Mastering Domain-Driven Design

Collaborative modeling with domain storytelling, event storming, and context mapping

Annegret Junker

www.bpbonline.com

First Edition 2025

Copyright © BPB Publications, India

ISBN: 978-93-65892-529

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:

www.bpbonline.com

Dedicated to

Stefan

and my beloved brother

Bernd

About the Author

Annegret Junker is Chief Software Architect at codecentric AG in Germany, who has an experience in software development for over 30 years. She worked in different roles and domains, such as automotive and insurance. Moreover, she is especially interested in Domain-Driven Design, microservices, and APIs.

Annegret talks regularly at conferences and publishes in leading journals.

She is working on a large integration project and defines APIs for partners. Annegret holds a PhD as an engineer.

About the Reviewer

Dmitrii Porozhniakov is a software engineer with over a decade of experience contributing to impactful projects at leading technology companies. Throughout his career, he has worked in roles that demanded technical expertise and strategic thinking, developing systems that are scalable, efficient, and innovative.

In his current role at Meta, Dmitrii focuses on backend development and cross-team workflows, playing a key part in initiatives that enhance data transparency and service understanding. Previously, he has played pivotal roles in designing robust payment infrastructures, streamlining large-scale data operations, and building scalable architectures that support millions of users. His contributions have consistently addressed complex technical challenges while aligning with broader organizational goals, ensuring seamless performance and long-term reliability.

In addition to his professional achievements, Dmitrii is the author of several articles on software development, sharing insights on architecture, processes, and industry practices. His work reflects a thoughtful and practical approach to solving modern engineering challenges.

Acknowledgement

I want to express my deepest gratitude to my family and friends, especially my brother Bernd and my lifelong friends Stefan and Stefan, for their unwavering support and encouragement as I wrote this book. I wrote it especially for my late husband, Stefan, who I will always miss.

Moreover, I am grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. Revising this book was a long journey, with valuable participation and collaboration of reviewers, technical experts, and editors.

I would also like to acknowledge the valuable contributions of my colleagues and co-workers during many years working in the tech industry, who have taught me so much and provided helpful feedback on my work. Moreover, I would like to thank my colleagues at codecentric AG for their support and the time they offered me to write this book.

Finally, I would like to thank all the readers who have taken an interest in my book and for their support in making it a reality. Your encouragement has been invaluable.

Preface

Writing software is a challenging task in all phases of design and implementation. It is not alone in writing code; it is about understanding each other. Developers need to understand business experts, business experts need to understand software architects, solution architects need to work with enterprise architects, etc. Moreover, the problems we solve nowadays in software are intrinsically complex. We need to tackle this complexity by applying Domain-Driven Design, as described by Eric Evans over twenty years ago. Meanwhile, we know methodologies and workshop formats that allow us to create and modernize software in a highly collaborative way. A comprehensive process that emphasizes collaboration and engagement of all stakeholders is presented in the book, from the first ideas up to the implementation of events and restful APIs.

This book introduces the process as Synergetic Blueprint and discusses its application using multiple examples.

Chapter 1: Introduction to Domain-Driven Design – This chapter discusses the history of this approach, the complexity of modern software applications, and how to tackle this complexity.

Chapter 2: Introduction to the Example Online Library – This chapter introduces an online library used as an example throughout the book to explain the single steps of the design process.

Chapter 3: Why Strategic Design – This chapter discusses the influence of software architecture on a system and how to achieve team independence. Organizational dependencies are discussed when talking about Conway's law. This discussion leads to how to reach better software and what better software means.

Chapter 4: Bounded Context and Domain – This chapter introduces the terms domain, subdomain, bounded context, and aggregate. The patterns in strategic design are discussed.

Chapter 5: Domain Storytelling – This chapter introduces domain storytelling as a workshop format. It discusses the domain story of the online library and its visual glossary and gives recommendations on how to conduct a workshop.

Chapter 6: Event Storming – This chapter introduces the reader to the event storming workshop format and discusses how to set one up. It explains in detail the steps of an event storming workshop, from finding and consolidating the events to assigning aggregates and reading models.

Chapter 7: Context Map – This chapter introduces the context map as a tool for discussing context map and discuss synchronous and asynchronous interfaces. It also shows the API definitions and how to add interfaces to the context map.

Chapter 8: Overview of Strategic Design – This chapter gives an overview of the strategic part of Domain-Driven Design. It discusses the importance of strategic design and the relationship between design and team dependencies. The chapter also offers tips and tricks for applying strategic design in daily work.

Chapter 9: Introduction to Tactical Design – This chapter gives a first overview of the tactical part of the design. It discusses the role of a model in software development and explains how to evolve the ubiquitous language. It also provides an overview of the tactical pattern of Domain-Driven Design.

Chapter 10: Aggregate, Entity, and Value Object – This chapter discusses the topic in detail and provides a model of the online library. It also provides and discusses the patterns repository, orchestrator, and factories.

Chapter 11: Exposing Aggregates via APIs – This chapter discusses the necessary synchronous APIs of a bounded context and how to define them. It also discusses the properties of a RESTful API and its specification. The theory is applied in practice using the online library as an example.

Chapter 12: Exposing Domain Events – This chapter discusses how to expose the domain events of a bounded context. The events are defined using AsyncAPI. The events of the online library are used as an example. Furthermore, the definition of services is discussed for the events.

Chapter 13: Pitfalls in Tactical Design – This chapter discusses the dependencies between team responsibilities and tactical design. It points to pitfalls and gives hints on how to avoid them.

Chapter 14: Usage of Domain-Driven Design in a Greenfield – This chapter gives a comprehensive overview of the process using the example of a bike rental shop. It offers hints on how to set up the workshops and find the right audience. Using the example, it describes how to document the workshop results and the architectural decisions. Furthermore, the chapter describes how to evolve a DDD process.

Chapter 15: Domain-Driven Design in a Brownfield Project – This chapter discusses the process in a brownfield project using the example of a classic community library. It describes prioritizing the capabilities and applying a step-by-step approach to the necessary modernization. It emphasizes again the required organizational changes.tic

Chapter 16: Summary – This chapter summarizes the content of the previous chapters and comprehensively describes the synergetic blueprint process. It gives an overview of strategic and tactical design and how to apply the method to projects. It also discusses the

Coloured Images

Please follow the link to download the
Coloured Images of the book:

https://rebrand.ly/14c67c

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

1. Introduction to Domain-Driven Design 1

Introduction 1

Structure 1

Objectives 1

Short history of Domain-Driven Design 2

Publication of the blue book 2

Domain models and their shortcomings 2

Microservices require a shift 3

Complexity of modern software systems 5

Handling complexity 5

Software complexity in a VUCA world 7

Tackling complexity 9

Design Thinking 9

Complexity and analysis 10

CAP theorem 14

Commensurate complexity 15

Software architecture 17

Conclusion 19

Points to remember 19

Multiple choice questions 20

Answers 20

2. Introduction to the Example Online Library 21

Introduction 21

Structure 21

Objectives 21

First ideas 22

Business plan model 26

Capability map 27

Conclusion ... 34
Points to remember ... 34
Multiple choice questions ... 34
Answers ... *35*

3. Why Strategic Design ... 37
Introduction ... 37
Structure ... 37
Objectives ... 38
Influence of software architecture in a system ... 38
Team independence ... 39
Conway's Law ... 41
Better software ... 42
Software quality ... *42*
Security ... *44*
Software quality aspects and DDD ... *45*
Conclusion ... 46
Points to remember ... 46
Multiple choice questions ... 47
Answers ... *48*

