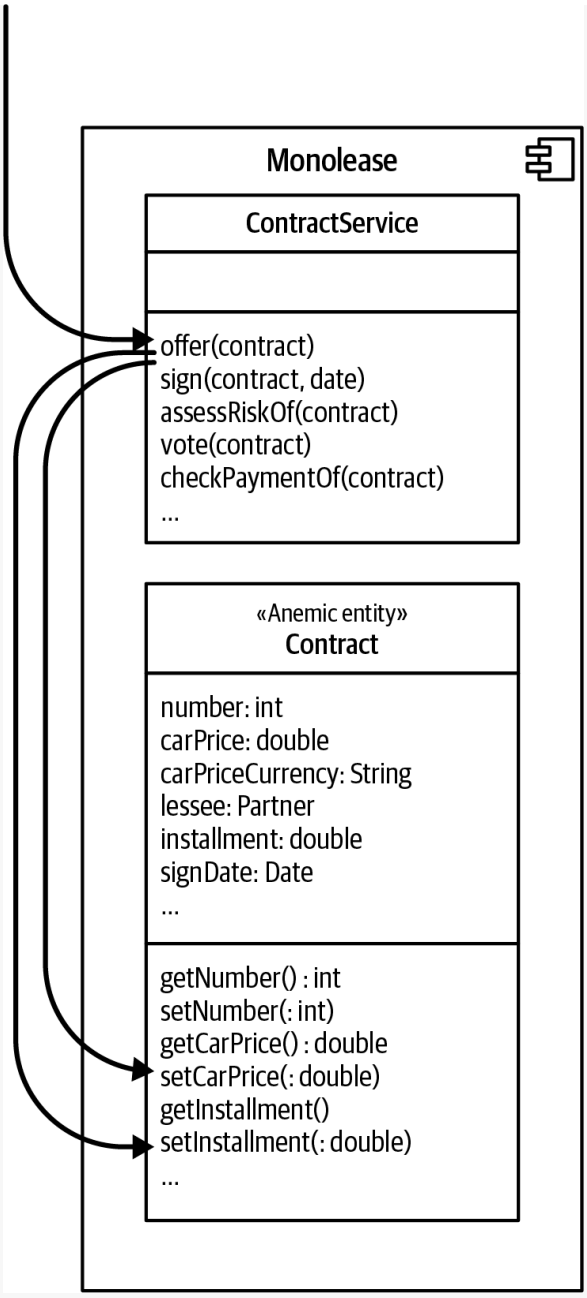


Figure 13-2. Extracting the context "offering"—initial situation.

Substep 4.4: Extracting Context

When starting the implementation, the following basic wisdoms related to refactoring and Agile software development should be kept in mind:

- Every refactoring should be prepared by *unit tests* of the existing functionality and secured in such a way that the old behavior remains when the change is made.
- After every change, the *tests must be run* and every *change, no matter how small, must be integrated* so that a runnable (i.e., production ready) state is available at all times.
- For *quality assurance*, the refactorings are implemented in pair programming or in mob or ensemble programming. If this option does not exist, at least a code review is performed to ensure that the four-eyes principle is maintained.

Let's take a look at the typical challenges that arise when performing refactorings.

Challenge: Disentangling Business Logic in Services

In order to preserve the functionality of the legacy system, we have to be appropriately careful when disentangling the logic. At the same time, it is clear that in an anemic domain model, the domain logic is located in the methods of the service classes and must be extracted from there.

We want to divide up these methods in such a way that only the part of the domain logic is moved into the respective bounded context that is also correct from a domain perspective. So, for each method, we have to ask ourselves in which bounded context it belongs or whether it has to be decomposed into individual parts.

How does this look in our car leasing example?

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: EXTRACTING A SERVICE

Our team Chewbacca starts with the just-planned sprint and gets to work. One pair grabs the backlog item for refactoring 1.3 and starts implementing. The two developers first put a new package `offering` next to the old package `monolease`, which represents the new context "offering" (see Figure 13-3).[1]

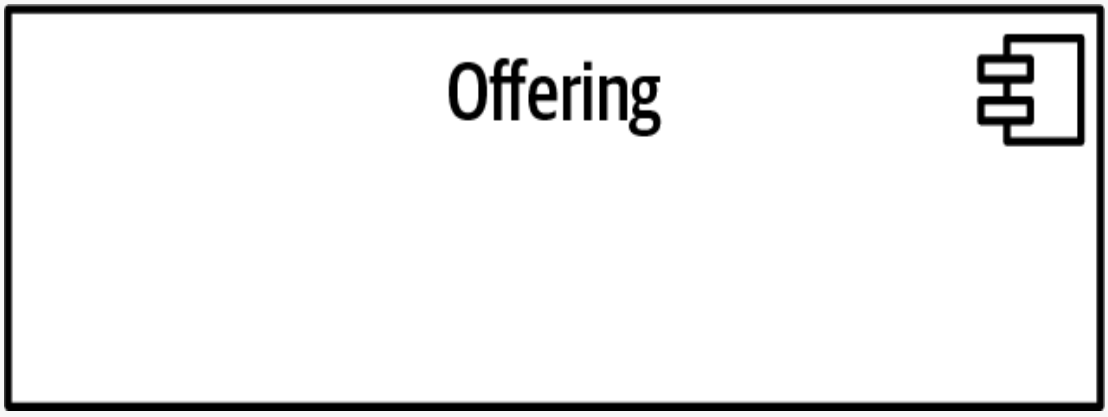


Figure 13-3. Extracting the context "offering"—new package for new context.

The goal of this refactoring is a modular monolith and not a microservices architecture (see Chapter 5)—so a new package is sufficient for the time being. Later, when the rebuild work has reached a good stage, the teams will discuss whether it makes sense and is necessary to move to a microservices architecture.

Next, the pair performs the steps of refactoring *Extract Specialized Service*:

After creating a new test class `offering.ContractServiceTest`, a new class `offering.ContractService` is created (see Figure 13-4 and Listing 13-1).

Listing 13-1. Extracting the context "offering"—new class for extracted service

```
package offering;
```

```
class ContractService {
}
```

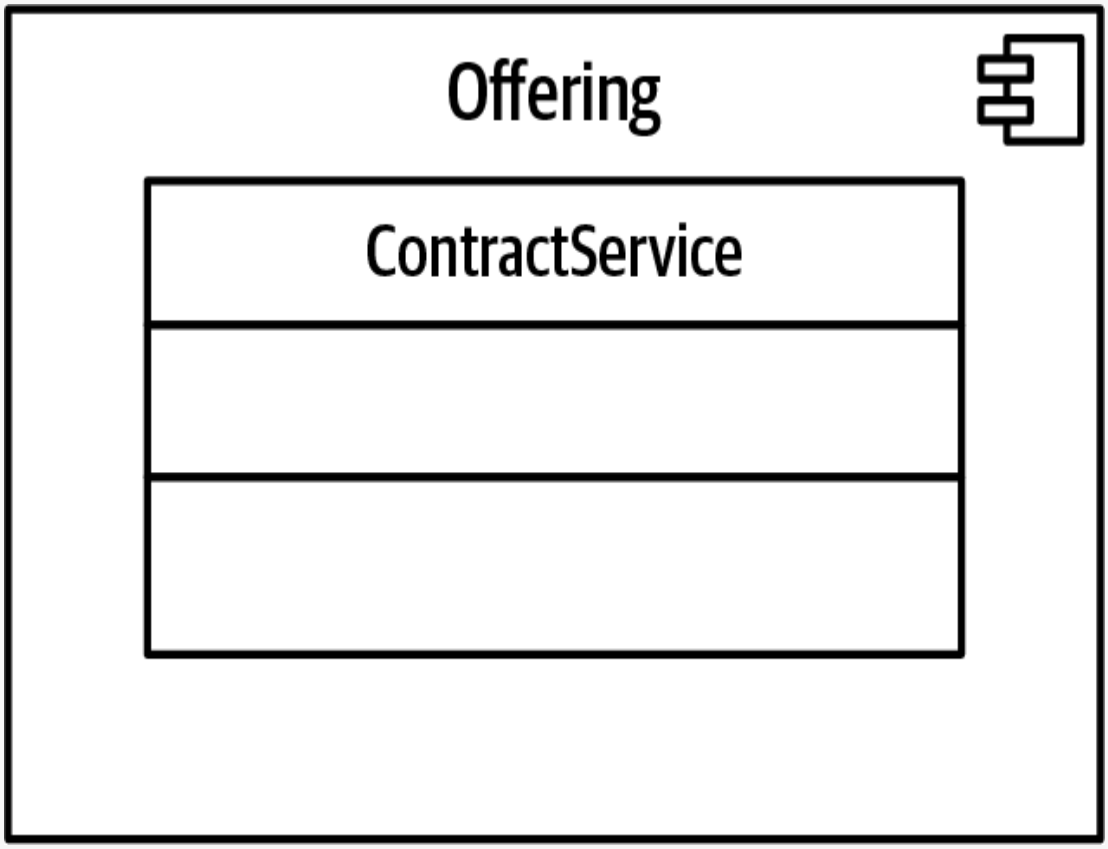


Figure 13-4. Extracting the context "offering"—new class for extracted service.

In the old class `monolease.ContractService`, a field `_TEMPORARY_offeringContractService` of the new type `offering.ContractService` is inserted. For such transition variables (or even methods) "ugly" names like in Listing 13-2 are often used to make it clear that they will be removed again later.

Listing 13-2. Extracting the context "offering"—creating a transition variable

```
    package monolease;

    class ContractService {
        // ...
        offering.ContractService
_TEMPORARY_offeringContractService;
        // ...
    }
```

The methods of the `ContractService` that are only needed in the context of "offering" can be moved using the standard refactoring *Move Method* [Fowler 2019, p. 198] built into most IDEs. In doing so, the methods of the old class are changed so that they forward to their counterparts in the new class and marked as deprecated (see `offer()` in Figure 13-5 and Listing 13-3).

Listing 13-3. Extracting the context "offering"—mark method to be moved as obsolete

```
package monolease;

class ContractService {
    // ...
    /**
     * @deprecated Use {@link
     *     offering.ContractService#offer()} instead.
     */
    @Deprecated
    public offer(Contract contract) {
        _TEMPORARY_offeringContractService.offer(contract);
    }
    // ...
}
```

Next, the pair goes step-by-step through the warnings that appear in the code due to the `@Deprecated` annotations. If we were correct in the original analysis, then only code that also belongs in the new bounded context should be detected here in the `offering` package.

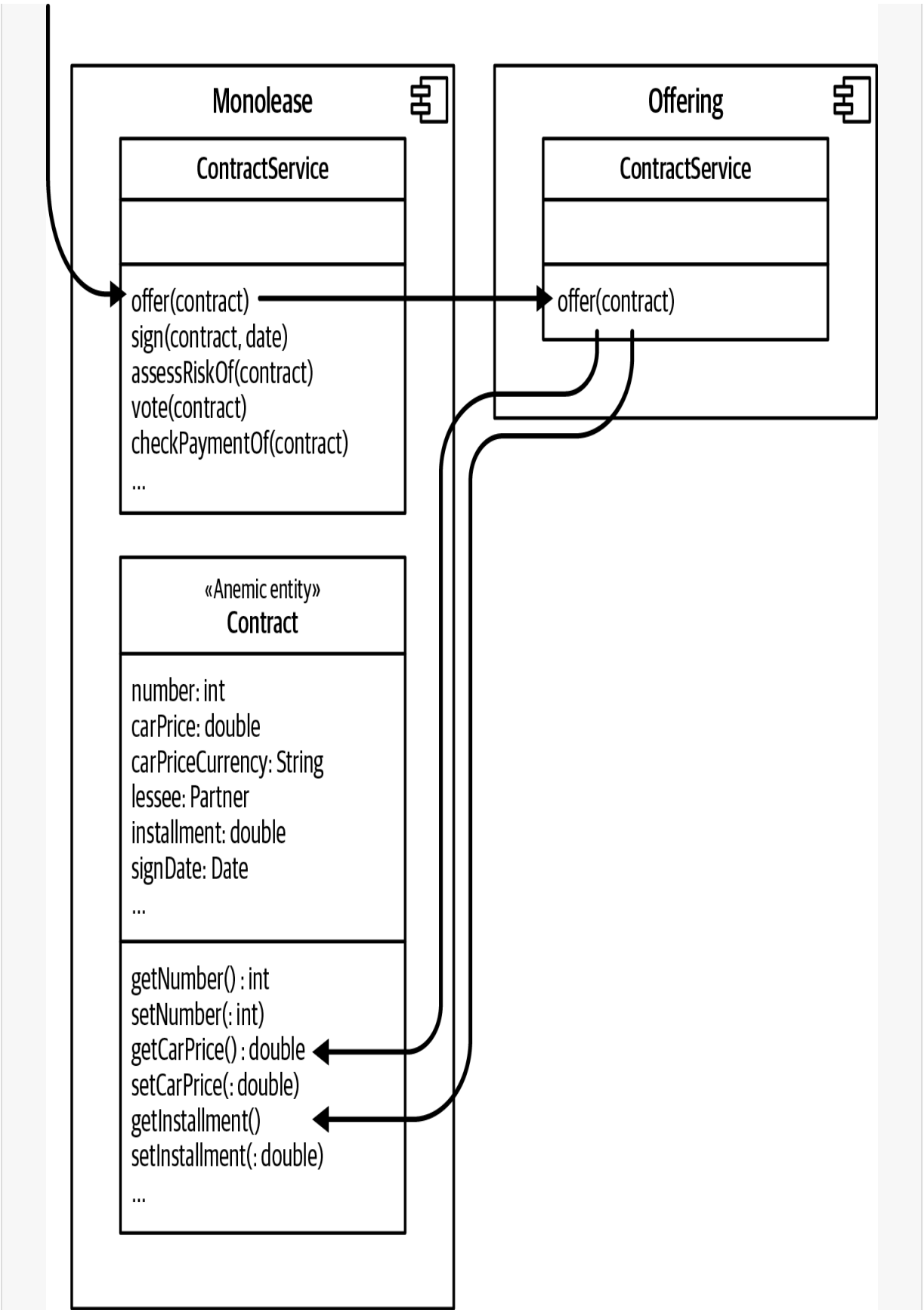

Monolease
ContractService
offer(contract)
sign(contract, date)
assessRiskOf(contract)
vote(contract)
checkPaymentOf(contract)
...
«Anemic entity»
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
...
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
getInstallment()
setInstallment(: double)
...
Offering
ContractService
offer(contract)

Figure 13-5. Extracting the context "offering"—Move Method `offer()`.

But for now we leave this code in the monolith and replace every call to `monolease.ContractService.offer()` with a call to `offering.ContractService.offer()` (see Figure 13-6). This requires that an object of the type `offering.ContractService` is also placed alongside the objects of type `monolease.ContractService`. If necessary, the objects of type `monolease.ContractService` can then be deleted.

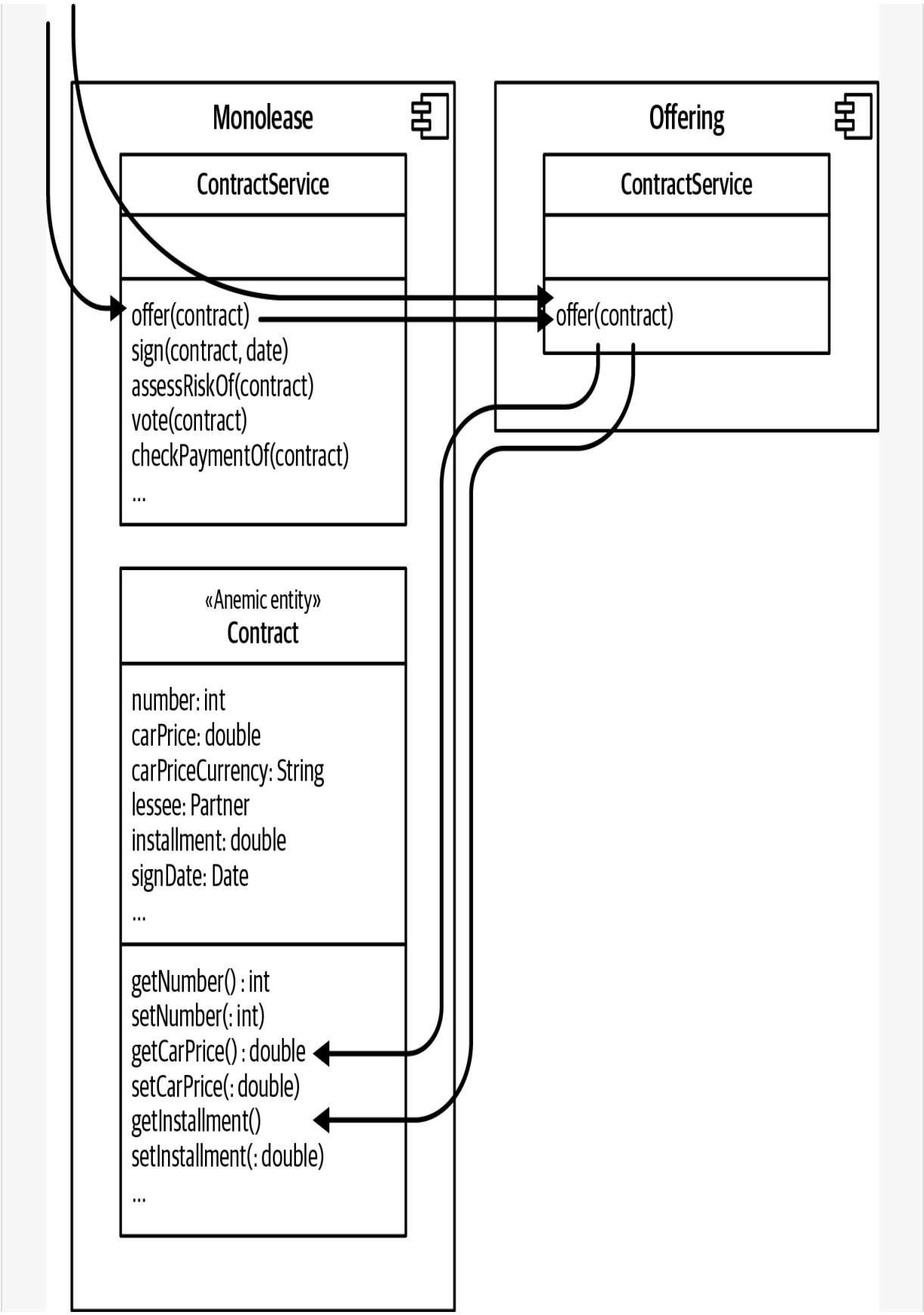

Monolease
ContractService
offer(contract)
sign(contract, date)
assessRiskOf(contract)
vote(contract)
checkPaymentOf(contract)
...
«Anemic entity»
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
...
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
getInstallment()
setInstallment(: double)
...
Offering
ContractService
offer(contract)

Figure 13-6. Extracting the context "offering"—step-by-step change calling code.

It can take quite a while until all calling places are changed and the old `offer()` method can safely be deleted (see Figure 13-7). If this takes longer than one sprint, then the individual steps should be planned as separate backlog items.

If you proceed in the same way as the pair in our example, then a viable and releasable state is present after each refactoring. The pair thus ensures that it always leaves the software in a state of evolvability between the steps.

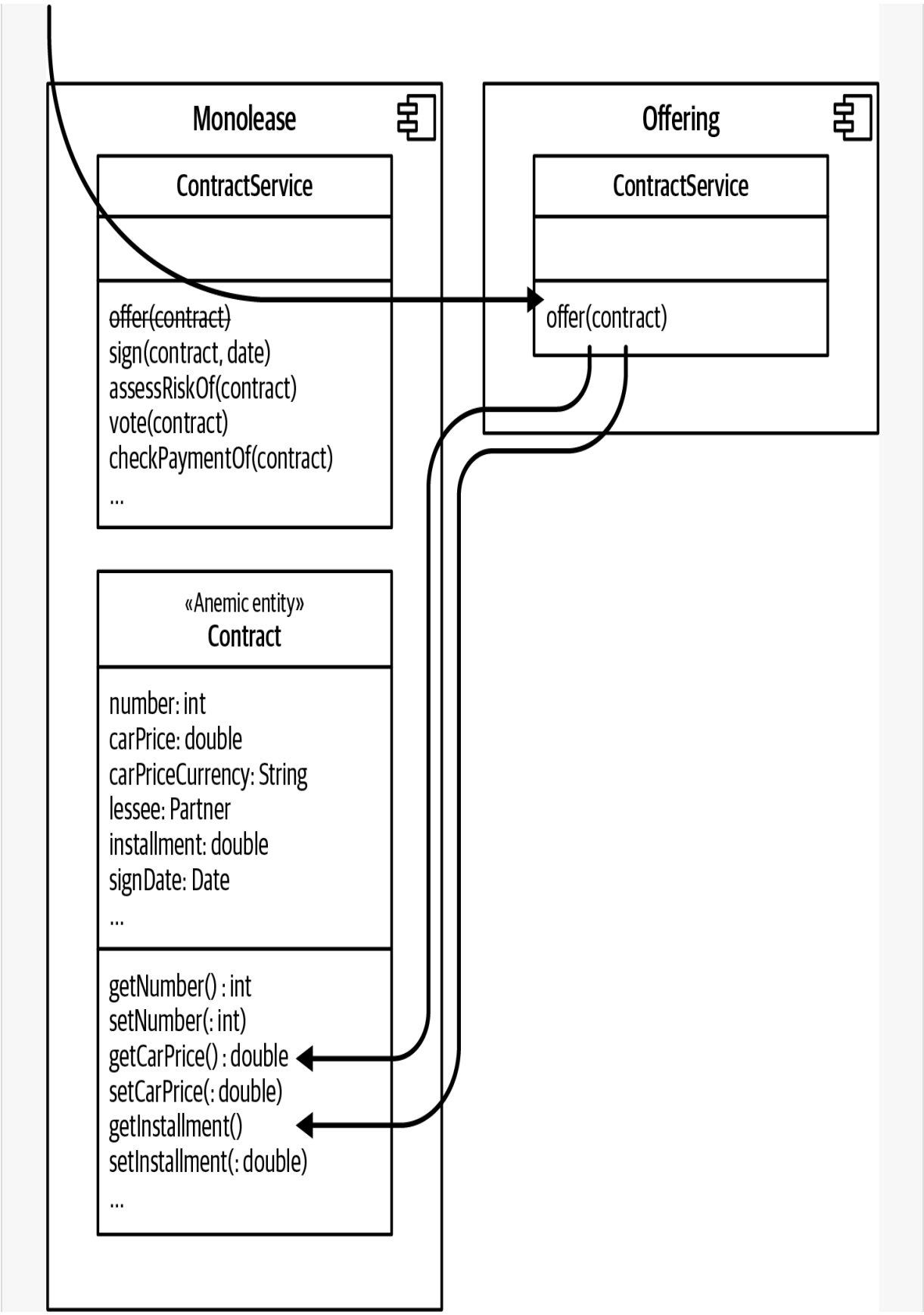

Monolease
ContractService
~~offer(contract)~~
sign(contract, date)
assessRiskOf(contract)
vote(contract)
checkPaymentOf(contract)
...
«Anemic entity»
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
...
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
getInstallment()
setInstallment(: double)
...
Offering
ContractService
offer(contract)

Figure 13-7. Extracting the context "offering"—no more calling code, old method can be deleted.

If the system is in the state of Figures 13-5, 13-6, or 13-7, and a requirement calls for a modification to the contract offering logic, that change can be made in `offering.ContractService.offer()`.

The larger the refactoring, the more important it is to proceed in small steps so that further development is possible in between (see substep 4.2).

Challenge: Moving Local Data, Setters, and Getters into a Context

Disentangling the business logic as the first step makes the design of the classes much clearer, so that it is now also easier to see which data fields, setters, and getters from the entities belong in which bounded context. Basically, we want to achieve high cohesion (see Chapter 5), so the data fields of the classes have to be moved to the new bounded context where their setters and getters are used.

Let's start with the smaller challenge and detach the parts of an entity with its data, setters, and getters from the monolith that are only used in *one* context. Then, in the subsequent section, we'll see how to handle the parts of an entity with its data, setters, and getters that are needed in *several* contexts.

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: DUPLICATING AN ENTITY

The service (or actually just a method of the service) is now extracted, but still uses the entity in the monolith. So the new bounded context "offering" is still far from being capable of running on its own.

For the following sprint, the product owner and the team therefore decide to next schedule the tactical refactoring 1.1 from Table 12-2 (separate anemic domain class `Contract` from the monolith and deconstruct it) in the sprint backlog. When carving out the domain class, a similar procedure is followed as in the refactoring for the service. However, not only methods—in this case getters and setters—are moved, but also data fields.

To do this, the pair performs the steps of the refactoring *Extract Specialized Anemic Entity*:

A new class `offering.Contract` is created (see Figure 13-8 and Listing 13-4).

Listing 13-4. Extracting the context "offering"—a new `Contract` class

```
package offering;

class Contract {
}
```

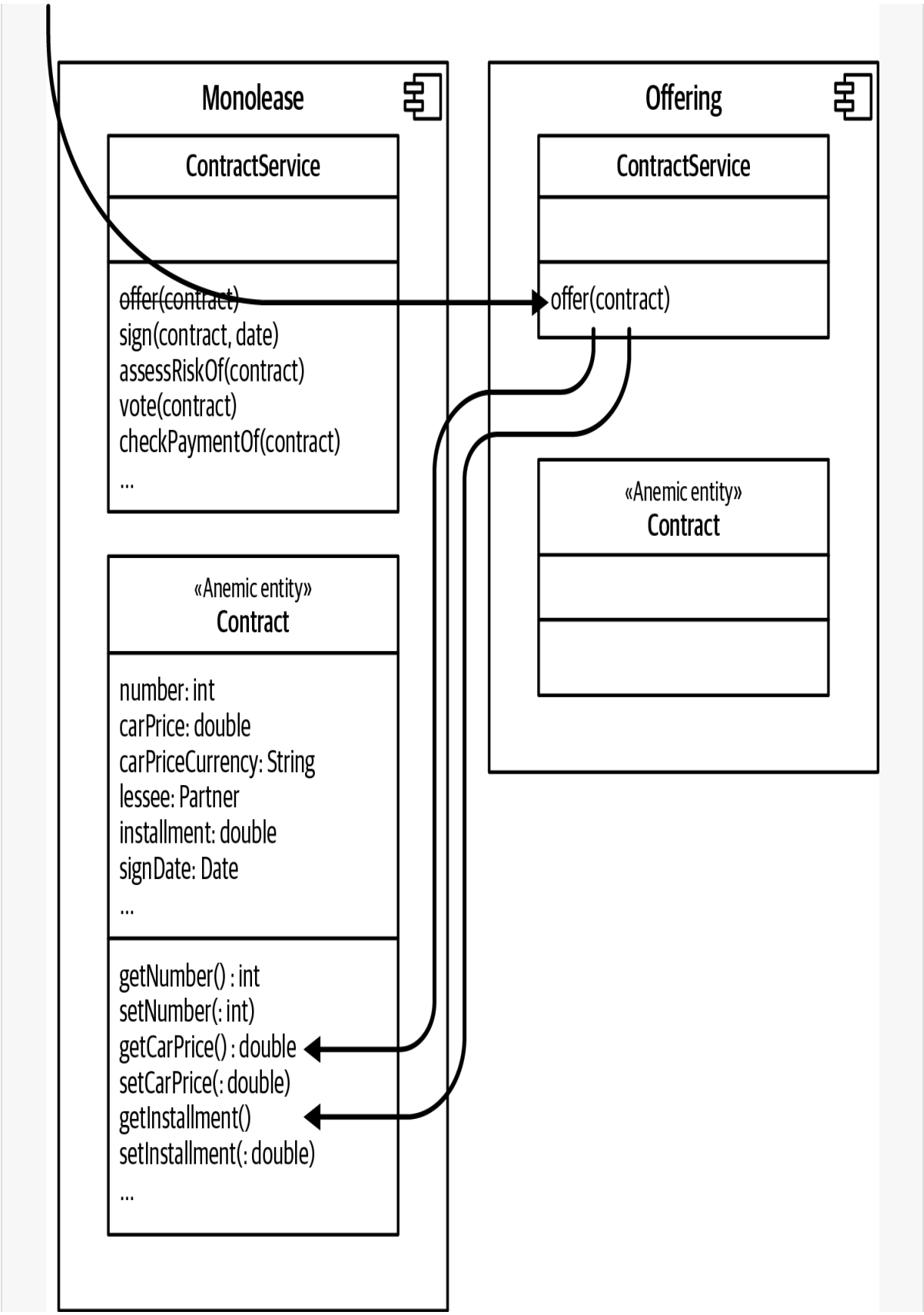

Monolease
ContractService
offer(contract)
sign(contract, date)
assessRiskOf(contract)
vote(contract)
checkPaymentOf(contract)
...
«Anemic entity»
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
...
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
getInstallment()
setInstallment(: double)
...
Offering
ContractService
offer(contract)
«Anemic entity»
Contract

Figure 13-8. Extracting the context "offering"—a new `Contract` class.

Into the old class `monolease.Contract` a field `_TEMPORARY_offeringContract` of type `offering.Contract` is inserted (see Listing 13-5).

Listing 13-5. Extracting the context "offering"—transition variable for `Contract`

```
package monolease;

class Contract {
    // ...
    offering.Contract _TEMPORARY_offeringContract;
    // ...
}
```

The data fields with setters and getters that are only needed in the logic of the `offering.Contract` class are moved from the `monolease.Contract` class in the monolith to the `offering.Contract` class in the new bounded context. The methods of the old class that access them (their getters and setters) are forwarded to the new class and marked as deprecated (see as an example `installment` in Figure 13-9 and Listing 13-6).

Listing 13-6. Extracting the context "offering"—mark method to be moved as obsolete

```
package monolease;

class Contract {
    // ...
    /**
     * @deprecated Use {@link
     *     offering.Contract#getInstallment()} instead.
     */
    @Deprecated
    public getInstallment() {
        return _TEMPORARY_offeringContract.getInstallment();
    }
    // ...
}
```

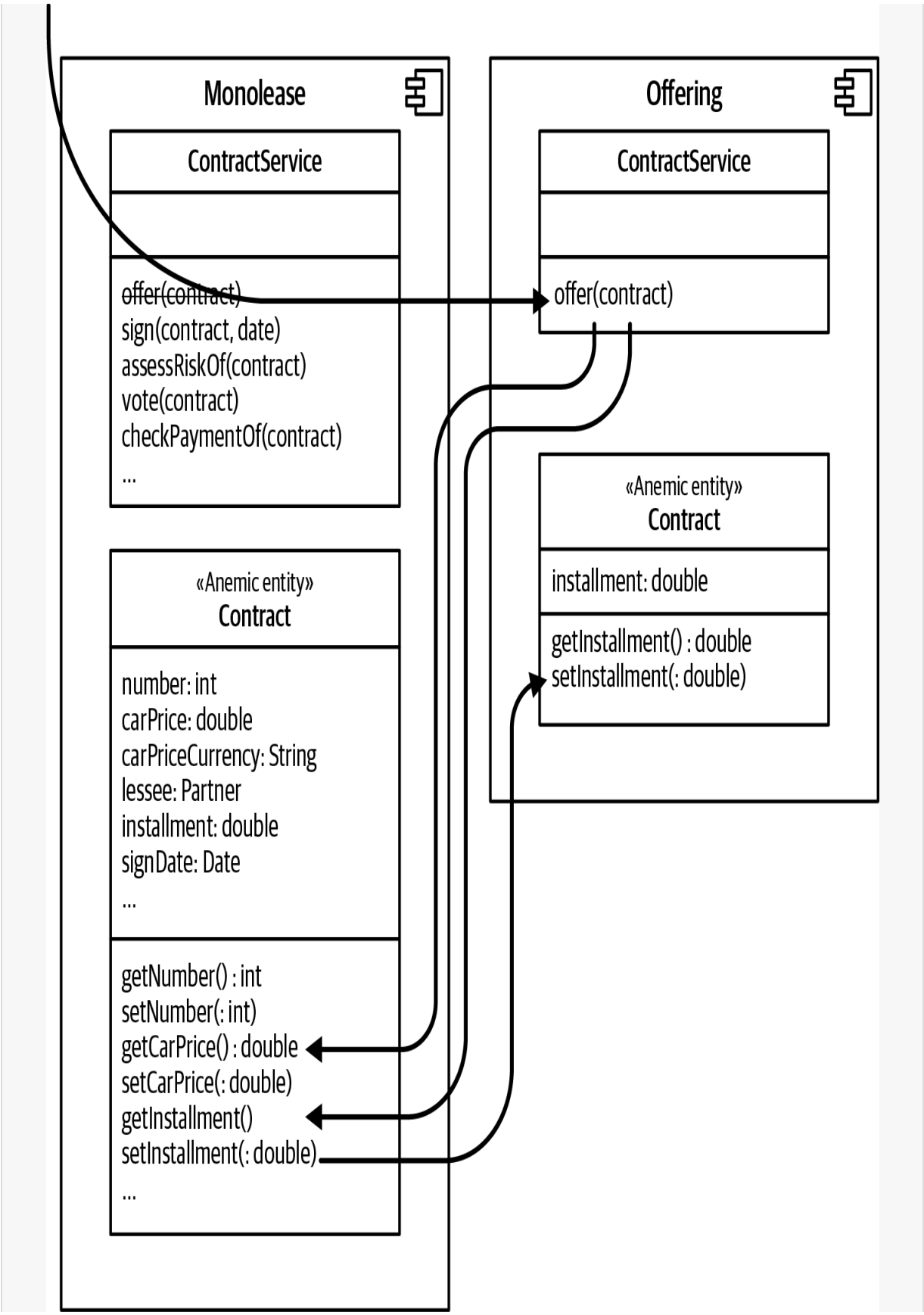

Monolease
ContractService
offer(contract)
sign(contract, date)
assessRiskOf(contract)
vote(contract)
checkPaymentOf(contract)
...
«Anemic entity»
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
...
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
getInstallment()
setInstallment(: double)
...
Offering
ContractService
offer(contract)
«Anemic entity»
Contract
installment: double
getInstallment() : double
setInstallment(: double)

Figure 13-9. Extracting the context "offering"—move data field as well as getter and setter.

Next, the calling code is changed and, as before, we go step-by-step through the warnings that appear in the code due to the `@Deprecated` annotations. Finally, we reach the state where the `offer()` method from `offering.ContractService` uses only the installment in the new context, and we can delete the `getInstallment()` and `setInstallment()` methods and the `installment` data field from the `Contract` class in the monolith (see Figure 13-10).

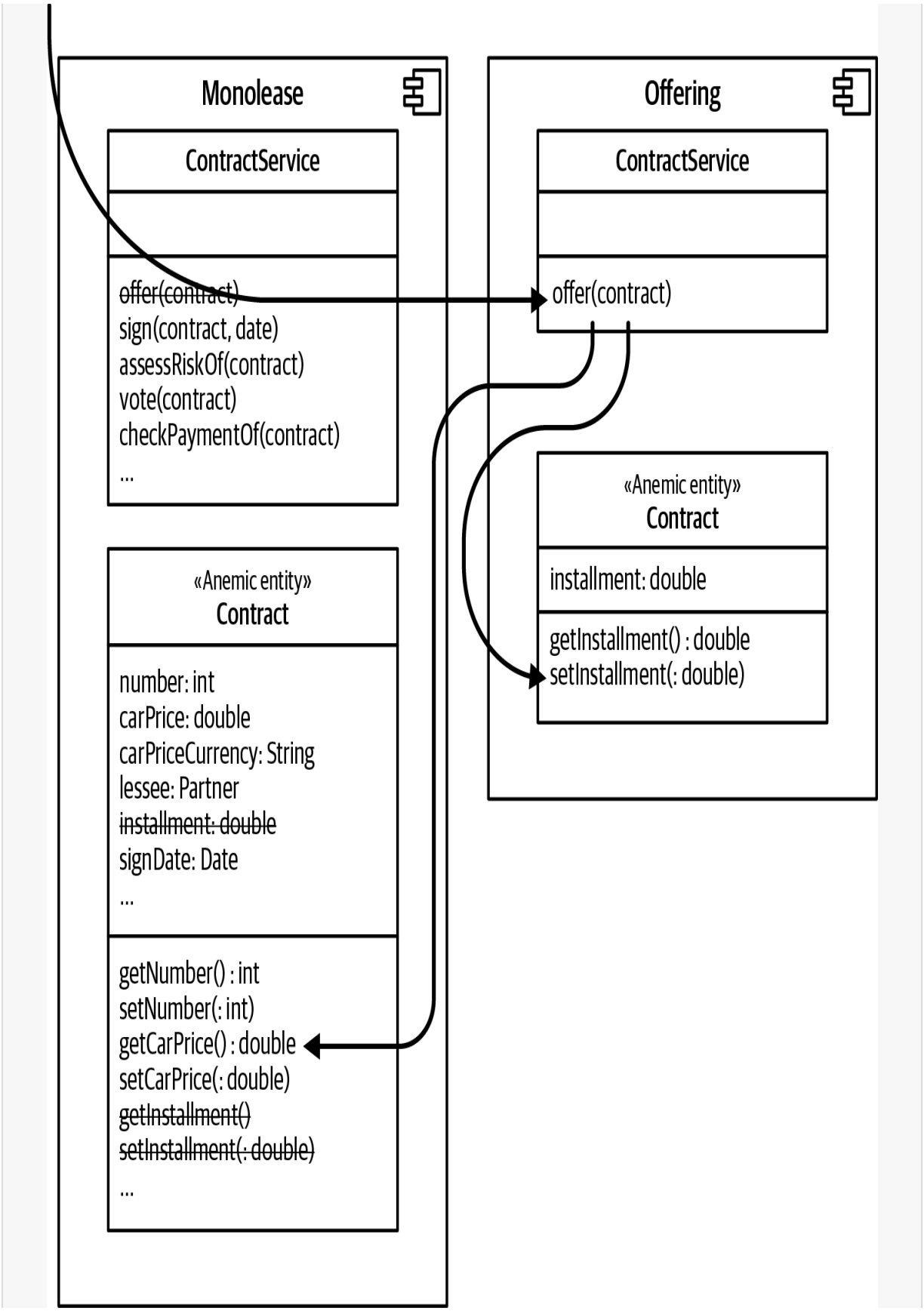

Monolease
ContractService
offer(contract)
sign(contract, date)
assessRiskOf(contract)
vote(contract)
checkPaymentOf(contract)
...
«Anemic entity»
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
...
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
getInstallment()
setInstallment(: double)
...
Offering
ContractService
offer(contract)
«Anemic entity»
Contract
installment: double
getInstallment() : double
setInstallment(: double)

Figure 13-10. Extracting the context "offering"—refactoring done.

Identity of Entities

In Chapter 8 we described the domain-oriented identity of entities as a means to make domain knowledge more explicit. At this point, the domain-oriented identity takes on another important role: it helps disentangle entities that are needed in two or more bounded contexts.

If there are multiple entities in different bounded contexts representing the same thing, then we need to ensure in the implementation and data structures that the identity, the meaning "this is the same thing," is explicit.

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: TRANSFERRING THE IDENTITY OF AN ENTITY

The identity of a leasing contract is defined by the contract number. So next we move into the class `offering.Contract` the field for the identity `number`. The constructor of the old class `monolease.Contract` is extended to initialize the field `_TEMPORARY_offeringContract` with a new object with the same contract number (see Figure 13-11 and Listing 13-7).