4. Bounded Context and Domain ... 49
Introduction ... 49
Structure ... 49
Objectives ... 49
Domains ... 50
Subdomains ... 51
Wardley Mapping ... *52*
Purpose ... *52*
Scope ... *52*
Users ... *53*
User needs ... *53*
Value chain ... *54*

Final step in mapping *56*

Distilling core, supportive, and generic domains *59*

Bounded context and aggregate 63

Patterns in strategic design 64

Shared Kernel *64*

Conformist *66*

Anti-Corruption Layer *67*

Open Host Service *68*

Conclusion 69

Points to remember 69

Multiple choice questions 70

Answers *72*

5. Domain Storytelling 73

Introduction 73

Structure 73

Objectives 73

Target of domain storytelling 74

Storytelling *75*

Visual language of domain storytelling *81*

Workshop setup 83

Workshop attendees *83*

Whiteboards *84*

Modeling tools *84*

Workshop length *86*

Enhancing domain storytelling for existing systems 87

Domain storytelling of the online library 88

Purchase of books *88*

Membership management *89*

Reading *90*

Discussion *91*

Recommendations *92*

Visual glossary 93

Recommendations for a domain storytelling workshop 96
Conclusion 97
Points to remember 98
Multiple choice questions 99
Answers 100

6. Event Storming 101

Introduction 101
Structure 101
Objectives 102
Target of an event storming workshop 102
Brainstorming 102
Event storming 103
Setup of the workshop 103
Scheduling 104
Tooling 104
Event storming in detail 105
Collection of events 106
Consolidation of events 107
Finding the commands 109
Roles and processes 110
Finding aggregates and views 111
Finding bounded contexts 112
Event storming of the online library 113
Purchase 114
Member management 115
Membership calculation 118
Visual glossary enhancement 118
Reading 120
Visual glossary enhancement 124
Discussions 125
Visual glossary enhancement 126
Recommendation 127

Bounded context canvas 128

Guidelines in event storming workshops 130

Conclusion 132

Points to remember 133

Multiple choice questions 133

Answers *134*

7. Context Map **135**

Introduction 135

Structure 135

Objectives 136

Context map 136

Creation of a context map 139

Domain events and REST interfaces 140

Synchronous communication *141*

Asynchronous communication *141*

West-east and north-south communication *142*

Applying communication patterns in context map *143*

API definitions 143

Purchase *144*

Member management *144*

Reading *146*

Discussions *148*

Context map and API definition 150

Conclusion 152

Points to remember 152

Multiple choice questions 153

Answers *154*

8. Overview of Strategic Design **155**

Introduction 155

Structure 155

Objectives 155

Overview of strategic design 156

Significance of strategic design 160

Development over the years 160

Development and purchase 161

Define the structure of development 161

Relationship of bounded context and team structure 162

Open host service 162

Anti-corruption layer 162

Conformist 163

Big Ball of Mud 163

Tips and tricks for applying strategic design 164

Definition of business capabilities 164

Training for the teams 164

API-first strategy 164

Governance process 165

Team structure 167

Conclusion 167

Points to remember 168

Multiple choice questions 168

Answers 169

9. Introduction to Tactical Design 171

Introduction 171

Structure 171

Objectives 172

Tactical design 172

Tactical design for modeling 172

Model-driven design 173

Further development of the ubiquitous language 177

Patterns provided by DDD 179

Conclusion 180

Points to remember 180

Multiple choice questions 180

Answers *181*

10. Aggregate, Entity, and Value Object **183**

Introduction 183

Structure 183

Objectives 184

Aggregate, entity, and value object 184

Aggregates *184*

Entity *186*

Value object *187*

Example model of the online library 187

Repository 193

Factories and orchestrators 194

Factories *195*

Orchestrator *196*

Choreography *197*

Conclusion 200

Points to remember 200

Multiple choice questions 201

Answers *201*

11. Exposing Aggregates via APIs **203**

Introduction 203

Structure 203

Objectives 203

Introduction to RESTful APIs and its specification 204

Application programming interface *204*

REST *204*

OpenAPI specification language for RESTful APIs *206*

Specification of the API of task management 208

Schema design *208*

Resource paths *211*

Tasks Get *211*

Tasks Post ... 212
Single task Get ... 213
Single task Delete ... 213
APIs for online library ... 214
API of Catalog Management ... 215
API Member Management ... 215
API of discussions ... 218
Notes API ... 224
Conclusion ... 227
Points to remember ... 227
Multiple choice questions ... 228
Answers ... 229

12. Exposing Domain Events ... 231
Introduction ... 231
Structure ... 231
Objectives ... 231
Introduction to event-driven architecture ... 232
Broker architecture ... 233
Event-streaming ... 234
Publish and subscribe ... 235
Defining events using AsyncAPI ... 235
Structure of an AsyncAPI ... 236
Skeleton of an AsyncAPI specification ... 237
Pure message definitions of the library ... 239
Header of the messages ... 239
Book purchased ... 240
Catalog entry created, catalog entry updated ... 241
Book lent, book returned ... 242
Member created ... 243
Amount paid ... 243
Reading position marked ... 245
Definitions of the services ... 245

Conclusion 246
Points to remember 246
Multiple choice questions 247
Answers *248*
Bibliography 248

13. Pitfalls in Tactical Design 249
Introduction 249
Structure 249
Objectives 249
Team responsibility and tactical design 250
Check implementation consistency by LLMs *250*
Ubiquitous language *253*
Pitfalls and how to avoid them 254
Driven by database design *254*
Purely out of the provider's point of view *254*
Purely technical solutions *255*
Trying to solve someone else's problem *255*
Conclusion 256
Points to remember 256
Multiple choice questions 257
Answers *257*

14. Usage of Domain-Driven Design in a Greenfield 259
Introduction 259
Structure 259
Objectives 260
Setting up a workshop 263
Capability map and Wardley Map *264*
Domain storytelling and event storming *266*
Context map *271*
API design *272*
Finding target audience for the workshops 273

Documenting workshop results 274
  Scope 275
    Qualitative requirements 275
    Stakeholders 276
  Constraints 277
  Context 278
    Capability map 278
    Business context 280
    Technical context 282
  Solution approach 283
Documentation of bounded contexts 285
  Bounded context canvas 285
  Architectural communication canvas 287
  Synchronous API specification 290
  Asynchronous specification 291
Documenting architectural decisions 291
Importance of documenting software 293
Evolving the DDD process 293
Conclusion 296
Points to remember 296
Multiple choice questions 297
  Answers 298

15. Domain-Driven Design in a Brownfield Project 299
Introduction 299
Structure 299
Objectives 300
Overview 300
Renaissance of the domain 303
  Vision statement 303
  Domain storytelling 305
    Purchasing books 306
    Onboarding of new members

Reservation of books 311
Lending of books 313
Context map 318
Wardley Map 319
Target architecture 321
Decoupling layer 324
Benefits of decoupling 324
Decoupling via capabilities 325
Decoupling catalog and lending 326
Modernization layer 327
Step-by-step approach for target architecture 328
Remaining functionality 330
Organizational changes 330
Conclusion 332
Points to remember 333
Multiple choice questions 333
Answers 335

16. Summary **337**
Introduction 337
Structure 337
Objectives 338
Synergetic Blueprint 338
Strategic design 338
Artifacts of the Synergetic Blueprint process 347
Ideation 348
Business planning 349
Requirements gathering 350
Solution design 350
Tactical design 351
Using AI in the process 352
Creating a RESTful API based on a visual glossary 353
Check on consistency 355

Using AI to generate models *356*
Conclusion 358
Points to remember 358
Multiple choice questions 358
Answers *359*

Index 361-367

CHAPTER 1

Introduction to Domain-Driven Design

Introduction

The chapter gives a brief overview about **Domain-Driven Design** (**DDD**). It introduces the readers to the complexity of software architecture and development and shows how DDD can help overcome such obstacles.

This chapter discusses what complexity means in a fast-changing and uncertain world. It explains what software architecture means and which tasks a software architect needs to fulfill and which chapters to refer to according to the tasks.

Structure

In this chapter, we will cover the following topics:

- Short history of Domain-Driven Design
- Complexity of modern software systems
- Tackling complexity

Objectives

By the end of this chapter, you will be able to understand the difficulties a software architect faces in their day-to-day work, what is needed to become a good software architect, and to refer to which chapter in the book to complete the tasks, and how DDD can help.

> **Any intelligent fool can make things bigger and more complex... It takes a touch of genius - and a lot of courage to move in the opposite direction.**
>
> *~E.F. Schumacher*

Short history of Domain-Driven Design

DDD was introduced by *Eric Evans* nearly two decades ago and has since become a fundamental approach to managing complexity in software systems. Subsequently, numerous papers and presentations have expanded on the original concept, introducing new thoughts and methodologies. This chapter will explore both the original principles and their subsequent enhancements. Currently, a variety of patterns and methodologies have evolved from *Eric Evans*' pioneering work.

Publication of the blue book

The term DDD was introduced by *Eric Evans* in his PhD thesis titled *Domain-Driven Design: Tackling Complexity in the Heart of Software*[1]. The publication is referenced by The Blue Book in the community. This concept became a cornerstone in software engineering literature. However, at that time, IT specialists and business experts did not collaborate closely; programmers were often seen merely as order takers, and projects were primarily planned upfront based on data models.

Domain models and their shortcomings

The term, **domain model**, was introduced by *Peter P. Gabriel*[2] , who is also known for inventing the **list processing** (**Lisp**) programming language. This idea was further popularized by *Martin Fowler*, particularly through his work on patterns of enterprise application architecture. A domain model forms a network of interconnected objects, each representing something meaningful and distinct, ranging in scope from an entire company to a single line on an order form[3].

Anyhow, the concept of domain design, already under development since the early 1980s, gained momentum with the advent of modeling languages, like **Unified Modelling Language** (**UML**) and object-oriented programming languages. Object-oriented analysis and design, notably discussed by *Grady Booch*[4] was a pivotal step in fostering a common language among participants in a development project, including developers, architects, and database specialists.

---

[1] Evans, E. (2004). *Domain-Driven Design Tackling Complexity in the Heart of Software.* Upper Saddle River, NJ: Addison-Wesley.

[2] Gabriel, R. P. (1996). *Patterns of Software.* Oxford: Oxford University Press.

[3] Fowler, M. (2003). *Patterns of Enterprise Application Architecture.* Boston: Addison-Wesley. p.116

[4] Booch, G. (1994). *Object-oriented Analysis and Design. With Applications.* Benjamin / Cummings Publishing Company, Inc.

Moreover, those domain models crafted by IT specialists lacking in-depth business knowledge posed comprehension challenges for business experts, who found the representations in UML difficult to grasp. Moreover, using UML represents the data and not the function combined with it. There were only nouns without verbs combining them.

Evans addressed that issue by advocating for a *ubiquitous language*, a terminology shared between business and IT professionals, ensuring that business concepts are accurately represented even in technical systems. The effectiveness of such an approach in bridging the communication gap between business and IT will be explored thoroughly in this book.

Eric Evans's seminal work on DDD was groundbreaking yet initially found to be challenging to read, leading to its underappreciation for nearly a decade. In mainstream circles, it was recognized merely for a handful of programming patterns, inadvertently becoming associated with over-engineering. While *Evans* emphasized the integration of domain knowledge with technical expertise, his work fell short in providing the level of practical guidance for tactical design and technical patterns that readers sought.

Microservices require a shift

However, the rise of cloud computing and microservices marked a significant shift. Microservices, characterized by their ability to create flexible and independent services where each fulfills a distinct business function, demanded a departure from traditional object-centric domain models to service-oriented architectures.

This evolution underscored the need for a clear methodology to effectively design and implement microservices. Addressing this gap, *Vaughn Vernon*[5] introduced a more accessible approach to DDD in 2013, offering practical guidance for applying these principles in the rapidly evolving landscape of software development. Vernon's contributions were timely, providing much-needed clarity and direction for professionals navigating the complexities of microservice architectures. His work highlighted the increasing demand for methodologies that were not only theoretically sound but also pragmatically applicable to the challenges of microservices.

The publication by *Scott Millet* and *Nick Tune*[6] represented another significant advancement in the realm of DDD design and microservices, offering a structured set of methodologies and techniques for navigating this previously uncharted territory. They introduced foundational patterns in software architecture, such as event sourcing and **Command Query Responsibility Segregation** (**CQRS**), which have since become cornerstones in the field.

---

[5] Vernon, V. (2013). *Implementing Domain-Driven Design*. Boston: Addison-Wesley.

[6] Millet, S., & Tune, N. (2015). *Patterns, Principles, and Practices of Domain-Driven Design*. Indianapolis, IN: Wrox.

By leveraging these patterns, the focus shifted from the minutiae of business function internals to the broader perspective of constructing comprehensive applications and domains. This facilitated a more collaborative and effective dialogue between IT specialists and business stakeholders, enabling the holistic development of systems that better align with organizational objectives and requirements.

Despite the advancements in methodologies and collaborative techniques, there has been a pressing need for practical workshop formats that effectively unite IT specialists and business experts within a shared space. Up until that point, a significant barrier had persisted due to the disconnect in understanding between the two groups: business experts often produced requirement documents that eluded the grasp of IT professionals, while IT specialists generated architecture design documents that remained obscure to business stakeholders. The solution lay in establishing common workshop formats designed to foster a mutual understanding of terms and boundaries, enabling both sides to collaboratively navigate the complexities of project requirements and solutions.

Alberto Brandolini[7] introduced an innovative approach to workshop formats called **Event Storming**, which was redefined ever since. The workshop format enables rapid exploration of business domains. Known for its power, engagement, efficiency, and simplicity, Event Storming fosters collaboration between business experts and IT professionals. This collaboration helps both groups to establish a common language and mutual commitment, streamlining the process of tackling complex problems together. We will discuss this workshop format in more detail in *Chapter 6, Event Storming,* of this book.

Despite the innovations, certain challenges persisted, particularly in finding the precise terminology for aggregates and views. To address this, *Stephan Hofer* and *Henning Schwentner,*[8] introduced a novel workshop format called **domain storytelling** in 2021. This method leverages the power of storytelling, based on the principle that people comprehend and connect with each other more effectively through narratives and visualizations. In a domain storytelling session, a subject matter expert narrates a story, while another specialist illustrates the narrative, sentence by sentence, through drawings. This interactive approach allows business experts to instantly verify their understanding, facilitating a deeper, mutual comprehension among all participants. We will discuss the workshop format domain storytelling in detail in *Chapter 5, Domain Storytelling,* of this book.

Today, a wide range of workshop formats and modeling approaches exist to facilitate the understanding of a domain. Beyond the previously discussed domain storytelling and Event Storming, methods such as the *Business Model Canvas*, *Wardley Mapping* and *User Story Mapping* offer additional perspectives. These approaches, among others, will be thoroughly examined in the subsequent chapters on strategic design, where they will be compared to alternative modeling methodologies.