Listing 13-7. Extracting the context "offering"—identity introduced

```
package offering;

class Contract {
    private final int number;

    public Contract(int contractNumber) {
        number = contractNumber;
    }
}


package monolease;

class Contract {
    // ...
    public Contract() {
        _TEMPORARY_offeringContract =
            new offering.Contract(number);
    }
    // ...
}
```

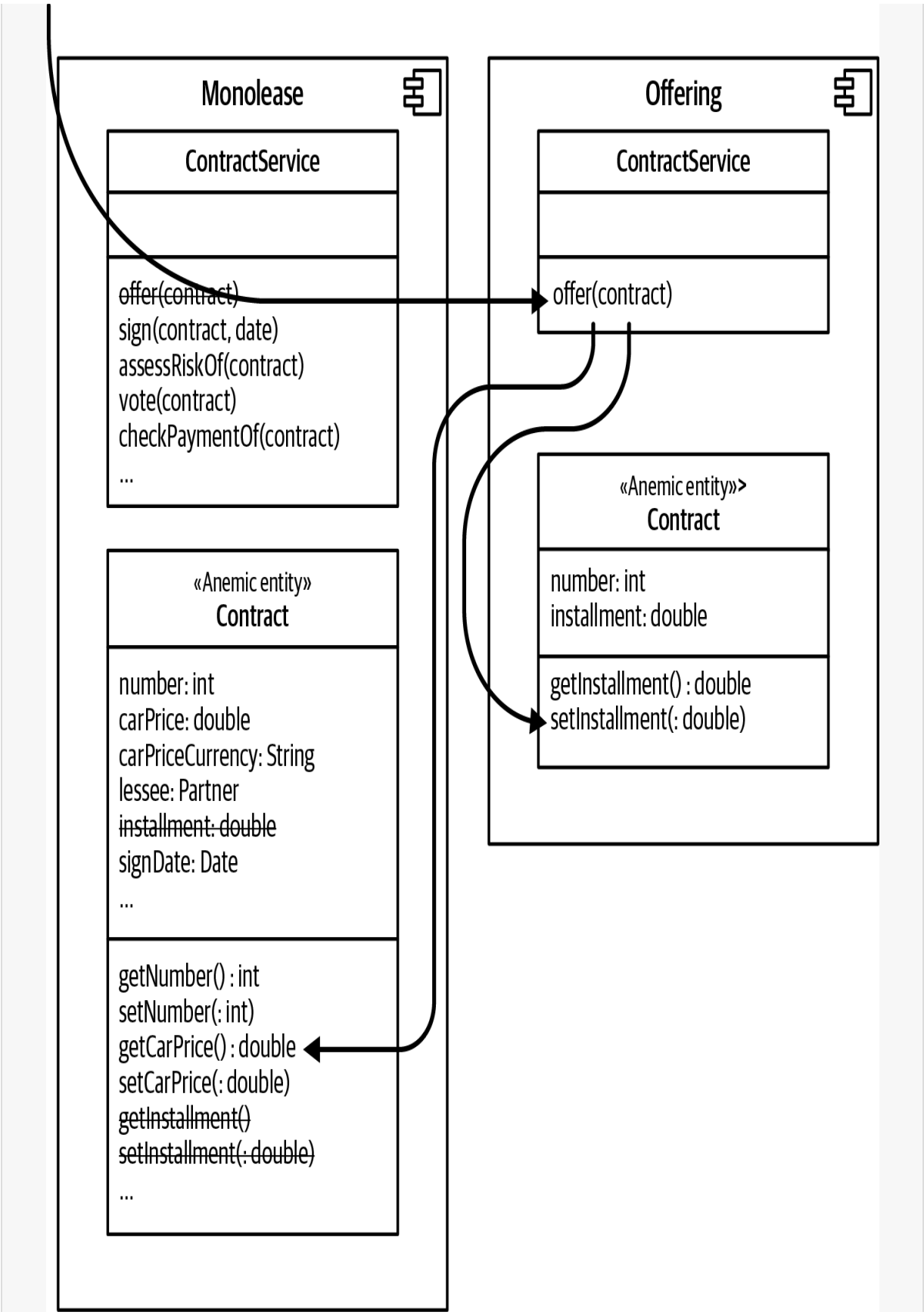

Monolease
ContractService
offer(contract)
sign(contract, date)
assessRiskOf(contract)
vote(contract)
checkPaymentOf(contract)
...
«Anemic entity»
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
...
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
getInstallment()
setInstallment(: double)
...
Offering
ContractService
offer(contract)
«Anemic entity»
Contract
number: int
installment: double
getInstallment() : double
setInstallment(: double)

Figure 13-11. Extracting the context "offering"—identity introduced.

Challenge: Duplicating Data, Setters, and Getters into Multiple Contexts

When decomposing, data and its getters and setters are often needed in different bounded contexts, and we therefore have to duplicate them. In order to keep the duplicated data fields in sync afterwards, we also need to introduce interfaces between the bounded contexts through which updates can be sent.

Let's take a look at the concrete implementation in the car leasing example.

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: INTRODUCE EVENT

After removing from the monolith the part of `Contract` that is only needed in the context `offering`, the pair turns to the more difficult task: the parts of `Contract` that are needed in multiple contexts. The `offer()` method in `ContractService` uses not only the `installment` but also the `carPrice`. However, the price is also needed by other parts in the monolith, namely by the context "risk assessment."

We perform this refactoring with the following steps. The `carPrice` field and its accessor methods are copied into the `offering.Contract` class, and the `offer()` method in `ContractService` is modified to call `setCarPrice()` on both sides (see Figure 13-12).

A detailed analysis of the algorithm in the `offer()` method reveals that the price is only set when the offer is created and remains unchanged afterwards. Our pair realizes that the price comes into the system through the context "offering" and the class `Contract` in Monolease only needs to be informed about setting the price.

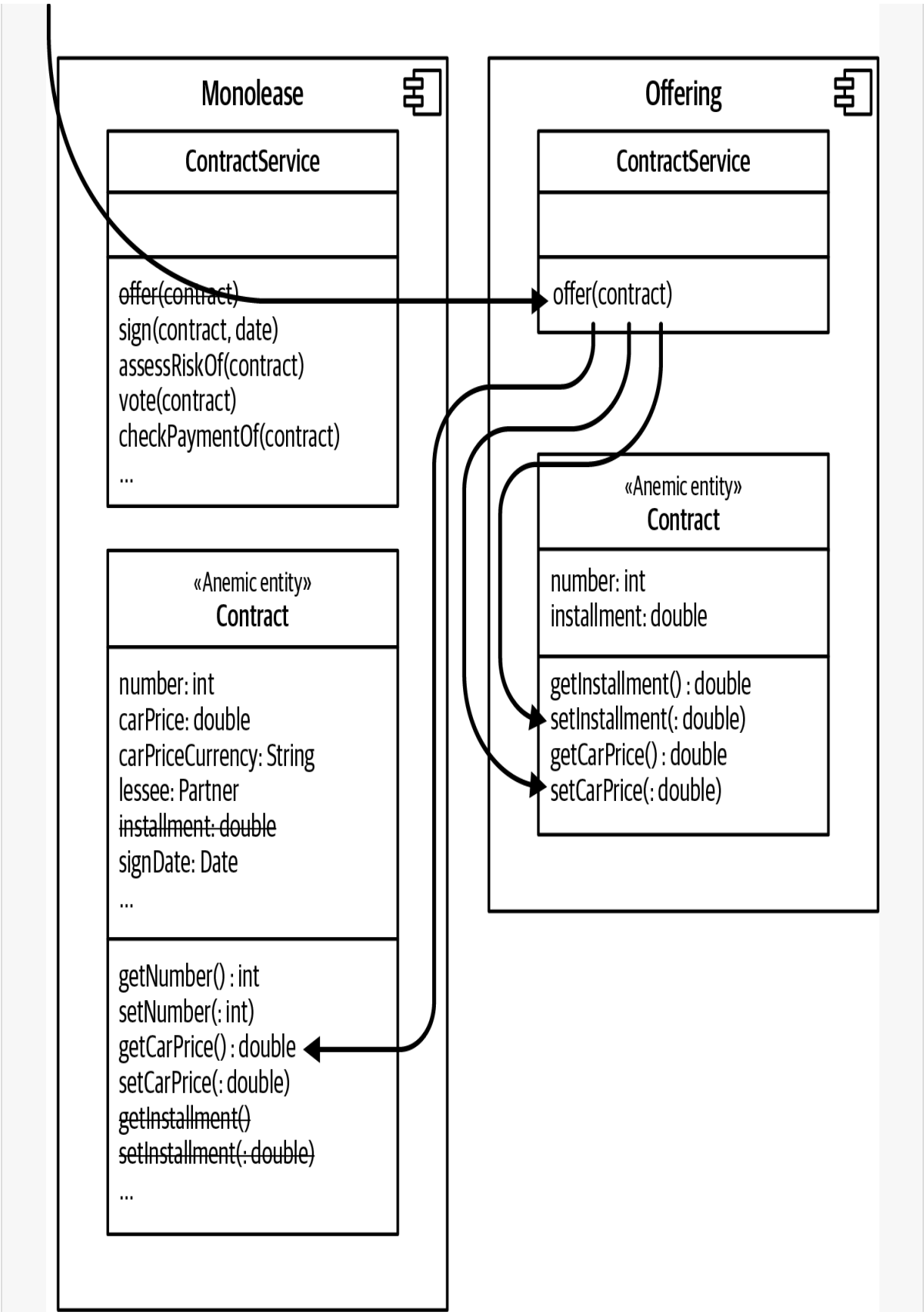

Monolease
ContractService
offer(contract)
sign(contract, date)
assessRiskOf(contract)
vote(contract)
checkPaymentOf(contract)
...
«Anemic entity»
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
...
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
getInstallment()
setInstallment(: double)
...
Offering
ContractService
offer(contract)
«Anemic entity»
Contract
number: int
installment: double
getInstallment() : double
setInstallment(: double)
getCarPrice() : double
setCarPrice(: double)

Figure 13-12. Extracting the context "offering"—data field and setter and getter duplicated.

To accomplish this, the pair introduces the first domain event (see Chapter 3) into the system and gives it the name `ContractSigned` (see Figure 13-13). Domain events of this type are sent by the "offering" context whenever a new contract is signed by the client. Because we are in a modular monolith, the team implements the event mechanism with an Observer pattern. In a distributed architecture of microservices, we would choose a different solution and send the event as a message over a message bus, for example.

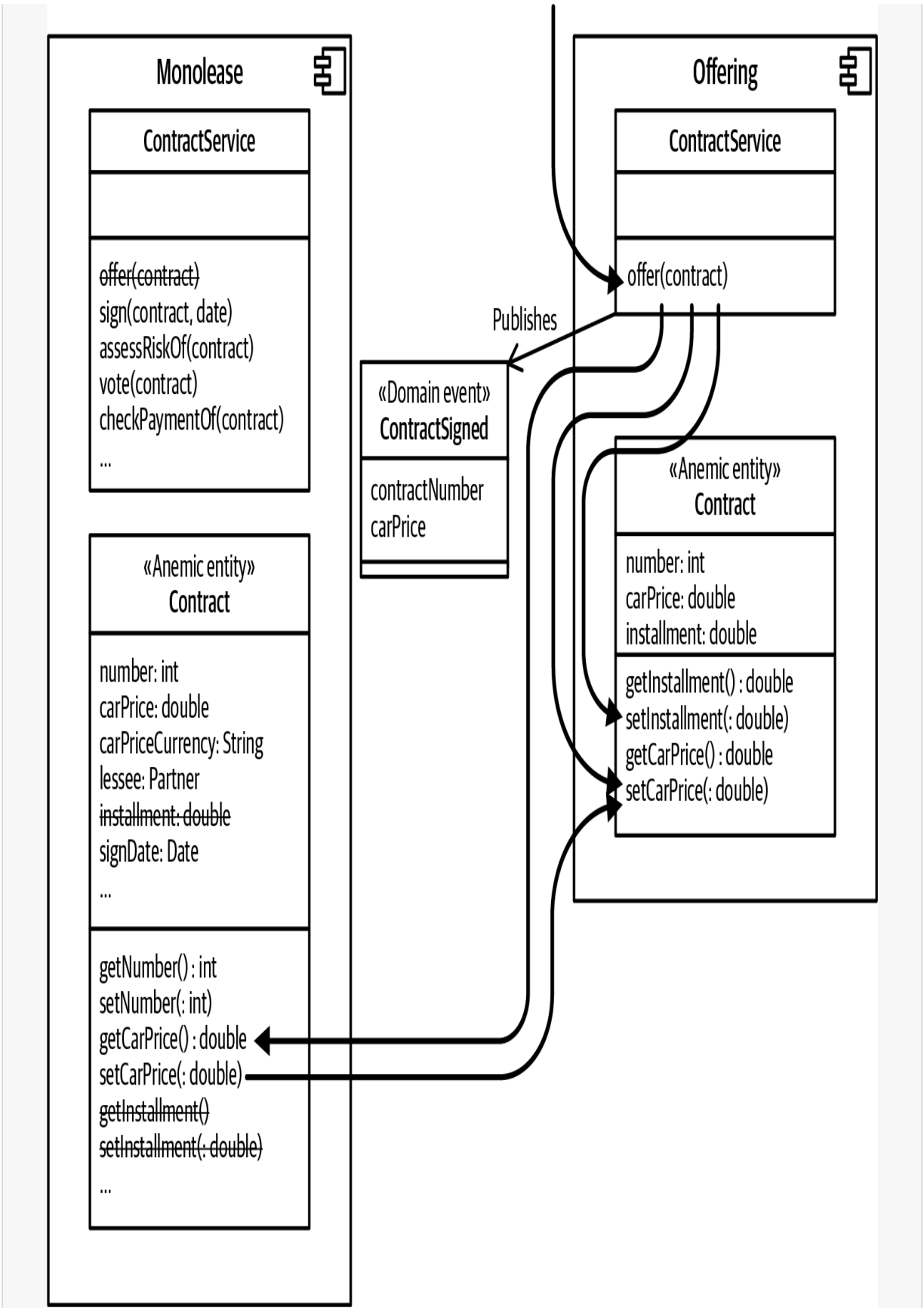

Monolease
ContractService
offer(contract)
sign(contract, date)
assessRiskOf(contract)
vote(contract)
checkPaymentOf(contract)
...
«Anemic entity»
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
...
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
getInstallment()
setInstallment(: double)
...
Publishes
«Domain event»
ContractSigned
contractNumber
carPrice
Offering
ContractService
offer(contract)
«Anemic entity»
Contract
number: int
carPrice: double
installment: double
getInstallment() : double
setInstallment(: double)
getCarPrice() : double
setCarPrice(: double)

Figure 13-13. Extracting the context "offering"—introduce a domain event.

The domain event `ContractSigned` is subscribed to by the monolith. For this purpose, a separate class `ContractSignedListener` is introduced, which processes the event. The `ContractSignedListener` loads the `Contract` by means of the field `contractNumber` from the database (ideally with the help of a repository) and then saves it again. This database access is omitted in the figures for space reasons. After loading the appropriate `Contract` object, the `ContractSignedListener` calls the `setCarPrice()` method on the `Contract` object in Monolease and thus sets the new price.

Now we can change the `ContractService.offer()` method in the new bounded context so that it no longer calls `setCarPrice()` from the `Contract` class in Monolease. Finally, we also cut the call from `setCarPrice()` in Monolease to `setCarPrice()` in the new context. This call is obsolete because the prices originate in the new bounded context "offering," and no backward path is needed anymore (see Figure 13-14).

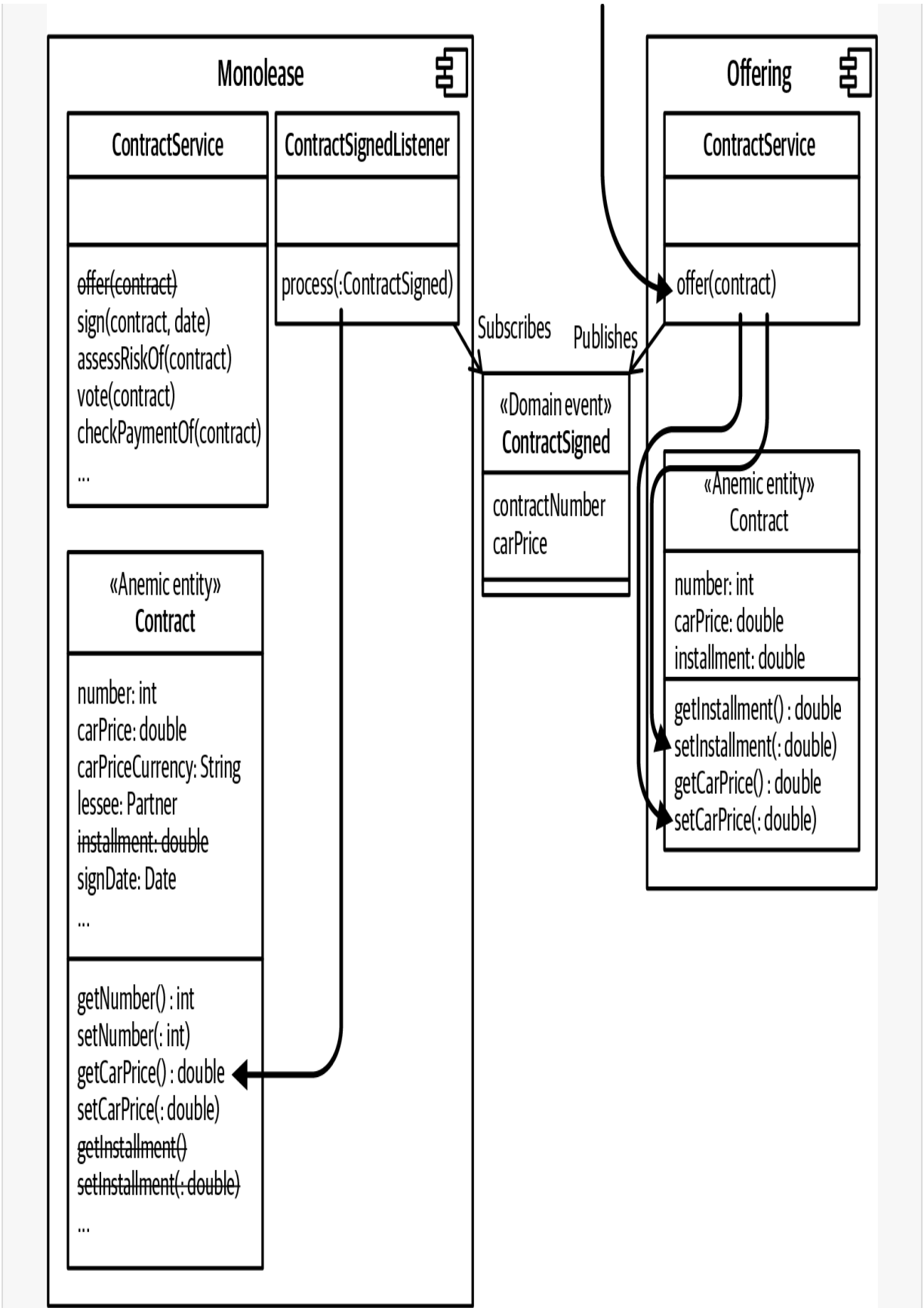

Monolease
ContractService
offer(contract)
sign(contract, date)
assessRiskOf(contract)
vote(contract)
checkPaymentOf(contract)
...
ContractSignedListener
process(:ContractSigned)
«Anemic entity»
Contract
number: int
carPrice: double
carPriceCurrency: String
lessee: Partner
installment: double
signDate: Date
...
getNumber() : int
setNumber(: int)
getCarPrice() : double
setCarPrice(: double)
getInstallment()
setInstallment(: double)
...
Subscribes
Publishes
«Domain event»
ContractSigned
contractNumber
carPrice
Offering
ContractService
offer(contract)
«Anemic entity»
Contract
number: int
carPrice: double
installment: double
getInstallment() : double
setInstallment(: double)
getCarPrice() : double
setCarPrice(: double)

Figure 13-14. Extracting the context "offering"—monolith subscribes to domain event.

Wow! Now there are no arrows pointing from the monolith to the new bounded context "offering," and no arrows pointing from "offering" to the monolith. The dependencies now only exist through the common interface (here, the event `ContractSigned`). This means that team Chewbacca has neatly extracted the bounded context "offering."

Of course, this applies initially only to the part of offering a contract, which is implemented with the `offer()` method. The team must now continue to work to split and move the other parts as well. We will not look at these individual steps in detail here. We hope that with this example we have given you a good idea of how the next reconstructions will proceed. After that, it's time to split the data model and UI layer; refactorings for this can also be found in the catalog.

The subdivided models are still anemic. Therefore, the next step is to strengthen the source code with domain knowledge, as described in Chapter 8, and to make a model with rich domain behavior out of it.

In the car leasing example, we just saw how you can separate services in a monolith and then use domain events to keep them synchronized. In most legacy systems, domain events at the service level are also a good means of decoupling in other ways.

Challenge: Introducing Domain Events Between Services

In most of the monoliths we have seen, services call each other in a variety of ways. Let's take a look at a real-world example of this.

CAROLA'S FIELD STORY: SERVICES STRONGLY COUPLED VIA INTERFACE

An impressive example is described in my book *Sustainable Software Architecture* (see [Lilienthal 2019]); Chapter 7.4.3 shows a system in which the developers have outsourced all relationships between services to interfaces.

In Figure 13-15 you can see several modules as squares, which are distributed over different layers, called user interfaces, business logic, framework, and the ServiceAPI. The arcs, green in the ebook and gray in the printed version, are the relationships between the modules, all running from top to bottom here. The ServiceAPI contains the interfaces for all services. The implementations of the services are hidden in the modules of the business logic and the framework layer.

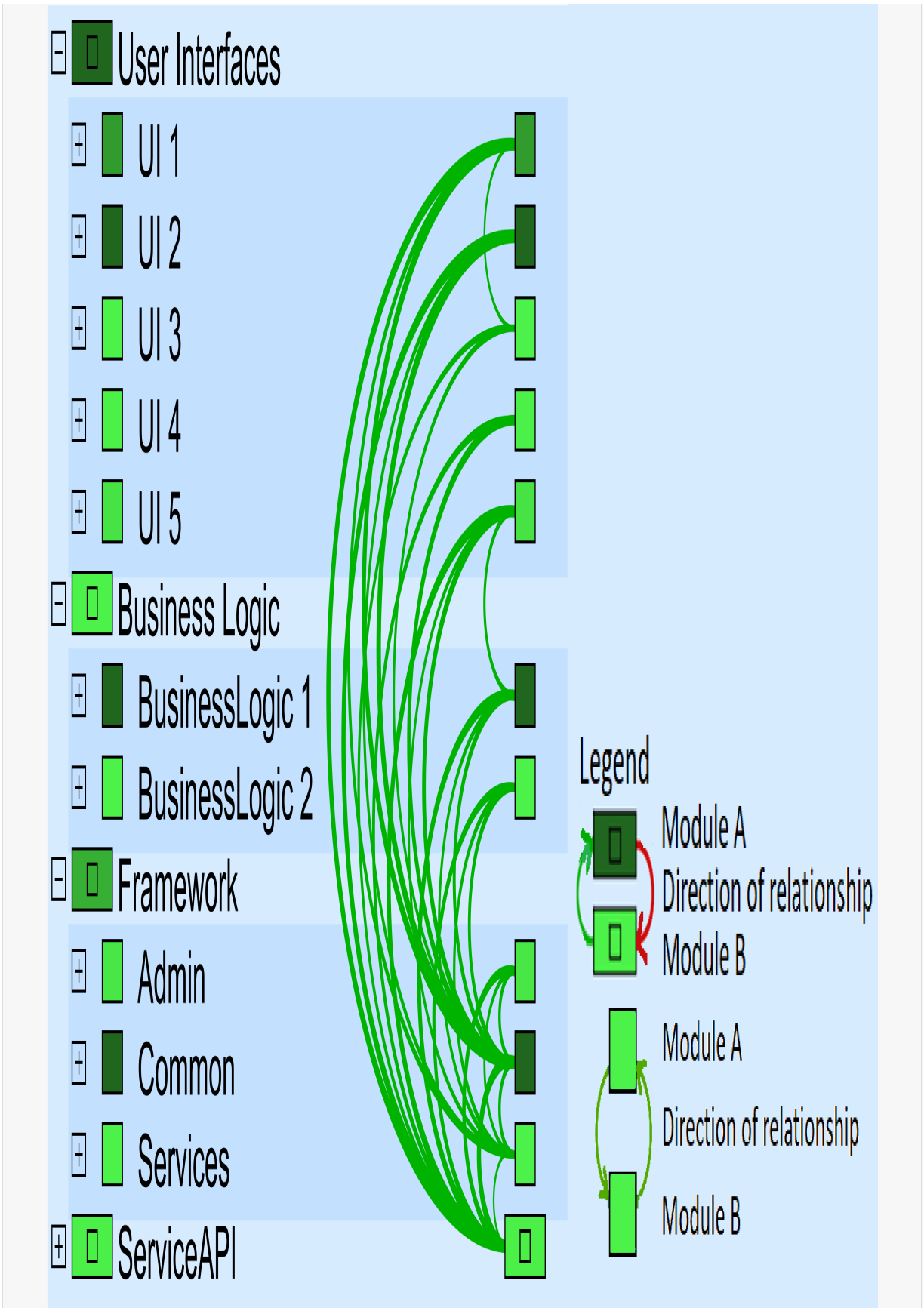

User Interfaces
UI 1
UI 2
UI 3
UI 4
UI 5
Business Logic
BusinessLogic 1
BusinessLogic 2
Framework
Admin
Common
Services
ServiceAPI
Legend
Module A
Direction of relationship
Module B
Module A
Direction of relationship
Module B

Figure 13-15. Screenshot of the architecture review tool: all projects use the ServiceAPI to call each other.

The basic rule of this architecture was that all service implementations must use other services exclusively through the interfaces in the ServiceAPI. You can see in Figure 13-15 that many calls from the business logic and framework layers go to the ServiceAPI. But services are also called from the UI layer through the ServiceAPI.

By itself, this is a simple architectural rule to gather all interfaces in the ServiceAPI, and it is also a frequently used pattern for the interface between client and server. However, by reusing this design in the server, i.e., between the services from the business logic and framework layers, the strong coupling between the services is obscured. If the interfaces are integrated into the respective domain-driven projects, a completely different picture emerges (see Figure 13-16).

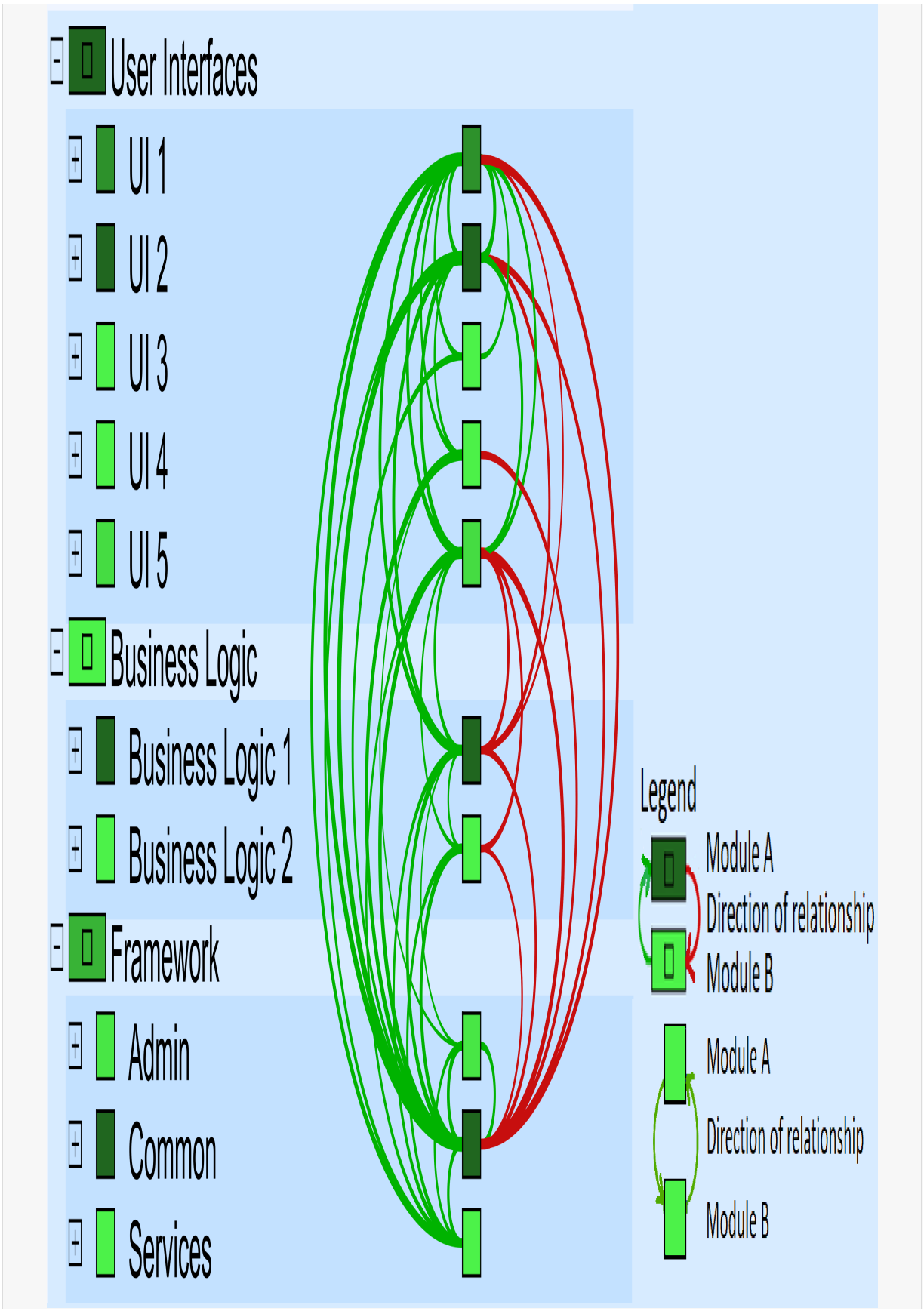

User Interfaces
UI 1
UI 2
UI 3
UI 4
UI 5
Business Logic
Business Logic 1
Business Logic 2
Framework
Admin
Common
Services
Legend
Module A
Direction of relationship
Module B
Module A
Direction of relationship
Module B

Figure 13-16. Screenshot of the architecture review tool: all projects use the ServiceAPI to call each other (interfaces integrated into domain-driven projects).

Now there are arcs not only on the left side of the axis (green in the ebook) but also on the right side of the axis (red in the ebook). Arcs on the left side of the axis go downwards, thus following the layering. Arcs on the right side of the axis go upwards, thus against the layering. What we see is a big ball of mud here! The various modules are strongly coupled with each other. The module Common in particular is used frequently and makes extensive use of the modules above it and, unfortunately, even the UI. The first step here should be to break down the relationships between server and client, i.e., to the user interface layer, but afterwards the strong coupling between the services should be addressed. Otherwise, this system will hardly be decomposable.

The structure in Figure 13-16 is certainly an extreme example of how services (and also the UI) can be entangled. However, we encounter this phenomenon in monoliths frequently. Even in some microservices architectures, we have found such service-level highly coupled structures via REST calls. Similarly problematic are microservices architectures where large amounts of events are sent back and forth, making them more like a distributed big ball of mud than a loosely coupled microservices architecture (see Chapter 5).

Our job in all these cases is to loosen the coupling between services in the monolith or between microservices. In the monolith, we should insert domain events where possible. Typical places for this are calls from one service to another where data or objects are passed without expecting a result. Instead of these calls, a domain event can be sent and thus the services can be decoupled.

Both in the monolith and in microservices architectures that are already decoupled via domain events, we have to check whether the domain events actually stem from the domain. The domain-oriented understanding of the business processes and the cuts in the business processes developed in step 1 help us do this. If domain events are sent out without the modeled

business processes having the business equivalent, the domain event must be examined at the respective point and, if necessary, the selected bounded contexts must be modified.

Challenge: Move Domain Logic Out of Services and into the Anemic Domain Model

In "The Method", we saw that there are basically two ways to approach Domain-Driven Transformation: tactical transformation first (path A) or strategic transformation first (path B):

- In Chapter 8, at the bank, we performed the tactical transformation first and then transformed the result strategically later in Chapter 12.
- For the leasing example, we have been transforming strategically since step 1 (see Chapter 10).

So for the leasing example, the question now is: how do we take up the fight against model anemia and improve our classes with tactical transformation?

Push Down Logic from Services into Anemic Entities

In an anemic domain model, we have "data bags"—the anemic entities—and service classes built on top of them. The business logic, which actually would belong in the entity and value object classes of a rich domain model, can be found in the service classes in an anemic model.

We therefore need to rebuild the domain classes and the service classes so that the business logic moves down from the service classes to the domain classes. To do this, we examine the logic in the services:

- What concerns only one aggregate that the service works with? That must be moved to the entities of the aggregate.

- What is overarching across aggregates? That needs to stay in the service.
- What protects the consistency and integrity of an aggregate? That moves down into the root entity of the aggregate on which the service operates.

If a method consists entirely of business logic that belongs in the entity, it is moved with *Move Logic from Service to Entity*, a special case of the refactoring *Move Method* [Fowler 2019, p. 198]. Often a method contains both parts of specialized logic, which belong into the service, and parts that belong in the entity. Then it makes sense to first extract the logic in the service with the refactoring *Extract Method* [Fowler 2019, p. 106] into its own method and then move it down.

In order to understand what applies for which code parts, it is often useful to model the domain precisely with Collaborative Modeling (see Chapter 4). For this purpose, fine-grained domain stories or Software Design EventStorming are suitable.

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: REFACTORING THE ANEMIC DOMAIN MODEL

To address the anemia in the domain model of the car leasing example, let's look at the class `Contract`. After decomposing the two contexts "offering" and "risk assessment," there are two classes each in the system, `Contract` and `ContractService`, each of which has been relieved of some of its fields and methods. Let us focus here on the context "offering" (see Figure 13-17) and on the elimination of domain model anemia within this context.

As we can see in the class diagram in Figure 13-17, `Contract` is a class that contains only fields and getters and setters to access those fields. It does not contain any business logic. Where is the business logic located? It is found in the class `ContractService`, and also in the class `Calculator`.

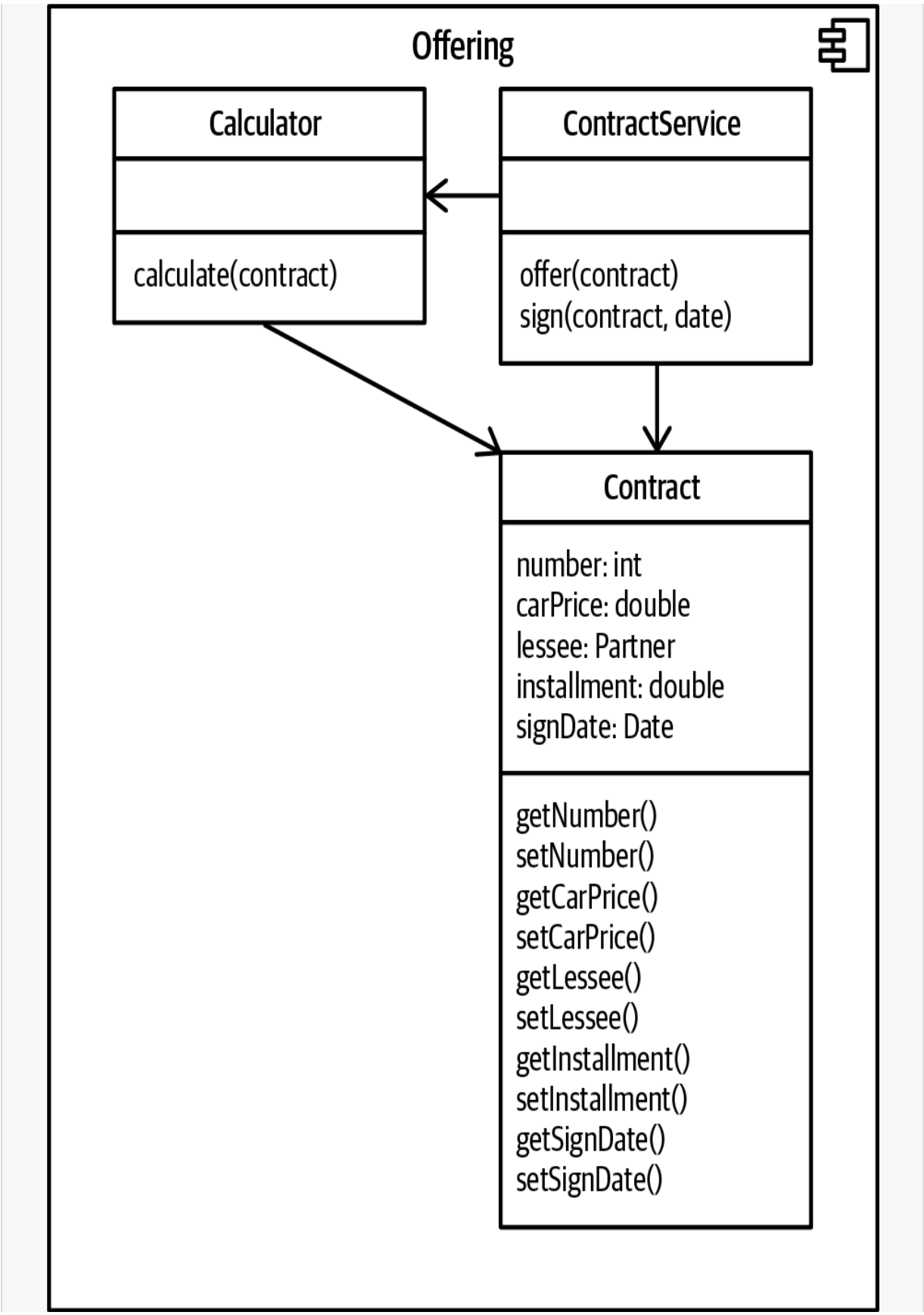

Offering
Calculator
calculate(contract)
ContractService
offer(contract)
sign(contract, date)
Contract
number: int
carPrice: double
lessee: Partner
installment: double
signDate: Date
getNumber()
setNumber()
getCarPrice()
setCarPrice()
getLessee()
setLessee()
getInstallment()
setInstallment()
getSignDate()
setSignDate()

Figure 13-17. Domain model anemia in the context "offering."

We use fine-grained domain stories to analyze the process in the context "offering" and find that the method `ContractService.offer()` currently implements three substeps. We pull these substeps out into their own methods and arrive at Figure 13-18.

For the newly extracted methods, we observe:

- The actual offering with `offerContractToCustomer()` is correctly placed here in the service, because it also needs data from the `Partner` (not shown in the figures).
- The method `filloutContract()` is wrong in the service and belongs one level deeper into the entity `Contract`.
- The method `calculateContract()` calls the class `Calculator` in the background to perform the business calculations. We will integrate this entire logic from the `Calculator` into the entity `Contract` and thus make the `Calculator` obsolete.

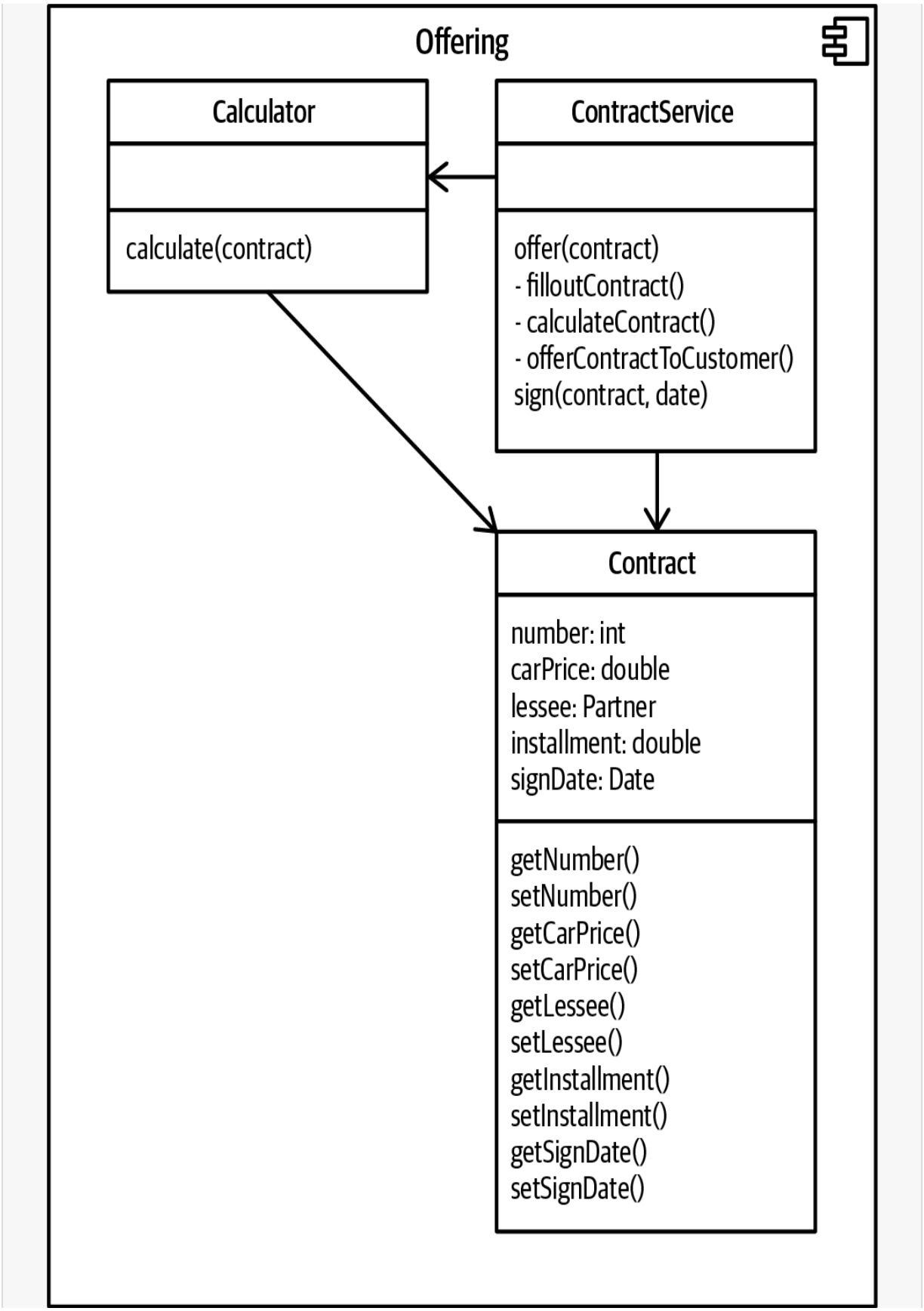

Offering
Calculator
calculate(contract)
ContractService
offer(contract)
- filloutContract()
- calculateContract()
- offerContractToCustomer()
sign(contract, date)
Contract
number: int
carPrice: double
lessee: Partner
installment: double
signDate: Date
getNumber()
setNumber()
getCarPrice()
setCarPrice()
getLessee()
setLessee()
getInstallment()
setInstallment()
getSignDate()
setSignDate()

Figure 13-18. Method `offer()` subdivided.