---

[7] Brandolini, A. (18. November 2013). *Introducing Event Storming*. Abgerufen am 24. March 2024 von https://ziobrando.blogspot.com/2013/11/introducing-event-storming.html

[8] Hofer, S., & Schwentner, H. (2021). *Domain Storytelling.*

Complexity of modern software systems

As complexity in modern software systems intensifies, it is essential for architects, business analysts, developers, and all professionals involved in these systems to understand the nature of this complexity and how to manage it effectively.

Handling complexity

Modern software systems embody the operations of businesses, facilitate complex technical systems like vehicles and provide entertainment in our leisure time.

Regardless of their intendent function, these systems are inherently complex. Let us explore the concept of complexity in greater detail.

Complexity is defined as something where cause and effect can only be perceived in hindsight.

This definition is taken out of the *Cynefin* framework introduced by *Snowden*[9]. *Figure 1.1* shows the areas defined by the Cynefin framework. It provides aid for decision making using the clarity or unclarity of the relationship of cause and effect. Cynefin is a Welsh word meaning *habitat*.

The following figure framework provides five decision-making domains: simple, complicated, complex, chaotic, and disorder:

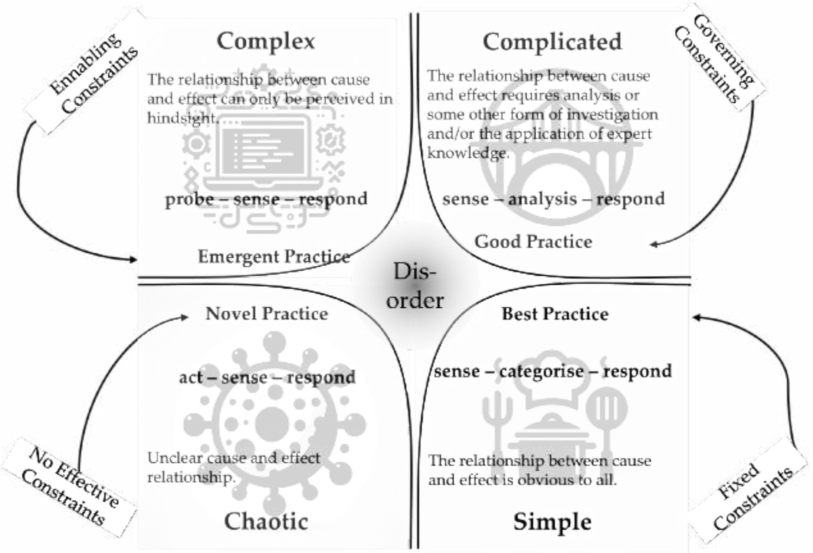

Ennabling Constraints
Complex
The relationship between cause and effect can only be perceived in hindsight.
probe – sense – respond
Emergent Practice
Complicated
Governing Constraints
The relationship between cause and effect requires analysis or some other form of investigation and/or the application of expert knowledge.
sense – analysis – respond
Good Practice
Dis-order
Novel Practice
act – sense – respond
Unclear cause and effect relationship.
Chaotic
No Effective Constraints
Best Practice
sense – categorise – respond
The relationship between cause and effect is obvious to all.
Simple
Fixed Constraints

***Figure 1.1**: The Cynefin framework*

***Source**: Snowden, 2010*

---

[9] Snowden, D. (10. July 2010). *The Cynefin Framework*. Retrieved March 24, 2024 from https://www.youtube.com/watch?v=N7oz366X0-8

The **simple domain** represents the *known knowns*. It means, there are best practices and rules in place which can be used. The solution is stable and the relationship between cause and effect is clear. Imagine you are following a cooking recipe: the steps are clear, and the outcome predictable. Just as adding too much salt can ruin a soup, straying from these known parameters in the simple domain leads to undesirable outcomes. The decision-making is in the realm of reason and logic: Find a proper rule and apply it.

The **complicated domain** is characterized by *known unknowns*. Understanding the relationship between cause and effect necessitates both analysis and expertise. However, through thorough analysis, accurate solutions can be identified. Consider the role of a bridge engineer, who, armed with extensive knowledge and expertise, knows how to construct a bridge that is guaranteed not to fail. This process involves identifying the problem, conducting a detailed analysis, and finding an appropriate solution with the guidance of an expert.

The **complex domain** is characterized by *unknown unknowns*. The causes of effects in this domain can only be understood in hindsight, allowing patterns to be identified post-factum. This necessitates a strategy of probing: testing to see if a chosen path leads to the desired outcome. If it does not, a new approach is necessary. Unfortunately, the software industry often operates within this complex domain, requiring constant adaptation and exploration.

In the **domain of chaos**, the relationship between cause and effect is obscured. One is often confronted with entirely novel circumstances that necessitate immediate action. After acting, it is crucial to assess whether your intervention has contributed to the stabilization of the situation. This is a common scenario during an epidemic, where the approach is to act and then observe the results for any signs of stabilization.

The **disorder domain** is a domain where it is not clear which approach can be applied. The situation requires that it is broken down, so that the other domain strategies can be applied.

In software development, we are fortunate to seldom encounter truly chaotic situations. However, we frequently navigate through complex scenarios and systems. Despite this complexity, it is possible to discern patterns in our work that can be adopted as good practices and implemented as governance mechanisms.

In the complex world of software development, there is a common tendency to deploy patterns as a universal solution. This approach stems from the belief that the more patterns we apply, the better our outcomes should be. However, this is not always the case, especially when dealing with systems that range from simple to complicated domains.

Applying a pattern without a thoughtful consideration of its impact is akin to using a map without knowing the terrain it represents. Software systems, at their core, are complex entities. Thus, merely adopting a pattern without understanding the system's intrinsic complexity often falls short of delivering the desired outcome.

The key to effective pattern application lies in a two-step process: probing and sensing. Before embracing a pattern, it is essential to assess whether it aligns with your objectives and will indeed move you closer to your goals. Once a pattern proves its value in this preliminary phase, you can then confidently integrate it into your practices. This iterative approach not only ensures the pattern's relevance but also enhances your ability to adapt it to future projects.

However, treat it carefully. Each software project presents a unique set of circumstances. What worked once may not work again due to the ever-changing nature of software development. Adaptability and discernment are your allies in navigating this complex landscape.

Through probe and detect approaches, we strive to navigate the management of software systems within a rapidly changing and uncertain world. Such environments, characterized by even greater levels of ambiguity and complexity, are referred to as VUCA worlds. It is within these dynamic contexts that software systems must operate.

Software complexity in a VUCA world

Bennis and *Nanus*[10] in 1985 described behavior as a world of **Volatility**, **Uncertainty**, **Complexity**, **And Ambiguity** (**VUCA**). We still live in such a fast-changing world.

Volatility takes a significant meaning in a world characterized by rapid changes and unpredictability, especially in software development. To craft solutions for such a dynamic environment, our systems must be designed with utmost flexibility in mind. This is where the concept of microservices shines. Microservices architecture offers adaptability and scalability, allowing for seamless interchangeability between components as technology evolves or business needs shift.

Consider the evolution in handling customer interactions within a telephony application. Not long ago, a basic rule engine, capable of responding to specific keywords, would have been sufficient. However, the landscape of technology and user expectations has transformed dramatically. Today, the same task is more effectively accomplished using advanced artificial intelligence, specifically a language processing robot. This shift not only reflects the rapid pace of technological advancement but also underscores the need for software solutions to be adaptable, ready to embrace new methods and technologies to meet changing demands.