We complete the refactoring and arrive at Figure 13-19. Here we can see the following:

- The `fillout()` method has been provided with two parameters, the `carPrice` and the `lessee`.
- We can delete the setters `setCarPrice()` and `setLessee()` in `Contract`, because they are replaced by `fillout()`.
- In the method `calculate()`, the leasing rate is calculated, which was previously determined by the `Calculator` and then set into the `Contract` class by `setInstallment()`.
- We can dispense the `setInstallment()` setter now.
- The class `Calculator` is no longer needed.

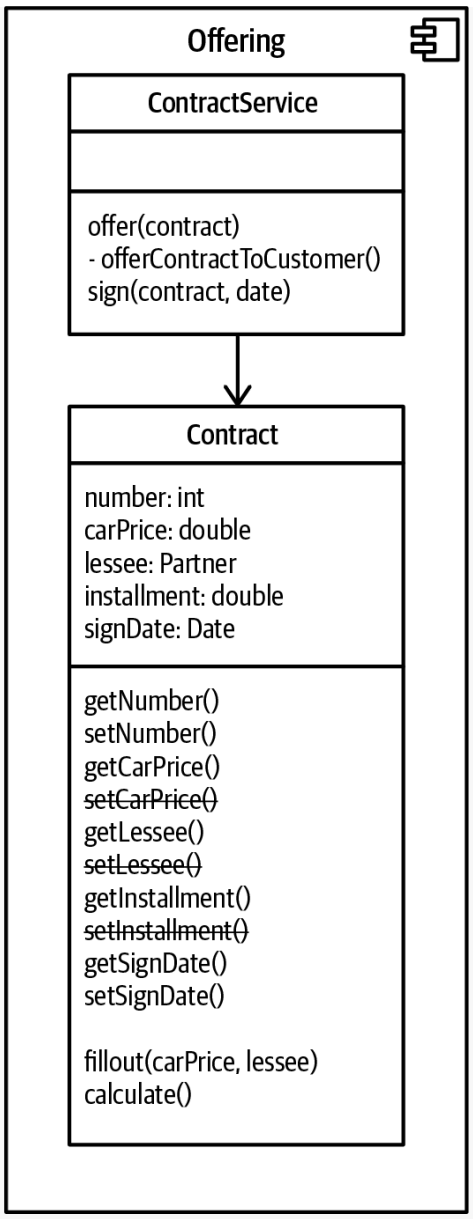


Figure 13-19. Anemia in the context "offering."

With this we have achieved encapsulation for the `carPrice`, `lessee`, and `installment` fields. This increases the robustness of the `Contract` class immensely, because the fields can no longer be set to an arbitrary value via the setter methods.

Further steps against model anemia are (see Chapter 8):

- Introduce Design by Contract and require a precondition that `calculate()` can only be executed if `fillout()` was

executed before. This way you can make sure that when `calculate()` is called, there is always already a value in the `carPrice` field, and the value of `installment` can be calculated in a meaningful way.

- Make the identity "contract number," which is hidden in the `number` field, explicit.
- Introduce value objects.
- Draw aggregate boundaries between contract and partner.

Challenge: Disentangling the Data Model

We have disentangled the parts of the domain layer, but what about the database?

Similar to the split of the (anemic) entities, we now split the database tables. As we learned in Chapter 12, database refactorings are harder than code refactorings, since we have to change three components (schema, data, and calling code).

CASE STUDY “CAR LEASING”

IN THE CAR LEASING EXAMPLE: DISENTANGLING THE DATA MODEL

We still have a monolithic data model in Monolease (see Figure 13-20).

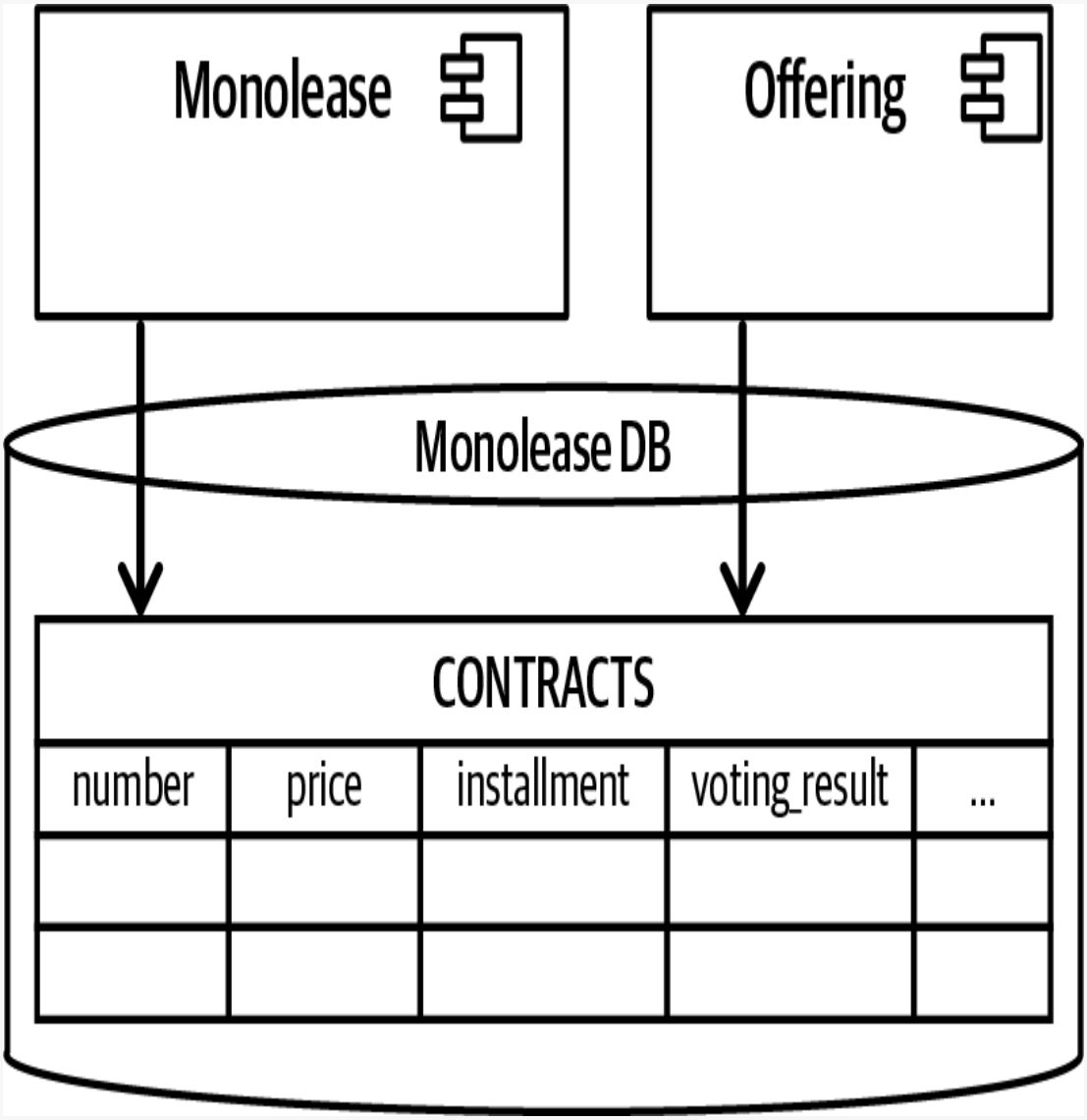


Figure 13-20. Monolithic database model.

From this monolithic model we want to separate a smaller model of `CONTRACTS`. That was planned as refactoring 1.2 in Table 12-2. As always, we move in baby steps; the first of these is to create an empty table `OFFERING_CONTRACTS` (see Figure 13-21).

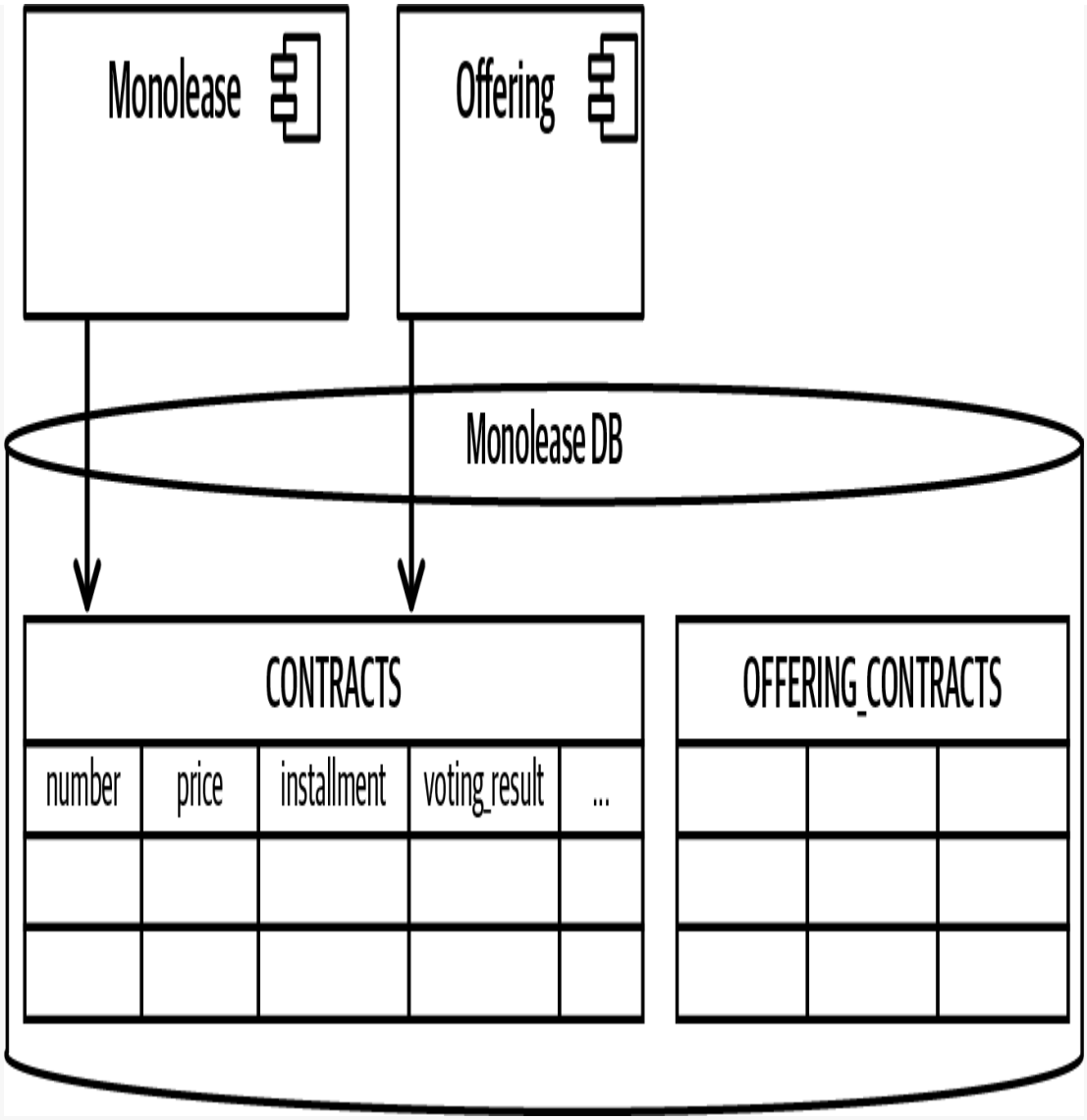


Figure 13-21. Adding empty database table `OFFERING_CONTRACTS`.

The new table gets the same primary key as the monolithic table: the contract number, which is represented by the column `number` (see Figure 13-22). For the time being the tables have a 1-to-0..1 relationship. The new table is still not used.

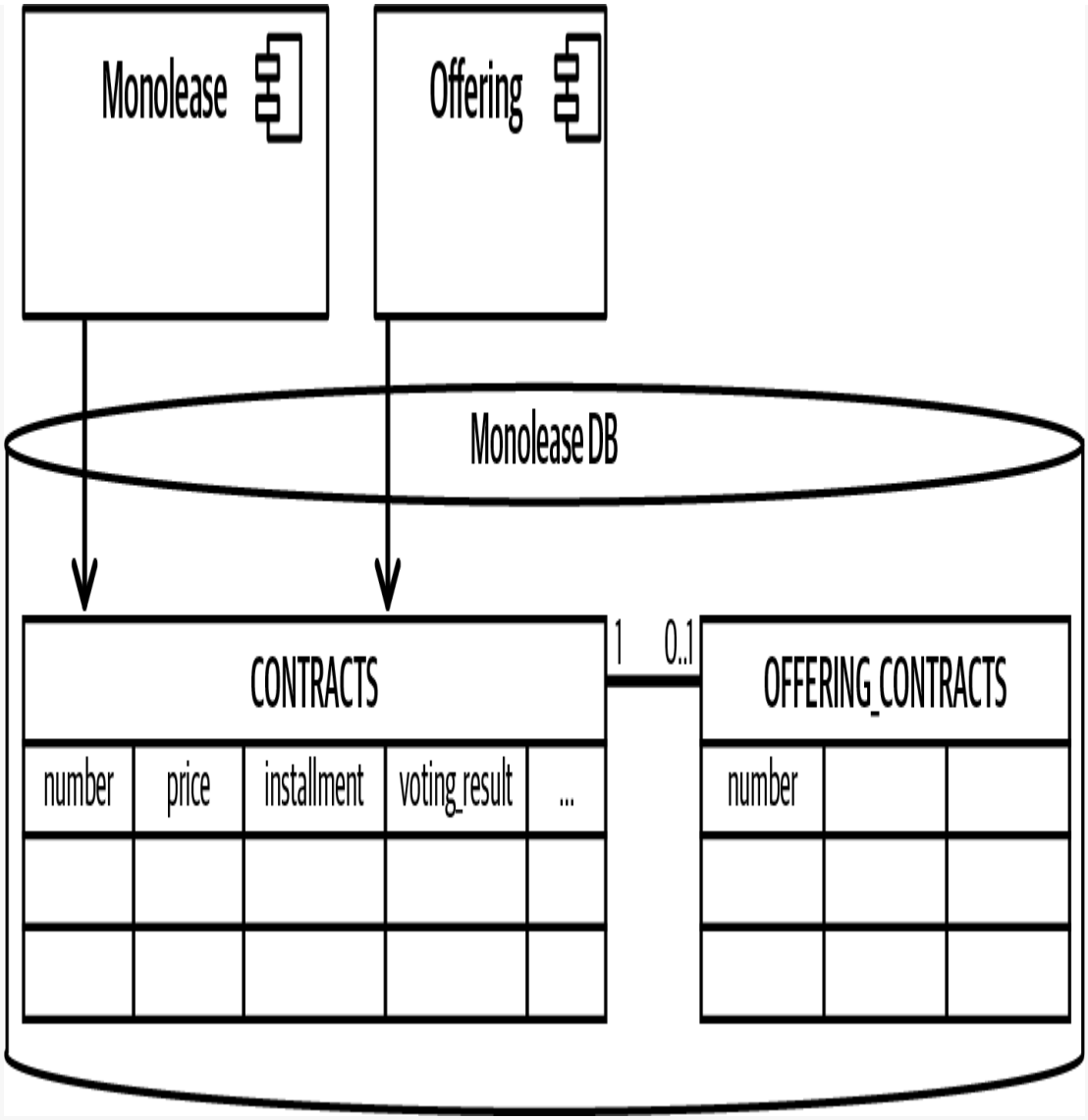


Figure 13-22. Adding primary key and relationship between tables.

The new table `OFFERING_CONTRACTS` is filled with the columns it needs. In the old table, the column `installment` that should be moved to the new context is marked with a drop date. On both tables, triggers are registered to update the respective other table when data changes (see Figure 13-23).

A data migration script is written that copies all required data from `CONTRACTS` to `OFFERING_CONTRACTS`. This would be those contracts that are needed in the bounded context “offering.”

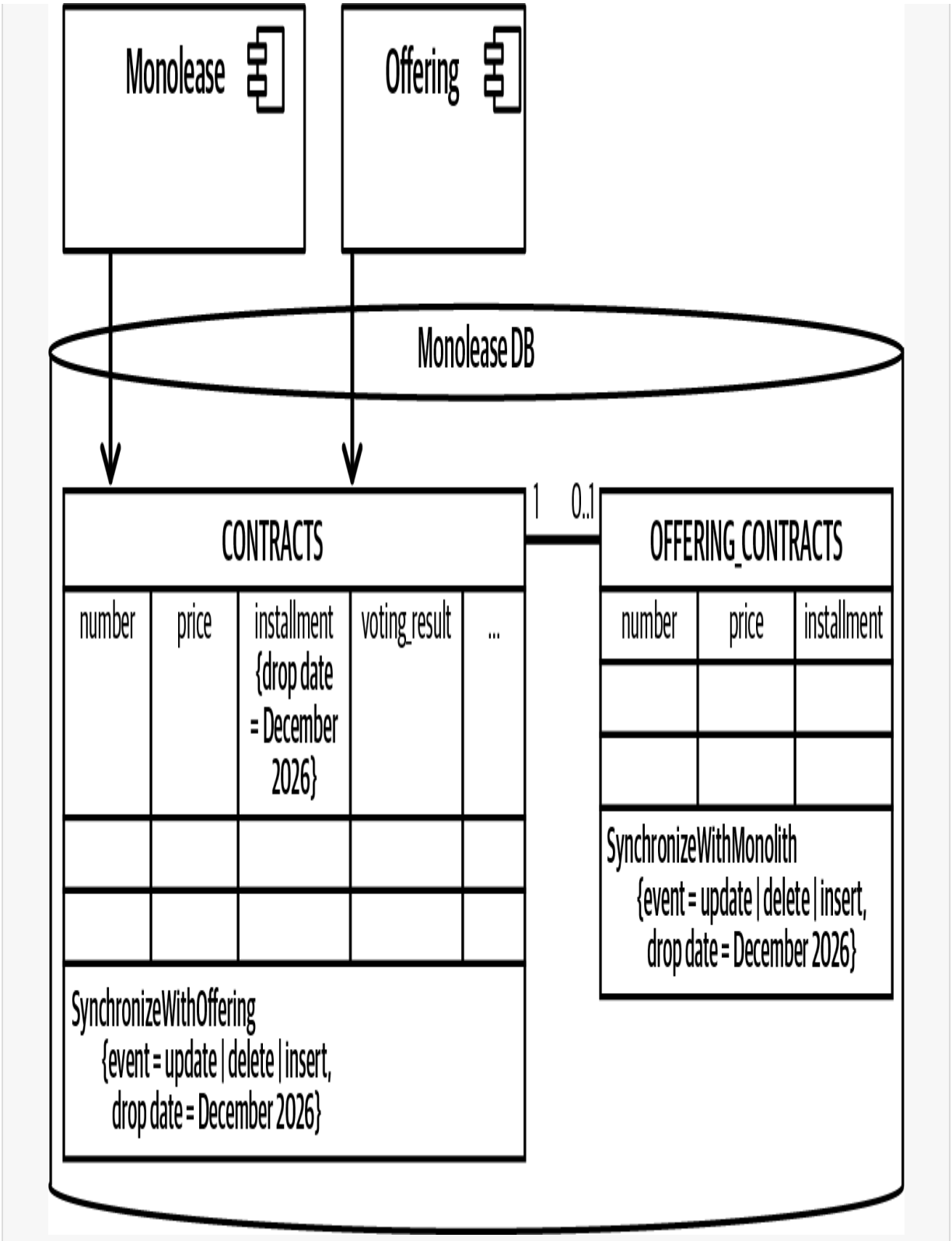


Figure 13-23. Adding required columns and synchronization triggers.

After all the preliminary requirements are met, the code that accesses the database tables is finally changed. Step-by-step, every call from the "offering" bounded context that previously accessed the `CONTRACTS` table is changed to access `OFFERING_CONTRACTS` instead. The result is Figure 13-24. As you can see, the database triggers are still in place.

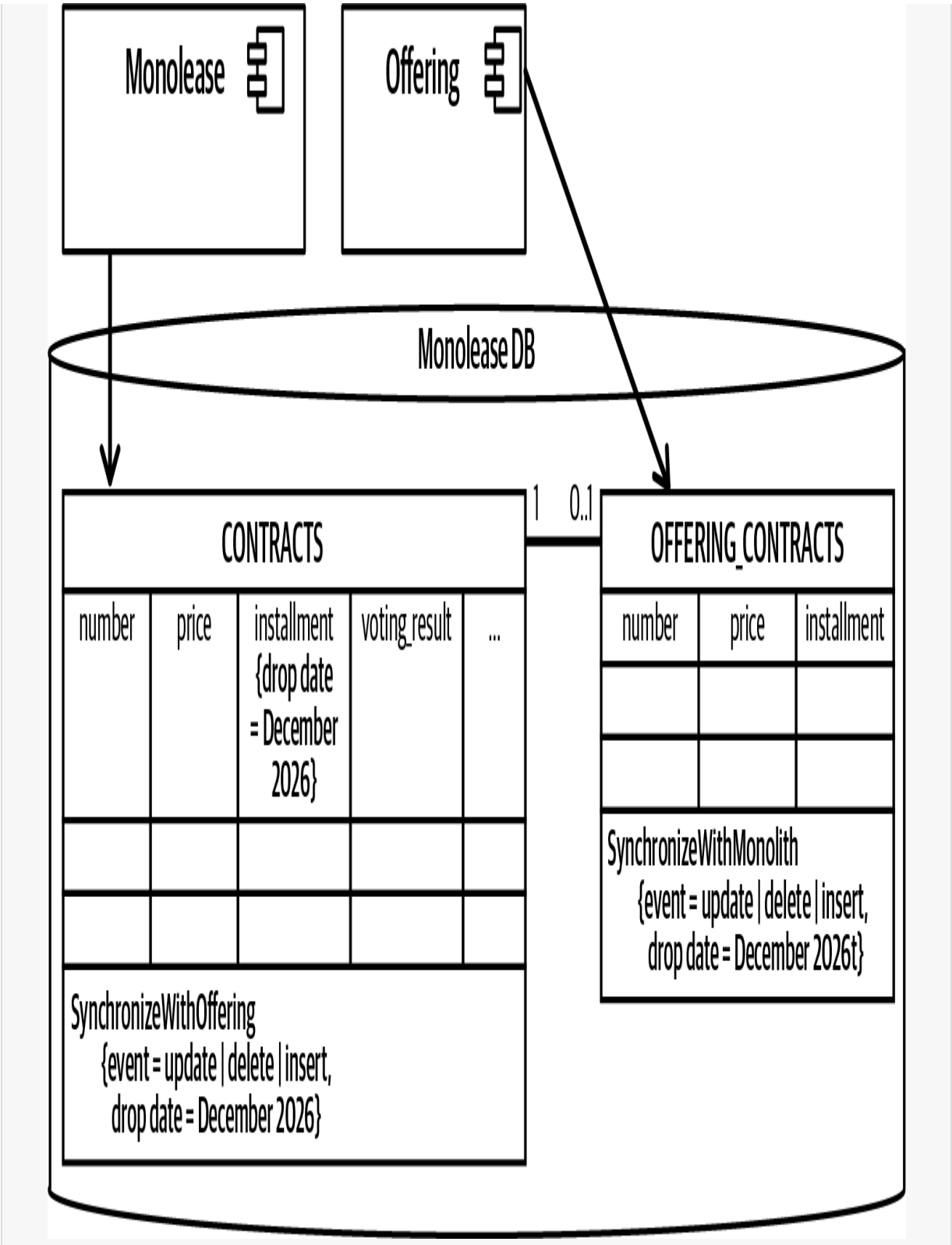


Figure 13-24. Altering the code in bounded context "offering" to access the new table instead of the old one.

When the transition period has passed (i.e., on the drop date), the now unused column `installment` is removed from the old table `CONTRACTS` since all the accesses now go to column `installment` in the new table `OFFERING_CONTRACTS`. That leads us to also remove the triggers to update data on both sides, and to remove the relationship between the tables (see Figure 13-25).

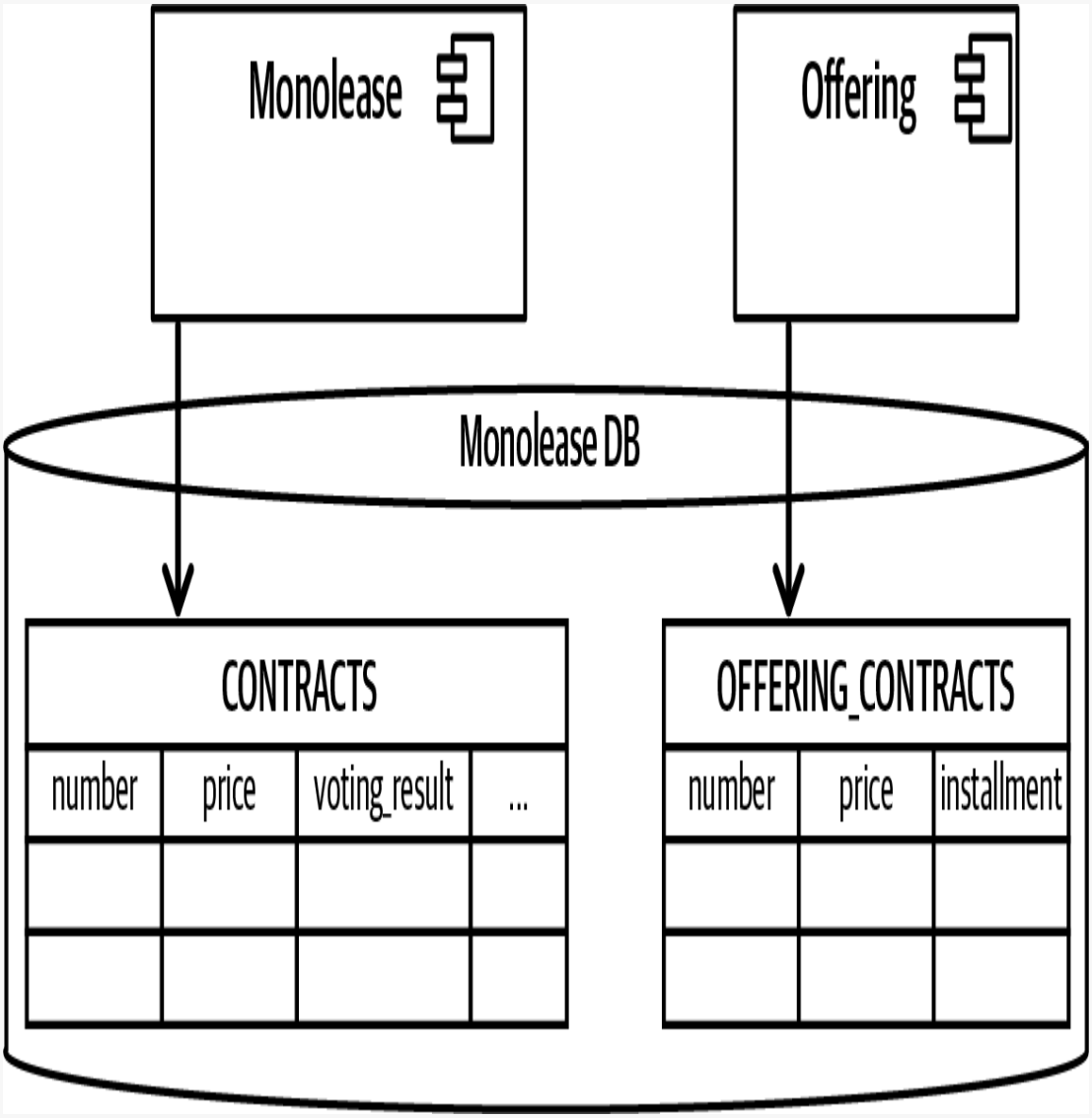


Figure 13-25. Removing unused columns from old table, synchronization triggers, and relationship.

As a final step, the new table `OFFERING_CONTRACTS` is moved to a new database (or database schema) with the name `offering` that belongs to the eponymous bounded context. To follow the idea of using

domain language everywhere, `OFFERING_CONTRACTS` is renamed to `CONTRACTS` (see Figure 13-26).

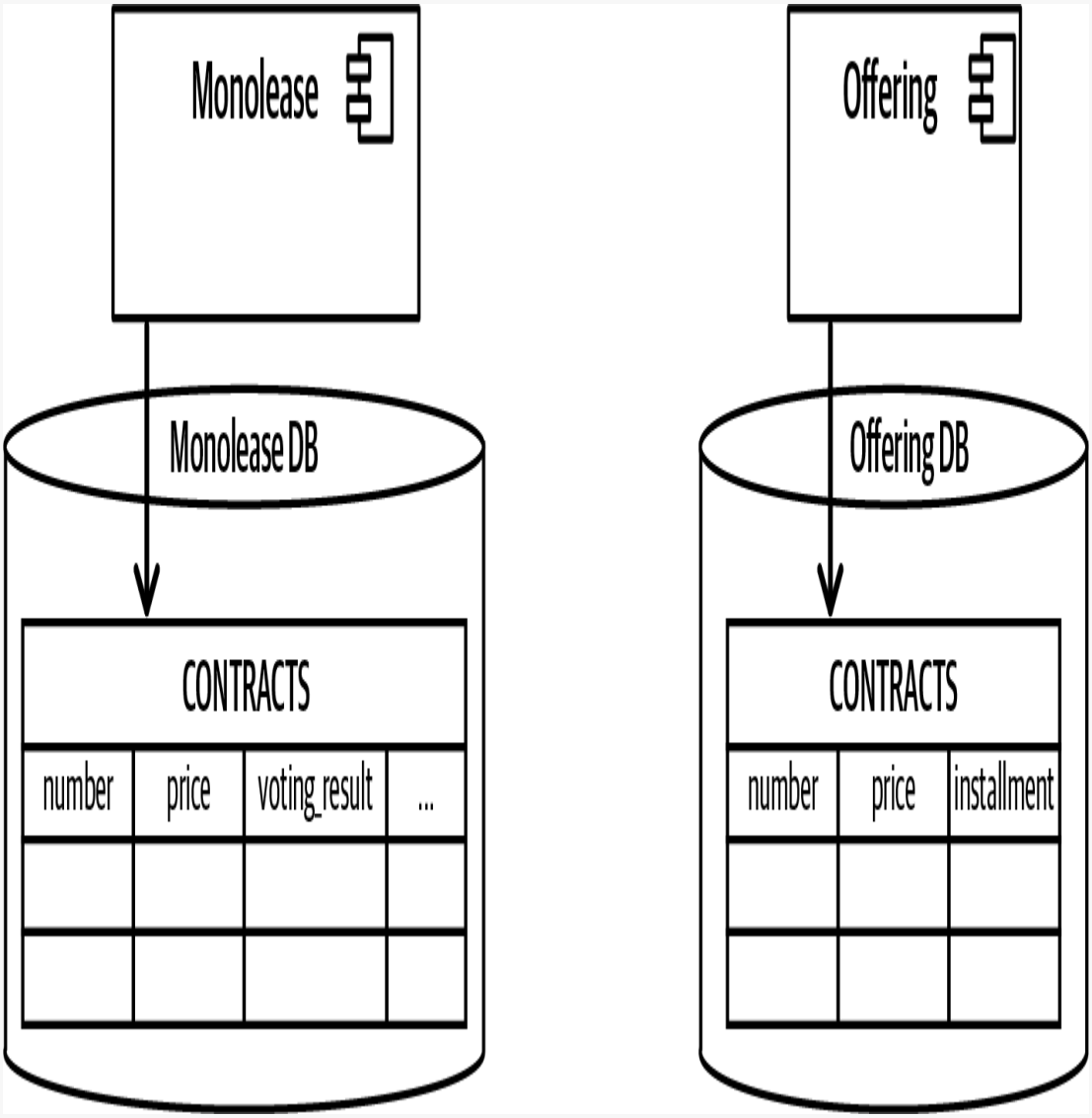


Figure 13-26. Moving the new table into a new database (schema) for the extracted bounded context.

All the steps that were taken are evolutionary in the sense that each intermediate result is viable, i.e., shippable to production.

Challenge: Selecting Findings at the Appropriate Moment

In order to tackle findings, we need to think about the appropriate strategic and tactical refactorings for the respective findings—so basically, repeat step 3 for the findings. But wait! That's exactly what we don't want; we don't want to compile a complete list of all useful refactorings.

Over the past decades of software development, up-front design has proven to be an approach that generates a lot of useless effort (see also "The Method"). Rather, we want to plan for the appropriate findings at the moments when management-level decisions are made to extend or change our system.

IN THE CAR LEASING EXAMPLE: TACKLE THE FINDINGS

At Alphorn, the management decides that the user interface with which the customer can make their leasing inquiries should also be usable on mobile devices. Our Monolease with its outdated UI technology is not able to generate responsive UIs that can be used on both computers and mobile devices. This is where our finding “outdated user interface technology” comes into play.

We certainly won’t be able to convert the entire Monolease UI to modern technology all at once—that would be far too time-consuming. But the “offering” bounded context, in which the customer selects their vehicle and signs the contract, could be the first to be converted to a modern UI technology.

We ask team Chewbacca to work out the strategic and tactical refactorings for the “offering” bounded context that match the finding. This probably requires a session with the architecture analyst and their tools, because architecture discussions are much faster with tool support. Subsequently, these refactorings run into the improvement backlog following the same procedure as described above and are scheduled into the sprints.

Once the team has had the experience of converting the outdated UI technology in one bounded context, it is much easier to estimate the right path and the effort required for the other bounded contexts. We learn iteratively, as we do everywhere in this book, and can shape the strategic and tactical refactorings for the next conversions.

In this way, findings from the architecture review are also gradually brought into the implementation.

Challenge: Keeping the Stamina

Even though in Scrum we talk about sprints, our major refactoring is more like a marathon that is divided into many stages. That's why the following is still important:

- The sheer *size of the task* is scary. Don't give up.
- Knowing that you *can't fix the system in whole* and in all places is frustrating. Establish continuous improvement as a daily habit and stay persistent! (We've already talked about the Scout Rule in Chapter 7.)
- Keeping *motivation* high over a long period of time, keeping making small changes and keeping going, is exhausting. Be proud of and celebrate even small successes! (Reading the Starfish Principle[2] is also helpful.)
- Even if the system is *not completely* decomposed, it's already worth a lot if we pull parts apart. Let's rejoice in these partial successes!
- Again and again it will happen to us that a refactoring is *too big for one sprint*. There is only one thing that helps: split, split, split! Often the following order applies: (strategic) bounded context refactorings split into (tactical) class refactorings, then split class refactorings into method and field refactorings.
- We are tackling something *complex* with decomposition, and we don't know a hundred percent from the start how to proceed and which steps will lead us to the goal. If the first idea doesn't work, we try a second or third. Let's learn by experimenting!

Somewhat disrespectfully spoken, so far, we've been drawing nice diagrams with boxes, stick figures, and arrows, and sticking colorful pieces of paper on the wall—now it's about *running code in production*.

Summary

In this chapter, we finally got our hands on the real source code and started changing it. This is time-consuming and tedious, but the results are very promising, even in the small example.

It is crucial that everyone understands what is happening: refactoring is never done—every improvement reveals the next one. If you want to decompose a legacy system, you have to work iteratively through the source code over a long period of time. It is important to keep in mind that each refactoring—no matter how small—is already a step forward. Don't let the size of the challenge discourage you: every improvement counts!

---

1 In Java, package names like `com.alphorn.monolease` and `io.leasingninja.offering` are usually common. In this book, we shorten the names to save space.

2 See *https://blog.doismellburning.co.uk/the-starfish-principle-every-little-bit-helps*.

Part IV. Conclusion

In this final part of the book, we want to both take a look into the future and summarize the whole method of Domain-Driven Transformation.

What we have described so far is distilled from our (and others') experiences and battle-tested. In the outlook (Chapter 14), we present new ideas that might help, though there is room for exploration. We invite you, dear reader, to try these out and look forward to your feedback!

In Chapter 15 we give a recap of the whole method of Domain-Driven Transformation. This is meant as both a summary and a quick reference.

Chapter 14. Outlook: Domain Patterns and Their Implementation in Bounded Contexts

Like a Domain-Driven Transformation is never “done,” the method Domain-Driven Transformation is also not “done.” In this chapter we want to share some new and speculative ideas with you that we think can help with future transformations and Domain-Driven Design in general.

In recent years, we have seen a number of domains with their subdomains and have decomposed them. The goal was always to divide one or more software systems among subdomains and to achieve a meaningful decomposition of the software system into bounded contexts based on the boundaries found in the domain. Thus, we were concerned with both the domains and their implementation into bounded contexts. Over time, we gained the following impressions:

- There are (sub)domains that are easier to decompose into (sub)subdomains, and those where it is significantly more difficult.
- There are implementations in bounded contexts that make it easier or harder to decompose the legacy system.
- The development of the domain and the software influence each other, and the legacy software is to be decomposed differently depending on the state of development.

In order to structure the whole thing for us, we have so far identified three *domain patterns*, and have thought about further aspects in the implementation in bounded contexts.[1] We are far from finished with our

considerations. We hope that our ideas will spark your interest. We have already had heated discussions with some of our colleagues mentioned in the introduction, and are now hoping for continued feedback from our readership and audiences at conferences. Only then can new ideas mature in our community.

We dealt with the following dimensions to analyze the patterns we perceived in domains and their translation into bounded contexts:

- Dimension 1: how is the *workflow in the domain* organized?
- Dimension 2: how is the *domain model* implemented in the software?
- Dimension 3: what is the *shape* of the domain?

We consider these dimensions in the following sections.

Dimension 1: Workflow in the Domain

This dimension is about how the users organize their work and their cooperation. What do they do on their own, and what do they do together?

Domain Pattern Pipeline

If the work in a domain is done step-by-step, partly also by different roles, this domain follows the domain pattern *Pipeline*[2] (see Figure 14-1). The steps have an order, and it is clear that the later step cannot be done without the step before it. Each role completes a work step and passes the work result to the next station, where someone takes another role and further processes the work result from the last step or uses it as a basis for the next work step.

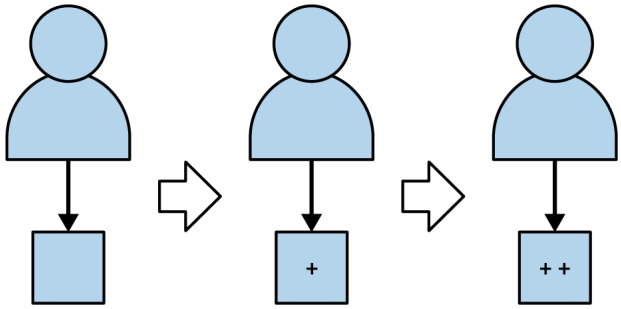


Figure 14-1. Domain pattern Pipeline*: a workpiece is processed consecutively by different people.*

Typical representatives of this domain pattern are:

- Logistics processes
- Purchase and delivery processes and trading platforms
- Conclusion of insurance or credit contracts
- Regulatory processes
- The cinema example, where the weekly schedule is created first, and then the tickets can be sold
- The leasing example: first comes the subdomain "offering," then "risk management"

In virtually every domain, parts of pipeline-like operations can be found; if the top level of a domain follows the domain pattern *Pipeline*, this domain can be analyzed and decomposed relatively easily with the Collaborative Modeling methods described in Chapter 4. This is because the boundaries in the business process, as we looked for in Chapter 10, are directly visible in *Pipeline* domains. This is certainly the reason why conference presentations and books like to use examples from domains that correspond to the *Pipeline* domain pattern at the top level.

Domain Pattern Blackboard

Much more difficult to analyze and decompose are domains where the various users work together with the software on one or more business objects. We call this domain pattern *Blackboard* in reference to [Buschmann

et al. 1996]. With the *Blackboard* pattern, an analysis of the work relationships between the individual roles involved does not reveal a recurring sequence of activities (see Figure 14-2).

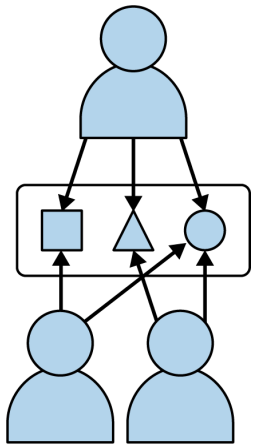


Figure 14-2. Domain pattern Blackboard*: different persons work on the same workpieces in undefined order and also in parallel.*

Rather, the overarching coarse-grained process consists of activities that are not performed sequentially but in parallel; i.e., they have no sequence, so significantly more state transitions must be possible in the system at all levels than in the *Pipeline* domain pattern. Examples are:

- Calculation software for cars or machines and plants
- Planning of major events (event management)
- Observation and monitoring in a control center, e.g., monitoring of plants, logistics processes, traffic flow, or the disruption of supply chains
- Software development
- Communication software, such as Teams, Zoom, and others
- Computer games, where many players meet in one world

In *Blackboard* domains, when analyzing and decomposing, you have to pay attention to the different actors and the steps of their cooperation in order to find the boundaries in the business processes.

Domain Pattern Dialog

In some domains, though mostly below the coarse-grained process, you may find a variant of the *Blackboard* domain pattern. This pattern is not just about working together on one or more business objects, but about explicitly negotiating a common plan to which multiple parties contribute information and respond to each other's information. We call this domain pattern *Dialog* (see Figure 14-3).

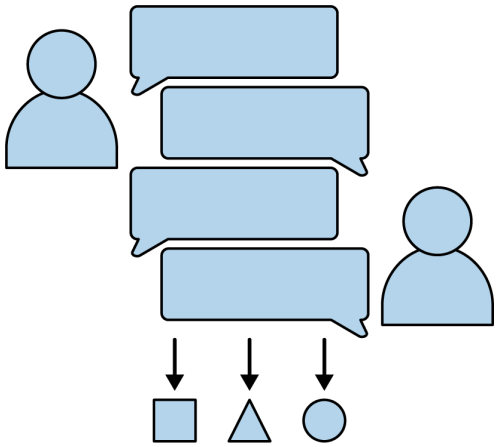


Figure 14-3. Domain pattern Dialog*: people negotiate a common plan.*

At the end of this coordination process, there is a result that all participants have explicitly agreed on. As a rule, this variant should be chosen instead of a *Blackboard* when the negotiation process takes place between different companies or administrative departments that have no authority over each other, or when no explicit administrative act is associated with the process on the part of the administrative department. Examples include:

- Negotiating ship voyage schedules between shipping companies and ports
- Negotiating the type, delivery, and maintenance cycles for major equipment between dispatchers and customers
- Negotiating long-term construction scheduling between management and construction applicants
- Some games also fall into this category if they are based on negotiation processes

In a *Dialog* domain, it is crucial to identify common parts in the negotiation process and make them subdomains or bounded contexts. Decomposing by actors or roles does not make sense for *Dialog* domains.

Evolution of Domains and the Patterns They Follow

During the evolution of a system, the pattern a domain follows may change. Let’s look at an example.

CAROLA'S FIELD STORY: CHANGE OF DOMAIN PATTERN IN A CONSTRUCTION COORDINATION SOFTWARE

In recent years, our company, WPS, has developed a product for coordinating construction measures in cities across organizational boundaries. In this product, the domain model is behaviorally very rich and well developed. But at the process level, the product is still in its infancy because none of the stakeholders had any idea of cross-organizational processes when it was developed. Initially, construction coordination functioned like a blackboard. In the beginning, construction site coordination across organizational boundaries was more of a construction measure mapping from all sides and negotiating with each other how to integrate the individual construction measures into an overall plan. This approach was relatively chaotic, and after two years of experience, the domain experts began to introduce processes with quality gates, i.e., certain states of the process at which there must be position statements and/or approvals.

It is interesting to observe here that on the one hand, the software drives the domain, while on the other hand, it lags behind in development steps. The domain evolves because the software provides the users with an overview of the large number of construction measures, thus enabling coordination. Here, the software is the driver. However, the next step in process support must first be built into the software, which in turn makes users impatient.

What we describe here is a young domain that is slowly evolving from a *Blackboard* domain into a *Dialog* domain. Since this domain involves coordination between different companies and authorities, i.e., agreement must be reached, and complex information is additionally required from these different participants, it is unlikely to develop into a *Pipeline* domain.

Dimension 2: The Domain Model in the Software

This dimension is about the implementation of the domain model in the software. Do all users work on the same work items or each on different ones? How are the work items modeled in the software? Are they anemic or behavior-rich (see Chapter 8)?

Central Domain Item Versus Diverse Domain Items

If you want to decompose a software system, you will have more trouble with systems where there is a central domain item than with domains that have naturally formed special domain items for each division or department. In Chapter 12 we wrestled with unbounded models, which in most cases are based on central domain items.

An example of such a central domain item is the container at the container terminal that we met in Chapter 1.

CASE STUDY “CONTAINER TERMINAL”

AT THE CONTAINER TERMINAL: A CENTRAL DOMAIN ITEM

All departments and all work steps of the container terminal require the domain concept “container” and its data in order to carry out their work steps: the delivery (rail/truck), the yard, the transport equipment on the terminal, the stowage planning, and the control center (see Figure 14-4).

Everywhere it is about containers, and so the domain model of the software systems used at container terminals will contain the central domain item “container.” This central domain item makes decomposition difficult. If the original development teams had thought of the container as something with different roles in different parts of the business, they could have avoided the central domain item. But that’s easier said than done afterwards, when the systems are in place.

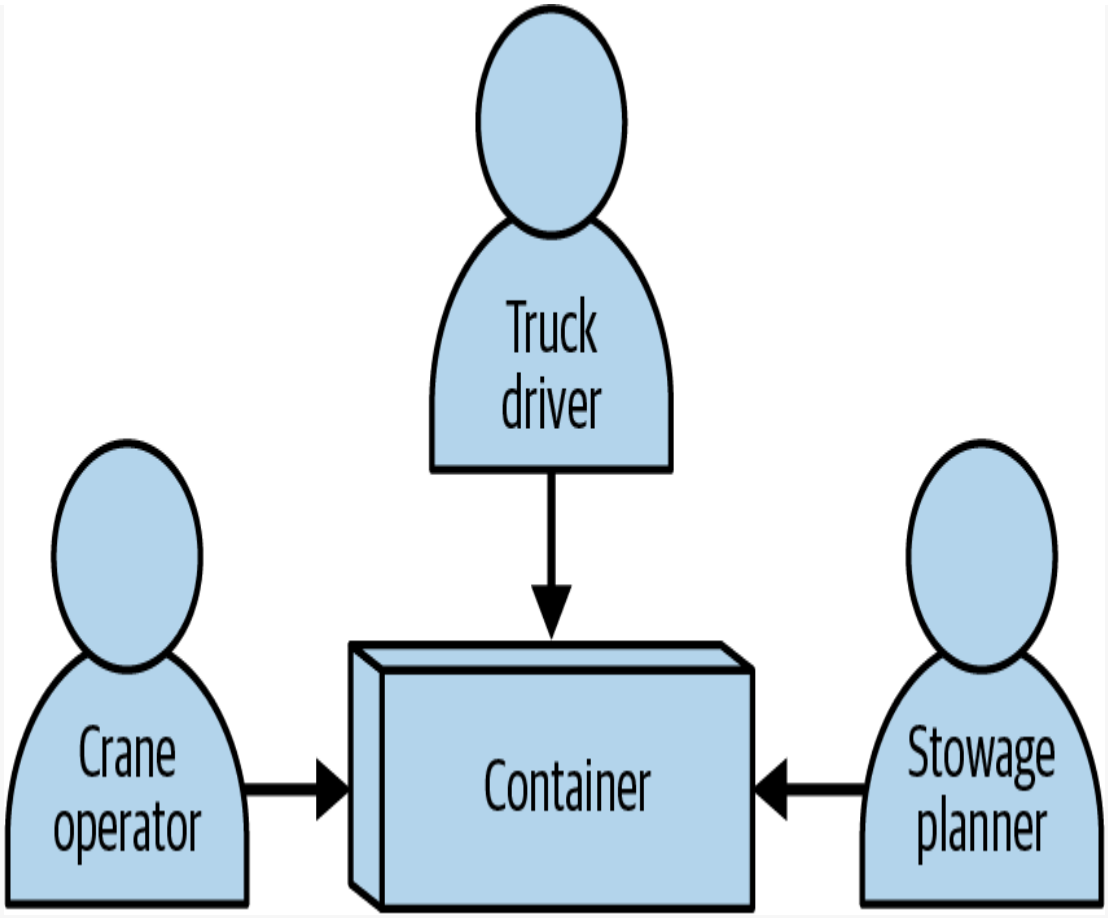


Figure 14-4. Example for a central domain item: the container is used by virtually everybody on the container terminal.

One domain where this problem is much less prevalent is the banking domain.

CASE STUDY “BANK”

IN THE BANK: DIVERSE DOMAIN ITEMS

Here there are various divisions, such as the lending business, stock business, customer service, payment transactions, etc., which all have to do with the customer, but whose products differ significantly (see Figure 14-5).

A loan is something completely different from a stock, so no one would try to create a unified abstraction for this. Also, using a single class `BankCustomer` in all subdomains would be unlikely—even if we constructed such a case as an example in Chapter 12.

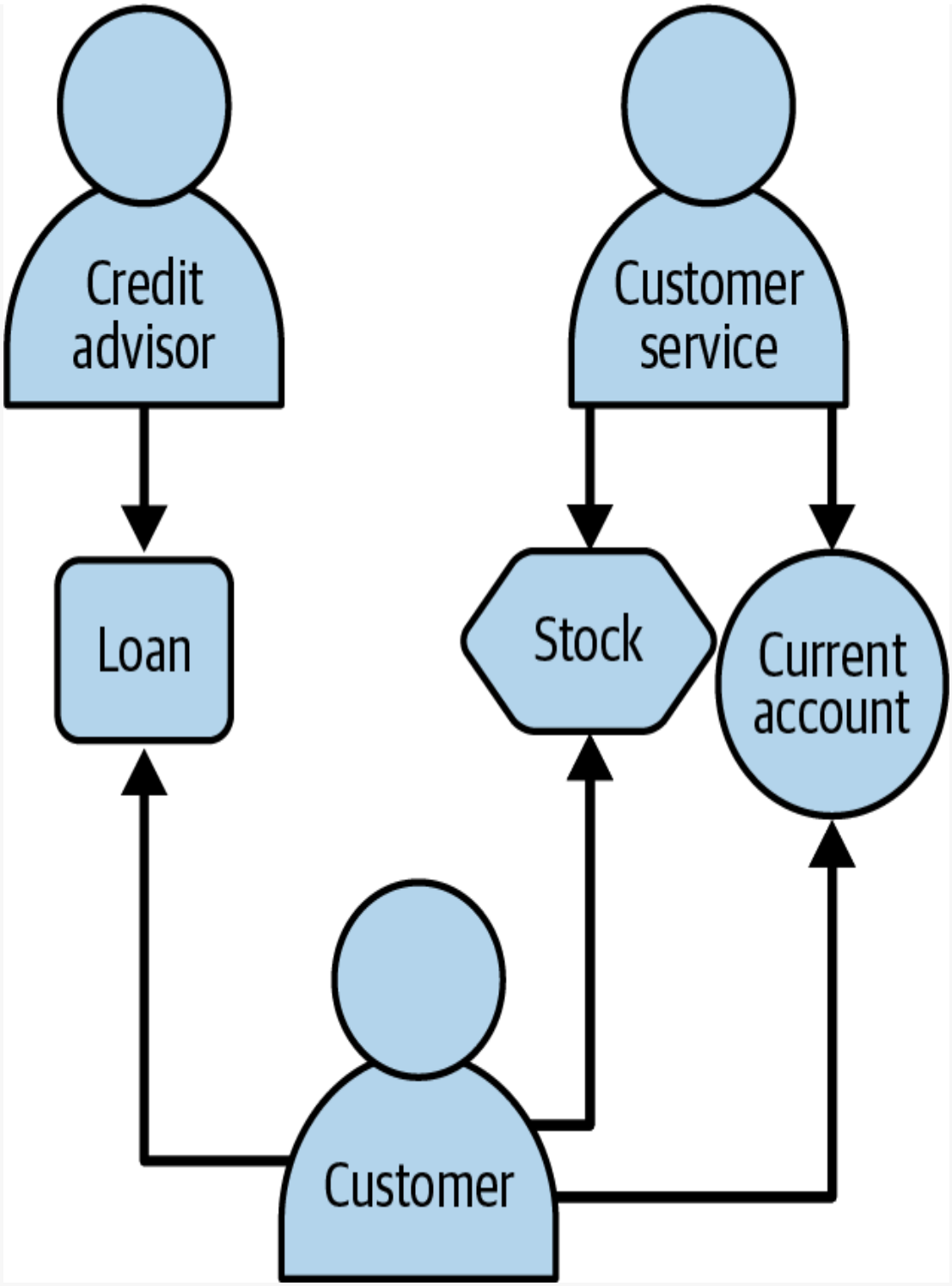


Figure 14-5. Example for a domain with diverse items: banking.

Insurance companies are also divided into lines of business with very different products (household insurance, life insurance, health insurance,

liability insurance, etc.). If an insurance company has software for these different lines of business, then you should definitely stick with it—the process of breaking up the cross-line of business systems at different insurance companies is too painful. There, the temptation to develop something across business lines is, unfortunately, too great. From our point of view, what tempts the architects and developers here is that in the insurance industry, all products are somehow called "insurance" and, with enough distance, look as if they have the same business process.

Admittedly, the extent of the central domain item also depends on the tendency of architects and developers to abstraction, i.e., to unification in class hierarchies. Also the degree of the intended reuse in a software system affects the decomposition of the central domain item.

> **TIP**
>
> For domains with central work items, it is especially important to focus on behavior rather than data. Our goal, especially for work items that appear central, is to always build specific models of the same things for different bounded contexts.

Form-Based Domain Model

In some software systems, it can be observed that the domain model consists mainly of forms. Such domain models are often found in administrative software, e.g., in the processing of applications of all kinds or in the recording, checking, and approval of tax returns (see Figure 14-6). These form-based domain models are often organized in a *Pipeline*-like manner. Such a domain model usually emerged because the previous paper-based work had been modeled in the software. Usually, these are old domains where people have worked with paper forms for a long time.

The implementation of these form-based domain models is most often done with CRUD applications.[3] The user fills data into a form and stores it. In many cases, such legacy systems are said to be a simple CRUD application, and therefore the division into modules/bounded contexts/separate domain models does not make sense. Certainly there are domains that need CRUD

applications, but from our experience, it's less often you would find them in practice.

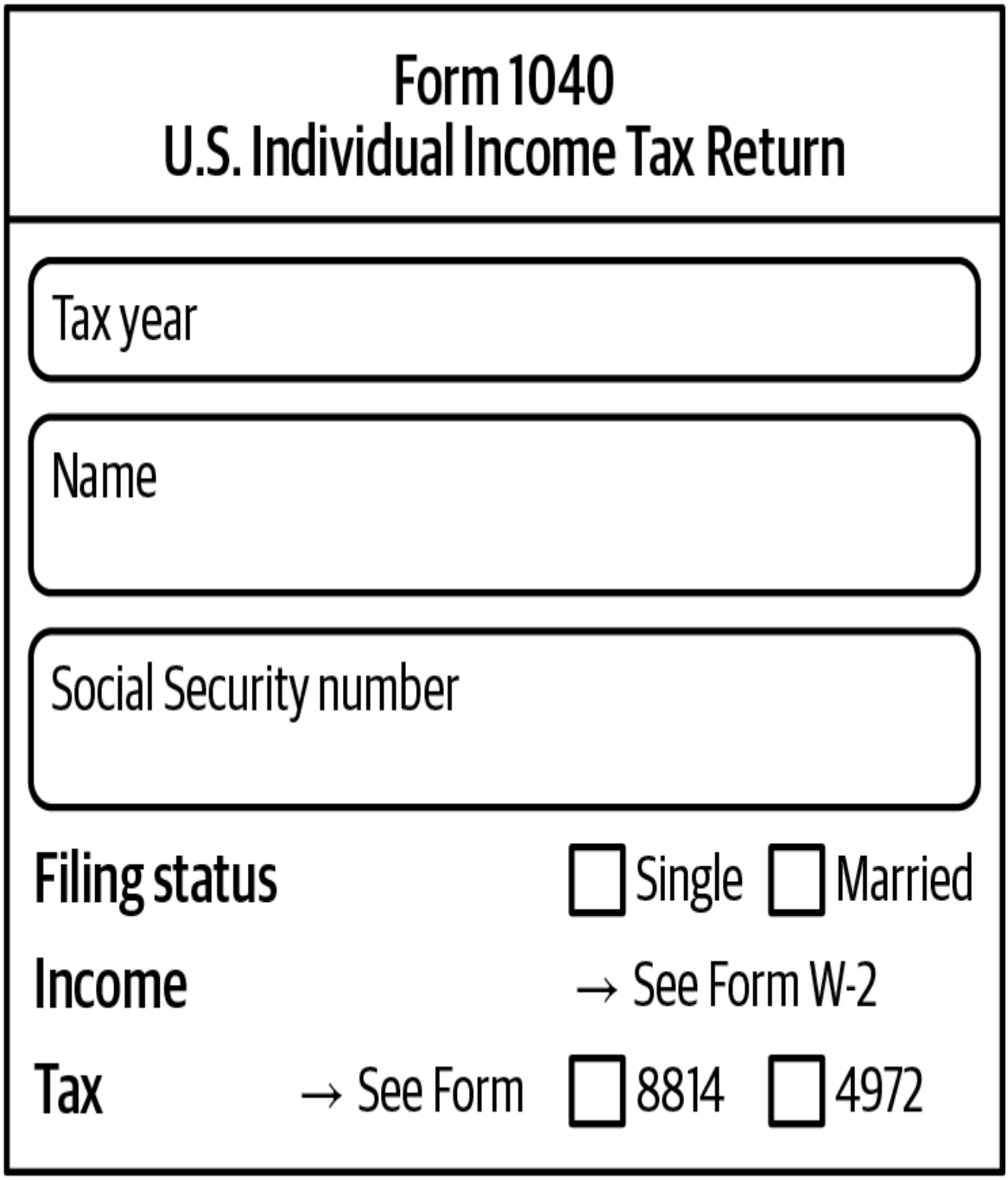


Figure 14-6. Example for a form-based domain: tax declaration.

> **NOTE**
>
> What must be understood about forms is that the forms themselves are already a model of the core concepts and processes of a domain. In a form, certain parts of the core concepts and processes are modeled.

In most CRUD applications, this leads to the fact that the domain behavior has slipped one architectural layer higher into the services, or if it is really bad, into the UI. The actual business logic is thus scattered across all layers.

If you were to clean up this scattering of business logic and, for example, look at which parts of the CRUD data objects are used in which bounded context, and pull the business logic parts from the UI and services down into the CRUD data objects, then a split would also make sense in the case of form-based domain models. However, this splitting is a complete reconstruction of the entire system and must be well weighed against a new implementation.

Dimension 3: Shape of the Domain

The last three topics we think about in the context of domains and bounded contexts are summarized in this section.

Age and Maturity of the Domain

Basically, you should make yourself aware of the age of the domain you are analyzing. Have the business processes and core concepts in this domain already existed for some time or has this domain only recently come into being? Depending on its age, the domain will have reached a different level of maturity in terms of the clarity of the core concepts and the support for business processes already implemented in software.

The domains already mentioned, such as banking, insurance, trading, and container transport, belong to the older domains. In some cases, there are elaborate and publicly available descriptions of the core terms and

processes for these domains. These documents should definitely be consulted during the analysis.

Younger domains are, for example, modern mobility concepts such as car sharing and the linking of different means of transport, the use of digital twins, drone transport, drone flights, social media, and much more.

In the Wardley Maps (see [Wardley 2020] and [Kaiser 2026]) there is an interesting consideration of the development of software. Wardley defines four statuses:

Genesis

A new, as yet unknown domain is being explored.

Custom built

Companies develop their own version.

Product

Software is offered on the market and differs depending on the provider.

Commodity

Highly standardized versions exist.

By placing the software for your domains in a Wardley Map, you can find out a lot about their degree of development.

Tangible Objects Domain Versus Fully Digitalized Domain

Some domains have been completely digitalized in recent years, like the insurance domain, many administrative processes such as tax collection and declaration, or large parts of the banking domain. In these domains, there are no longer any objects in the real world, only data and processes in the digital world.

Many other domains are built on real-world objects, such as containers, cars, goods, leasing, and transportation. For these representational domains, the domain models in the software are always a representation, and often a simulation, of reality, which must be kept parallel to varying degrees. The software supporting such representational domains should contain state machines in the part of the domain model that represents the real objects, which represent the possible changes of the real objects.

CASE STUDY "CONTAINER TERMINAL"

AT THE CONTAINER TERMINAL: TANGIBLE OBJECTS

In the port, the important business concepts can be seen and touched (see Figure 14-7).

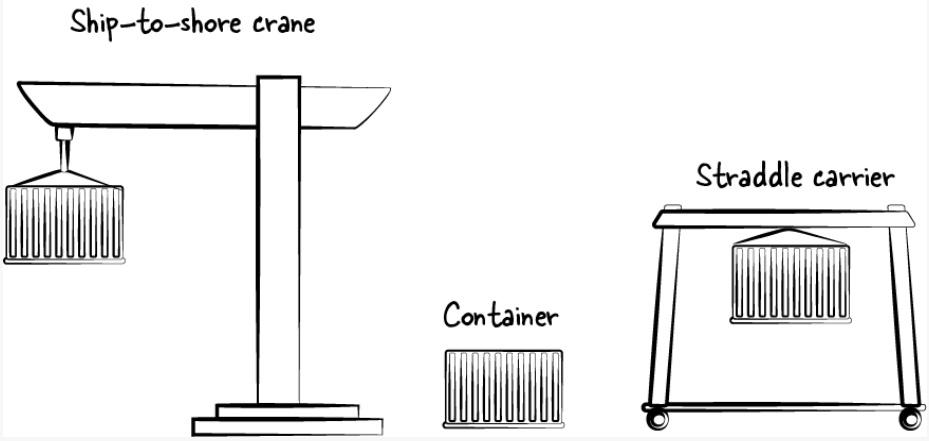


Figure 14-7. Example of a tangible objects domain: the container terminal.

There are also domains that have never existed in the real world, but exist only in the digital, for example, social media and software development tools. Here it is particularly difficult to build a ubiquitous language and a real domain model. It helps here to work a lot with metaphors.

Room for Transformation

Finally, there appear to be different possibilities for transformation in the bounded contexts. Some have a poorly developed understanding of the

domain knowledge in the domain model and can evolve at this point, such as the form-based domain models. Other bounded contexts have a domain model that is well developed in terms of domain expertise, but there is still a lot of transformation to be done at the process level. In the case of retail, where a high degree of organization of the processes and the domain model has been achieved through decades of experience, room for development can generally only be found through new business models.

CAROLA'S FIELD STORY: ROOM FOR TRANSFORMATION IN AN EVENT MANAGEMENT SYSTEM

One of our current company projects involves working on software for event management. The requirements specifications are mostly entity-driven, and complex validation rules are specified for the entities.

The domain model is behavior-rich here, but the processes are not sufficiently supported by the software, and exist "solely" in the minds of the users. Inexperienced users can hardly find their way around the software because it was created with a focus on the individual entities.

Such systems often have a menu consisting of one menu item per entity. The individual menu items per entity contain the activities that can be performed with the individual entities—but the overall context or processes are missing.

This system has a clear potential for evolution at the level of business processes to make it easier for users in the future.

Summary

In this chapter, we have gathered our ideas about different categories of how to consider, classify, and analyze domains and their software implementations.

[1] We thank Olaf Zimmermann for this name idea. We had initially used “types of domains” as the title and also told people at talks that we were not yet happy with it. Olaf then said, “But these are domain patterns.” What we like about it is that you can hear the similarity to design patterns, analysis patterns, and architectural patterns.

[2] Pipeline: here in the sense of an assembly line.

[3] CRUD stands for “Create, Read, Update, Delete”—the standard database operations.

Chapter 15. Summarizing Domain-Driven Transformation

Coming to the end of the book, this chapter distills the method we presented.

Even if legacy systems are in poor condition or lying on the ground as a wreck, there is still a *treasure of domain knowledge* slumbering in them. In most cases, this domain treasure is valuable enough to bring it out of hiding into the light again, instead of replacing it with a redevelopment. This is because with a redevelopment, we are dealing with so many unknown unknowns that the actual effort exceeds the estimates many times over.

Like redevelopment, improving a legacy system is very time-consuming. With Domain-Driven Transformation, this undertaking can be carried out in small steps and thus with less risk. We have shown you how to do that in this book. In order to unearth the treasure of domain knowledge in a legacy system, we have to identify the parts of the source code that contain the valuable business knowledge of our domain. We want to strengthen these parts with domain knowledge, thus freeing them from the solution dust of years of decline, and transfer them, brightly polished, into the new world. Very poetic, isn't it?

We have divided our approach for the transformation of legacy systems into two parts with corresponding subtasks (see Figure 15-1), and in "The Method", we gave you the tools to decide how to proceed in detail:

Technical, tactical, and team-organizational Domain-Driven Transformation

> If your legacy system is a big ball of mud in poor general condition, improve the fundamentals first before moving on to strategic decomposition:

Technical stabilization

Bring your system up to date with libraries and infrastructure for development. With this quick win, you can move on to test strategy, build automation, and increase robustness.

Strengthening the source code with domain knowledge

Lift the treasure of domain knowledge in your software and make your domain model expressive by using domain language and rich domain behavior.

Change organization

Review your team structure. Smaller teams that focus on a single area of domain expertise and take an iterative approach achieve better results when transforming a legacy system.

Strategic Domain-Driven Transformation

If your legacy system is fundamentally stable and structured, you can decompose it into bounded contexts:

Domain rediscovery

Get a clear understanding of the business processes in your domain without considering the current implementation in the source code. In this way, you will find the real bounded contexts in the domain.

Modeling the domain-oriented target architecture

Model the target vision for the domain-oriented architecture of your system on the basis of the business processes.

Alignment of the target and the actual architecture

Align your legacy system with the target vision, determining findings as well as strategic and tactical refactorings.

Prioritization and implementation of the transformation measures

Clarify which strategic refactoring you will tackle first, and begin test-driven implementation.

Visually, the tasks you need to do in Domain-Driven Transformation can be summarized as shown in Figure 15-1.

On paper, Domain-Driven Transformation is easy to follow and sounds logical and simple, but in practice it is usually much more complex. Nevertheless, in most cases it is worth going down this path. That's because big balls of mud are typically "too big to fail." They are so big that a complete rebuild would take years. That is too long to leave the legacy system in the same state. Instead, it will be developed further. So the new system cannot keep up. That's why we prefer the small-step approach described here, in which the legacy system is not replaced by a new system, but rather the legacy system transforms into the new system and evolves.

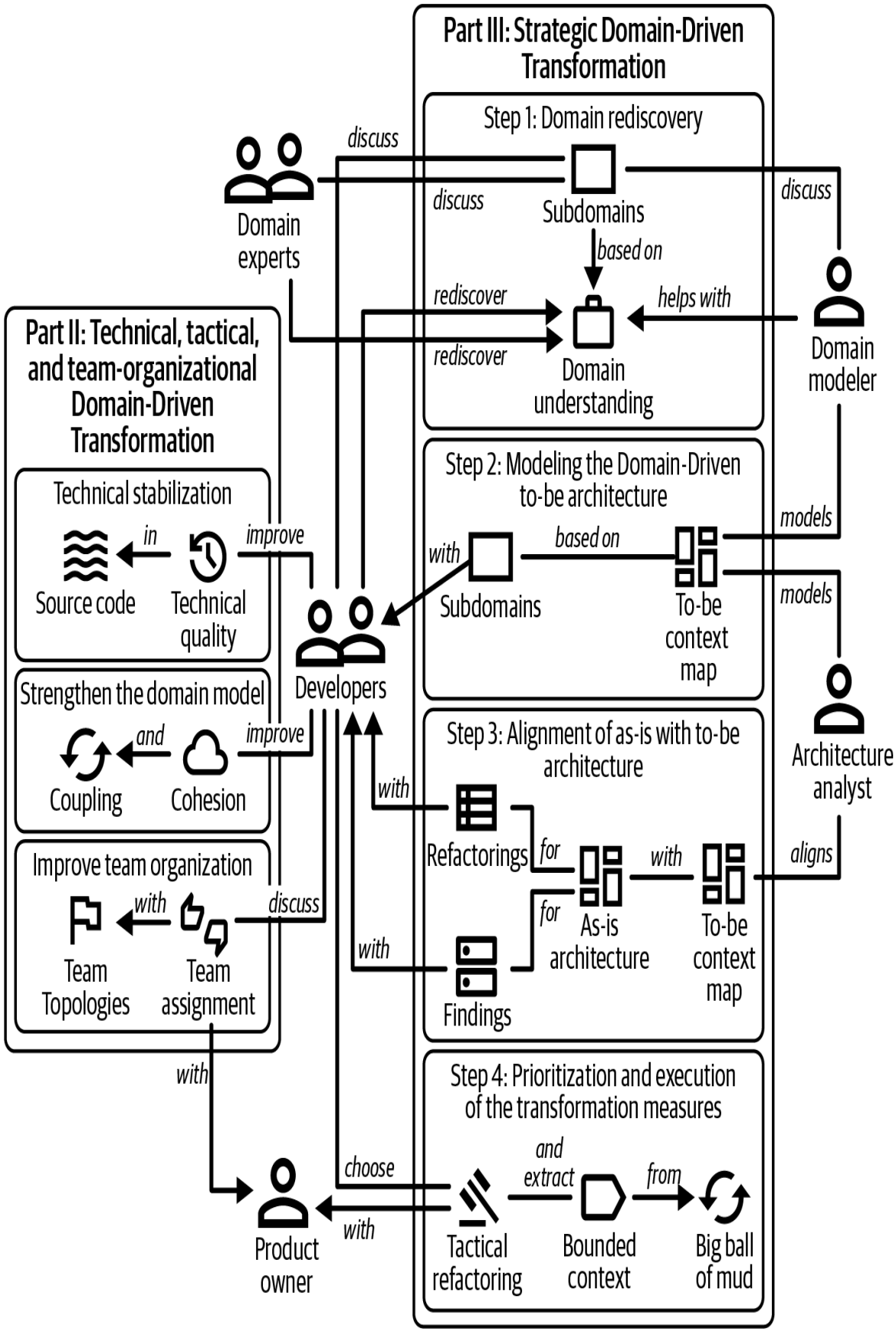

Part III: Strategic Domain-Driven Transformation
Step 1: Domain rediscovery
Domain experts
discuss
discuss
Subdomains
discuss
based on
rediscover
helps with
rediscover
Domain understanding
Domain modeler
Part II: Technical, tactical, and team-organizational Domain-Driven Transformation
Technical stabilization
in
improve
Source code
Technical quality
Step 2: Modeling the Domain-Driven to-be architecture
models
with
based on
Subdomains
To-be context map
models
Developers
Strengthen the domain model
and
improve
Coupling
Cohesion
Architecture analyst
Step 3: Alignment of as-is with to-be architecture
with
Refactorings
for
with
aligns
Improve team organization
with
discuss
for
As-is architecture
To-be context map
Team Topologies
Team assignment
with
Findings
with
Step 4: Prioritization and execution of the transformation measures
and extract
from
choose
with
Product owner
Tactical refactoring
Bounded context
Big ball of mud

Figure 15-1. Technical, tactical, team-organizational, and strategic Domain-Driven Transformation.

This can also work for technology and even programming language changes. For example, old mainframe systems implemented in COBOL are also migrated to a more modern programming language and moved to the cloud. Here, too, migration is done step-by-step, i.e., bounded context by bounded context. The important thing here is that the goal is not necessarily complete migration, but to maintain (or restore) future viability. In most cases, this results in a mixed system in which individual bounded contexts are implemented in the new world and others in the old world. Evolution can only take place step-by-step, with each step representing a viable version of the system.

Once a successful module decomposition has been achieved with Domain-Driven Transformation, it should be possible to assign the extension of a system with new features to precisely one team and its bounded context in each case.

Of course, this is an idealized situation—in practice, new features can lead to reconsidering the module decomposition because the new feature overturns the current domain-oriented decomposition. Or the feature, which is probably a bit too big, has to be broken down into smaller features in such a clever way that different teams can implement their respective part of the big feature independently of the other teams. The respective subfeatures then hopefully make sense on their own and can be delivered independently. The added value of the big feature, when all the teams involved are done, will delight the users.

Now, at the latest, it is time to start the transformation in your own project as well. We wish you success and look forward to your reports!

Appendix A. Overview of the Catalog of Domain-Driven Refactorings

On his website, *https://hschwentner.io/domain-driven-refactorings*, Henning lists the refactorings that help with Domain-Driven Transformation in an ever-growing catalog. For quick reference, we have collected an excerpt from that catalog with the refactorings that are referenced in this book.

The catalog is divided into four categories (and in this book four appendices):

Strategic refactorings (Appendix B)

Help with splitting a monolithic architecture into bounded contexts.

Tactical refactorings that support strategic refactorings (Appendix C)

The smaller steps that the strategic refactorings are built from.

Sociotechnical refactorings (Appendix D)

Reorganize the teams. This is often enabled by and/or accompanies strategic refactorings.

Tactical refactorings that strengthen domain knowledge (Appendix E)

Change the inner implementation of a bounded context.

Appendix B. Strategic Refactorings

For strategic transformation to be successful, you need smaller steps that can be taken—strategic refactorings that describe how the bounded context can be achieved. That is what we present in this appendix.

There are generally two options for a bounded context to come to life besides the big ball of mud:

- Carve out, see *Extract Bounded Context*
- Build new, see *Implement Bounded Context from Scratch*

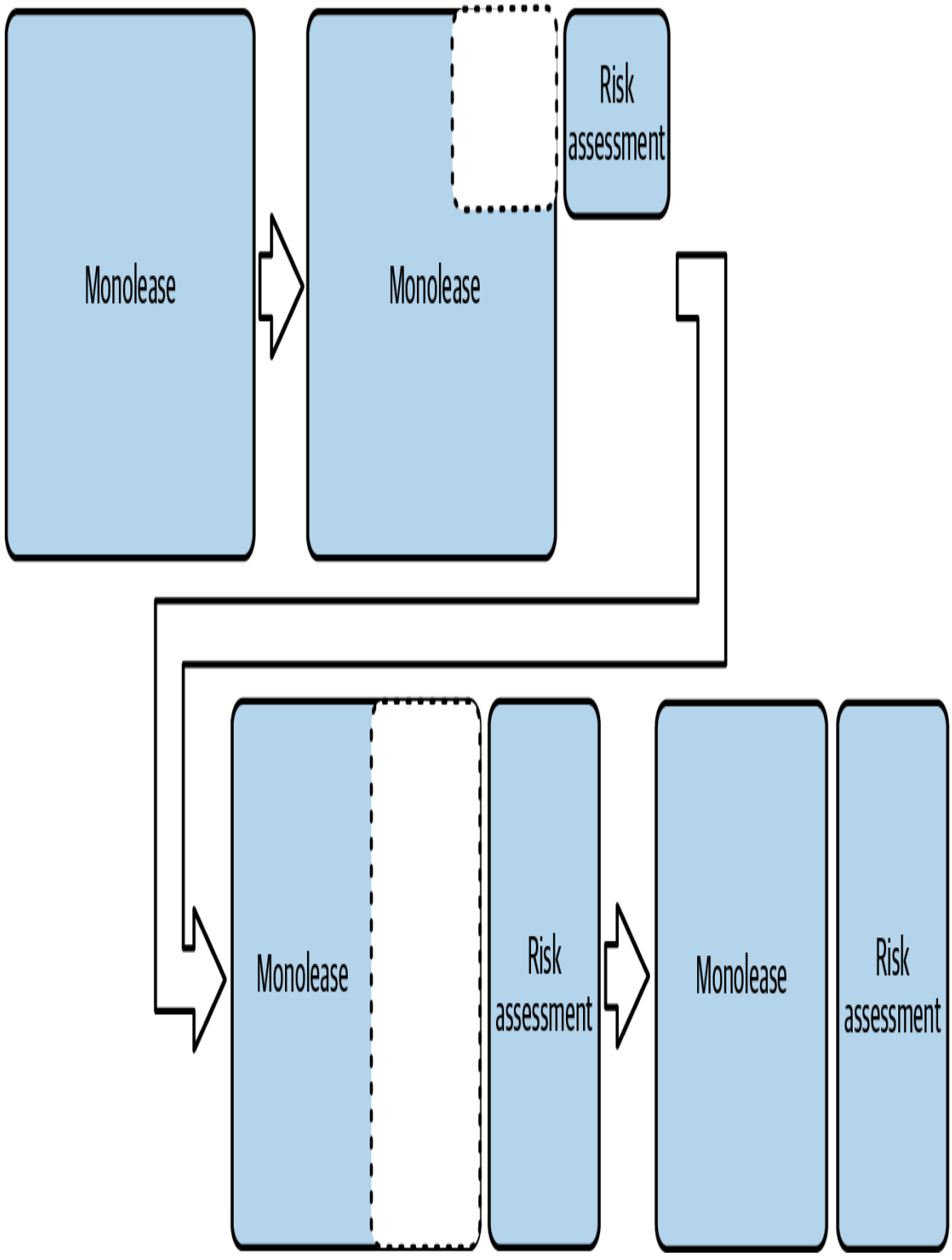
Strategic refactorings are still big and have to be divided into smaller steps—tactical refactorings that support the strategic ones. A strategic refactoring serves as an umbrella to group tactical refactorings that belong together. A strategic refactoring is only done when all the supporting tactical refactorings are done. To find those, see Appendix C.

Strategic refactorings are often accompanied by team reorganization. You can find matching sociotechnical refactorings in Appendix D.