DDD helps us to overcome volatility, because models are separated and insofar highly reduced coupling. To implement such an approach microservices can be used. However, microservices are only of use when they are tailored along the business capabilities of an entire domain. Single business functions can be technology-wise exchanged without endangering the entire business. Tailoring is crucial to that flexibility. It can be discovered by using the strategic part of DDD.

---

[10] Bennis, W. G., & Nanus, B. (1985).

Uncertainty is the quintessence of environments where the future is a puzzle, with software development being a prime example. This field is inherently unpredictable, thanks to its complex web of stakeholder interactions, intricate technical implementations, and the multifaceted systems at play. Predicting the next turn is a challenge that eludes even the most experienced professionals.

In navigating the muddy fields of software development, simplicity becomes our beacon. By deconstructing complexity into more manageable pieces, such as adopting a microservices architecture, we create nimble systems that can adapt and evolve. Similarly, fostering a common language between business and technology teams ensures a unified approach to problem-solving. These strategies do not just mitigate uncertainty; they empower us to move forward with confidence, making the unpredictable a bit more navigable.

Complexity, as we have talked about within the Cynefin framework, is about how everything in software systems and their managing organizations is closely connected and depends on each other. This complex nature highlights the many different relationships and factors that can affect what happens in software systems. Due to this, making decisions and solving problems needs careful thought and a detailed approach.

Technical professionals must comprehend the perspectives of their business counterparts. They need to grasp the business processes independently from the technical aspects, and this understanding must be clear and explicit. When a software architect listens to a business expert, they should subsequently articulate in their own words what they have understood. This allows the business expert to confirm whether the understanding is accurate or not. Through such a reciprocal process, a shared and explicit understanding between technical and business teams is cultivated.

Often, the time for those discussions is not spent. It is often seen as ineffective. The business experts write large documents with listed requirements, which are received by technical people, but the technical people do not understand the requirements because of a lack of business understanding. They try to solve the problem, but add more complexity to the problem, because they want to react to each possible situation which might occur based on the partly understood description.

Moreover, there are two types of complexity: accidental complexity and the complexity of the desired functionality. The misunderstood requirements add accidental complexity to the anyhow complex business domain. Even more, whereas the business complexity or essential complexity grows in the beginning of a development, however with time the accidental complexity grows faster as the essential complexity. That means, the overall complexity growth constantly with the time even though the business problem is solved, and the essential complexity is almost constant (*Zimarev*, 2019[11]).

To avoid accidental complexity, the technical teams and business experts need to collaboratively establish a shared understanding. Business experts need to describe clearly the problem area to ensure they are well understood. DDD provides the methodology

---

[11] Zimarev, A. (2019).

to define boundaries of complexity and to identify and manage the essential complexity within those boundaries.

Ambiguity embraces the inherent uncertainty and variability in the meaning of words or concepts, particularly when used across different contexts. For instance, the domain storytelling approach vastly differs from that of a mystery novel. Attempting to create a universal glossary that spans an entire enterprise is an exercise in futility. Instead, we must recognize the need for characteristic boundaries, identifying specific areas where defining terms proves meaningful and valuable.

The usage of an unambiguous language is only one part. The usage of a ubiquitous language, which is common to all stakeholders—technical and businesspeople—is the other part. Language is important, however, one needs to be aware that language belongs to a context. When one speaks of an address, it could mean a **Uniform Resource Locator** (**URL**), it could mean a mail address, it could mean an e-mail address or even a speech to the diplomatic choir.

We could define those nouns in lengthy tables, but we would miss the meaning when it comes to actions on those nouns. For example, it would be important if the address to the diplomatic choir is planned or has been done already. It is important if a letter should be sent to an address or was sent from one. Therefore, we need to define not only the nouns, but we also need to define the relationships between them. The relationships are the glue which combine different objects, and the glue are verbs, and those relationships are expressed as verbs. In *Chapter 5, Domain Storytelling*, the methodology of a **visual glossary** is introduced where nouns are combined by verbs in a graphical and easy to grasp manner.

Tackling complexity

As previously observed, we inhabit a complex world where software systems also embody complexity. Addressing this complexity is crucial for the development of successful software systems. The next chapter will explore various strategies for managing this complexity.

Design Thinking

An approach to create valuable software is *Design Thinking*[12]. Design Thinking is less a process, more a way to think about problems. Design Thinking states four principles of design (HART principles):

- **Human rule:** When we want to design a software, we need to have customers in mind. We design software for people—not for machines. Obviously, to create something for humans is more complex than just to create some algorithm a machine can follow. Even though the machine might be incredibly complex,

---

[12] Plattner, H., Meinel, C., & Leifer, L. (2010). *Design Thinking: Understand - Improve - Apply (Understanding Innovations).* New York, NY: Springer.

however, the machine following such an algorithm is predictable. Overall, we do not create software for machines, we create software for human beings.

- **Ambiguity rule**: As discussed previously, ambiguity presents a significant threat to software success. To mitigate this, the used language must be precise and unequivocal. In our software design, we employ not just one model, but a variety of models, each within its own context. Throughout the software development process, these multiple models are integral. DDD, at its core, is about creating models that reflect the domain intricately. Therefore, finding suitable models and establishing a ubiquitous language within defined boundaries is a collaborative effort between business experts and IT specialists.
- **Redesign rule: All design is redesign**: The redesign rule encourages us to think about already existing solutions. Probably an existing solution, an existing pattern can be applied. We refine already found solutions and shape them along the problem at hand. Designing software not only means to find new solutions, but also to find solutions that already exist and can be tuned to the current problem.
- **Tangibility rule: Make ideas tangible to facilitate communication**: Software design is shown in code. However, code is difficult to read—it is usually not tangible. Discussions about coarse-grained components, design rationales, etc. is almost impossible while only using code. Architecture can be made tangible using models, prototypes, mockups etc., so people can discuss them. Using simple models for things to be discussed is a great way to discuss software architecture.

The *Hart* principles create a base for our software design. Anyhow we need more methodologies and processes to be successful. However, throughout our way, we can ask ourselves if we still follow those principles.

Complexity and analysis

In the section, *Complexity of Modern Software Systems,* we saw that software systems are complex. They are not only complex, but they also need to be built in a volatile, uncertain, complex, and ambiguous world. When we want to develop stable software systems, we need to handle both aspects.

First, the problem area needs to be analyzed. Afterwards the solution needs to be designed. However, we cannot solve our problem only by analyzing, because it is complex. We need to postulate a solution and define a test to proof if the solution is correct or not. To go this try-and-error path for the entire postulated solution would be too expensive. The implementation of the solution might cost ten person years, we will need to wait too long until we get any result.

Therefore, we need to find smaller parts of the problem which can be solved. For those areas, we can start with our try-and-error path. So first, we need to tailor our problem in smaller parts. Those parts should solve the basic problems of the customer. Later, we can add more features and enrich our software. When we make mistakes, we can roll back and try another solution without losing too much time and money. The principle is called **release early, release often**.

This principle was introduced by *Raymond*[13] in 2001. Raymond discussed the following two software development models for open-source projects:

- **The Cathedral model**: The source code is available after each release. Code developed between releases is restricted to an exclusive group.
- **The Bazaar model**: The code is developed over the internet in view of the public. *Linus Torvald* introduced such a model for developing the *Linux* kernel.