Extract Bounded Context

Also known as: Carve Bounded Context Out of Monolith

Alternative to Implement Bounded Context from Scratch


Monolease
Monolease
Risk assessment
Monolease
Risk assessment
Monolease
Risk assessment

Motivation

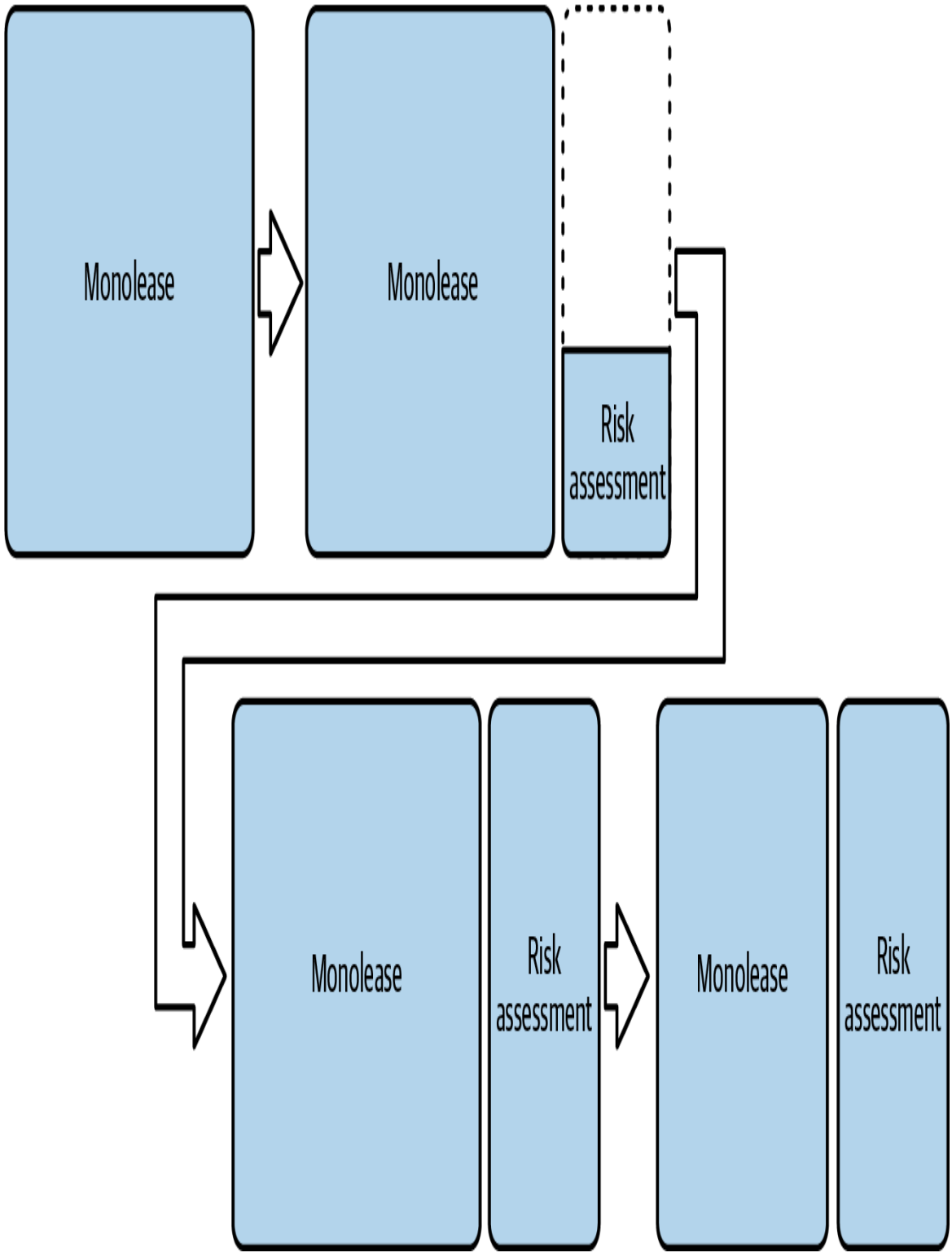
A legacy system is full of domain knowledge that is probably documented nowhere else. Sometimes there isn't even a domain expert who has this knowledge. To bring that knowledge into the modern world, transform it into a new bounded context.

Mechanics

- Analyze the domain to find out which bounded contexts should be there.
- Identify the first carved-out candidate.
- Apply the tactical refactorings for strategic transformation that are right for your situation. Often these are *Extract Specialized Entity*, *Extract Specialized Anemic Entity*, *Extract Specialized Service*, and/or *Extract Specialized Table*.

Implement Bounded Context from Scratch

Alternative to Extract Bounded Context


Monolease
Monolease
Risk assessment
Monolease
Risk assessment
Monolease
Risk assessment

Motivation

When the idealized context map has been drawn and we know which bounded contexts we want to implement, for every bounded context, we can either carve it out or build it from scratch.

Implementing from scratch is the only possibility when the modernization is also moving to another language. This often happens, for example, when a project moves from COBOL to Java or C#.

Building a bounded context from scratch risks losing domain knowledge that is hidden in the monolith. Also, such a bounded context may suffer from the *second-system effect* (see [Brooks 1995, chap. 5]).

Mechanics

- Start with the idealized context map to know which bounded context and which relationships to other bounded contexts (and the monolith) must be built.
- Choose the bounded context to carve out and assign a team with the fitting sociotechnical refactoring, e.g., *Form Cross-Functional Team out of Layer-Team Members*, *Form Second Team out of Only Layer-Team Members*, or *Form Second Cross-Functional Team out of Partly Layer-Team and First-Team Members*.
- Implement a first prototype.
- Build the outer interface in a way that is right for other bounded contexts that are in place.
- Add a connection layer to the monolith that behaves like the interface of the other (not yet carved out) bounded contexts would.

Appendix C. Tactical Refactorings That Support Strategic Refactorings

Strategic refactorings are still big and have to be divided into smaller steps. In this appendix we present these smaller steps—tactical refactorings that support the strategic ones. A supporting tactical refactoring is always carried out as part of a strategic refactoring.

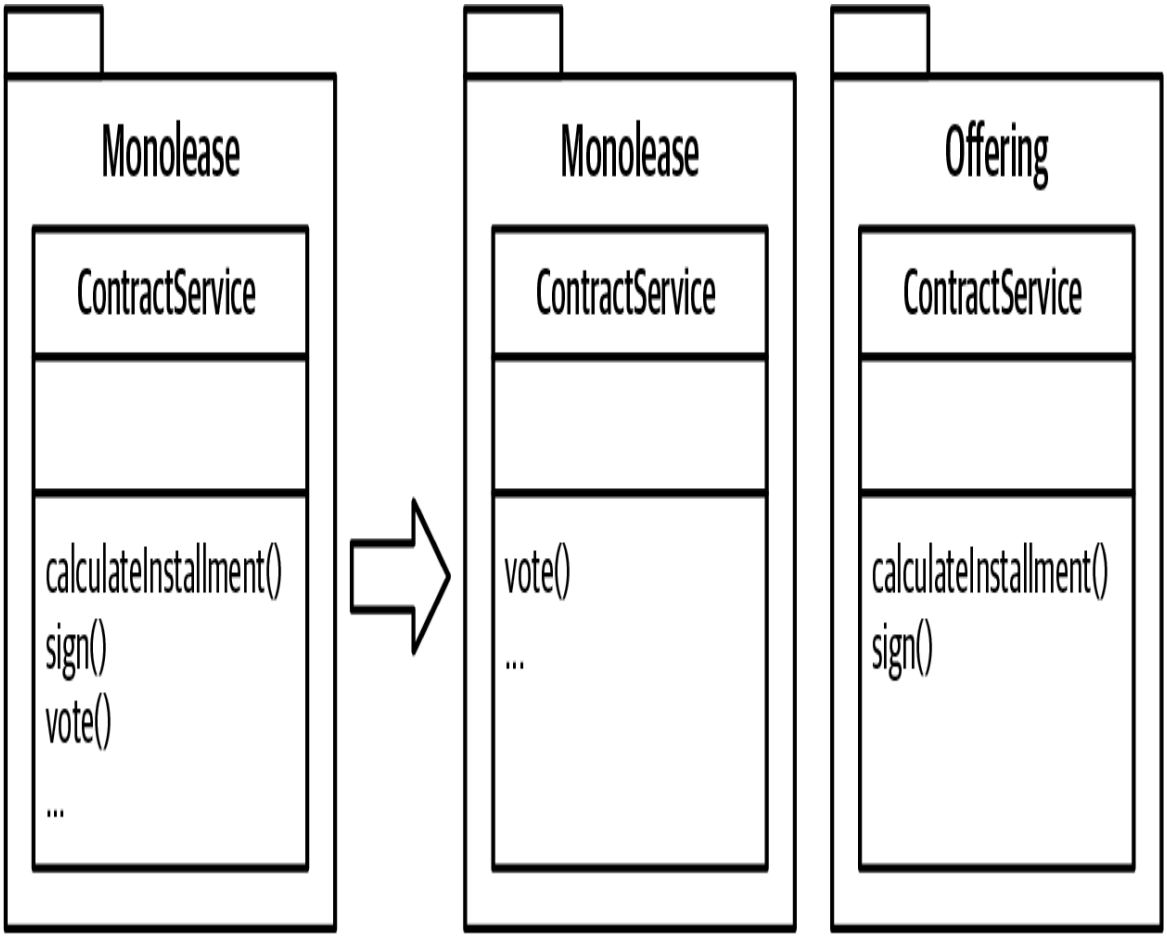
When extracting a bounded context, you can encounter different situations:

- An anemic domain model. Then *Extract Specialized Anemic Entity* and *Extract Specialized Service*, can help you.
- A domain model with rich behavior. This can be split with *Extract Specialized Entity* and with *Extract Specialized Service*.
- A monolithic data model. Tackle it with *Extract Specialized Table*.

Extract Specialized Service

Also known as: Carve Specialized Anemic Service Out of Monolithic Service

A special case of Extract Class *[Fowler 2019, p. 182]*



Motivation

As one step of *Extract Bounded Context*, you've found a domain service that has become too big. Its functionality belongs partly in the bounded context that is extracted. The decision has been made to split it.

Mechanics

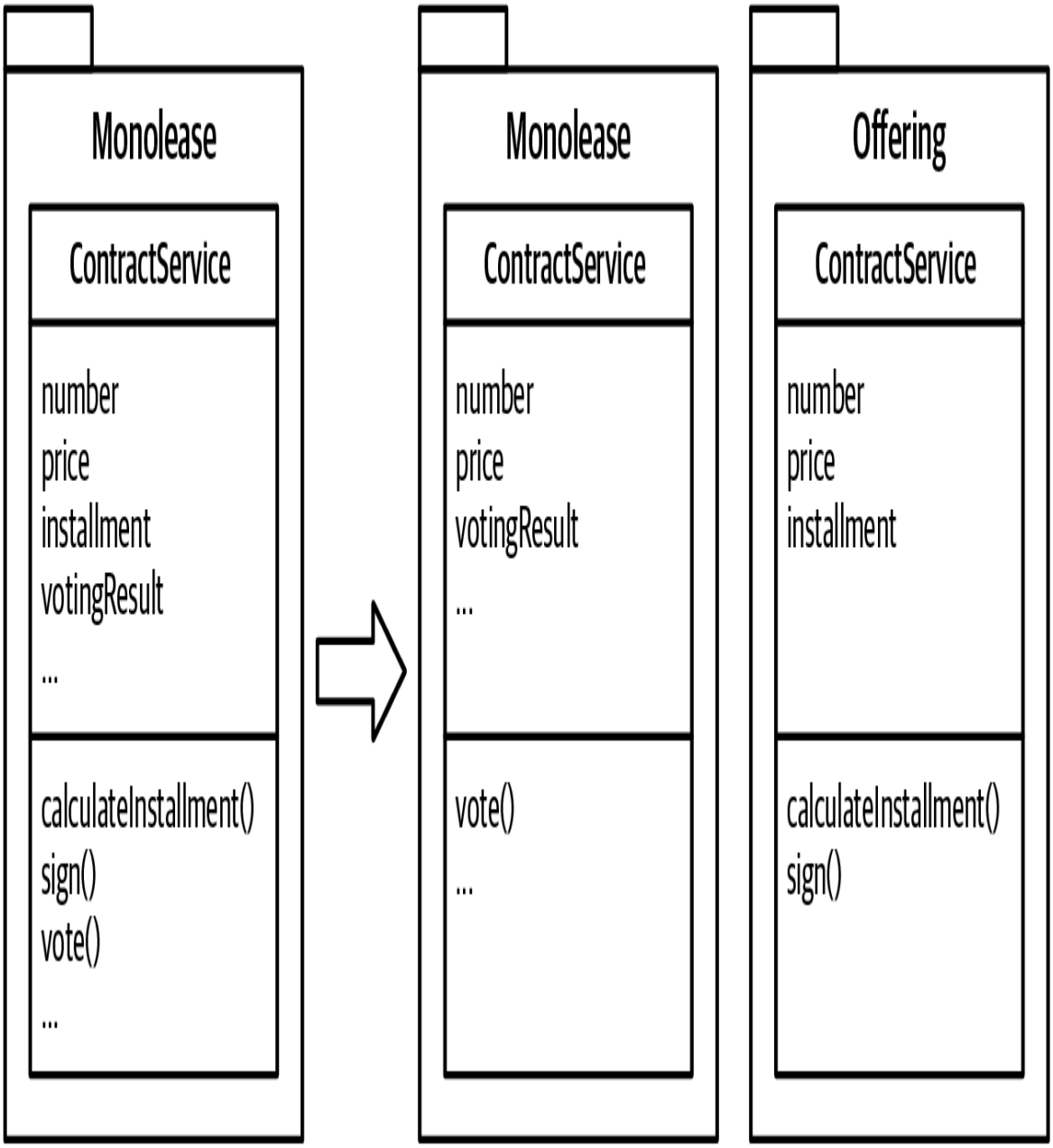
- Create an empty new class in the carved-out context.
- Add an instance field of type new class to the old class.
- Copy the first to-be-moved method to the new class with *Move Method* [Fowler 2019, p. 198].

- Replace the method body in the old class with a forward to the method in the new class.
- Step-by-step replace calls to the method in the old class with calls to the method in the new class.
- Delete the implementation in the old class.
- Repeat with other to-be-moved methods.
- Remove the instance field of type new class from the old class.

Extract Specialized Entity

Also known as: Carve Specialized Entity Out of Monolithic Entity

A special case of Extract Class *[Fowler 2019, p. 182]*



Motivation

As one step of *Extract Bounded Context*, you've found a monolithic domain model. Here you have identified an entity that has become too big. The decision has been made to split it.

If you have an anemic domain model instead of a behavioral rich domain model, apply *Extract Specialized Anemic Entity*.

This is often accompanied by *Extract Specialized Table*.

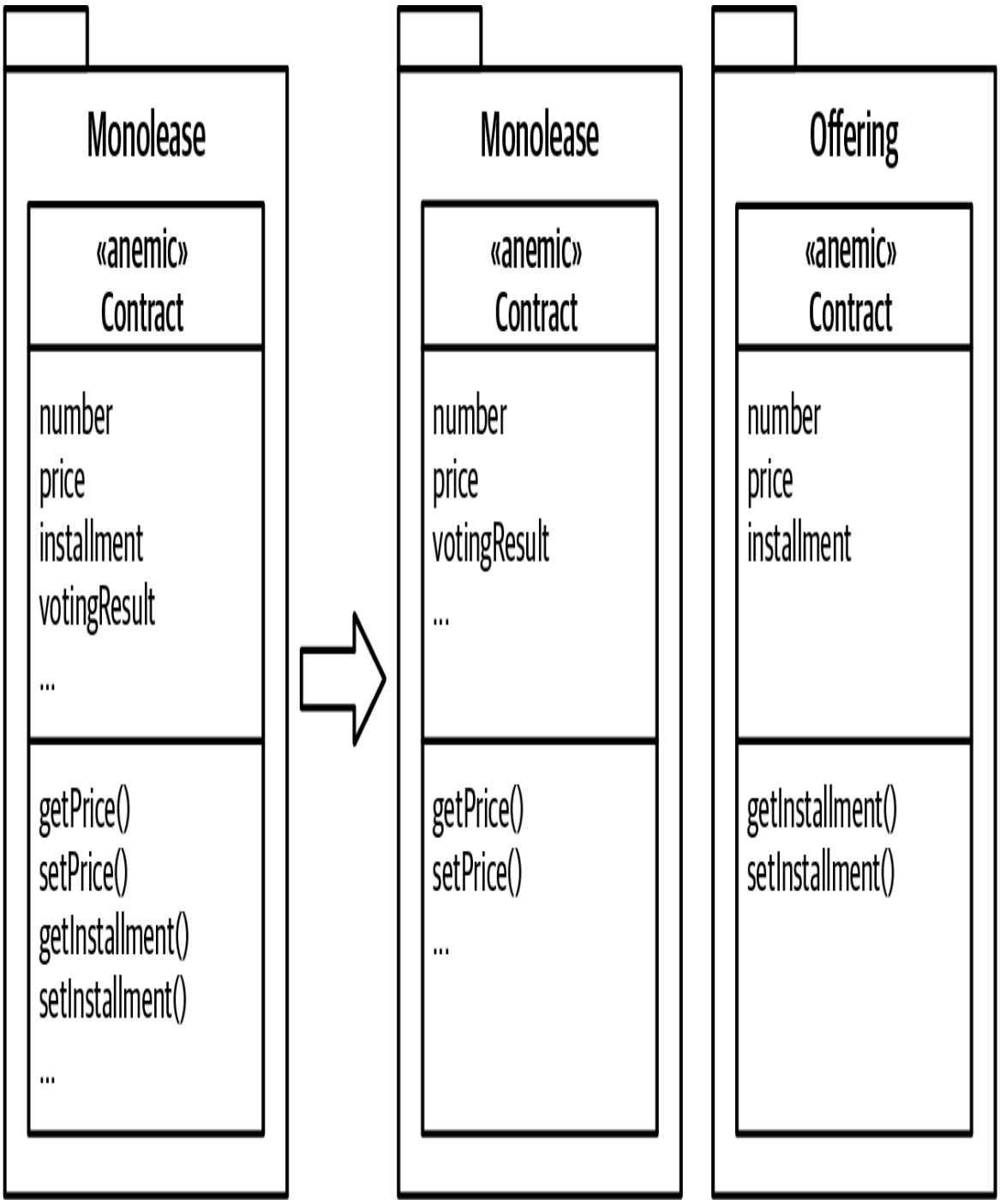
Mechanics

- Create an empty new class in the carved-out context.
- Add an instance field of type new class to the old class.
- Copy the to-be-moved fields from the old to the new class with *Move Field* [Fowler 2019, p. 207].
- Copy the first to-be-moved method to the new class with *Move Method* [Fowler 2019, p. 198].
- Replace the method body in the old class with a forward to the method in the new class.
- Step-by-step replace calls to the method in the old class with calls to the method in the new class.
- Delete the implementation from the old class.
- Delete the now unused fields from the old class.
- Repeat with other to-be-moved methods.
- Remove the instance field of type new class from the old class.

Extract Specialized Anemic Entity

Also known as: Carve Specialized Anemic Entity Out of Monolithic Entity

A special case of Extract Class *[Fowler 2019, p. 182]*



Motivation

As one step of *Extract Bounded Context*, you've found a monolithic anemic domain model. Here you have identified an anemic entity (a "data bag") that has become too big. The decision has been made to split it.

If you have a behavioral rich domain model instead of an anemic domain model, apply *Extract Specialized Entity*.

This is often a follow up to *Extract Specialized Service* and accompanied by *Extract Specialized Table*. After the extraction, it is often a good idea to *Heal Entity Anemia*.

Mechanics

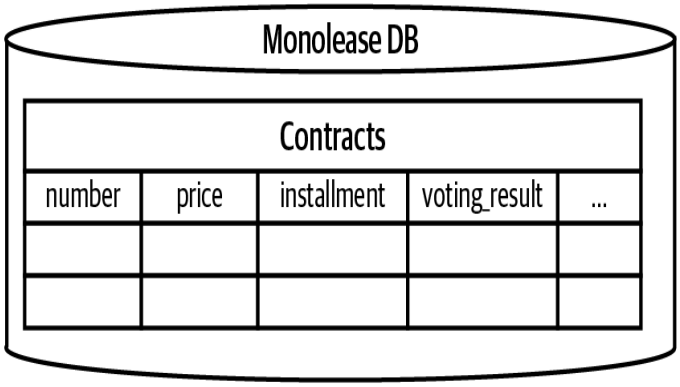
- Create an empty new class in the carved-out context.
- Add an instance field of type new class to the old class.
- Copy the to-be-moved fields from the old to the new class with *Move Field* [Fowler 2019, p. 207].
- Copy the first to-be-moved method to the new class with *Move Method* [Fowler 2019, p. 198].
- Replace the method body in the old class with a forward to the method in the new class.
- Step-by-step replace calls to the method in the old class with calls to the method in the new class.
- Delete the implementation from the old class.
- Delete the now unused fields from old class.
- Repeat with the other to-be-moved methods.
- Remove the instance field of type new class from the old class.

Extract Specialized Table

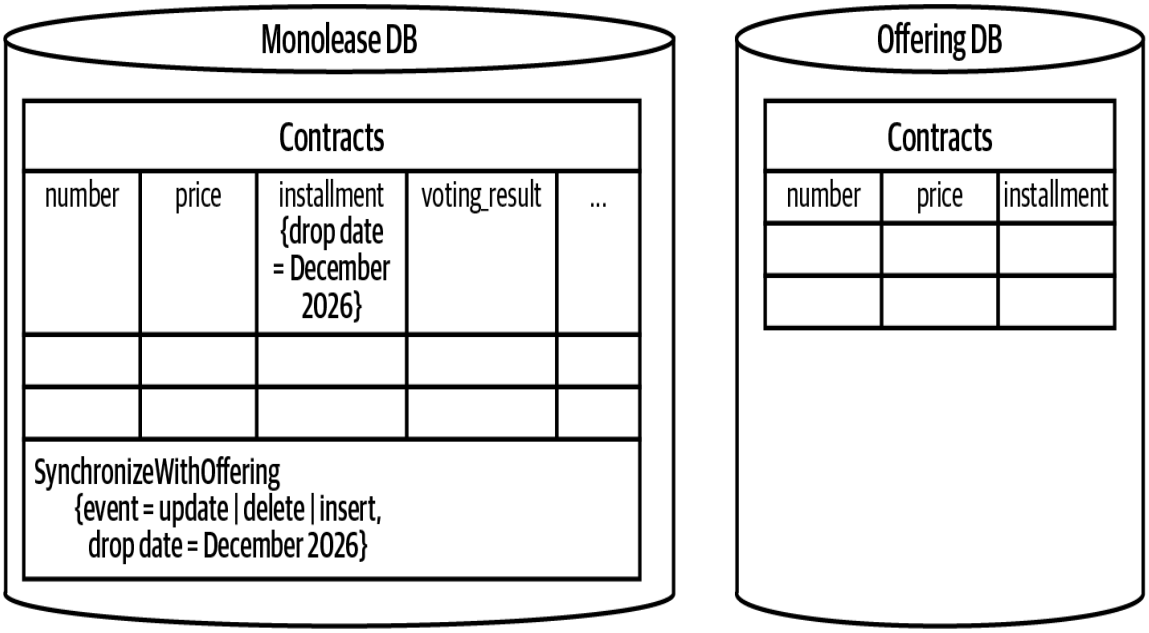
Type: Database refactoring

A special case of Split Table *[Ambler and Sadalage 2006, p. 145]*

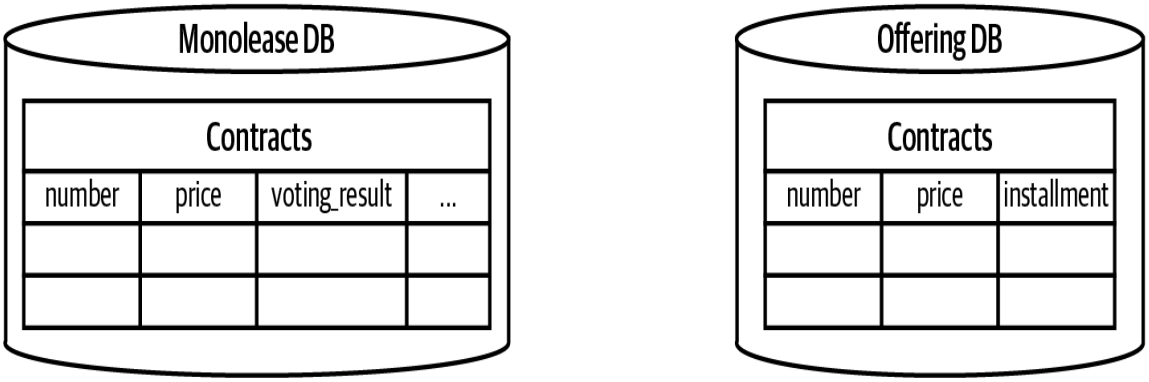
Original schema



Transition period



Resulting schema



Motivation

As one step of *Extract Bounded Context*, you've found a monolithic data model (often used to store a monolithic domain model). In this data model you have identified a table that has become too big. The decision has been made to separate the parts for the newly extracted bounded context.

Schema Update Mechanics

- Create an empty new table in the database scheme of the carved-out context.
- Copy the to-be-duplicated columns from the old to the new table.
- Copy the to-be-moved columns from the old to the new table by repeatedly applying *Move Column* [Ambler and Sadalage 2006, p. 103].
- Introduce synchronizing triggers on both the monolithic and the carved-out table. They must be invoked by any change to the data in the tables. Beware of the triggers triggering each other.
- Add a drop date to the triggers.

Data-Migration Mechanics

- Copy the data from the duplicated columns to the new table. Potentially not all rows existing in the monolith should also be present in the carved-out bounded context.
- Copy the data from the moved columns to the new table with *Move Data* [Ambler and Sadalage 2006, p. 192].

Access Program Mechanics

- Analyze the accessing program code thoroughly and update it accordingly during the transition period.
- If applicable, extract a specialized repository for the carved-out bounded context.

Appendix D. Sociotechnical Refactorings

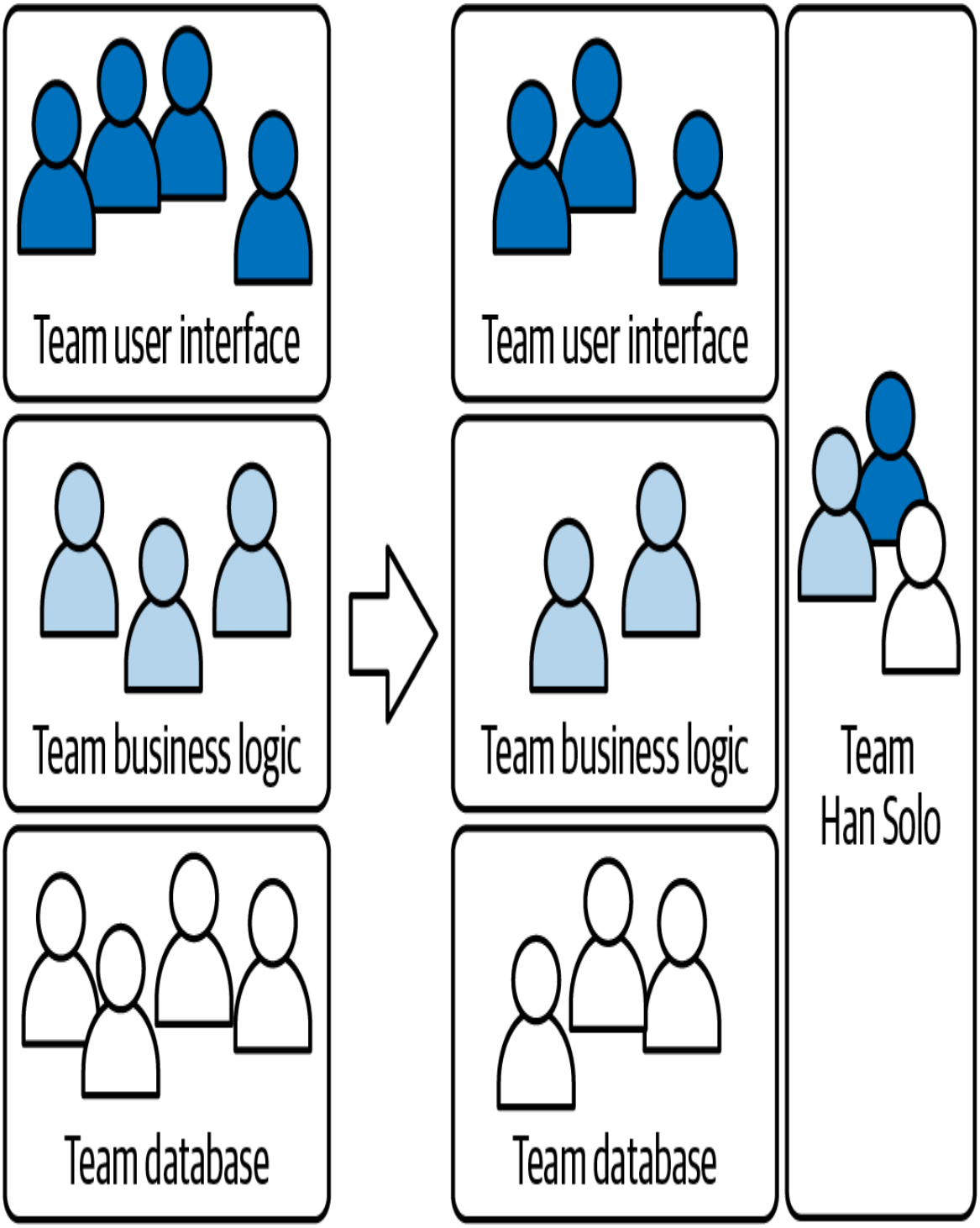
Since software and its architecture are heavily influenced by the organization of the people that build it, a piece of software is nowadays viewed not as a merely technical system, but as a sociotechnical system. This implies that when you transform such a system, you typically have to also transform the team organization. This appendix contains the refactorings for that reorganization. “Refactoring” is used here in a broad sense, since we are changing the system only indirectly.

The refactorings help with changing an organization that has teams organized according to technological layers. You start with *Form Cross-Functional Team out of Layer-Team Members* and then decide to use either of the following:

- *Form Second Cross-Functional Team out of Partly Layer-Team and First-Team Members*
- *Form Second Team out of Only Layer-Team Members*

Sociotechnical refactorings are typically companions with strategic refactorings.

Form Cross-Functional Team Out of Layer-Team Members



Motivation

When the teams in your projects are organized following technological layers, Conway's Law (see [Conway 1968]) will inevitably bite you. Even

the smallest change to your system will usually involve more than one team. The addition of one simple field, for example, will cause work for the UI team, business logic team, and database team alike.

When you apply *Extract Bounded Context* or *Implement Bounded Context from Scratch*, the question arises, "Which team will be responsible for the newly created bounded context?" None of the layer teams is fit for this purpose, because the new bounded context is a *vertical* and has UI, business logic, and database parts. On the other hand, we can't assign the bounded context to all the teams, because one of the rules is that it *should not belong to more than one team*.

The solution here is to form a new team that has all the required knowledge —a *cross-functional* team.

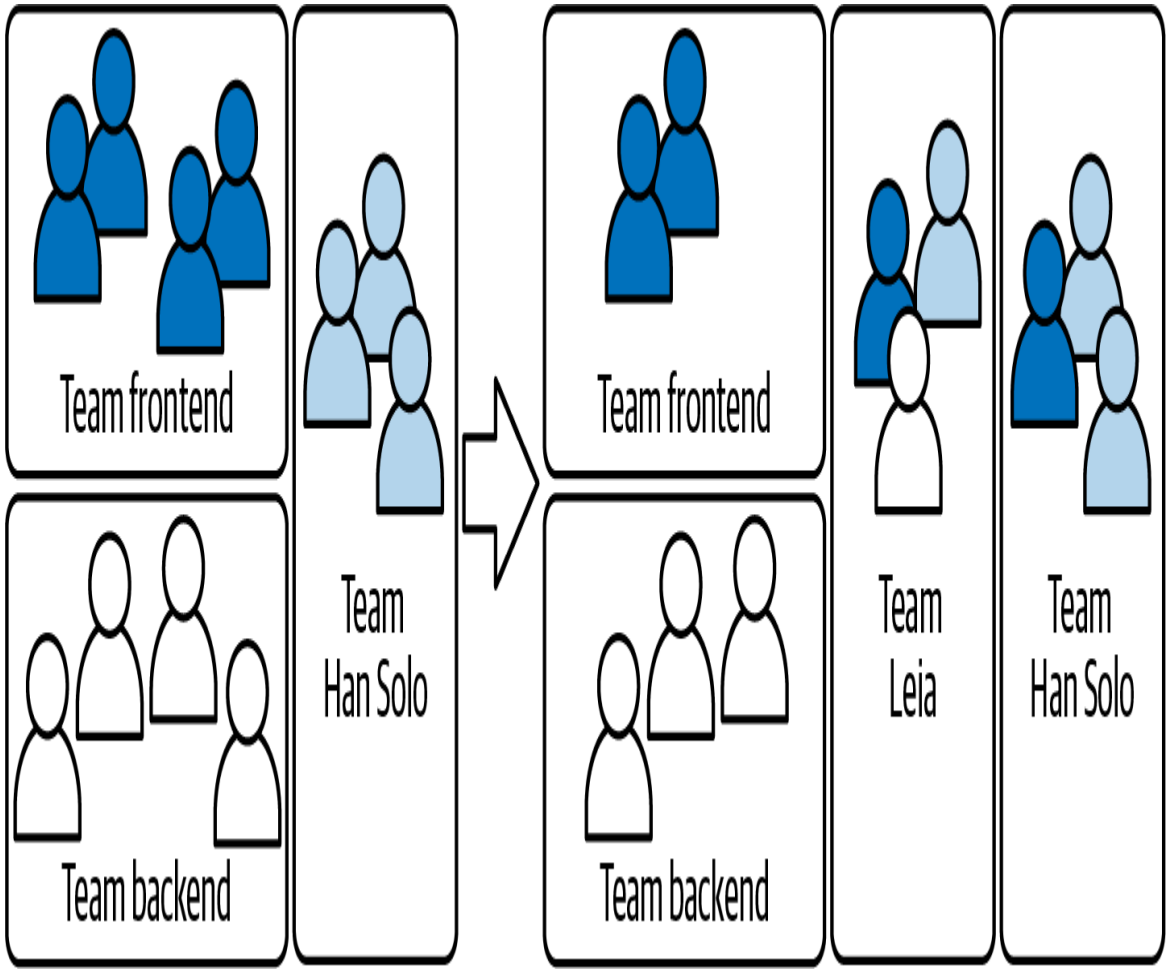
This is an application of the Isolation pattern from *Dynamic Reteaming* (see [Helfland 2020, chap. 7]).

Mechanics

- Choose (at least) one person from every layer team.
- You'll want the early adopters here, not the skeptics (see [Manns and Rising 2015]).
- Form a new team of fewer than nine people.
- Let the new team take responsibility for the new bounded context.
- In the new team, encourage the development of a t-shaped skill profile and other Agile techniques to break knowledge silos.
- For the time being, let the rest of the teams remain organized as they are. The monolith is still there and has to be maintained.

Form Second Cross-Functional Team Out of Partly Layer-Team and First-Team Members

Alternative to Form Second Team Out of Only Layer-Team Members



Motivation

After applying *Form Cross-Functional Team out of Layer-Team Members*, a first cross-functional team has been ramped up. To form the next teams, two different approaches can be taken: either take someone out of the first new team to be part of the second new team, or leave the first team as is and fill the new team with layer team members.

The advantage of sending a member of the first cross-functional team to the second team is that the knowledge that was gained by doing the change the first time can thus be transferred to the second team.

The disadvantage is that we have to rip apart the relatively newly formed first team and send it again into a storming-forming-norming-performing

cycle (see [Tuckman 1965]).

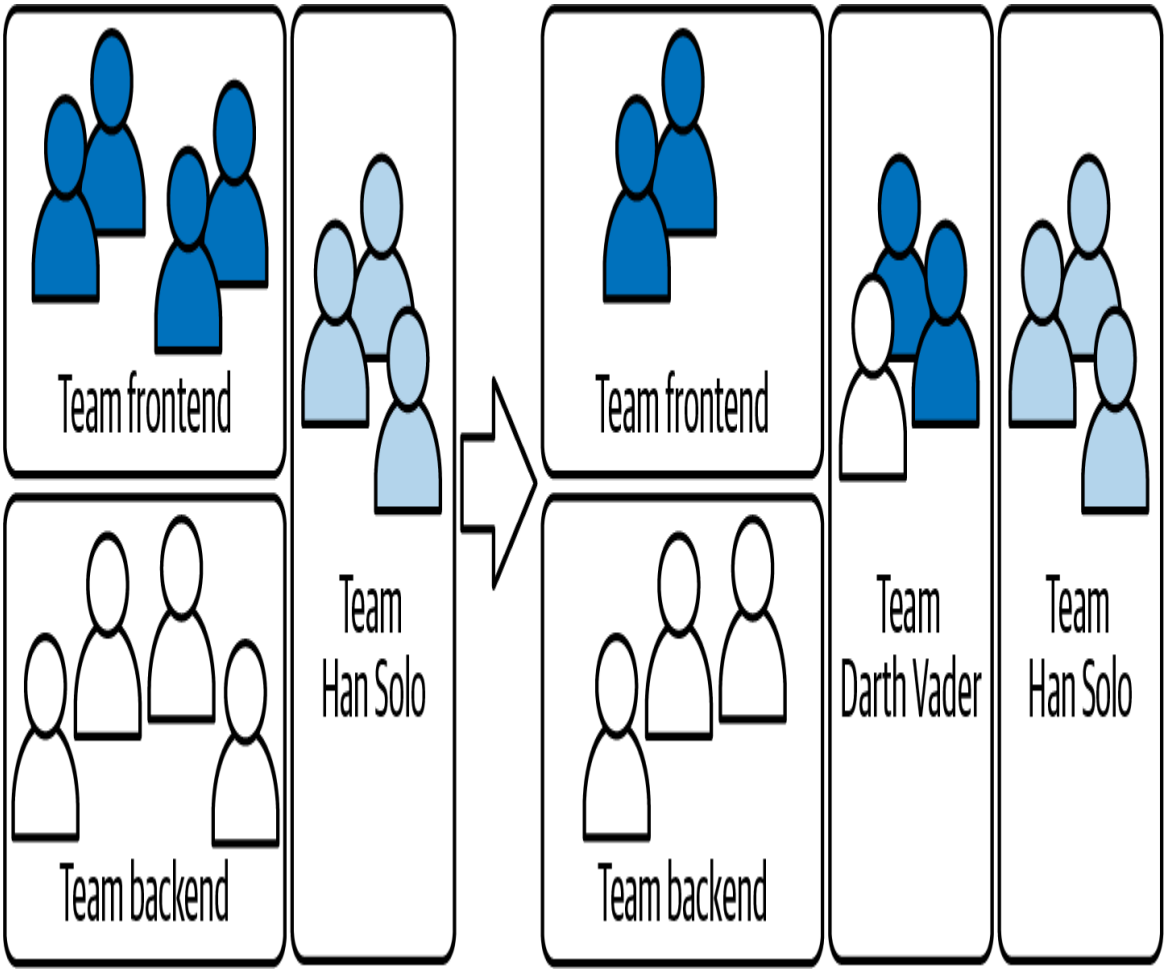
There is a connection to both the Grow-and-Split pattern and the Merging pattern from *Dynamic Reteaming* here (see [Helfland 2020, chap. 6 and 8]).

Mechanics

- Choose (at least) one person from the first cross-functional team.
- You'll want the "teachers" here—the people who like to pass on knowledge to peers.
- Fill up the second cross-functional team with members from the layer teams.
    - Check that all the required knowledge is present in the team. For example, if no UI expert has come from the first team, send someone from the old UI layer team.
    - Form a new team of fewer than nine people.
- Let the new team take responsibility for a new bounded context.
- Let the rest of the teams remain organized as they are. They should already be shrinking in a significant way.

Form Second Team Out of Only Layer-Team Members

Alternative to Form Second Cross-Functional Team Out of Partly Layer-Team and First-Team Members



Motivation

After applying *Form Cross-Functional Team out of Layer-Team Members*, a first cross-functional team has been ramped up. To form the next teams, two different approaches can be taken: either take someone out of the first new team to be part of the second new team, or leave the first team as it is and fill the new team with layer team members.

Forming follow-up cross-functional teams out of only layer teams has the following advantages: the existing cross-functional teams can work undisturbed, and several new cross-functional teams can be formed at once.

The disadvantage is that every new team has to learn everything about the team change from scratch, since the team members that have already experienced a team reorganization stay in their team.

Mechanics

- Follow the mechanics from *Form Cross-Functional Team out of Layer-Team Members*.