One of the most popular thesis of Raymond is *given enough eyeballs, all bugs are shallow*. In the Cathedral model, you need more time for testing, because the software is only available to a handful of people. Often software products cannot be developed in an open-source model. Questions of intellectual property, etc. might arise, but still today, open-source products are successful worldwide. In such a way, it is worth thinking about it.

A further step-by-step approach enhancing the release-early-release-often-principle was introduced by *Ries* called **Minimum Viable Product** (**MVP**) in 2009[14]. It proposes to create an early product with only those features (and no more) that allows you to ship the product. Afterwards, you can add more features to it and the customers get excited. *Kniberg* commented on it 2016[15], that it is not only about features. When a feature alone makes no sense, you cannot give it to a customer. A customer who wants to go from A to B cannot handle just a wheel, even though a wheel is an important feature for a car. However, when you give the customer a skateboard, they can travel from A to B. They might not be happy with the skateboard in the first step. It, however, fulfills the basic need. Afterwards you provide them with a scooter, bicycle, motorcycle, and finally with a car.

---

[13] Raymond, E. S. (2001). *The Cathedral & the Bazaar—Musing on Linux & Open Source by Accidental Revolutionary.* Farnham: O'Reilly Media.

[14] Ries, E. (23. March 2009). *Venture Hacks interview: What is the minimum viable product?*. Retrieved March 29, 2024 from **https://www.startuplessonslearned.com/2009/03/minimum-viable-product.html**

[15] Kniberg, H. (25. January 2016). *Making sense of MVP (Minimum Viable Product)—and why I prefer Earliest Testable/Usable/Lovable*. Retrieved March 29, 2024 from **https://blog.crisp.se/2016/01/25/henrikkniberg/making-sense-of-mvp**

With each step, the need is fulfilled more conveniently, and the customer becomes excited. Refer to *Figure 1.2*:

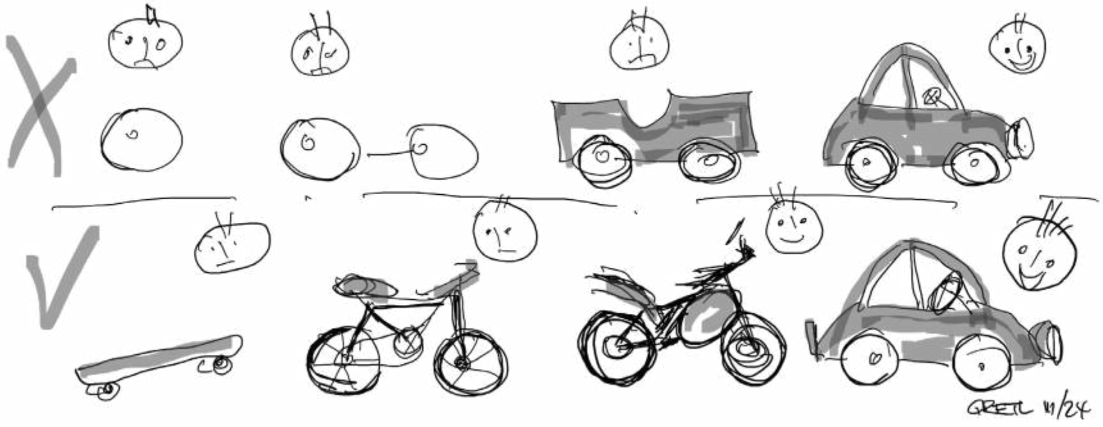


***Figure 1.2**: MVP ideas after Kniberg*

Keep in mind that you need to know the needs of your customers, which you can test quite early. You can adapt your software step by step. The features added can be small and when you do not cater to the needs of the customer, you can roll back without losing too much money and resources. There are some successful companies using an MVP approach: Spotify, Dropbox, AirBnB, Uber etc.

Using the MVP approach, we can iterate to learn. An iteration can be as brief as a few minutes or as long as a few days. Obviously, shorter cycles are to be preferred over longer ones, but sometimes more time is needed for thorough research. Every iteration follows the same steps, as proposed by *Keeling* 2017:

- **Think**: First you need to find the right questions: *What do we want to learn? What questions do we need to answer?* Based on that, you can create a plan, to learn what needs to be learned or to find the answers to your questions.
- **Do**: Do translates to how you execute the plan. You create something tangible that quickly and cheaply uncovers information needed to check our thinking and share our ideas.
- **Check**: You need to examine your results critically. You will find new questions to be answered and new things to be learned. Software is never finished. You need to start the iteration again.

Figure 1.3 shows the development process following the iteration idea described before:

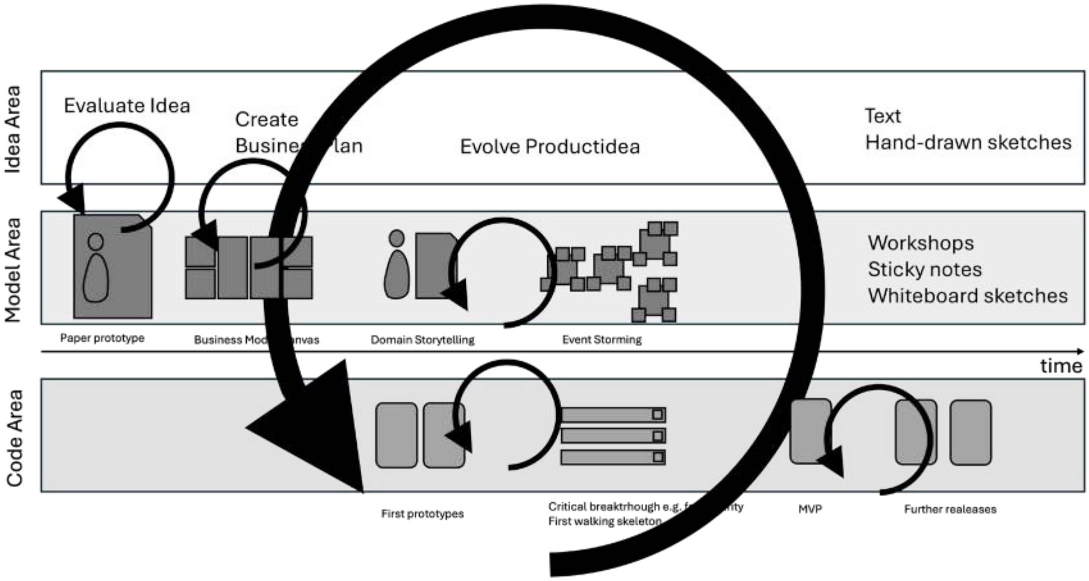


Figure 1.3: *Development process following iterations*

If you have a product idea, and you think it is great and can be a product, you discuss it, using paper prototypes and roughly sketched ideas. You will discuss them with specialists from banks, people from the business domain your product idea comes from, with friends and most probably with your family. The idea will evolve.

Now you need to create a business plan. Probably you do not like to have a business plan, but you need to persuade investors of your idea. So, you write your idea down. Anyhow to make it more tangible, you use a model—a business model canvas as introduced by *Strategyzer*[16]. Using such a model makes it easier to discuss your idea with investors as well as with important stakeholders.

Let us assume you could convince your stakeholders, and you can go further with your idea. Until now you only had a rough sketch in your small notebook. Therefore, you start to ask business experts, future users, friends to discuss with you how they want to use your application. To do so, you start a domain storytelling and get great ideas out of it. Domain storytelling (*Hofer and Schwentner*) and Event Storming (*Brandolini*) will be discussed in *Chapter 5, Domain Storytelling* and *Chapter 6, Event Storming*.

You need to detail your ideas further. You implement first prototypes to show what the software is supposed to do. You implement mockups so that someone can use the software. And you need to know better how to tailor your software. Therefore, you perform Event Storming workshops. You still have questions, how certain critical things e.g., security can

---

[16] Strategyzer. (2023). *The Business Model Canvas*. Retrieved March 29, 2024 from **https://www.strategyzer.com/library/the-business-model-canvas**

be implemented, therefore, you implement some critical breakthroughs to get the answers. Moreover, you implement a first walking skeleton, which can be used on the client side as well as on the server side with quite limited functionality. Regardless, you see how it could work.

Now you can implement your first MVP and release it! Your product is in the world. Later you can add more features and releases. You will step back to iterate via prototypes, domain stories, or even idea sketches. The iterative approach allows you to learn and to become better.

Thinking about complexity in software development, you most probably want to have consistent data, a high availability, and even probably an ever-working network. You cannot have that all at once. Why you cannot have it all at once and what you can have, will be discussed in the next section.

CAP theorem

The CAP theorem is often cited when the discussion comes to complexity in software applications. It was stated by *Eric Brewer* in 2001[17]. The theorem asserts that any networked shared-data system can only have only two of three desirable properties: **Consistency** (**C**), high **Availability** (**A**), or tolerance to network **Partitions** (**P**).

- Consistency means that every read from any part of the entire system receives the most recent write or an error.
- Availability means that every request is handled. It receives a (non-error) response. It does not matter if it gets the most recent write or even not.
- Partition tolerance means that the system still operates despite an arbitrary number of messages dropped or delayed. That can be achieved by a divided system, where each part can work independently. When the connection between the parts is restored, the system works correctly again.

Traditional systems like relational databases prefer consistency over availability, whereas more modern systems like online applications or *NoSQL* databases prefer availability over consistency.

Eventual consistency is a consistency model used in distributed applications to achieve high availability. Those systems follow a **Basically Available, Soft-State, Eventual Consistency** (**BASE)** semantics. The BASE semantic contrasts with **Atomicity, Consistency, Isolation, Durability** (**ACID**) used in traditional databases (see *Braun*[18]).

- **Basically available**: Reading and writing operations are available as much as possible.

---

[17] Brewer, E. (July / August 2001). Lessons from Giant-Scale Services. *IEEE Internet Computing*, S. 46-55.

[18] Braun, S. (2022), *Mastering Eventual Consistency*. IEEE 19th International Conference on Software Architecture Companion, ISCA-C 2022

- **Soft-state**: Even though there is no guarantee, after a certain amount of time, the state can be known, but it has not converged yet.
- **Eventually consistent**: That if we write something and we wait long enough (and the system works), all system nodes reach the same state. The waiting time is not guaranteed. Any further reads will deliver the same value.

The combination of two of the desired properties gives the according systems their name. Today, most of the web applications are designed as **Availability and Partition Tolerance** (**AP**) systems because the availability requirements are extremely high, whereas eventual consistency can be applied.

Figure 1.4 shows the CAP theorem as *Euler* diagram:

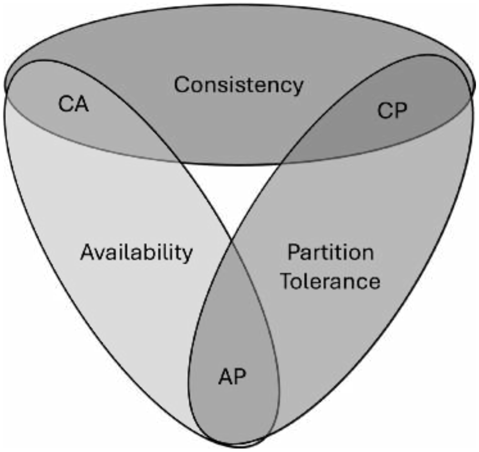


***Figure 1.4**: CAP theorem shown as Euler diagram*

Obviously, such an eventual consistent system can create conflicts. Therefore, conflict resolutions need to be implemented. How the reconciliation will be done and when (e.g., with a read detecting the conflict or with a write, automatic or manual) deeply depends on the application and needs to be defined during the software design phase.

Commensurate complexity

Thorngate's postulate[19] of commensurate complexity is a description of a phenomenon in social science. A model can never have all three properties simple, general, or accurate.

---

[19] Thorngate, W. (February 1976). "In general" vs. "it depends": Some comments on the Gergen-Schlenker debate. *Personality and Social Psychology Bulletin*, S. 404-410.

Only two of them can be fulfilled. Based on this postulate, *Weick*[20] created the picture of a clock in 1995. Refer to *Figure 1.5*:

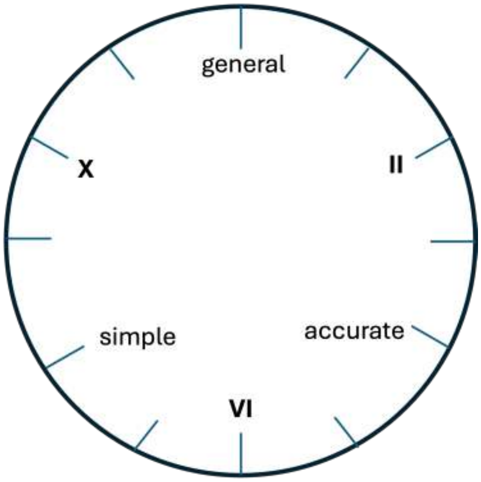


Figure 1.5: *Image of a clock showing model properties after Weick 1995*

Using the picture, it is easy to imagine that a 10-clock-model is simple and general, whereas a 2-clock-model is simple and accurate. The more the pointer wanders to 12 o'clock, the model becomes more general. The more it wanders to 4 o'clock, the more accurate it becomes.

If we want to use that model in software, we can find many samples.

A typical 10 o'clock model is loose coupling. Loose coupling has no context; therefore, it is general. It is simple, at least to understand, even though it is hard to implement. Loose coupling is not accurate, because it does not say anything about the practical implementation.

Another 10 o'clock model is the agile manifest. All four value pairs and the twelve principles are context-independent and simple. Nevertheless, there are many frameworks and discussions about the agile manifesto which show that it is not accurate.

If we want accuracy, we need to give up one of the properties. We can give up simplicity. A sample of such might be the inversion-of-control-container. It is still general, but it is not simple. However, it is clear how it can be used and what you can do with it. Therefore, it is accurate. It is a 2 o'clock model.

Comparable thoughts can be done about **Test Driven Design** (**TDD**). TDD do not create statements about the context of an application, but the concept is generally applicable. TDD is not easy. The cognitive performance of humans using TDD is high. Insofar, TDD is a 2 o'clock model as well.

---

[20] Weick, K. E. (1995). *Der Prozess des Organisierens.*

The 10 o’clock and 2 o’clock model samples are given by *Beine* in 2021[21].

Now, combine simplicity and accuracy and we get the 6 o’clock models. Obviously, those models are domain models. Domain models need to be easy, because they need to be understood by business experts and IT specialists in the same way. They need to be accurate, because they need to show the context as precisely as possible. In the end, we want to have easy to understand models using a precise and ubiquitous language. To create those models, we need DDD.

Software architecture

When we talk about DDD, we talk about software architecture.

The simplest way to understand *Software Architecture* is by understanding the work of software architects. The transition to becoming a software architect is not always clear-cut.