Appendix E. Tactical Refactorings That Strengthen Domain Knowledge

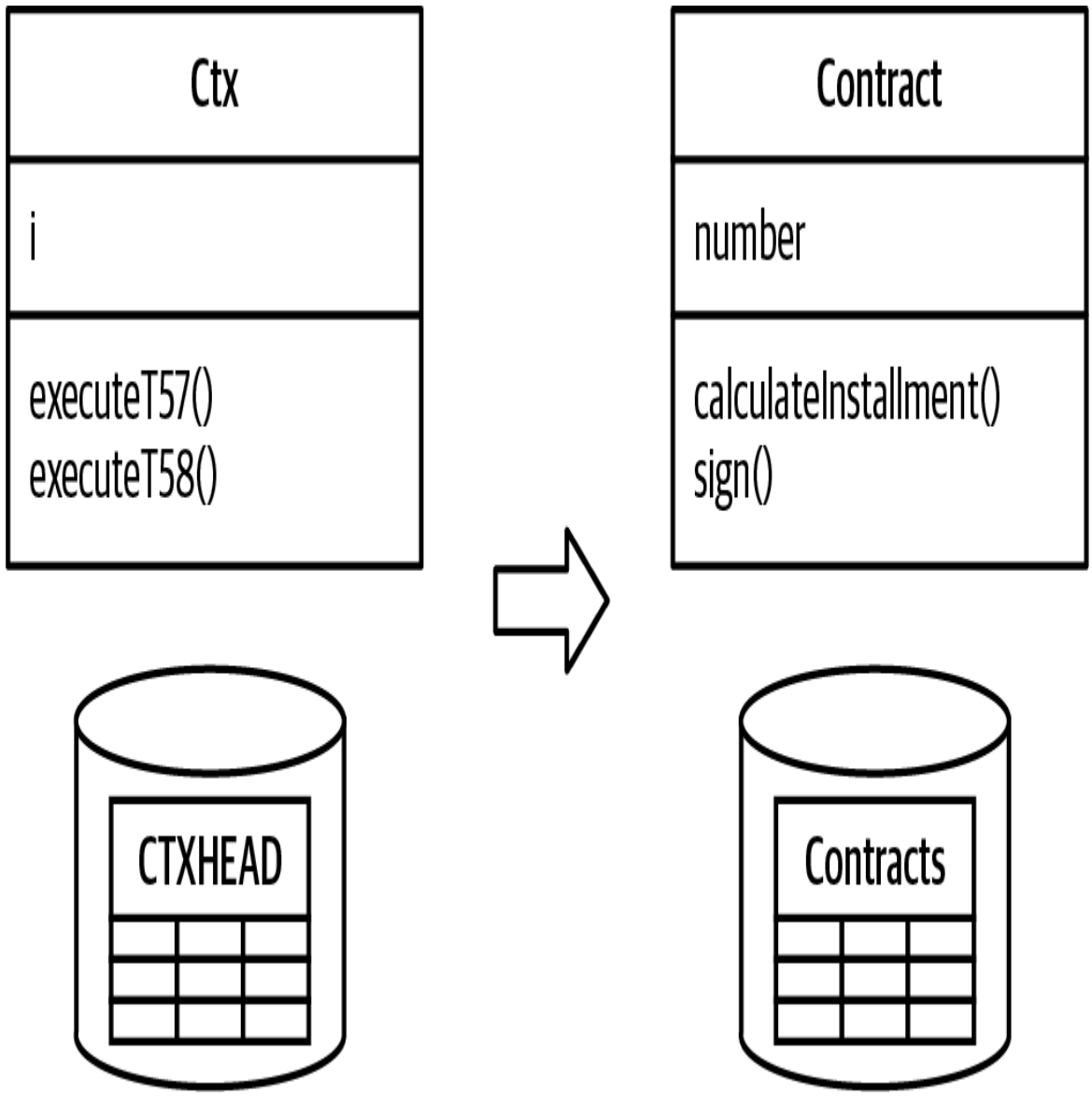
Code that makes it hard to understand for what purpose it was built also makes it hard to maintain. Therefore we want our code to clearly express the concepts of the domain it is built for. The refactorings in this appendix are meant to help you transform muddy code into crisply expressed concepts that make it easy for the reading programmer to understand and change them.

To achieve this beautiful and behavior rich domain model, different actions can be taken:

- Bring domain language into the code with *Enforce Ubiquitous Language*.
- Make the domain model more expressive through *Replace Primitive with Value Object*, *Heal Entity Anemia*, and *Introduce Contract*.
- Separate business logic from technical code with *Split Active Record into Aggregate and Repository* and *Split Repository into Interface and Implementation* to separate the database access, or *Extract Entity from Smart UI* and *Extract Service from Smart UI* to separate the user interface code.

These refactorings usually take place before or after tactical refactorings to support the strategic refactorings that are carried out.

Enforce Ubiquitous Language



Motivation

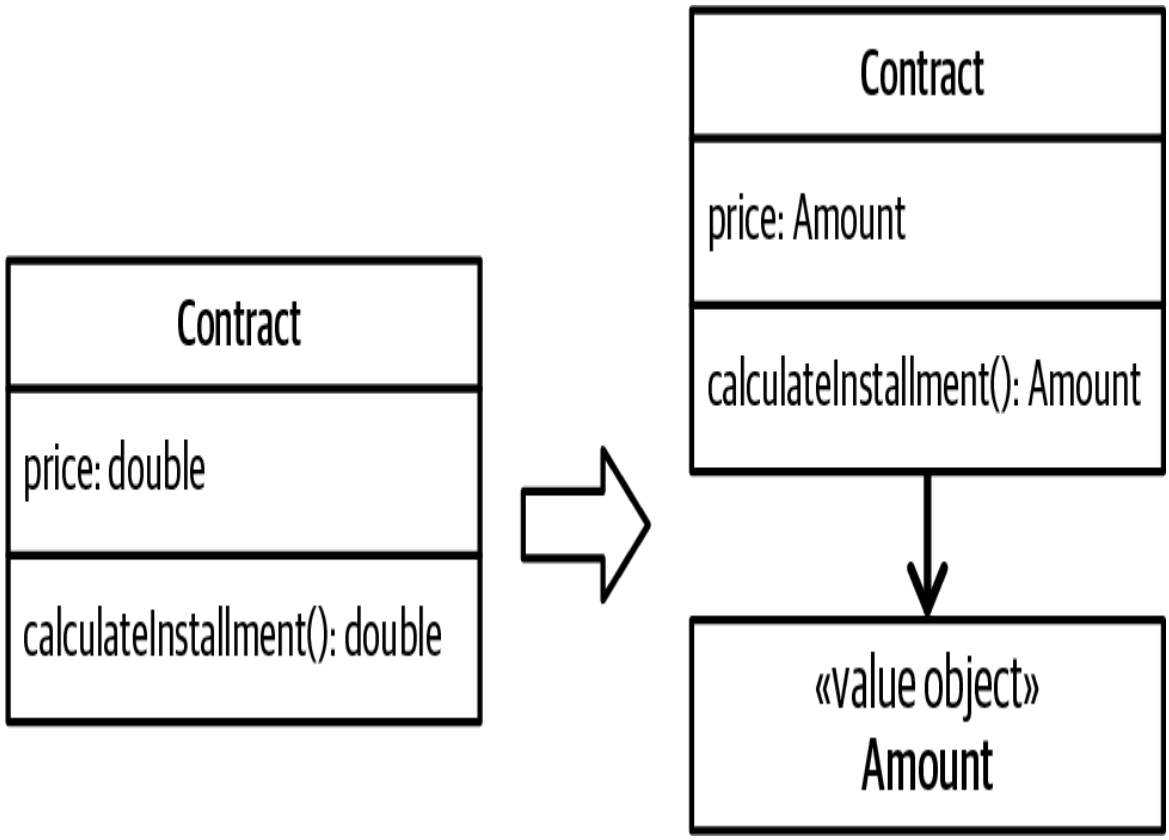
Whenever the code uses bad names, obscure technical names, strange abbreviations, or names that do not reflect your current understanding of the domain, change them accordingly. This is part of DDD's philosophy of "Refactoring Toward Deeper Insight."

Mechanics

- When the code uses the wrong words, apply *Rename Class*, *Rename Method* [Fowler 2019, p. 124], *Rename Field* [Fowler 2019, p. 244], *Rename Variable* [Fowler 2019, p. 137], and the like to the source code.
- For the data model use *Rename Table* [Ambler and Sadalage 2006, p. 113], *Rename Column* [Ambler and Sadalage 2006, p. 109], and *Rename View* [Ambler and Sadalage 2006, p. 117].
- Change the user interface accordingly.

Replace Primitive with Value Object

A special case of Replace Primitive with Object *[Fowler 2019, p. 174]*



Motivation

Many systems excessively use primitive types to express value-like domain concepts. Everywhere you will find naked `ints` and `Strings`. (Martin Fowler calls this bad smell Primitive Obsession.) A better way is to define types in their own right for these domain concepts, and thus make the code both more expressive and less error-prone.

Mechanics

- Create a simple value object class.
- Make sure the class is both identity-less and immutable.
- Stepwise use the new value object type where a primitive type was used.
- Where applicable, move domain logic (typical cases are validation and arithmetic) into the value object type with *Extract Method* [Fowler 2019, p. 106], *Move Method* [Fowler 2019, p. 198], and the like.

Heal Entity Anemia

Umbrella refactoring for Replace Setter, Remove Setter, *and* Move Logic from Service to Entity

Motivation

The anemic domain model is an antipattern that separates domain logic and domain data. In it, we typically find services and "data bags." A "data bag" or anemic entity is a class that represents a domain concept and models all its data, which it makes accessible via getters and setters. All the domain logic is placed in the services.

In such an architecture the anemic entities cannot ensure their consistency since they don't encapsulate their data but expose it.

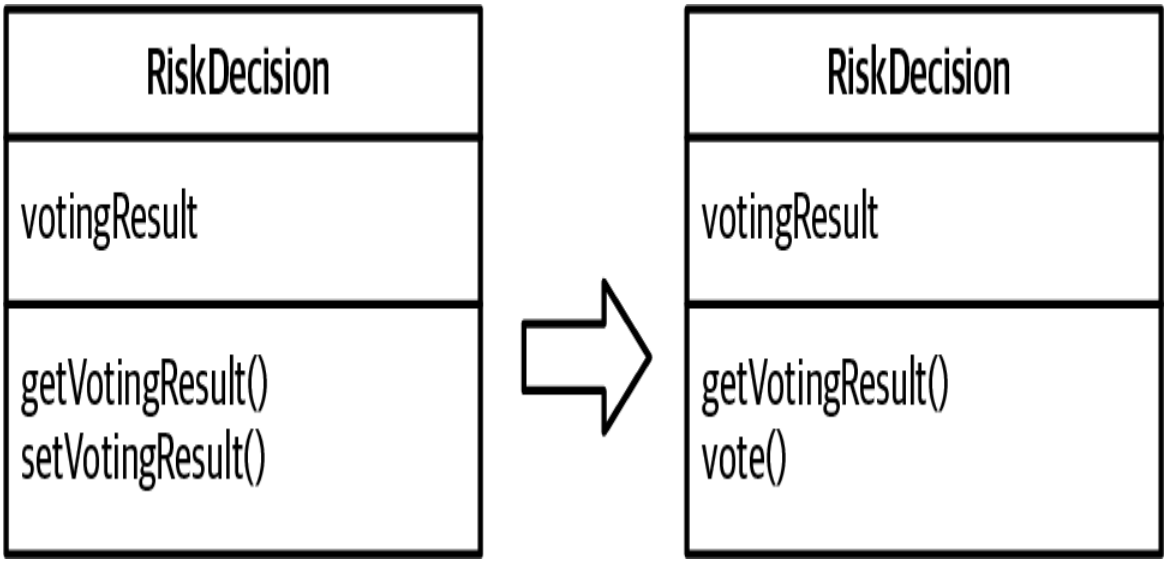
A group of smaller refactorings helps to get rid of this model anemia and enriches the entities with behavior.

Depending on what you do first—strategic transformation or tactical transformation—this is often a follow-up to *Extract Specialized Anemic Entity* or followed by *Extract Specialized Entity*.

Mechanics

- Encapsulate data in the anemic entities with *Replace Setter* or *Remove Setter*.
- Bring behavior to the entities with *Move Logic from Service to Entity*.

Replace Setter



Motivation

As one of the first lessons in programming, we have learned that it's a good thing to protect variables and make them as private as possible. (This is a direct application of Parnas' *information hiding*; see [Parnas 1972].)

Instance variables or fields in classes are therefore, by default, declared `private`.

> **WARNING**
>
> Unfortunately, many programmers have also learned that the next thing is to write a getter and setter for just that nicely encapsulated variable, thus breaking the encapsulation. Just as unfortunately, modern IDEs support this bad behavior with automatic generation of these accessor methods. If you're on the JVM, the Java Beans convention also enforces it.

While the getter might be OK, the setter yields serious problems. A better (and more domain-driven) approach is to build our classes with methods that resemble domain behavior and have names that express this behavior. The verbs in the method names should come from the domain, and *set* is a verb that is just not used in most domains.

If the setter is never used and is just there because it has been generated automatically, apply *Remove Setter* directly.

There are some similarities to *Rename Method* [Fowler 2019, p. 124] here. A difference is that often the new method does not just set the field but may add some other business behavior as well.

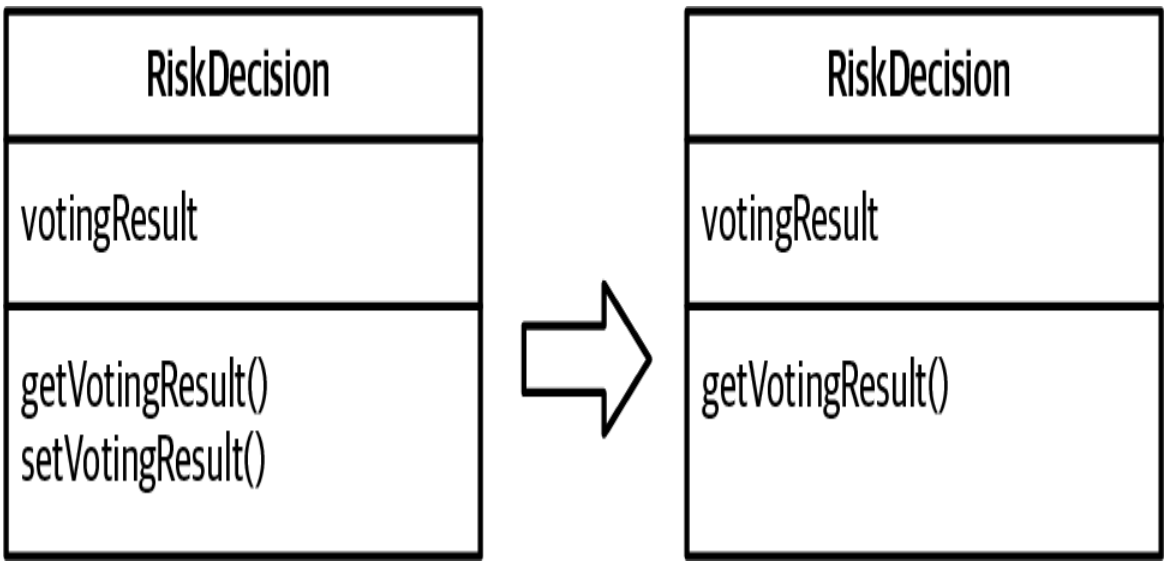
Mechanics

- Identify the setter to be replaced.
- Model with your domain experts the corresponding business process to find out what the right name for the new method will be.
- Introduce the new method into the to-be-changed class.
- Find the calls to the setter.
- Step-by-step see how you can replace the call to the setter with a call to the domain-named method.

- When all calls to the setter are replaced, remove it from the class.
- Find out if the newly created method has pre- and/or postconditions, and bring them to the code with *Introduce Contract*.

Remove Setter

A special case of Remove Setting Method *[Fowler 2019, p. 331]*



Motivation

Unfortunately, many programmers (and IDEs) tend to generate setters for all instance variables or fields. (See *Replace Setter* for reasons for that bad behavior.)

Mechanics

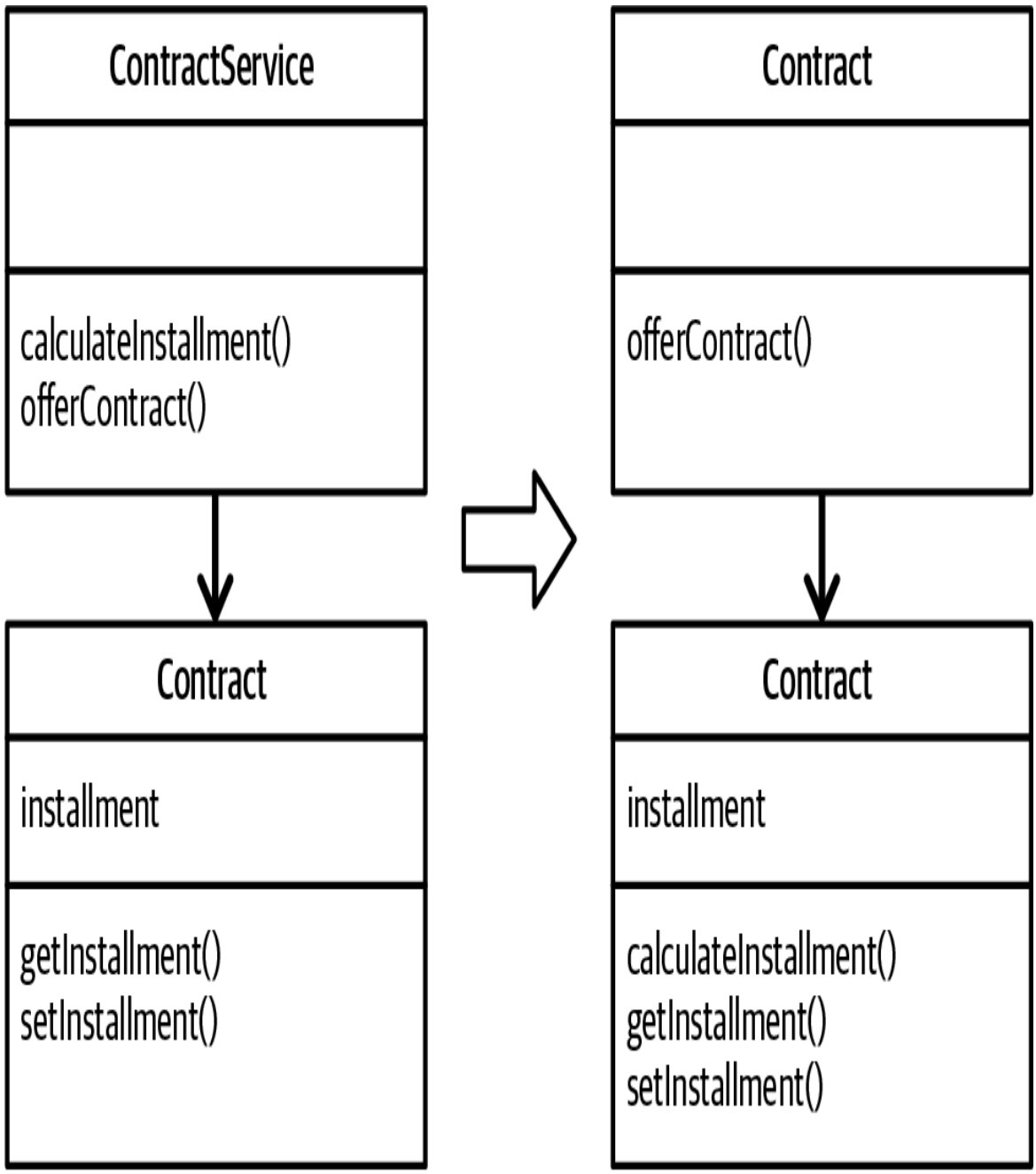
- Find all calls to the setter.
- If there are none, remove it.

WARNING

You may not find all calls with static code analysis, as many frameworks rely on getters and setters to work their reflection magic.

Move Logic from Service to Entity

A special case of Move Method *[Fowler 2019, p. 198]*



Motivation

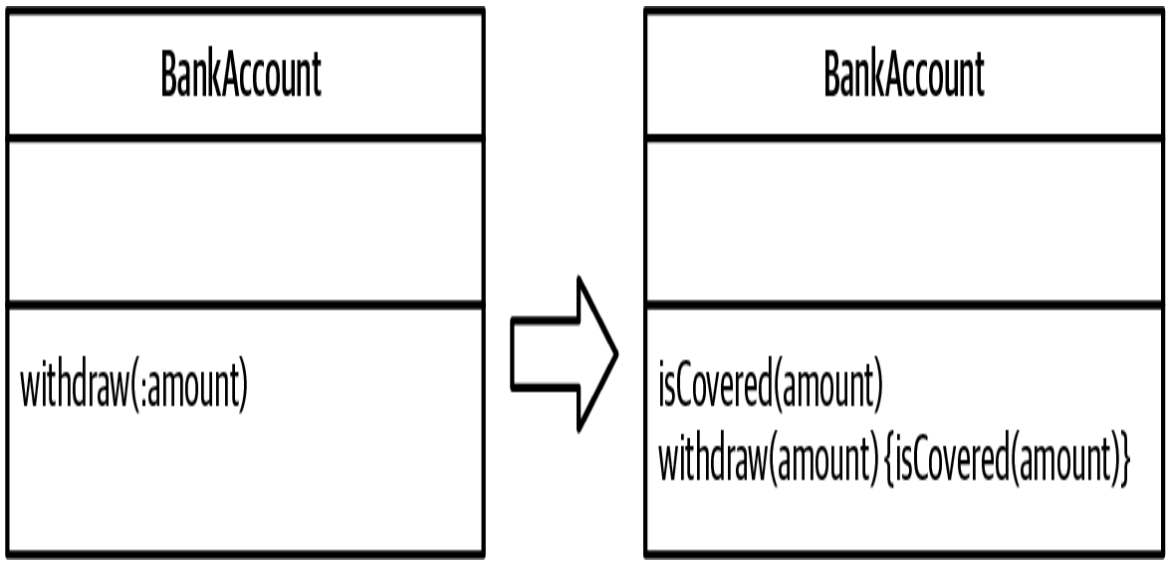
A domain service is a class that contains pure domain logic. This domain logic often operates on entities. If a piece of domain logic operates only on one entity and forces that entity to expose its data, it might make sense to move that piece of domain logic down into the entity. Thus, the cohesion of the entity can be increased, and the data can be encapsulated in the entity.

This is one of the small refactorings to *Heal Entity Anemia*.

Mechanics

- Copy the method from the entity to the service.
- Change the implementation of the to-be-moved-down method in the service so that it just forwards to the method in the entity.
- Identify the code that is calling the method and check if it makes sense that it directly works on the entity. If yes, change it to call the method on the entity directly.
- If no more code is calling the method in the service, delete it.
- Check if you can now get rid of setters by encapsulating data with *Replace Setter* or *Remove Setter*.

Introduce Contract



Motivation

Every method (and every operation in general, for that matter) can only operate under certain conditions. If we express these pre- and

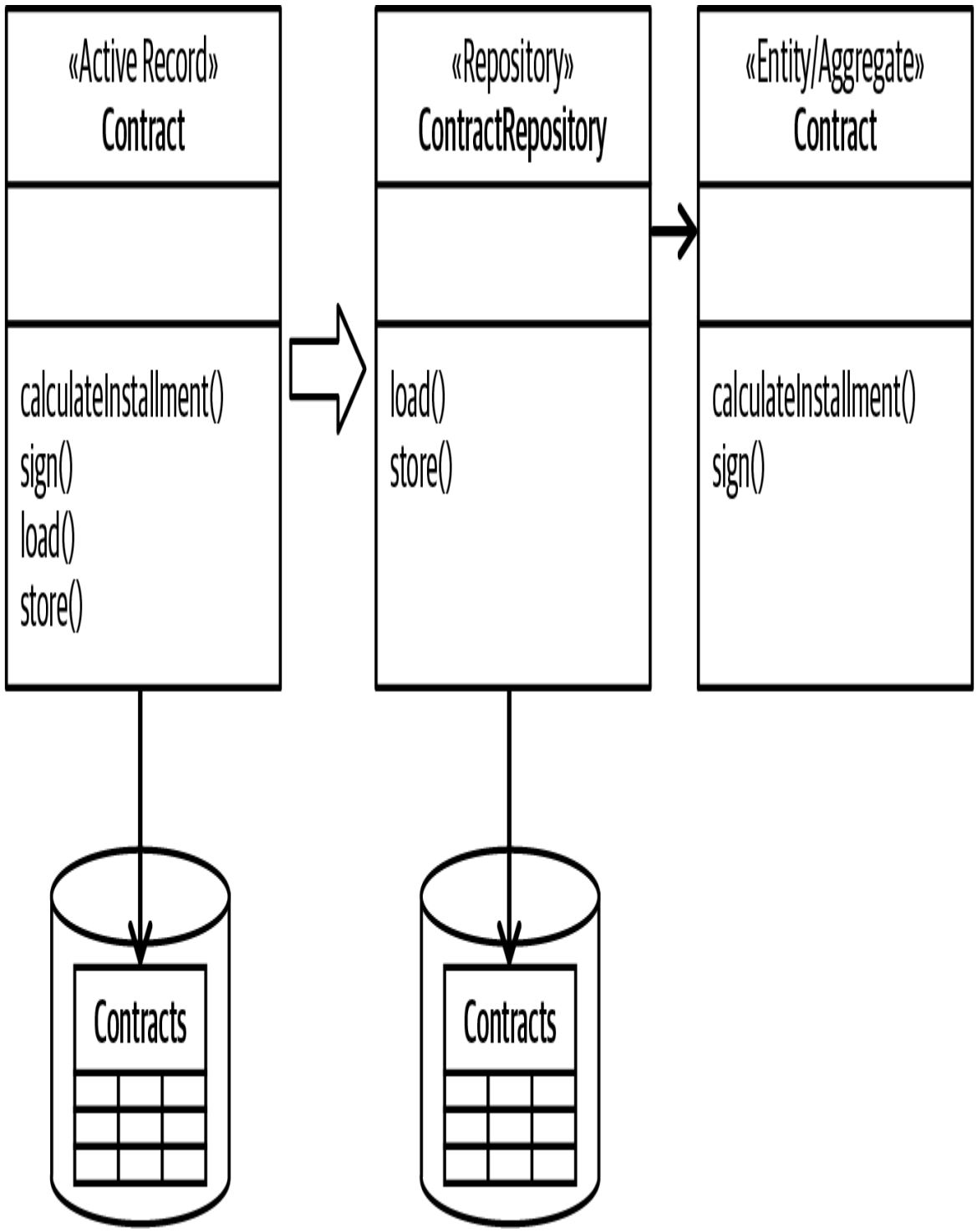
postconditions in the code (following the idea of Design by Contract, see [Meyer 1997]), we make that code both easier to understand and less error-prone.

> **WARNING**
>
> Unfortunately, most programming languages support only a poor man's Design by Contract with assertions. In that situation, it is often helpful to at least document contracts in comments.

Mechanics

- Document the method's preconditions with *Introduce Assertion* [Fowler 2019, p. 302].
- Document the method's postconditions with *Introduce Assertion* [Fowler 2019, p. 302].
- Check if the class that contains the method follows invariants, and document them with *Introduce Assertion* [Fowler 2019, p. 302], too.
- For every condition, if it makes sense, introduce a predicate method that gives the check a name. If applicable, use *Extract Method* [Fowler 2019, p. 106] for that.
- Expose the predicate method by giving it the same visibility as the changed method.

Split Active Record into Aggregate and Repository



Motivation

An Active Record (described in [Fowler 2003a]) is a part of the domain logic that does two things: it represents a concept from the domain, and

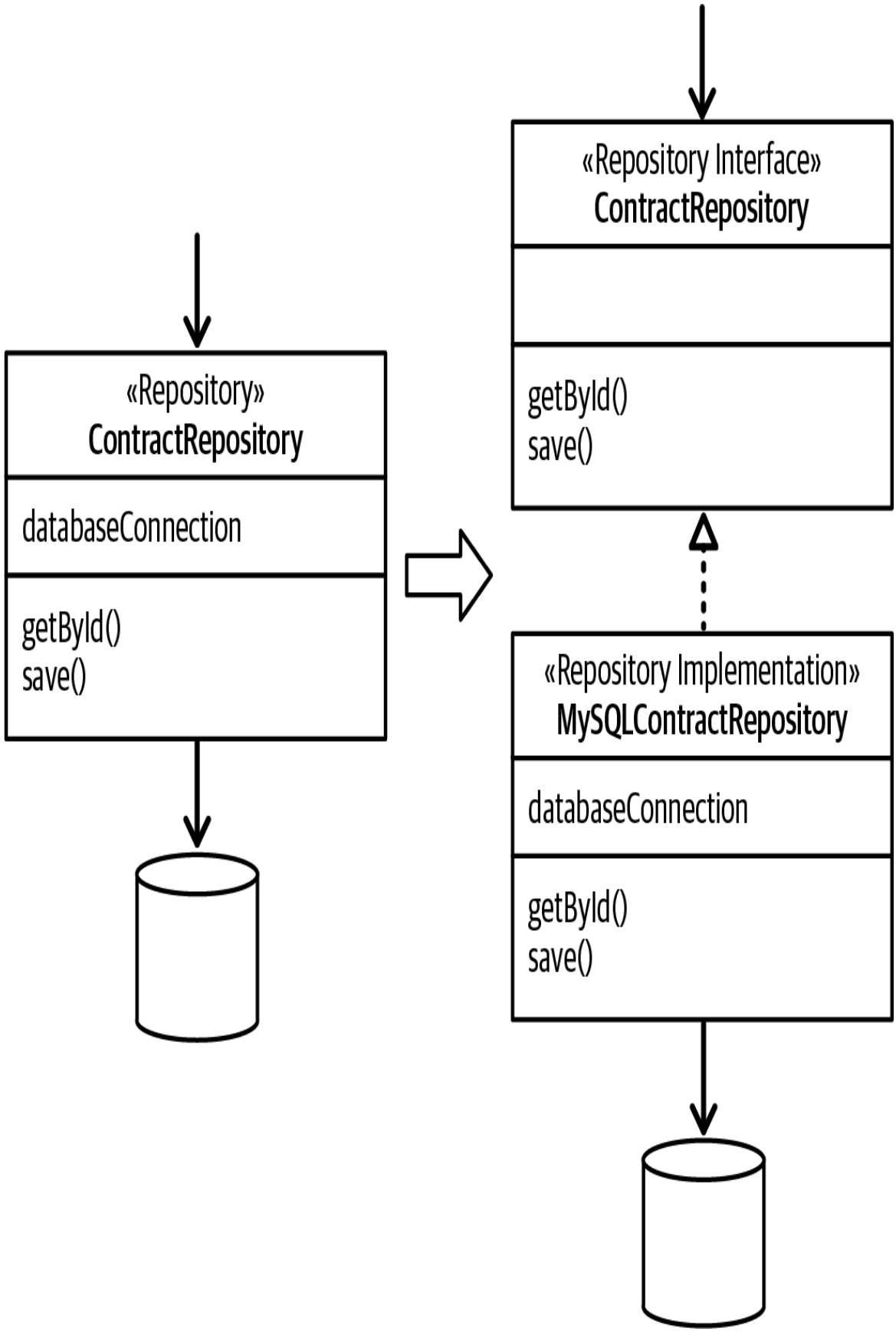
manages the storage of that concept. This refactoring separates these two responsibilities from each other.

Mechanics

- Identify methods in the active record as either domain logic or data access logic. If they are mixed, use *Extract Method* [Fowler 2019, p. 106] first.
- Create a repository class and copy all data access logic into it.
- Mark all data access methods in the active record as `@Deprecated`, `[Obsolete]`, or the like.
- Replace every call to a data access method of the active record with a call to the corresponding method in the repository.
- Remove data access methods from the active record, thus turning it into an aggregate or entity.

Often, a follow-up step to this refactoring is *Split Repository into Interface and Implementation*.

Split Repository into Interface and Implementation


«Repository»
ContractRepository
databaseConnection
getById()
save()
«Repository Interface»
ContractRepository
getById()
save()
«Repository Implementation»
MySQLContractRepository
databaseConnection
getById()
save()

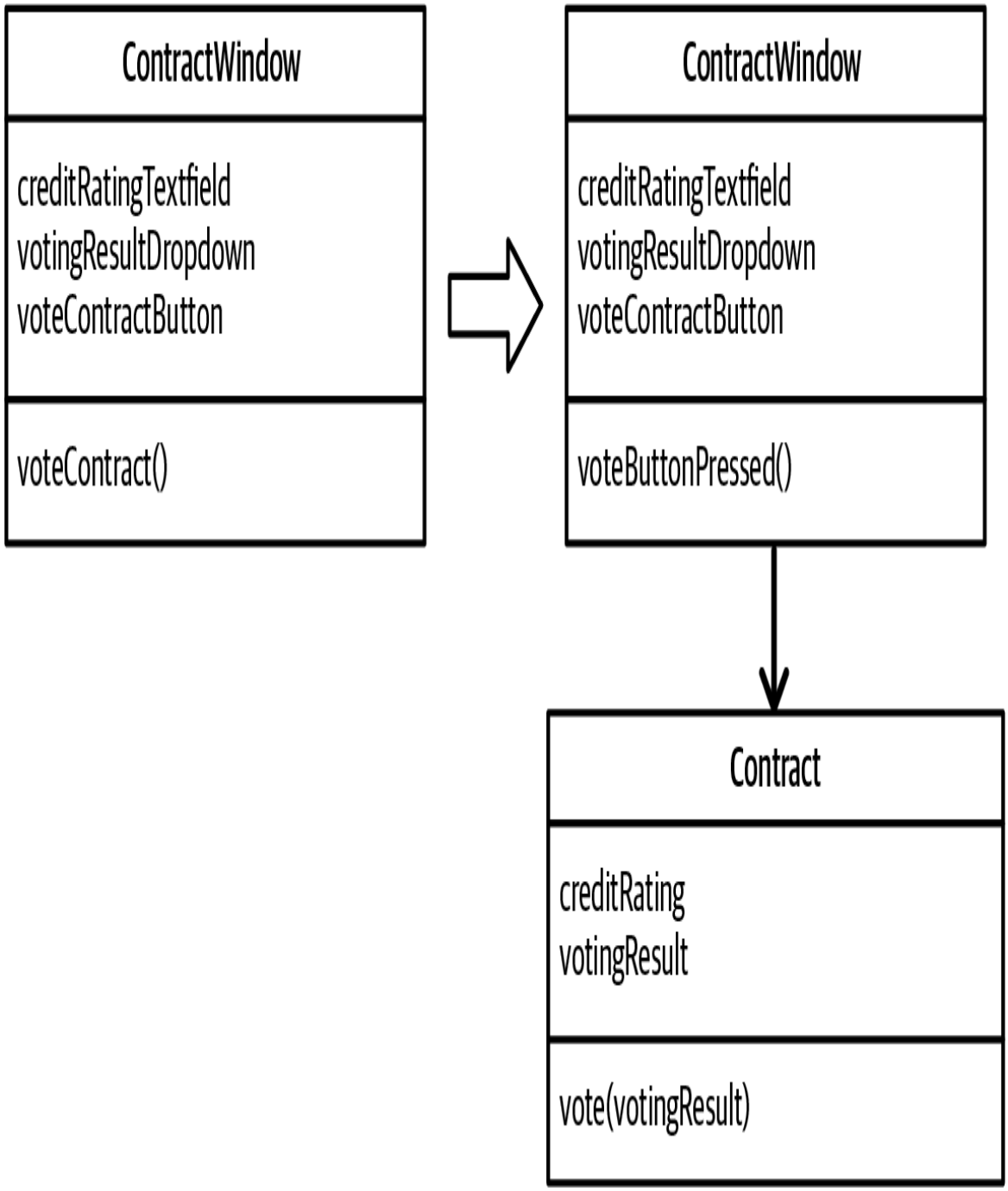
Motivation

A repository is a place in your domain layer where you can keep your entities or aggregates when you don't need them. A typical implementation for a repository is to store the aggregate's data into a database. Database access is a technical detail that the domain layer should have no knowledge of. Thus, separate the definition of the repository (domain layer) from its implementation (infrastructure layer).

Mechanics

- Apply *Extract Interface* [Fowler 1999, p. 341] on the repository.
- Move the implementation to the infrastructure layer; keep the interface in the domain layer.
- Move all uses of the old repository to the new interface. Only the the dependency injection configuration should know about the repository implementation.

Extract Entity from Smart UI



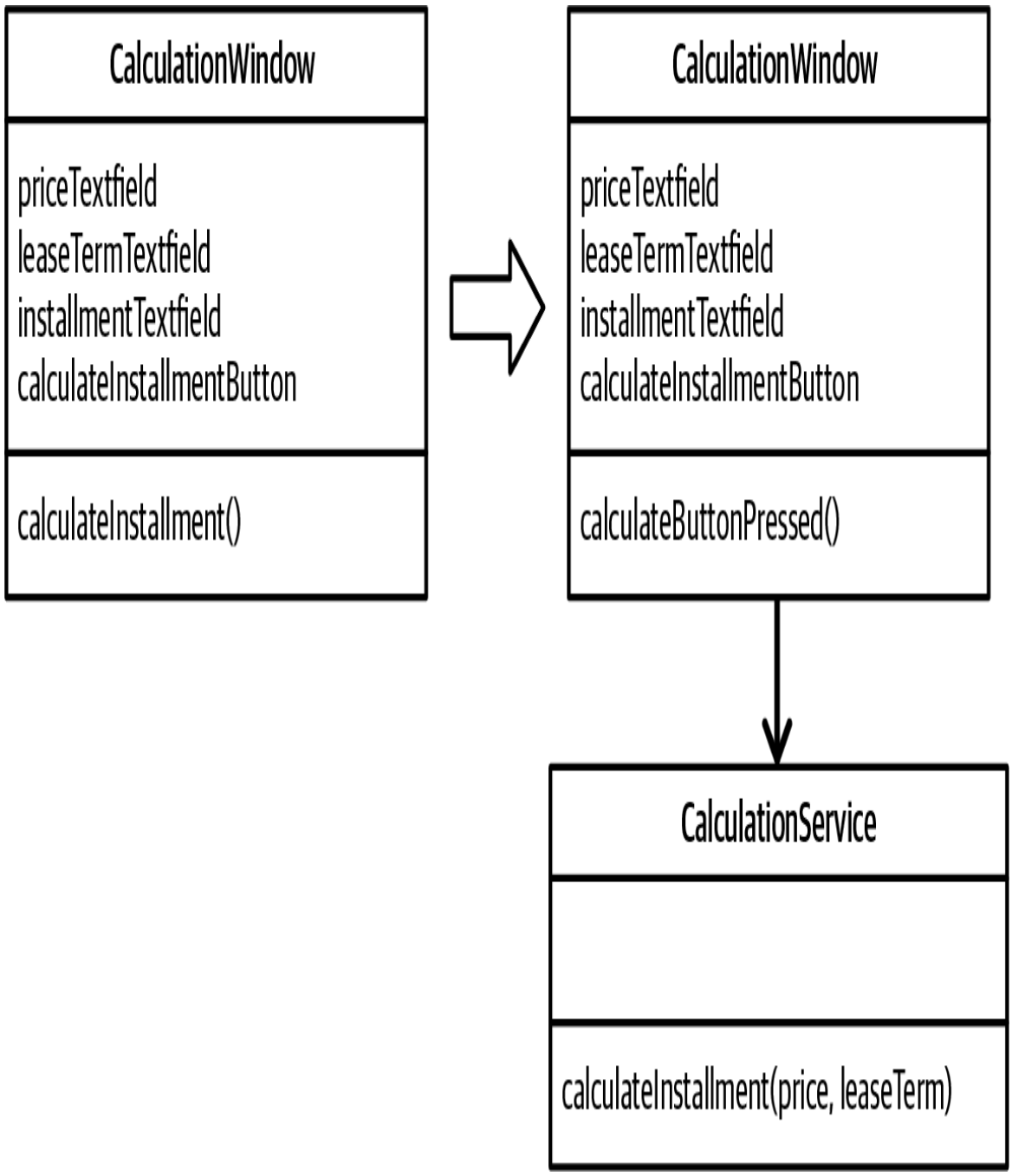
Motivation

Smart UI is a pattern (in DDD considered an "antipattern," see [Evans 2004, p. 76]) that mixes up UI code and business logic. This refactoring helps to separate the two concerns.

Mechanics

- In the smart UI class, separate the business logic from the UI logic with *Extract Method* [Fowler 2019, p. 106].
- Examine the extracted business logic method. Does it work on fields from the class? Then this refactoring is right; but if it works only on its parameters, *Extract Service from Smart UI* might fit better.
- Apply *Extract Class* [Fowler 2019, p. 182] and move the business logic and business data into the newly created entity.

Extract Service from Smart UI



Motivation

Smart UI is a pattern (in DDD considered an “antipattern,” see [Evans 2004, p. 76]) that mixes up UI code and business logic. This refactoring helps to separate the two concerns.

Mechanics

- In the smart UI class, separate the business logic from the UI logic with *Extract Method* [Fowler 2019, p. 106].
- Examine the extracted business logic method. Can it work only on parameters without using fields from the class? Then this refactoring is right; otherwise, *Extract Entity from Smart UI* might fit better.
- Apply *Extract Class* [Fowler 2019, p. 182] and move the business logic method into the newly created service.

Bibliography

Adzic, Gojko. 2011. *Specification by Example: How Successful Teams Deliver the Right Software*. Shelter Island, NY: Manning.

Ambler, Scott W., and Pramod J. Sadalage. 2006. *Database Refactorings: Evolutionary Database Design*. Upper Saddle River, NJ: Addison-Wesley.

Baas-Schwegler, Kenny, and João Rosa, eds. 2020. *Visual Collaboration Tools*. Published by the editors. *https://leanpub.com/visualcollaborationtools*.

Baas-Schwegler, Kenny, and Marco Heimeshoff. 2021. "EventStorming: Emerging Pivotal Events for a Big Picture." *Virtual DDD Design Heuristics*. *https://www.virtualddd.com/heuristics/eventstorming-emerging-pivotal-events-for-a-big-picture*.

Beck, Kent. 2024. *Tidy First?: A Personal Exercise in Empirical Software Design*. Sebastopol, CA: O'Reilly.

Beck, Kent, and Cynthia Andres. 2005. *Extreme Programming Explained: Embrace Change*. 2nd ed. Boston: Addison-Wesley.

Brandolini, Alberto. 2013. "Introducing Event Storming." *Ziobrando's Lair (Blog)*. *http://ziobrando.blogspot.com/2013/11/introducing-event-storming.html*.

———. 2021. *Introducing EventStorming*. Published by the author. *https://leanpub.com/introducing_eventstorming*.

Bråten, Stein. 1973. "Model Monopoly and Communications: Systems Theoretical Notes on Democratization." *Acta Sociologica* 16 (2).

Brooks, Frederick P., Jr. 1986. "No Silver Bullet: Essence and Accident in Software Engineering." In *The Mythical Man Month: Essays on Software Engineering, Anniversary Edition*, edited by Frederick P. Brooks Jr., 177–

203. Boston (1995): Addison-Wesley. *http://www.cs.unc.edu/techreports/86-020.pdf*.

———. 1995. *The Mythical Man Month: Essays on Software Engineering, Anniversary Edition*. Boston: Addison-Wesley.

Brown, Simon. Forthcoming. *The C4 Model*. Santa Rosa, CA: O'Reilly.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. New York: Wiley.

Ciceri, Christian, Dave Farley, Neal Ford, Andrew Harmel-Law, Michael Keeling, Carola Lilienthal, João Rosa, Alexander von Zitzewitz, Rene Weiss, and Eoin Woods. 2022. *Software Architecture Metrics: Case Studies to Improve the Quality of Your Architecture*. Sebastopol, CA: O'Reilly.

Cockburn, Alistair. 2005. "Hexagonal Architecture." *https://alistair.cockburn.us/hexagonal-architecture*.

Cockburn, Alistair, and Juan Manuel Garrido de Paz. 2024. *Hexagonal Architecture Explained: How the Ports & Adapters Architecture Simplifies Your Life, and How to Implement It*. Salt Lake City, UT: Humans and Technology Press.

Conway, Melvin E. 1968. "How Do Committees Invent?" *Datamation* 14 (5): 28–31. *https://www.melconway.com/Home/Conways_Law.html*.

da Silva, Eduardo, and Nick Tune. 2023. "Architecture Modernization Enabling Teams (AMET)." *Medium*. *https://medium.com/nick-tune-tech-strategy-blog/architecture-modernization-enabling-teams-amet-23a7f867b284*.

Dijkstra, Edsger. 1968. "Go To Statement Considered Harmful." *Communications of the ACM* 11 (3): 147–48.

Drotbohm, Oliver. 2022. "Architecturally Evident Applications: How to Bridge the Model-Code Gap?" *https://xmolecules.org/aea-paper.pdf*.

Design Council, n.d. "Eleven Lessons: Managing Design in Eleven Global Brands."

https://www.designcouncil.org.uk/fileadmin/uploads/dc/Documents/ElevenLessons_Design_Council%2520%25282%2529.pdf.

Evans, Eric. 2004. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley.

———. 2013. "Getting Started with DDD When Surrounded by Legacy Systems." *Domain Language*. *https://www.domainlanguage.com/ddd/surrounded-by-legacy-software*.

———. 2015. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Published by the author. *https://www.domainlanguage.com/ddd/reference*.

Feathers, Michael C. 2004. *Working Effectively with Legacy Code*. Upper Saddle River, NJ: Prentice Hall.

Floyd, Christiane. 1993. "STEPS: A Methodical Approach to PD." *Communications of the ACM* 63 (4): 83. *https://doi.org/10.1145/153571.214826*.

Foote, Brian, and Joseph Yoder. 1997. "Big Ball of Mud." In *Fourth Conference on Patterns Languages of Programs (PLoP '97)*. Technical Report #WUCS-97-34. Department of Computer Science, Washington University. *http://www.laputan.org/mud/*.

Fowler, Martin. 1997. *Analysis Patterns: Reusable Object Models*. Boston: Addison-Wesley.

———. 1999. *Refactoring: Improving the Design of Existing Code*. 1st ed. Boston: Addison-Wesley.

———. 2003a. *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley.

———. 2003b. "Anemic Domain Model." *Bliki*. *https://www.martinfowler.com/bliki/AnemicDomainModel.html*.

———. 2004a. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd ed. Boston: Addison-Wesley.

———. 2004b. “Strangler Fig Application.” *Bliki*. *https://martinfowler.com/bliki/StranglerFigApplication.html*.

———. 2005. “EventSourcing.” *Bliki*. *https://martinfowler.com/eaaDev/EventSourcing.html*.

———. 2011. “CQRS.” *Bliki*. *https://martinfowler.com/bliki/CQRS.html*.

———. 2014. “Replacing Throwing Exceptions with Notification in Validations.” *Bliki*. *https://martinfowler.com/articles/replaceThrowWithNotification.html*.

———. 2017. “Event-Driven.” *Bliki*. *https://martinfowler.com/articles/201701-event-driven.html*.

———. 2019. *Refactoring: Improving the Design of Existing Code*. 2nd ed. Boston: Addison-Wesley.

———. 2021. “On the Diverse and Fantastical Shapes of Testing.” *https://martinfowler.com/articles/2021-test-shapes.html*.

Gafert, Peter. 2023. “Unit Test Your Java Architecture.” *Open Source Project*. *https://www.archunit.org*.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Graca, Herberto. 2017. “DDD, Hexagonal, Onion, Clean, CQRS, … How I Put It All Together.” *Blog*. *https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together*.

Helfland, Heidi. 2020. *Dynamic Reteaming: The Art & Wisdom of Changing Teams*. Sebastopol, CA: O’Reilly.

Hoare, C.A.R. 2009. “Null References: The Billion Dollar Mistake.” In *QCon London 2009, Track: Historically Bad Ideas*. *https://qconlondon.com/london-2009/qconlondon.com/london-2009/tracks/show_trackd04f.html?trackOID=232*.

Hofer, Stefan, and Henning Schwentner. 2022. *Domain Storytelling: A Collaborative, Visual, and Agile Way to Develop Domain-Driven Software*. Boston: Addison-Wesley.

Hofer, Stefan, Henning Schwentner, and Heinz Züllighoven. 2020. "Distilling the Essence of Collaborative Modeling." *Medium*. *https://medium.com/@wps_de/distilling-the-essence-of-collaborative-modeling-ad1dd5204112*.

Hohpe, Gregor. 2020. *The Software Architect Elevator: Redesigning the Architect's Role in the Digital Enterprise*. Sebastopol, CA: O'Reilly.

Humble, Jez, and David Farley. 2011. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Upper Saddle River, NJ: Addison-Wesley.

Hunt, Andrew, and David Thomas. 2019. *The Pragmatic Programmer: Your Journey to Mastery, 20th Anniversary Edition*. Boston: Addison-Wesley.

Jackson, Cam. 2019. "Micro Frontends." *Bliki*. *https://martinfowler.com/articles/micro-frontends.html*.

Junker, Annegret. 2025. *Mastering Domain-Driven Design: Collaborative Modeling with Domain Storytelling, Event Storming, and Context Mapping*. India: BPB.

Kaiser, Susanne. 2026. *Architecture for Flow: Adaptive Systems with Domain-Driven Design, Wardley Mapping, and Team Topologies*. Boston: Addison-Wesley.

Kelle, Evelyn van, Gien Verschatse, and Kenny Baas-Schwegler. 2025. *Collaborative Software Design: How to Facilitate Domain Modeling Decisions*. Shelter Island, NY: Manning.

Khononov, Vlad. 2022. *Learning Domain-Driven Design*. Sebastopol, CA: O'Reilly.

———. 2024. *Balancing Coupling in Software Design: Universal Design Principles for Architecting Modular Software Systems*. Boston: Addison-

Wesley.

Koch, Jörn. 2022. “Scenario Casting.” *https://scenario-casting.org*.

Krilic, Mufrid. 2024. “Building an Application with TDD, DDD and Hexagonal Architecture—Isn’t It a Bit Too Much?” *Medium*. *https://mufridk.medium.com/building-an-application-with-tdd-ddd-and-hexagonal-architecture-isnt-it-a-bit-too-much-f4b55110f0e3*.

Kurtz, Cynthia F., and David J. Snowden. 2003. “The New Dynamics of Strategy: Sense-Making in a Complex and Complicated World.” *IBM Systems Journal* 42 (3): 462–83.

Lilienthal, Carola. 2008. “Komplexität von Softwarearchitekturen: Stile und Strategien.” Doctoral dissertation, University of Hamburg. *https://ediss.sub.uni-hamburg.de/handle/ediss/2160*.

———. 2019. *Sustainable Software Architecture: Analyze and Reduce Technical Debt*. Heidelberg: dpunkt.

Liskov, Barbara, and Jeannette Wing. 1994. “A Behavioral Notion of Subtyping.” *ACM Transactions on Programming Languages and Systems* 16 (6): 1811–41.

Manns, Mary Lynn, and Linda Rising. 2015. *More Fearless Change*. Boston: Addison-Wesley.

Martin, Robert C. 2010. “The Boy Scout Rule.” In *97 Things Every Programmer Should Know: Collective Wisdom by the Experts*, edited by Kevlin Henney. Sebastopol, CA: O’Reilly.

———. 2013. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice Hall.

———. 2018. *Clean Architecture: A Craftsman’s Guide to Software Structure and Design*. Boston: Prentice Hall.

Martraire, Cyrille. 2019. *Living Documentation: Continuous Knowledge Sharing by Design*. Boston: Addison-Wesley.

Meyer, Bertrand. 1997. *Object-Oriented Software Construction*. 2nd ed. Upper Saddle River, NJ: Prentice Hall.

Millet, Scott, and Nick Tune. 2015. *Patterns, Principles, and Practices of Domain-Driven Design*. Birmingham: Wrox.

Myers, Glenford J. 1978. *Composite/Structured Design*. New York: Van Nostrand Reinhold.

Newman, Sam. 2020. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. Sebastopol, CA: O'Reilly.

———. 2022. *Building Microservices: Designing Fine-Grained Systems*. 2nd ed. Sebastopol, CA: O'Reilly.

Palermo, Jeffrey. 2008. "The Onion Architecture." *https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1*.

Parnas, David L. 1972. "On the Criteria to Be Used in Decomposing Systems into Modules." *Communications of the ACM* 15 (12): 1053–58.

Patton, Jeff. 2014. *User Story Mapping: Discover the Whole Story, Build the Right Product*. Sebastopol, CA: O'Reilly.

Plöd, Michael. 2020. *Hands-on Domain-Driven Design—by Example*. Published by the author. *https://leanpub.com/ddd-by-example*.

———. 2025. "Bounded Contexts Are All About Cohesion." *https://www.michael-ploed.com/blog/bounded-contexts-are-all-about-cohesion*.

Rayner, Paul. 2024. *The EventStorming Handbook: Unlocking Creativity, Collaboration, and Communication for Your Teams*. Published by the author. *https://leanpub.com/eventstorming_handbook*.

Read, Jacqui. 2023. *Communication Patterns: A Guide for Developers and Architects*. Sebastopol, CA: O'Reilly.

Richards, Mark, and Neal Ford. 2020. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA: O'Reilly.

Roock, Stefan, and Martin Lippert. 2006. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Chichester, England: Wiley.

Schimak, Martin. 2019. “Storystorming.” *Medium*. *https://medium.com/plexiti/story-storming-191756f57387*.

Schwaber, Ken, and Jeff Sutherland. 2020. *The Scrum Guide*. *https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf*.

Schwentner, Henning. 2008. “Wider die enge Kopplung: Große Refactorings in einem Prozess zur Entflechtung von Anwendung und Rahmenwerk.” Master’s thesis, University of Hamburg. *https://swa.informatik.uni-hamburg.de/files/abschlussarbeiten/diplomarbeit-henning_schwentner_mit_deckblatt.pdf*.

Siedersleben, Johannes. 2004. *Moderne Softwarearchitektur: Umsichtig planen, robust bauen mit Quasar*. 3rd ed. Heidelberg: dpunkt.

Skelton, Matthew, and Manuel Pais. 2019. *Team Topologies: Organizing Business and Technology Teams for Fast Flow*. Portland, OR: IT Revolution.

Snowden, David J., and Mary E. Boone. 2007. “A Leader’s Framework for Decision Making.” *Harvard Business Review* 85 (11): 68–76.

Spolsky, Joel. 2003. “Exceptions.” *Joel on Software (Blog)*. *https://www.joelonsoftware.com/2003/10/13/13*.

Starke, Gernot. 2022a. “Arc42.” *Open Source Project*. *https://www.arc42.org*.

———. 2022b. “Architecture Improvement Method.” *Open Source Project*. *https://aim42.github.io*.

———. 2024. *Effektive Softwarearchitekturen: Ein praktischer Leitfaden*. Munich: Hanser.

Stevens, Wayne P., Glenford J. Myers, and Larry L. Constantine. 1974. "Structured Design." *IBM Systems Journal* 13 (2): 115–39. *https://doi.org/10.1147/sj.132.0115*.

Stroustrup, Bjarne. 2024. *Programming: Principles and Practice Using C++*. 3rd ed. Hoboken, NJ: Addison-Wesley.

Tornhill, Adam. 2015. *Your Code as a Crime Scene: Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs*. Dallas: Pragmatic Bookshelf.

———. 2018. *Software Design X-Rays: Fix Technical Debt with Behavioral Code Analysis*. Raleigh, NC: Pragmatic Bookshelf.

Tuckman, Bruce W. 1965. "Developmental Sequence in Small Groups." *Psychological Bulletin* 63 (6): 384–99. *https://doi.org/10.1037/h0022100*.

Tune, Nick, and Jean-Georges Perrin. 2024. *Architecture Modernization: Socio-Technical Alignment of Software, Strategy, and Structure*. Shelter Island, NY: Manning.

Vernon, Vaughn. 2013. *Implementing Domain-Driven Design*. Upper Saddle River, NJ: Addison-Wesley.

———. 2015. *Reactive Messaging Patterns with the Actor Model: Application and Integration in Scala and Akka*. Boston: Addison-Wesley.

———. 2016. *Domain-Driven Design Distilled*. Boston: Addison-Wesley.

———. 2017. *Domain-Driven Design kompakt*. Translated by Carola Lilienthal and Henning Schwentner. Heidelberg: dpunkt.

Vernon, Vaughn, and Tomasz Jaskuła. 2022. *Strategic Monoliths and Microservices: Driving Innovation Using Purposeful Architecture*. Boston: Addison-Wesley.

Voss, Markus, Andreas Hess, and Bernhard Humm. 2006. "Towards a Framework for Large Scale Quality Architecture." In *Perspectives in Software Architecture Quality. Short Papers of the 2nd International Conference on the Quality of Software Architectures (QoSA 2006)*, edited by Christine Hofmeister, Ivica Crnkovic, Ralf Reussner, and Steffen

Becker, 52–58. Internal Report. Department of Informatics, University of Karlsruhe. *https://doi.org/10.5445/IR/1000005146*.

Wardley, Simon. 2020. *Wardley Maps: Topographical Intelligence in Business*. Published by the author. *https://github.com/HiredThought/wardley-maps-ebook/raw/main/bin/Wardley%20Maps%20-%20Simon%20Wardley.pdf*.

Wlaschin, Scott. 2018. *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#*. Raleigh, NC: Pragmatic Bookshelf.

Wolff, Eberhard. 2017a. *A Practical Guide to Continuous Delivery*. Boston: Addison-Wesley.

———. 2017b. *Microservices: Flexible Software Architecture*. Boston: Addison-Wesley.

Wynne, Matt. 2015. “Introducing Example Mapping.” *Cucumber Blog*. *https://cucumber.io/blog/bdd/example-mapping-introduction*.

Yourdon, Edward, and Larry L. Constantine. 1978. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*. 2nd ed. New York: Yourdon Press.

Züllighoven, Heinz. 2005. *Object-Oriented Construction Handbook: Developing Application-Oriented Software with the Tools and Materials Approach*. San Francisco: Morgan Kaufmann.

Index

A

accidental complexity, Sources of Accidental Complexity, Dealing with Essential and Accidental Complexity-Dealing with Essential and Accidental Complexity, DDD Concepts Against Complexity-DDD Concepts Against Complexity, Where Does the Big Ball of Mud Come From?-Where Does the Big Ball of Mud Come From?

activity arrow, domain stories, Domain Storytelling

actors, Domain Storytelling

- in actual versus target architecture, Schematic Description of Step 3
- in domain rediscovery, Schematic Description of Step 1
- for prioritizing and implementing transformation, Schematic Description of Step 4
- in target architecture, Schematic Description of Step 2
- virtual actors thought experiment, Technique: Thought Experiment "Virtual Actors"-Technique: Thought Experiment "Virtual Actors"

actual versus target architecture, Comparing Actual Architecture with Target Architecture-Comparing Actual Architecture with Target Architecture, Summarizing Domain-Driven Transformation

- alignment of, Step 3—Alignment of Actual with Target Architecture-Substep 3.3: Compiling Refactorings and Findings
- identifying actual architecture, Substep 3.1: Identifying the Actual Architecture-Actual Architecture and AI

mapping of target to actual, Modularity in the MMI

schema of, Schematic Description of Step 3

Acyclic Dependency Principle (ADP), The Good Old Layered Architecture

aggregate, Tactical Design

and cohesion and coupling, Cohesion and Coupling in the Domain

ID references at boundaries of, Introduce ID References at Aggregate Boundaries-Introduce ID References at Aggregate Boundaries

one microservice per aggregate problem with SOA, Service-Oriented Architectures-Service-Oriented Architectures

splitting active record into, Motivation

aggregate root, Tactical Design

aggregate stickies, Software Design EventStorming, Deeper into the Details: Software Design EventStorming

agile process, Domain-Driven Transformation as, What Has Just Happened?

Agile software development, Origin of Complexity: Problem and Solution Space

AI (artificial intelligence)

and actual architecture, Actual Architecture and AI

as decomposition tool, Discussing Decomposition with AI

and domain knowledge, Does the AI have Knowledge About My Domain?

preconditions support in Design by Contract, Introduce Design by Contract

processing IDE warnings with, Resolving Compiler and IDE Warnings

for programming language updates, Updating the Programming Language

support for CoMo, Collaboration Modeling and AI

test writing for legacy software, The Scout Rule (Formerly Known as the Boy Scout Rule)

for updating software dependencies, Updating Software Dependencies

AI-aided transformation, AI-Aided Transformation

Alphorn Auto Leasing, case study, Schematic Description of Step 1, Substep 2.1: Creating a Context Map, Actual Architecture and AI, Substep 4.2: Adding Tactical Refactorings to the Product Backlog (or Improvement Backlog)

AMET (Architecture Modernization Enabling Teams), Architecture Modernization Enabling Teams-Architecture Modernization Enabling Teams

Analysis Patterns (Fowler), Documenting Architecture in Code

anemic domain model, Enrich Modeling with Domain Knowledge-Enrich Modeling with Domain Knowledge, Challenge: The Unbounded Anemic Domain Model-Challenge: The Unbounded Anemic Domain Model, Challenge: Move Domain Logic Out of Services and into the Anemic Domain Model-Push Down Logic from Services into Anemic Entities

anemic entity, extracting specialized, Motivation

annotations, domain stories, Domain Storytelling

anticorruption layer, context mapping pattern, Context Mapping

application service, domain layer, Tactical Design

architecture (see domain architecture; software architecture concepts)

architecture analyst, Overview of Strategic Transformation

Architecture Modernization (Tune), Architecture Modernization Enabling Teams

Architecture Modernization Enabling Teams (AMET), Architecture Modernization Enabling Teams-Architecture Modernization Enabling Teams

architecture review tool, Comparing Actual Architecture with Target Architecture, Decomposition from Inside to Outside

ArchUnit, Documenting Architecture in Code

as-is processes, Scope Factors for Domain Stories, Step 1—Domain Rediscovery, Substep 1.1: Getting to Know the As-Is Situation

assertions, source code, Introduce Design by Contract

automatic refactoring, Refactoring

automating build and deployment, Automating Build and Deployment-Automating Build and Deployment

autonomy, as prerequisite for working in flow, Working in the Flow

B

baby steps, in XP, Substep 4.2: Adding Tactical Refactorings to the Product Backlog (or Improvement Backlog)

backend team, Software as a Sociotechnical System-Vertical Architecture/Team Organization

banking case study, Chaotic Exploration, Service-Oriented Architectures, Enrich Modeling with Domain Knowledge, Types of Topologies, Problem Cause: Reusability Misunderstood, Central Domain Item Versus Diverse Domain Items

Beck, Kent, Breaking Dependencies

big ball of mud (BBoM), Big Ball of Mud-Coupling and Reuse

architecture level, At the Architecture Level

context mapping pattern, Context Mapping

microservices architecture transformation, From Monolith to Microservices-From Monolith to Microservices

in MMI hierarchy, MMI 0–4: Big ball of mud-MMI 0–4: Big ball of mud, Hierarchy in the MMI

and tactical transformation, Path A: Tactical transformation first

big bang replacement approach to transformation, How to Approach the Transformation-Replacement—Big Bang

Big Bank case study, Chaotic Exploration, Service-Oriented Architectures, Enrich Modeling with Domain Knowledge, Types of Topologies, Problem Cause: Reusability Misunderstood, Central Domain Item Versus Diverse Domain Items

Big Picture EventStorming, EventStorming-Enforcing the Timeline, The Color Coding-The Color Coding

boundaries

aggregate, Tactical Design, Introduce ID References at Aggregate Boundaries-Introduce ID References at Aggregate Boundaries

between bounded contexts, Bounded context

crossing in unbounded models, Domain model and bounded contexts, Challenge: Unbounded Models

drawing for subdomains, Drawing Boundaries-Drawing Boundaries, Substep 1.4: Finding Subdomains in the Business Processes

indicators, Indicators for Subdomain Boundaries

in software teams, Software as a Sociotechnical System, Vertical Architecture/Team Organization

boundaries, drawing for subdomains, Drawing Boundaries

bounded contexts, Determining Target Architecture, Bounded context

assigning to teams, Substep 2.3: Assigning Bounded Contexts to Teams-Substep 2.3: Assigning Bounded Contexts to Teams

bubble context, Bubble Context

context mapping, Context map-Context Mapping

and domain model, Domain model and bounded contexts-Domain model and bounded contexts

domain patterns in, Outlook: Domain Patterns and Their Implementation in Bounded Contexts-Room for Transformation

microservices support through domain division, Microservices

MMI metric for relationships between, Hierarchy in the MMI

refactoring activities, Comparing Actual Architecture with Target Architecture, Substep 4.4: Extracting Context, Motivation, Motivation

and ubiquitous language, Ubiquitous Language

upstream-downstream, Context Mapping

Brandolini, Alberto, EventStorming

breaking dependencies, legacy systems, Breaking Dependencies-Breaking Dependencies, Break Cycles and Dependencies

bubble context, Bubble Context

bugs in legacy software, Resolving Compiler and IDE Warnings-Different Kinds of Tests

in anemic domain model, Enrich Modeling with Domain Knowledge

improving error robustness, Improving Error Robustness-Introduce Design by Contract

resolving compiler and IDE warnings, Resolving Compiler and IDE Warnings

test coverage for fixing, Increase Test Coverage When Fixing Bugs-Different Kinds of Tests

build and deployment, automating, Automating Build and Deployment-Automating Build and Deployment

building blocks, tactical design, Tactical Design-Tactical Design, Inside-Outside Architectures and Tactical DDD-Software Architecture Hamburger

business domain/problem space, DDD and the Business Domain-Core domain charts

architecture (see domain architecture)

domain model (see domain model)

knowledge base (see domain knowledge)

optimizing processes, Substep 1.3: Optimizing the Business Processes

reducing inheritance in source code, Reduce Inheritance in Business Source Code

strategic design in, Strategic Design on the Business Side-Strategic Design on the Business Side

subdomains (see subdomains)

teams (see teams)

ubiquitous language, Ubiquitous Language-English or the local language?, Indicators for Subdomain Boundaries

business logic, Separating Business Source Code and Technical Source Code, Challenge: Disentangling Business Logic in Services-Challenge: Disentangling Business Logic in Services

business versus technical source code, Separating Business Source Code and Technical Source Code-Separating Business Source Code and Technical Source Code

C

C# 8, Updating the Programming Language, Build Robustness Against Null

C# 9/10, Updating the Programming Language

C4 Model, Example: A Legacy System in a Container Port, Comparing Actual Architecture with Target Architecture-Comparing Actual Architecture with Target Architecture, Architecture Documentation with the C4 Model-Level 4: Code

canonical model, Domain model and bounded contexts, From IT Landscapes to SOA and Back Again

car leasing case study, Schematic Description of Step 1, Substep 2.1: Creating a Context Map, Actual Architecture and AI, Substep 4.2: Adding Tactical Refactorings to the Product Backlog (or Improvement Backlog)

case studies

Alphorn Auto Leasing, Schematic Description of Step 1, Substep 2.1: Creating a Context Map, Actual Architecture and AI, Substep 4.2: Adding Tactical Refactorings to the Product Backlog (or Improvement Backlog)

Big Bank, Chaotic Exploration, Service-Oriented Architectures, Enrich Modeling with Domain Knowledge, Types of Topologies, Problem

Cause: Reusability Misunderstood, Central Domain Item Versus Diverse Domain Items

Cinema (Cine2001), DDD Concepts Against Complexity, Domain Storytelling, Level 1: Context, Introduce Design by Contract, Types of Topologies, Indicators for Subdomain Boundaries

container port terminal, Domain Rediscovery-Domain Rediscovery, Central Domain Item Versus Diverse Domain Items

Green Urban Supplier, Introducing a Case Study: Green Urban Supplier

central domain item, Central Domain Item Versus Diverse Domain Items-Central Domain Item Versus Diverse Domain Items

challenges element, strategic transformation

in actual versus target architecture, Schematic Description of Step 3

in domain rediscovery, Schematic Description of Step 1

for prioritizing and implementing transformation, Schematic Description of Step 4

in target architecture, Schematic Description of Step 2

chaotic exploration, EventStorming, Chaotic Exploration

CI/CD server, introducing to legacy system, Automating Build and Deployment, Updating the Programming Language

Cinema case study (Cine2001), DDD Concepts Against Complexity, Domain Storytelling, Level 1: Context, Introduce Design by Contract, Types of Topologies, Indicators for Subdomain Boundaries

classes

big ball of mud, At the Class Level-At the Class Level

cohesion and coupling, Cohesion and Coupling in DDD

MMI metric for number per cycle, Hierarchy in the MMI

standalone class and cohesion and coupling, Cohesion and Coupling in the Domain

subclasses, The Liskov Substitution Principle

superclass, The Liskov Substitution Principle

in unbounded domain model, Decomposition from Inside to Outside

Clean Architecture, Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More

cleaning up legacy code as you go, Cleaning Up the Code as You Go

Cockburn, Alistair, Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More

code level, C4 Model, Level 4: Code

cognitive load control, as prerequisite for working in flow, Working in the Flow

cohesion, increasing, Cohesion and Coupling-Cohesion and Coupling in the Domain, Strengthening the Source Code with Domain Knowledge-Strengthening the Source Code with Domain Knowledge, Increasing Cohesion-Make the Identity of Entities Explicit

cohesion through coupling, Modularity in the MMI

Design by Contract, Introduce Design by Contract-Introduce Design by Contract

enriching modeling with domain knowledge, Enrich Modeling with Domain Knowledge-Enrich Modeling with Domain Knowledge

making identity of entities explicit, Make the Identity of Entities Explicit-Make the Identity of Entities Explicit

using value objects, Utilize Value Objects-Utilize Value Objects

collaboration team interaction mode, Team Interaction Modes

Collaborative Modeling (CoMo), Collaborative Modeling-Summary

AI support for, Collaboration Modeling and AI

bringing the right people together, Bringing the Right People Together-Bringing the Right People Together

and DDD, Collaborative Modeling and DDD

domain identity in, Make the Identity of Entities Explicit

Domain Storytelling (see Domain Storytelling)

EventStorming (see EventStorming)

fundamental concepts, Fundamental Concepts of Collaborative Modeling-Fundamental Concepts of Collaborative Modeling

methods for transformation, Collaborative Modeling Methods for Transformation-Collaborative Modeling Methods for Transformation, Step 1—Domain Rediscovery

model monopoly, Modeling Tools and the Model Monopoly-Modeling Tools and the Model Monopoly

other methods, Other Collaborative Modeling Methods

remote, Collaborative Modeling Remote

Scenario Casting, Scenario Casting-The Second Round of Scenario Casting, Substep 1.4: Finding Subdomains in the Business Processes

which method to use when, Which Method to Use When?-Which Method to Use When?

collecting domain terms, ubiquitous language, Collecting domain terms-Collecting domain terms

color coding

EventStorming, The Color Coding-Deeper into the Details: Software Design EventStorming

Scenario Casting, Step 2: Focusing

command stickies, Software Design EventStorming, Deeper into the Details: Software Design EventStorming

commodity status, Wardley Maps, Age and Maturity of the Domain

compiler warnings, resolving bug issues, Resolving Compiler and IDE Warnings

complexity in software development, Mastering Complexity-The Fundamental Methods

accidental, Sources of Accidental Complexity, Dealing with Essential and Accidental Complexity-Dealing with Essential and Accidental Complexity, DDD Concepts Against Complexity-DDD Concepts Against Complexity, Where Does the Big Ball of Mud Come From?-Where Does the Big Ball of Mud Come From?

decision areas, Decision Areas of Software Architecture-Dealing with Essential and Accidental Complexity

essential, Type of Complexity: Essential Versus Accidental-Type of Complexity: Essential Versus Accidental, Dealing with Essential and Accidental Complexity

fundamental methods for dealing with, The Fundamental Methods-The Fundamental Methods

origin dimension, Origin of Complexity: Problem and Solution Space-Origin of Complexity: Problem and Solution Space

types of, Type of Complexity: Essential Versus Accidental-Dealing with Essential and Accidental Complexity

complicated subsystem team, Types of Topologies

components level, C4 Model, Level 3: Components

conformist context mapping pattern, Context Mapping

consistency, in anemic domain model, Enrich Modeling with Domain Knowledge

container level, C4 Model, Level 2: Containers

container port terminal case study, Domain Rediscovery-Domain Rediscovery, Central Domain Item Versus Diverse Domain Items

context (see bounded contexts)

context level, C4 Model, Level 1: Context

context map/mapping, Determining Target Architecture, Context map-Context Mapping, Substep 2.1: Creating a Context Map, Substep 3.2: Comparing the Target Context Map and the Source Code-Substep 3.2: Comparing the Target Context Map and the Source Code

Conway, Mel, Software as a Sociotechnical System

core domain, Path A: Tactical transformation first, Core domain-Core domain charts, Software Architecture Hamburger, Substep 2.2: Categorizing Subdomains, Pattern: Proceed with a Core Domain

core domain charts, Core domain charts-Core domain charts, Substep 2.2: Categorizing Subdomains

coupling and reuse, BBoM, Coupling and Reuse

coupling, reducing, Cohesion and Coupling-Cohesion and Coupling in the Domain, Strengthening the Source Code with Domain Knowledge-

Strengthening the Source Code with Domain Knowledge, Reduce Coupling-The Liskov Substitution Principle

breaking cycles and dependencies, Break Cycles and Dependencies

LSP, The Liskov Substitution Principle-The Liskov Substitution Principle

metrics for improving system implementation, Improving Internal Structure Using Metrics

reducing inheritance in source code, Reduce Inheritance in Business Source Code

using ID references at aggregate boundaries, Introduce ID References at Aggregate Boundaries-Introduce ID References at Aggregate Boundaries

cross-functionality of teams, Vertical Architecture/Team Organization, Motivation, Motivation

CRUD applications, Form-Based Domain Model

custom built status, Wardley Maps, Age and Maturity of the Domain

customer-supplier context mapping pattern, Context Mapping

cycles, breaking, Break Cycles and Dependencies

D

da Silva, Eduardo, Architecture Modernization Enabling Teams

dashboards, metrics for improving code structure, Improving Internal Structure Using Metrics-Improving Internal Structure Using Metrics

data bag, Motivation

data minimization, Reuse, duplication, and redundancy

data model

disentangling, Challenge: Disentangling the Data Model-Challenge: Disentangling the Data Model

unbounded, Challenge: Unbounded Models

database queries in business logic, Separating Business Source Code and Technical Source Code

database team, Software as a Sociotechnical System-Vertical Architecture/Team Organization

databases

business logic in, Separating Business Source Code and Technical Source Code

implementing persistence, Tactical Design

decision areas, software architecture, Decision Areas of Software Architecture

decomposing legacy systems, What Has Just Happened?, MMI 8–10: Domain-driven modularization

(see also Collaborative Modeling)

AI as tool for, Discussing Decomposition with AI

team organization, Improving Team Organization-Where Do You Stand?—Made Easy

unbounded domain model, Decomposition from Inside to Outside-Decomposition from Inside to Outside

user interface, Challenge: Decomposing the User Interface-UI in Context: Microfrontends

by work item, Challenge: Do Not Decompose by Work Items-Challenge: Do Not Decompose by Work Items

decoupling (see coupling, reducing)

dependencies

breaking in legacy systems, Breaking Dependencies-Breaking Dependencies, Break Cycles and Dependencies

MMI, Break Cycles and Dependencies

reducing to create test coverage, The Scout Rule (Formerly Known as the Boy Scout Rule)

subdomain context map, Strategic Design in the Software

updating, Updating Software Dependencies-Resolving Compiler and IDE Warnings

dependency cycle, BBoM, At the Class Level

deployment pipeline, automating, Automating Build and Deployment-Automating Build and Deployment

deployment units

C4 Model, Level 2: Containers

monolithic versus microservices, Monolith

Design by Contract, Introduce Design by Contract-Introduce Design by Contract, Introduce Design by Contract-Introduce Design by Contract

Design Thinking, Collaborative Modeling Methods for Transformation

developers (see domain experts and developers)

digital aspects of processes, Substep 1.2: Extracting the Domain Knowledge (Purification)

digitalized domain stories, Scope Factors for Domain Stories-Scope Factors for Domain Stories

distilling the business process, Categorizing Subdomains—What Is Core?, Scope Factors for Domain Stories, Distilling Ubiquitous Language

distributed big ball of mud, From Monolith to Microservices

divergence-convergence, Design Thinking, Collaborative Modeling Methods for Transformation, Enforcing the Timeline

diverse domain item, Central Domain Item Versus Diverse Domain Items-Central Domain Item Versus Diverse Domain Items

domain architecture, DDD and the Software Solution-Domain Events

- age and maturity of, Age and Maturity of the Domain
- bubble context, Bubble Context
- C4 Model, Example: A Legacy System in a Container Port, Comparing Actual Architecture with Target Architecture-Comparing Actual Architecture with Target Architecture, Architecture Documentation with the C4 Model-Level 4: Code
- cohesion and coupling in, Cohesion and Coupling in the Domain
- context mapping, Context Mapping-Context Mapping
- decision areas, Decision Areas of Software Architecture-Dealing with Essential and Accidental Complexity
- documenting in code, Documenting Architecture in Code-Documenting Architecture in Code
- domain events, Domain Events-Domain Events, Chaotic Exploration-Enforcing the Timeline, Challenge: Introducing Domain Events Between Services-Challenge: Introducing Domain Events Between Services

domain model (see domain model)

goals of, Goals of the Right Size-Indicators for the Right Size

layered architecture (see layered architecture)

strategic design (see strategic design)

tactical design, Fundamentals, Tactical Design-Tactical Design, Inside-Outside Architectures and Tactical DDD-Inside-Outside Architectures and Tactical DDD

domain events, Domain Events-Domain Events, Chaotic Exploration-Enforcing the Timeline, Challenge: Introducing Domain Events Between Services-Challenge: Introducing Domain Events Between Services

domain experts and developers

dividing software systems to support, Preface

domain modeling, The Domain Model

gathering, Challenge: Gathering the Right Domain Experts

groups of, Indicators for Subdomain Boundaries

importance as creators of domain model, Collaborative Modeling and DDD

leading, Challenge: Leading Domain Experts Back to the Actual Domain-Challenge: Leading Domain Experts Back to the Actual Domain

model monopoly issue in CoMo, Modeling Tools and the Model Monopoly-Modeling Tools and the Model Monopoly

transformation process for container port example, Comparing Actual Architecture with Target Architecture-Prioritization and Implementation of the Transformation

and ubiquitous language, Ubiquitous Language, Collecting domain terms, Pollution of domain language with technological terms-English or the local language?

domain knowledge

adding, Reuse, duplication, and redundancy

AI tools supporting, Does the AI have Knowledge About My Domain?

enriching modeling with, Enrich Modeling with Domain Knowledge-Enrich Modeling with Domain Knowledge

purification of, Substep 1.2: Extracting the Domain Knowledge (Purification)-Distilling Ubiquitous Language

refactorings that strengthen, Tactical Refactorings That Strengthen Domain Knowledge-Mechanics

strengthening source code with, Strengthening the Source Code with Domain Knowledge-Where Do You Stand?—Made Easy

domain language (see ubiquitous language)

Domain layer, The Layered Architecture in DDD

domain logic, moving out of services, Challenge: Move Domain Logic Out of Services and into the Anemic Domain Model-Push Down Logic from Services into Anemic Entities

domain model, The Domain Model-Reuse, duplication, and redundancy

and bounded contexts, Domain model and bounded contexts-Domain model and bounded contexts

central versus diverse, Central Domain Item Versus Diverse Domain Items-Central Domain Item Versus Diverse Domain Items

decision areas, Decision Areas of Software Architecture-Dealing with Essential and Accidental Complexity

enriching with domain knowledge, Enrich Modeling with Domain Knowledge-Enrich Modeling with Domain Knowledge

finding in EventStorming, Finding the Domain Model-Finding the Domain Model

finding with Domain Storytelling, Finding the Domain Model

form-based, Form-Based Domain Model-Form-Based Domain Model

only one ubiquitous language per model, Collecting domain terms

pure modeling, Alternative: Modeling Pure in the First Place

reuse, duplication, and redundancy, Reuse, duplication, and redundancy-Reuse, duplication, and redundancy

in software, Dimension 2: The Domain Model in the Software-Form-Based Domain Model

structural similarity between domain and source code, The Domain Model

and target architecture, Step 2—Modeling the Domain-Driven Target Architecture-Putting It All Together

domain modeler, Overview of Strategic Transformation

architectural role, Step 2—Modeling the Domain-Driven Target Architecture

domain rediscovery role, Step 1—Domain Rediscovery, Challenge: Leading Domain Experts Back to the Actual Domain

domain modules (see modules)

domain patterns, Outlook: Domain Patterns and Their Implementation in Bounded Contexts-Room for Transformation

domain rediscovery, Step 1—Domain Rediscovery-Discussing Decomposition with AI, Summarizing Domain-Driven Transformation

container port legacy system, Domain Rediscovery-Domain Rediscovery

gathering domain experts, Challenge: Gathering the Right Domain Experts

schema of, Schematic Description of Step 1-Schematic Description of Step 1

domain service building block, Tactical Design

domain shape, Dimension 3: Shape of the Domain-Room for Transformation

Domain Storytelling, Domain Rediscovery, Fundamentals, Domain Storytelling-Outlook

best use of, Which Method to Use When?

container port example, Domain Rediscovery-Domain Rediscovery

digitalized domain stories, Scope Factors for Domain Stories-Scope Factors for Domain Stories

finding domain model, Finding the Domain Model

groups for drawing boundaries, Drawing Boundaries-Drawing Boundaries, Substep 1.4: Finding Subdomains in the Business Processes

pure domain stories, Scope Factors for Domain Stories, Scope Factors for Domain Stories-Finding the Domain Model, The Color Coding

safety compilers example, Technique: Thought Experiment “Virtual Actors”

with Scenario Casting, What's Next in the Scenario-Based Approach?-What's Next in the Scenario-Based Approach?

scenario-based approach (one picture—one story), One Picture—One Story

scope factors, Scope Factors for Domain Stories-Scope Factors for Domain Stories

domain terms, in ubiquitous language, Collecting domain terms-Collecting domain terms, Indicators for Subdomain Boundaries

Domain-Driven Design (DDD), Comparing Actual Architecture with Target Architecture, Fundamentals-Fundamentals, The Fundamental Methods, Domain-Driven Design-Summary

accidental complexity, Sources of Accidental Complexity, Dealing with Essential and Accidental Complexity-Dealing with Essential and Accidental Complexity, DDD Concepts Against Complexity-DDD Concepts Against Complexity, Where Does the Big Ball of Mud Come From?-Where Does the Big Ball of Mud Come From?

cohesion and coupling of modules, Cohesion and Coupling in DDD, Cohesion and Coupling in the Domain, Architectural Styles for Legacy Systems

domain architecture (see domain architecture)

domain modeling (see domain modeling)

and inside-outside architectures, Inside-Outside Architectures and Tactical DDD-Inside-Outside Architectures and Tactical DDD

microservices domain division into bounded contexts, Microservices

object graph, Tactical Design

requirements analysis, Decision Areas of Software Architecture-Dealing with Essential and Accidental Complexity

Domain-Driven Design (Evans), Fundamentals, Mastering Complexity

Domain-Driven Transformation, Introduction: Domain-Driven Transformation in a Nutshell-Summary, Summarizing Domain-Driven Transformation-Summarizing Domain-Driven Transformation

- approaching a legacy system for transformation, How to Approach the Transformation-Summary
- big bang replacement approach, How to Approach the Transformation-Replacement—Big Bang
- container port example, Example: A Legacy System in a Container Port-What Has Just Happened?
- with Kanban, Domain-Driven Transformation with Kanban
- as method for different architectures, The Method-MMI 8–10: Domain-driven modularization
- MMI 8–10 path, MMI 8–10: Domain-driven modularization-MMI 8–10: Domain-driven modularization
- MMI paths through, Paths Through Domain-Driven Transformation-MMI 8–10: Domain-driven modularization
- refactorings (see refactorings)
- reshaping approach to transformation, Reshaping-Reshaping
- step-by-step replacement approach, Replacement—Step-by-Step-Replacement—Step-by-Step
- strategic (see strategic transformation)
- strategic design's role in (see strategic design)
- tactical transformation, MMI 0–4: Big ball of mud, Path A: Tactical transformation first, Path A or path B?

double-diamond design, Design Thinking, Collaborative Modeling Methods for Transformation

DRY principle, Reuse, duplication, and redundancy

duplicate work, transformation approaches to avoid, Replacement—Big Bang, Replacement—Step-by-Step, Reshaping

duplicated code, removing from legacy systems, Cleaning Up the Code as You Go

duplicating data, Challenge: Duplicating Data, Setters, and Getters into Multiple Contexts-Challenge: Duplicating Data, Setters, and Getters into Multiple Contexts

duplication and reuse, domain model, Reuse, duplication, and redundancy-Reuse, duplication, and redundancy

E

Eiffel, Introduce Design by Contract

emergence, Design Thinking, Enforcing the Timeline

empty lists, protecting against, Introduce Design by Contract

enabling team, Types of Topologies

enforcing timeline, EventStorming, Enforcing the Timeline-Enforcing the Timeline

English or local language question, ubiquitous language, English or the local language?-English or the local language?

entities

- extracting from smart UI, Motivation
- extracting specialized, Motivation

identity of, Identity of Entities

making identity explicit, Make the Identity of Entities Explicit-Make the Identity of Entities Explicit

moving logic from service to, Motivation

entity building block, Tactical Design

error robustness, improving in legacy systems, Improving Error Robustness-Introduce Design by Contract

errors (see bugs in legacy software)

essential complexity, Type of Complexity: Essential Versus Accidental-Type of Complexity: Essential Versus Accidental, Dealing with Essential and Accidental Complexity

Evans, Eric, Fundamentals, Mastering Complexity, Cohesion and Coupling in DDD, Introduce Design by Contract

Event Sourcing, Domain Events

event-driven flow, CoMo, EventStorming

EventStorming, Fundamentals, EventStorming-Other Formats and Outlook

adding people and systems, People and Systems

best use of, Which Method to Use When?

Big Picture format, Exploring with Big Picture EventStorming

chaotic exploration, Chaotic Exploration

color coding, The Color Coding-The Color Coding

dividing domain into subdomains, Drawing Boundaries, Substep 1.4: Finding Subdomains in the Business Processes

enforcing timeline, Enforcing the Timeline-Enforcing the Timeline

explicit walkthrough, Explicit Walkthrough

finding domain model, Finding the Domain Model-Finding the Domain Model

other formats, Other Formats and Outlook

with Scenario Casting, What's Next in the Scenario-Based Approach?-What's Next in the Scenario-Based Approach?

Software Design EventStorming, Deeper into the Details: Software Design EventStorming-Deeper into the Details: Software Design EventStorming

value-added steps, Value-Added Steps

Example Mapping, CoMo, Collaborative Modeling Methods for Transformation

exceptions, improving use of, Improve the Use of Exceptions-Improve the Use of Exceptions

executing modernization step, AMET, Architecture Modernization Enabling Teams

Explicit Architecture (Graca), Different Styles Together—Explicit Architecture-Different Styles Together—Explicit Architecture

explicit mapping of patterns, MMI metric, Pattern Consistency in the MMI

explicit walkthrough, EventStorming, Explicit Walkthrough

external environment errors, Improve the Use of Exceptions

Extract Bounded Context refactoring, Motivation

Extract Method refactoring, Push Down Logic from Services into Anemic Entities

Extract Specialized Anemic Entity refactoring, Motivation

Extract Specialized Entity refactoring, Motivation

Extract Specialized Service refactoring, Motivation

Extract Specialized Table refactoring, Motivation

F

facilitating team interaction mode, Team Interaction Modes

factory building block, Tactical Design

Feathers, Michael, Increase Test Coverage When Fixing Bugs, Breaking Dependencies-Breaking Dependencies, Substep 4.2: Adding Tactical Refactorings to the Product Backlog (or Improvement Backlog)

findings

- compiling, Substep 3.3: Compiling Refactorings and Findings-Substep 3.3: Compiling Refactorings and Findings
- selecting, Challenge: Selecting Findings at the Appropriate Moment

finish of modernization step, AMET, Architecture Modernization Enabling Teams

form-based domain model, Form-Based Domain Model-Form-Based Domain Model

Fowler, Martin, Refactoring, Enrich Modeling with Domain Knowledge, Documenting Architecture in Code

framework team, Special case framework team-Special case framework team

free bounded context, Context Mapping

frontend team, Software as a Sociotechnical System-Vertical Architecture/Team Organization

fully digitalized domain, Tangible Objects Domain Versus Fully Digitalized Domain

G

Gang of Four, Tactical Design

generic subdomain, Generic and supporting subdomain, Substep 2.2: Categorizing Subdomains

genesis status, Wardley Maps, Age and Maturity of the Domain

getter methods (getters), Enrich Modeling with Domain Knowledge, Challenge: The Unbounded Anemic Domain Model, Challenge: Moving Local Data, Setters, and Getters into a Context-Challenge: Moving Local Data, Setters, and Getters into a Context, Challenge: Duplicating Data, Setters, and Getters into Multiple Contexts-Challenge: Duplicating Data, Setters, and Getters into Multiple Contexts

Git for version control, Automating Build and Deployment

GitHub Copilot, Does the AI have Knowledge About My Domain?

glossaries, ubiquitous language, Glossaries

goals/reason element, strategic transformation

- in actual versus target architecture, Schematic Description of Step 3
- in domain rediscovery, Schematic Description of Step 1
- for prioritizing and implementing transformation, Schematic Description of Step 4
- in target architecture, Schematic Description of Step 2

Graca, Herberto, Different Styles Together—Explicit Architecture-Different Styles Together—Explicit Architecture

granularity scope factor, domain stories, Scope Factors for Domain Stories

Green Urban Supplier case study, Introducing a Case Study: Green Urban Supplier

groups in domain stories, Drawing Boundaries-Drawing Boundaries

H

Hexagonal Architecture, Tactical Design, Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More, Different Styles Together—Explicit Architecture

hierarchies, Reduce Inheritance in Business Source Code

high cohesion and loose coupling, Cohesion and Coupling in the Domain, Modularity in the MMI

Hoare, Tony, Build Robustness Against Null

horizontal architecture, for teams, Horizontal Architecture/Team Organization-Problems with the horizontal decomposition

hot spots, EventStorming, The Color Coding

I

ID references at aggregate boundaries, Introduce ID References at Aggregate Boundaries-Introduce ID References at Aggregate Boundaries

IDE (integrated development environment), Updating the Integrated Development Environment, Resolving Compiler and IDE Warnings

Implement Bounded Context from Scratch refactoring, Motivation

Implementing Domain-Driven Design (Vernon), Fundamentals

improvement backlog, adding refactoring to, Substep 4.2: Adding Tactical Refactorings to the Product Backlog (or Improvement Backlog)

information hiding, Modularity, Motivation

Infrastructure as Code (IaC), Automating Build and Deployment, Updating the Programming Language

inheritance relationships, Separating Business Source Code and Technical Source Code, Reduce Inheritance in Business Source Code

inside-outside architectures, Inside-Outside Architectures and Tactical DDD-Inside-Outside Architectures and Tactical DDD

integrated development environment (IDE), Updating the Integrated Development Environment, Resolving Compiler and IDE Warnings

intention-revealing interface, Enrich Modeling with Domain Knowledge

interaction modes, team, Team Interaction Modes-Team Interaction Modes

internal programming errors, Improve the Use of Exceptions, Introduce Design by Contract

invariants, source code, Introduce Design by Contract

is-a relationship, The Liskov Substitution Principle

IT landscapes and SOA, From IT Landscapes to SOA and Back Again-From IT Landscapes to SOA and Back Again

J

Java, Preface

Java 14, Updating the Programming Language

Java 9, Updating the Programming Language

JetBrains AI Assistant, Does the AI have Knowledge About My Domain?

K

Kanban, Domain-Driven Transformation with Kanban

kick-off for modernization step, AMET, Architecture Modernization Enabling Teams

know-how loss, and transformation approaches, Replacement—Big Bang, Replacement—Step-by-Step, Reshaping

knowledge crunching, Collaborative Modeling and DDD

Koch, Jörn, Scenario Casting

Kotlin, Build Robustness Against Null

L

language boundary, Strategic Design on the Business Side

large refactoring, Substep 4.2: Adding Tactical Refactorings to the Product Backlog (or Improvement Backlog)

late deployment, and transformation approaches, Replacement—Big Bang, Replacement—Step-by-Step, Reshaping

layer teams, Horizontal Architecture/Team Organization, Motivation, Motivation, Motivation

layered architecture, Comparing Actual Architecture with Target Architecture, Architectural Styles for Legacy Systems

- decision areas, Decision Areas of Software Architecture-Dealing with Essential and Accidental Complexity
- in DDD, The Layered Architecture in DDD
- MMI 4–8 path, MMI 4–8: Technical layered architecture-Path A or path B?

legacy systems

- AI-aided transformation, AI-Aided Transformation

approaching for transformation, How to Approach the Transformation-Summary

common problems with, Technical, Tactical, and Team-Organizational Domain-Driven Transformation- Technical, Tactical, and Team-Organizational Domain-Driven Transformation

complexity in, Complexity in Legacy Systems

decomposing (see decomposing legacy systems)

replacement approach, How to Approach the Transformation-Replacement—Step-by-Step

reshaping approach, Reshaping-Reshaping

software architecture and styles of, Architectural Styles for Legacy Systems

strengthening with domain knowledge, Strengthening the Source Code with Domain Knowledge-Where Do You Stand?—Made Easy

team organization to decompose, Improving Team Organization-Where Do You Stand?—Made Easy

technical stabilization, MMI 0–4: Big ball of mud, Technical Stabilization of Legacy Software-Summary

libraries and frameworks, updating, Updating Libraries and Frameworks

Lilienthal, Carola, Break Cycles and Dependencies

link seam, breaking dependencies, Breaking Dependencies

Liskov Substitution Principle (LSP), The Liskov Substitution Principle-The Liskov Substitution Principle

Liskov, Barbara, The Liskov Substitution Principle

local data, moving, Challenge: Moving Local Data, Setters, and Getters into a Context-Challenge: Moving Local Data, Setters, and Getters into a Context

loose coupling, Cohesion and Coupling in the Domain, Architectural Styles for Legacy Systems, Modularity in the MMI

M

Martin, Robert C., Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More, The Liskov Substitution Principle

metrics for improving legacy internal structure, Improving Internal Structure Using Metrics-Improving Internal Structure Using Metrics

Meyer, Bertrand, Introduce Design by Contract, Introduce Design by Contract

microfrontends, UI in Context: Microfrontends-UI in Context: Microfrontends

microservices, Fundamentals, Monolith, Microservices-From Monolith to Microservices, Service-Oriented Architectures-Service-Oriented Architectures

mixed sprints, Substep 4.3: Planning Achievable Refactorings into the Sprint Backlog

model monopoly, CoMo, Modeling Tools and the Model Monopoly-Modeling Tools and the Model Monopoly

modeling, relationship to programming, Domain-Driven Design

(see also domain model)

Modularity Maturity Index (MMI), The Modularity Maturity Index: Where Do You Stand?-Calculation of the MMI

calculation of, Calculation of the MMI-Calculation of the MMI

container port legacy software example, Comparing Actual Architecture with Target Architecture-Comparing Actual Architecture with Target Architecture

cycles and dependencies in, Break Cycles and Dependencies

hierarchy category, Hierarchy in the MMI

modularity category, Modularity in the MMI

paths in, Paths Through Domain-Driven Transformation-MMI 8–10: Domain-driven modularization

MMI 0–4, big ball of mud, MMI 0–4: Big ball of mud-MMI 0–4: Big ball of mud

MMI 4–8, technical layered architecture, MMI 4–8: Technical layered architecture-Path A or path B?

MMI 8–10, domain-driven modularization, MMI 8–10: Domain-driven modularization-MMI 8–10: Domain-driven modularization

pattern consistency category, Pattern Consistency in the MMI

modularity, in software architecture, Modularity, Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More-Inside-Outside Architectures and Tactical DDD, The Modularity Maturity Index: Where Do You Stand?-Calculation of the MMI

modules, Tactical Design-Tactical Design, Modularity, Inside-Outside Architectures and Tactical DDD-Inside-Outside Architectures and Tactical DDD

(see also bounded contexts)

monolithic systems, Monolith, From Monolith to Microservices-From Monolith to Microservices

monolithic UI, Monolithic UI

Move Method refactoring, Push Down Logic from Services into Anemic Entities

mutually dependent bounded contexts and teams, Context Mapping

N

narrator, EventStorming, Explicit Walkthrough

notation for domain stories, Domain Storytelling

null references, building robustness against, Build Robustness Against Null

NullPointerException, Build Robustness Against Null

NullReferenceException, Build Robustness Against Null

O

object graph, DDD, Tactical Design

object seam, breaking dependencies, Breaking Dependencies

one-sided sprints, Substep 4.3: Planning Achievable Refactorings into the Sprint Backlog

Onion Architecture, Tactical Design, Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More, Different Styles Together—Explicit Architecture

open host service (OHS) context mapping pattern, Context Mapping

optimizing build process, legacy system, Automating Build and Deployment

overall view, Challenge: Unbounded Models

overlooked functionality, and approach to transformation, Replacement—Big Bang, Replacement—Step-by-Step, Reshaping

P

Pais, Manuel, How to Perform This Team Reorganization?

Palermo, Jeffrey, Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More

paper process thought experiment, Technique: Thought Experiment “Paper Process”-Technique: Thought Experiment “Paper Process”

Parnas, David, Modularity

partnership context mapping pattern, Context Mapping

people, EventStorming color coding, The Color Coding

percent proportion MMI metric

- classes and packages in cycles, Hierarchy in the MMI
- mapping of source code to a pattern, Pattern Consistency in the MMI
- violations in relationships between bounded contexts, Hierarchy in the MMI

percentage of violations in technical layering, MMI metric, Hierarchy in the MMI

Perforce for version control, Automating Build and Deployment

PHP 8.1 and 8.2, Updating the Programming Language

Pipeline, in workflow in the domain, Dimension 1: Workflow in the Domain-Domain Pattern Pipeline

pivotal events, EventStorming, Enforcing the Timeline, Substep 1.4: Finding Subdomains in the Business Processes

platform team, Types of Topologies

Plöd, Michael, Cohesion and Coupling in DDD

point-in-time scope factor, domain stories, Scope Factors for Domain Stories

pollution of domain language with technological terms, Pollution of domain language with technological terms-Pollution of domain language with technological terms

polymorphism, and breaking dependencies, Breaking Dependencies

Ports & Adapters Architecture, Tactical Design, Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More, Different Styles Together—Explicit Architecture

postconditions, source code, Introduce Design by Contract

preconditions, source code, Introduce Design by Contract

primitive types, Utilize Value Objects, Motivation

prioritizing and implementing transformation, Prioritization and Implementation of the Transformation, Step 4—Prioritization and Implementation of the Transformation Measures-Challenge: Keeping the Stamina, Summarizing Domain-Driven Transformation

problem space, Origin of Complexity: Problem and Solution Space-Origin of Complexity: Problem and Solution Space

problem-inherent complexity, Origin of Complexity: Problem and Solution Space-Origin of Complexity: Problem and Solution Space, Challenge: Leading Domain Experts Back to the Actual Domain

Process Modeling EventStorming, EventStorming, Other Formats and Outlook

product backlog, adding refactoring to, Substep 4.2: Adding Tactical Refactorings to the Product Backlog (or Improvement Backlog)

product status, Wardley Maps, Age and Maturity of the Domain

programming language, updating, Updating the Programming Language

programming, relationship to modeling, Domain-Driven Design

published language context mapping pattern, Context Mapping

pure domain stories, Scope Factors for Domain Stories, Scope Factors for Domain Stories-Finding the Domain Model, The Color Coding

purification of domain processes, Substep 1.2: Extracting the Domain Knowledge (Purification)-Distilling Ubiquitous Language

Q

quality assurance, Substep 4.4: Extracting Context

Quality Software Architecture (Quasar), Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More

R

redundancy and reuse, domain model, Reuse, duplication, and redundancy-Reuse, duplication, and redundancy

refactorings

- adding to product backlog, Substep 4.2: Adding Tactical Refactorings to the Product Backlog (or Improvement Backlog)
- anemic domain model, Enrich Modeling with Domain Knowledge-Enrich Modeling with Domain Knowledge
- bounded contexts in, Comparing Actual Architecture with Target Architecture, Substep 4.4: Extracting Context, Motivation, Motivation
- catalog of, Overview of the Catalog of Domain-Driven Refactorings
- compiling, Substep 3.3: Compiling Refactorings and Findings-Substep 3.3: Compiling Refactorings and Findings

container port example, Comparing Actual Architecture with Target Architecture

for database queries, Separating Business Source Code and Technical Source Code

estimating, Estimating Refactorings

improvement backlog, Substep 4.2: Adding Tactical Refactorings to the Product Backlog (or Improvement Backlog)

pausing in emergencies, Let the Refactoring Pause Only in Emergencies

planning in sprint backlog, Substep 4.3: Planning Achievable Refactorings into the Sprint Backlog-Domain-Driven Transformation with Kanban

and reshaping approach to transformation, Reshaping-Reshaping

selecting, Substep 4.1: Selecting the First Strategic Refactoring-Pattern: Proceed with a Core Domain

sociotechnical, Sociotechnical Refactorings-Mechanics

strategic, Substep 3.3: Compiling Refactorings and Findings, Strategic Refactorings-Mechanics

tactical, Tactical Refactorings That Support Strategic Refactorings-Access Program Mechanics, Tactical Refactorings That Strengthen Domain Knowledge-Mechanics

for team reorganization, How to Perform This Team Reorganization?

in technical stabilization of legacy software, Refactoring-Improving Internal Structure Using Metrics

for user interfaces, Separating Business Source Code and Technical Source Code

relationships

bounded contexts, Hierarchy in the MMI

determining, Substep 2.4: Determining Relationships

inheritance, Separating Business Source Code and Technical Source Code, Reduce Inheritance in Business Source Code

use case classes, Decomposition from Inside to Outside

relaxed versus strict layered architecture, The Good Old Layered Architecture

remote online approach, CoMo, Collaborative Modeling Remote

renaming methods, refactoring approach, Refactoring

repository

as building block in domain model, Tactical Design, Inside-Outside Architectures and Tactical DDD-Inside-Outside Architectures and Tactical DDD

checking in, only without warnings, Resolving Compiler and IDE Warnings

splitting, Motivation-Mechanics

requirements analysis, Decision Areas of Software Architecture-Dealing with Essential and Accidental Complexity

reshaping approach to transformation, Reshaping-Reshaping

responsibilities for program units, and MMI, Modularity in the MMI

results element, strategic transformation

in actual versus target architecture, Schematic Description of Step 3

in domain rediscovery, Schematic Description of Step 1

for prioritizing and implementing transformation, Schematic Description of Step 4

in target architecture, Schematic Description of Step 2

reusability, Problem Cause: Reusability Misunderstood

reuse and coupling, BBoM, Coupling and Reuse

reuse, duplication, and redundancy, domain model, Reuse, duplication, and redundancy-Reuse, duplication, and redundancy

rich domain events, Domain Events

root entity, Tactical Design

S

scenario backlog, The Core Idea

Scenario Casting, Scenario Casting-The Second Round of Scenario Casting, Substep 1.4: Finding Subdomains in the Business Processes

best use of, Which Method to Use When?

Green Urban Supplier case study, Introducing a Case Study: Green Urban Supplier

orientation scenarios, putting into action, What's Next in the Scenario-Based Approach?-What's Next in the Scenario-Based Approach?

second round, The Second Round of Scenario Casting-The Second Round of Scenario Casting

Step 1 brainstorming, Step 1: Brainstorming-Step 1: Brainstorming

Step 2 focusing, Step 2: Focusing-Step 2: Focusing

Step 3 assembling, Step 3: Assembling-Step 3: Assembling

workshop flow, Workshop Flow: From Exploration to Implementation-Workshop Flow: From Exploration to Implementation

scenario-based approach (one picture—one story) (see Domain Storytelling)

scope factors, Domain Storytelling, Scope Factors for Domain Stories-Scope Factors for Domain Stories

scout rule for fixing bugs, The Scout Rule (Formerly Known as the Boy Scout Rule)-The Scout Rule (Formerly Known as the Boy Scout Rule), Refactoring

scribe, EventStorming, Explicit Walkthrough

seam model for breaking dependencies, Breaking Dependencies-Breaking Dependencies

security gaps, closing in legacy systems, Updating Software Dependencies

Self-contained Systems (SCS), Self-Contained Systems, From IT Landscapes to SOA and Back Again

separate ways context mapping pattern, Context Mapping

sequence numbers, domain stories, Domain Storytelling

server application, C4 Model, Level 3: Components

service-oriented architecture (SOA), Architectural Styles for Legacy Systems, Service-Oriented Architectures-From IT Landscapes to SOA and Back Again

services

business logic in, Challenge: Disentangling Business Logic in Services-Challenge: Disentangling Business Logic in Services

domain events between, Challenge: Introducing Domain Events Between Services-Challenge: Introducing Domain Events Between

Services

extracting from smart UI, Motivation

extracting specialized, Motivation

moving domain logic out of, Challenge: Move Domain Logic Out of Services and into the Anemic Domain Model-Push Down Logic from Services into Anemic Entities

moving logic to entity from, Motivation

setter methods (setters), Enrich Modeling with Domain Knowledge, Challenge: The Unbounded Anemic Domain Model, Challenge: Moving Local Data, Setters, and Getters into a Context-Challenge: Moving Local Data, Setters, and Getters into a Context, Challenge: Duplicating Data, Setters, and Getters into Multiple Contexts-Challenge: Duplicating Data, Setters, and Getters into Multiple Contexts

shadow IT, Replacement—Big Bang

shared kernel context mapping pattern, Context Mapping

Siedersleben, Johannes, Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More

size metrics, improving system implementation with, Improving Internal Structure Using Metrics

Skelton, Matthew, How to Perform This Team Reorganization?

SOA (service-oriented architecture), Architectural Styles for Legacy Systems, Service-Oriented Architectures-From IT Landscapes to SOA and Back Again

sociotechnical refactorings, Sociotechnical Refactorings-Mechanics

sociotechnical system, software as, Software as a Sociotechnical System-How to Perform This Team Reorganization?

software architecture concepts

- BBoM (see big ball of mud)
- C4 Model for documenting architecture, Architecture Documentation with the C4 Model-Level 4: Code
- cohesion and coupling, Cohesion and Coupling-Cohesion and Coupling in the Domain
- inside-outside architectures, Inside-Outside Architectures and Tactical DDD-Inside-Outside Architectures and Tactical DDD
- IT landscapes to SOA and back, From IT Landscapes to SOA and Back Again-From IT Landscapes to SOA and Back Again
- legacy system styles, Architectural Styles for Legacy Systems
- microservices, Microservices-From Monolith to Microservices
- modularity and MMI, Modularity, Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More-Inside-Outside Architectures and Tactical DDD, The Modularity Maturity Index: Where Do You Stand?-Calculation of the MMI
- monolithic systems, Monolith, From Monolith to Microservices-From Monolith to Microservices
- Self-contained Systems, Self-Contained Systems
- service-oriented architectures, Architectural Styles for Legacy Systems, Service-Oriented Architectures-Service-Oriented Architectures

Software Architecture Hamburger, Software Architecture Hamburger-Software Architecture Hamburger

Software Design EventStorming, EventStorming, Deeper into the Details: Software Design EventStorming-Deeper into the Details: Software Design

EventStorming

SOLID principles, Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More

solution space, Origin of Complexity: Problem and Solution Space-Origin of Complexity: Problem and Solution Space, DDD and the Software Solution-Domain Events, Challenge: Solution-Space Induced Contexts-Challenge: Solution-Space Induced Contexts

solution-dependent complexity, Origin of Complexity: Problem and Solution Space-Origin of Complexity: Problem and Solution Space, Challenge: Leading Domain Experts Back to the Actual Domain

Sonargraph, Comparing Actual Architecture with Target Architecture, At the Class Level

SonarQube, Improving Internal Structure Using Metrics

source code

- business versus technical, Separating Business Source Code and Technical Source Code-Separating Business Source Code and Technical Source Code
- and Design by Contract, Introduce Design by Contract-Introduce Design by Contract
- documenting domain architecture in, Documenting Architecture in Code-Documenting Architecture in Code
- strengthening with domain knowledge, Strengthening the Source Code with Domain Knowledge-Where Do You Stand?—Made Easy
- structural similarity with domain code, The Domain Model
- versus context map, Substep 3.2: Comparing the Target Context Map and the Source Code-Substep 3.2: Comparing the Target Context Map and the Source Code

Spolsky, Joel, Improve the Use of Exceptions

sprint backlog, planning refactorings in, Substep 4.3: Planning Achievable Refactorings into the Sprint Backlog-Domain-Driven Transformation with Kanban

standalone class, and cohesion and coupling, Cohesion and Coupling in the Domain

standstill for users, reshaping approach to transformation, Reshaping

standstill on one hand/moving target on the other, Replacement—Big Bang, Replacement—Step-by-Step

step-by-step replacement approach to transformation, Replacement—Step-by-Step-Replacement—Step-by-Step

sticking to old solutions, as reshaping transformation's drawback, Reshaping

Storystorming, CoMo, Collaborative Modeling Methods for Transformation

Strangler Fig Application, Replacement—Step-by-Step

strategic design, Comparing Actual Architecture with Target Architecture, Fundamentals

    bounded context, Bounded context

    business side, Strategic Design on the Business Side-Strategic Design on the Business Side

    context map, Context map

    domain modeling, Strategic Design on the Business Side-Strategic Design on the Business Side

    in the software, Strategic Design in the Software-Context map

strategic refactorings, Substep 3.3: Compiling Refactorings and Findings, Strategic Refactorings-Mechanics

strategic transformation, Strategic Domain-Driven Transformation-The Structure in This Part of the Book, Summarizing Domain-Driven Transformation

- alignment of actual to target architecture (see actual versus target architecture)
- domain rediscovery (see domain rediscovery)
- MMI 4–8 situation, Path B: Strategical transformation first
- modeling target architecture, Step 2—Modeling the Domain-Driven Target Architecture-Putting It All Together
- prioritizing and implementing transformation, Prioritization and Implementation of the Transformation, Step 4—Prioritization and Implementation of the Transformation Measures-Challenge: Keeping the Stamina, Summarizing Domain-Driven Transformation
- process of, Overview of Strategic Transformation

stream-aligned team, Types of Topologies, Goals of the Right Size

strict versus relaxed layered architecture, The Good Old Layered Architecture

Stroustrup, Bjarne, Mastering Complexity

structural similarity between domain and source code, The Domain Model

Struts user interface library, Updating Libraries and Frameworks

subclasses, The Liskov Substitution Principle

subdomains

- boundary indicators, Indicators for Subdomain Boundaries

categorizing, Substep 2.2: Categorizing Subdomains-Substep 2.2: Categorizing Subdomains

and determining target architecture, Determining Target Architecture-Determining Target Architecture

dividing domain with boundaries, Drawing Boundaries

EventStorming to find, Drawing Boundaries

finding in business processes, Substep 1.4: Finding Subdomains in the Business Processes-Indicators for Subdomain Boundaries

finding in domain stories, Drawing Boundaries-Drawing Boundaries

generic subdomain, Generic and supporting subdomain, Substep 2.2: Categorizing Subdomains

Scenario Casting for finding, What's Next in the Scenario-Based Approach?-What's Next in the Scenario-Based Approach?

strategic design dividing of, Strategic Design on the Business Side-Strategic Design on the Business Side, Strategic Design in the Software

supporting subdomain, Generic and supporting subdomain, Substep 2.2: Categorizing Subdomains

substeps element, strategic transformation

in actual versus target architecture, Schematic Description of Step 3

in domain rediscovery, Schematic Description of Step 1

in target architecture, Schematic Description of Step 2

for prioritizing and implementing transformation, Schematic Description of Step 4

superclass, The Liskov Substitution Principle

supporting subdomain, Generic and supporting subdomain, Substep 2.2: Categorizing Subdomains, Pattern: Start with a Supporting Subdomain

Sustainable Software Architecture (Lilienthal), Break Cycles and Dependencies

T

table, extracting specialized, Motivation

tactical design, Tactical Design-Tactical Design, Inside-Outside Architectures and Tactical DDD-Inside-Outside Architectures and Tactical DDD

tactical refactorings, Substep 3.3: Compiling Refactorings and Findings, Tactical Refactorings That Support Strategic Refactorings-Access Program Mechanics, Tactical Refactorings That Strengthen Domain Knowledge-Mechanics

tactical transformation, MMI 0–4: Big ball of mud, Path A: Tactical transformation first, Path A or path B?

tangible objects domain, Tangible Objects Domain Versus Fully Digitalized Domain

target architecture

- versus actual architecture (see actual versus target architecture)
- determining, Modularity in the MMI
- modeling, Step 2—Modeling the Domain-Driven Target Architecture-Putting It All Together, Summarizing Domain-Driven Transformation
- schema of, Schematic Description of Step 2
- subdomain identification, Determining Target Architecture-Determining Target Architecture

team topologies, Team Topologies-Architecture Modernization Enabling Teams

AMET, Architecture Modernization Enabling Teams-Architecture Modernization Enabling Teams

team evolution over time, Team Evolution Over Time-Team Evolution Over Time

team interaction modes, Team Interaction Modes-Team Interaction Modes

types of, Types of Topologies-Types of Topologies

working in the flow, Working in the Flow

Team Topologies (Skelton and Pais), How to Perform This Team Reorganization?

teams, Origin of Complexity: Problem and Solution Space

(see also domain experts and developers)

and software as a sociotechnical system, Software as a Sociotechnical System-How to Perform This Team Reorganization?

assigning bounded contexts to, Substep 2.3: Assigning Bounded Contexts to Teams-Substep 2.3: Assigning Bounded Contexts to Teams

complexity issue for, Dealing with Essential and Accidental Complexity

structure issues, Improving Team Organization-Where Do You Stand?—Made Easy

technical stabilization of legacy software, MMI 0–4: Big ball of mud, Technical Stabilization of Legacy Software-Summary

automating build and deployment, Automating Build and Deployment-Automating Build and Deployment

cleaning up code as you go, Cleaning Up the Code as You Go

dependency updating of software, Updating Software Dependencies-Resolving Compiler and IDE Warnings

error robustness, improving, Improving Error Robustness-Introduce Design by Contract

IDE updating, Updating the Integrated Development Environment

refactoring, Refactoring-Improving Internal Structure Using Metrics

test coverage for fixing bugs, Increase Test Coverage When Fixing Bugs-Different Kinds of Tests

technical versus business source code, Separating Business Source Code and Technical Source Code-Separating Business Source Code and Technical Source Code

terminal operating system (TOS), Example: A Legacy System in a Container Port-Example: A Legacy System in a Container Port

test coverage for fixing bugs in legacy software, Increase Test Coverage When Fixing Bugs-Different Kinds of Tests

thin domain events, Domain Events

thinnest viable platform (TVP), Types of Topologies

thought experiments, Technique: Thought Experiment “Paper Process”-Technique: Thought Experiment “Virtual Actors”

tidyings (cleanup of code), Cleaning Up the Code as You Go

timeline enforcing, EventStorming, Enforcing the Timeline-Enforcing the Timeline

to-be business processes, Step 1—Domain Rediscovery

to-be domain stories, Scope Factors for Domain Stories

tools element, strategic transformation

in actual versus actual architecture, Schematic Description of Step 3

in domain rediscovery, Schematic Description of Step 1

in target architecture, Schematic Description of Step 2

for prioritizing and implementing transformation, Schematic Description of Step 4

TOS (terminal operating system), Example: A Legacy System in a Container Port-Example: A Legacy System in a Container Port

Tune, Nick, Architecture Modernization Enabling Teams, Architecture Modernization Enabling Teams

TVP (thinnest viable platform), Types of Topologies

U

U-shaped architecture, Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More

ubiquitous language, Ubiquitous Language-English or the local language?, Distilling Ubiquitous Language

collecting domain terms, Collecting domain terms-Collecting domain terms

enforcing, Motivation

English or local language question, English or the local language?-English or the local language?

glossaries, Glossaries

pollution of domain language with technological terms, Pollution of domain language with technological terms-Pollution of domain language with technological terms

in source code, Enrich Modeling with Domain Knowledge

as specific to domain model, Domain model and bounded contexts

umbrella refactoring, Motivation

unbounded models, Challenge: Unbounded Models-Decomposition from Inside to Outside

anemic domain model, Challenge: The Unbounded Anemic Domain Model-Challenge: The Unbounded Anemic Domain Model

context problem, Domain model and bounded contexts

data model, Challenge: Unbounded Models, Challenge: The Unbounded Data Model

domain model, Comparing Actual Architecture with Target Architecture, Challenge: Unbounded Models, Decomposition from Inside to Outside-Challenge: The Unbounded Domain Model

services, Challenge: Unbounded Models

unexpected behavior types in code, Improve the Use of Exceptions

unit tests, Different Kinds of Tests-Different Kinds of Tests, Substep 4.4: Extracting Context

unknown unknowns, and approach to transformation, Replacement—Big Bang, Replacement—Step-by-Step, Reshaping

upstream-downstream bounded context, Context Mapping

user interface

access to in business logic, Separating Business Source Code and Technical Source Code

decomposing, Challenge: Decomposing the User Interface-UI in Context: Microfrontends

User Story Mapping, CoMo, Collaborative Modeling Methods for Transformation

users

frustrations with big bang replacement approach, Replacement—Big Bang

impact of reshaping approach, Reshaping

impact of step-by-step replacement approach, Replacement—Step-by-Step

V

validated narrative, EventStorming, Explicit Walkthrough

value objects, Tactical Design, Cohesion and Coupling in DDD, Utilize Value Objects-Utilize Value Objects, Motivation

value-added steps, EventStorming, Value-Added Steps

Vernon, Vaughn, Fundamentals

version changes, programming, Updating the Programming Language

version control systems, applying modern, Automating Build and Deployment

vertical architecture, for teams, Vertical Architecture/Team Organization-Vertical Architecture/Team Organization

virtual actors thought experiment, Technique: Thought Experiment “Virtual Actors”-Technique: Thought Experiment “Virtual Actors”

visual legend, EventStorming stickies as, The Color Coding

W

Wardley Maps, Age and Maturity of the Domain

waterfall model, Origin of Complexity: Problem and Solution Space

well-balanced proportions for program units, MMI, Modularity in the MMI

whole value, Tactical Design, Utilize Value Objects

work object, domain stories, Domain Storytelling

workflow patterns in the domain, Evolution of Domains and the Patterns They Follow-Evolution of Domains and the Patterns They Follow

- Blackboard, Domain Pattern Blackboard-Domain Pattern Blackboard
- Dialog, Domain Pattern Dialog-Domain Pattern Dialog
- Pipeline, Dimension 1: Workflow in the Domain-Domain Pattern Pipeline

Working Effectively with Legacy Code (Feathers), Breaking Dependencies, Substep 4.2: Adding Tactical Refactorings to the Product Backlog (or Improvement Backlog)

workshops, CoMo (see EventStorming)

X

X as a service team interaction mode, Team Interaction Modes

xMolecules libraries, Documenting Architecture in Code

Z

Züllighoven, Heinz, Inside-Outside Architectures: Hexagonal/Ports & Adapters, Onion, Clean, and More

About the Authors

Carola Lilienthal is CEO of WPS – Workplace Solutions GmbH in Hamburg, where she is responsible for the field of software architecture. She received her master's degree in computer science from the University of Hamburg in 1995, where she wrote her doctoral thesis on the complexity of software architectures. Since 2003, she has been analyzing the architecture of software systems internationally on behalf of her customers and advising development teams on how they can improve the sustainability of their software systems. In total, Carola and her team have analyzed over 300 software systems with between 30,000 and 15 million LOC in Java, C++, C#, ABAP, PHP, and TypeScript.

Carola is the author of *Sustainable Software Architecture* (dpunkt). She is particularly interested in the education of software architects and developers, which is why she is an active member of iSAQB, the International Software Architecture Qualification Board, and regularly shares her knowledge from over 25 years of software development at conferences, in articles, and at training courses.

Henning Schwentner is a programmer who has been into computers ever since he got an Amiga 500 in the early '90s. He was lucky enough to turn this passion into a profession and works as a coder, coach, and consultant at WPS – Workplace Solutions. He helps teams bring structure to their existing software or build new systems with a sustainable architecture from scratch.

Henning is the author of *Domain Storytelling* (Addison-Wesley) and LeasingNinja.io and a co-organizer of ComoCamp. He is the proud father of six children in a very special patchwork situation.

Colophon

The animal on the cover of *Domain-Driven Transformation* is an Atlas moth (*Attacus atlas*), one of the largest tropical silkworm moths. They are native to Southeast Asia and the Malay archipelago.

The moths are named after Atlas, the Greek Titan, due to their size. The wingspan of the Atlas moth measures up to 9.4 inches and is only surpassed by *Thysania agrippina* and *Attacus caesar*. Females are larger and heavier than males, while males have broader antennae with chemoreceptors that help them locate females. They have reddish-brown wings, each of which has a pale triangular marking bordered in black. The tip of each forewing has a protrusion that resembles the head of a snake. When threatened, the moth moves in such a way that the appearance of two snake heads can scare off predators.

Female Atlas moths lay 130–170 eggs on the underside of leaves. After two weeks, green caterpillars hatch. They have white, waxy spines along their backs and a bright orange ring on the abdomen above the final set of prolegs. The caterpillars first eat their egg shells and then the foliage on which they hatched. They eat constantly during this larval stage to build up fat reserves for their adult stage. They grow to 4.5 inches long and approximately 1 inch in thickness.

After 6 or 7 weeks, the caterpillar spins a cocoon and enters the pupal stage. Adult moths emerge about 4 weeks later. The proboscis of the moth is nonfunctional, so they are unable to eat. They survive for 1 to 2 weeks on the food they consumed as caterpillars. To conserve energy, male Atlas moths rest during the day and only fly at night, in search of a mate. Females typically stay near the cocoons they emerged from and wait for a mate to find them. They live the majority of their lives, a span of about 100 days, on a single tree.

The Atlas moth has an uncertain conservation status and has not been formally assessed by the IUCN Red List. Many of the animals on O’Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Monica Kamsvaag, based on an antique line engraving from *Birds and Beasts*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.