Usually, developers evolve into software architects by supplementing their technical abilities with communication and business knowledge. Similarly, business analysts move to architectural roles by enhancing their skills with technical expertise. Product owners, on the other hand, typically expand their knowledge to include both technical and business insights. Therefore, if you are working as a developer, business analyst, or product owner, it is beneficial to acquire some software architectural knowledge in order to progress in your career or to effectively engage with software architects on an equal footing. For those who are already practicing as software architects, it is recommended to read this book and integrate DDD into your skillset. This approach is increasingly becoming a foundational skill for software architects in the current and foreseeable future.

As professionals in the software industry, it is crucial that we possess a strong understanding of both business and technology. We must comprehend our customers' needs as thoroughly as we grasp the intricacies of the technology we work with. Moreover, it is our responsibility to lead our customers toward effective solutions by helping them fully understand the nature of their problems.

It is a common scenario: customers often come forward with solutions in mind for their challenges. However, to provide a genuinely suitable solution, one must first thoroughly understand the problem. This requires a deep dive into the domain and its specific terminology. For instance, in the insurance industry, grasping the meaning of terms like *policy* or *claims* is essential. In the automotive sector, *platform*s have a completely different connotation than it might in IT. Once you have a solid understanding of the customer's needs, the next step is to propose a technical solution that is not just effective, but also appropriate in terms of cost, time, and the available skill set.

As an architect you need to cover a high number of functions throughout the entire software development process.

---

[21] Beine, G. (May 2021). Die Angemessenheit von Komplexität.

Software architects define the problem from an engineering perspective. They tailor the system and assign responsibilities to the single parts. They keep an eye on the bigger picture—be it the entire system running in production, the perception by users, or the revenues coming out of it. They decide about trade-offs among quality attributes like consistency and availability. They manage technical debts and enable implementation teams to do architectural work (see *Keeling*[22]). One might think that architects are superheroes, but they are not. Usually, they are seasoned professionals, who have the ability to communicate concerns and solutions.

Software architecture is defined by the fundamental structures of an application, which includes the components and their interrelations. It employs methods like prototyping and developing *walking skeletons* to flesh out these structures. Effective architecture must be tested—this is where probing with sample implementations comes into play. Pose hypotheticals to the application, such as, *what if a response takes 10 seconds?* The acceptability of this delay varies by application: it may be tolerable in an accounting software, pending business expert approval, but in a real-time trading system, the delay could result in a substantial financial loss. Even in a calculator app, while there is no monetary loss, a 10-second delay could be unacceptable to users. Therefore, asking *what if* is crucial to balance quality attributes like performance, availability, and accuracy, among others.

Keeling stated six points which makes software amazing. We will discuss how to apply them into our day-to-day lives:

- **Software architecture turns a big problem into smaller, more manageable**:

  As we already discussed, large problems are difficult to handle and difficult to solve. Therefore, we need bounded contexts, which divide a domain in smaller areas. You will learn how to find bounded contexts and use a context map in *Chapter 6, Event Storming* and *Chapter 7, Context Map*.

- **Software architecture shows people how to work together**:

  As software architects, we need to talk to business people, to project managers, to developers, testers, DevOps engineers and so on. Throughout the book, we will learn a couple of workshop methodologies, which help to organize and moderate such difficult discussions. Domain storytelling workshop is explained in *Chapter 5, Domain Storytelling*.

- **Software architecture provides a vocabulary for talking about complex ideas**:

  As software architects we need to develop a ubiquitous, unambiguous language together with IT specialists and business experts. Such a language can be documented in a visual glossary. The methodology of a visual glossary is discussed in *Chapter 5, Domain Storytelling* and *Chapter 6, Event Storming*.

---

[22] Keeling, M. (2017). *Design It!* Pragmatic Bookshelf.

- **Software architecture looks beyond features and functionality**:

  We will discuss how to handle quality attributes in tactical design, how to design aggregates and domain events in *Chapter 10, Aggregate, Entity, and Value Object*.

- **Software architecture helps you avoid costly mistakes**:

  We will discuss those mistakes in the brown-field example in *Chapter 15, Usage of Domain-Driven Design in the Brownfield*.

- **Software architecture enables agility**:

  Software architects are not only responsible for good software. They are responsible for organizing teams along the desired software architecture. If they do not do so, the product will fail according to *Conway's law* 1968[23]. We will discuss the organizational aspects together agility and team responsibility in *Chapter 14, Usage of Domain-Driven Design in a Greenfield Project*.

Conclusion

The chapter provided a brief overview of how DDD has evolved over the last 20 years. It explained the significance of understanding the nature of complexity in a rapidly changing world and offered strategies for addressing it in software development.

DDD is a fundamental component of software architecture, making it a valuable skill to acquire.

In the next chapter, we will introduce an online library that will serve as a consistent example throughout the entire book.

Points to remember

- DDD is employed to manage complexity in software design.
- Complexity is an intrinsic aspect of software and is inescapable.
- While specific patterns may contribute to the intricacy of parts of a software system, the system as a whole remains complex.
- DDD is instrumental in preventing accidental complexity not just at the outset of development but throughout the entire lifecycle of a system.
- DDD fosters collaboration between business professionals and IT specialists, enabling them to forge a shared, ubiquitous language.

---

[23] Conway, M. E. (April 1968). How do committees invent?

Multiple choice questions

1. **Which workshop formats are introduced during the development of DDD**
    a. Event Storming and domain storytelling
    b. Design Thinking and Mind Storming
    c. World café and Open Space
    d. Unconf and Clean Café
2. **Which domains are mentioned in the Cynefin framework?**
    a. Complex, complicated, and simple
    b. Complex, complicated, simple, and disorder
    c. Complex, complicated, simple, disorder, and chaotic
    d. Chaotic and complex
3. **Why is the complexity of software increasing constantly?**
    a. Because the world is becoming more and more complex.
    b. Because the used technology is not adequate anymore.
    c. Because complexity is increasing everywhere.
    d. Because over time more and more accidental complexity is added when no particular actions are taken.
4. **How many requirements of consistency, availability, and partition tolerance can be fulfilled at the same time?**
    a. Nothing of them—all three are idealisms
    b. Only 1
    c. Only 2
    d. All three of them need to be fulfilled
5. **When loose coupling is simple and general, how would you describe DDD on Weicks clock?**
    a. Simple and general
    b. General and accurate
    c. Accurate, simple and general
    d. Accurate and simple

Answers

1. a
2. b
3. d
4. c
5. d

CHAPTER 2

Introduction to the Example Online Library

Introduction

This chapter introduces online library, which will be used as an example throughout the entire book. It is introduced by first idea sketches and a paper prototype. The business idea is outlined by a business model canvas and a capability map.

Using the capability map core, supportive, and generic domains are found and adapted to the needs of an online library application.

Structure

This chapter will cover the following topics:

- First ideas
- Business plan model
- Capability map
- Capabilities and domains

Objectives

You will learn how to sketch a business idea for a new product or service and how to use detect capabilities out of it. Additionally, you will learn how to detect core domains and others for your application.

> **When all else fails, give up and go to the library.**
>
> *~Stephen King*

First ideas

Throughout the book we will use the example of an online library open for students and other readers. We want to make our library a place where people can read, chat, and exchange summaries.

The library should be implemented as a mobile application in the same way as a web application. *Figure 2.1* presents the initial sketch for the digital library. It is sketchy and unclear as it covers the very first ideas and discussions. Users should be able to search for books using a single search field that accepts various criteria, such as author names, titles, and **International Standard Book Number** (**ISBN**). Upon selecting a book, it can be opened for reading within the application. Additionally, users have the capability to annotate the text, with the convenience of exporting these annotations to other note-taking applications. For books that includes images, an option view and magnify option should be available.

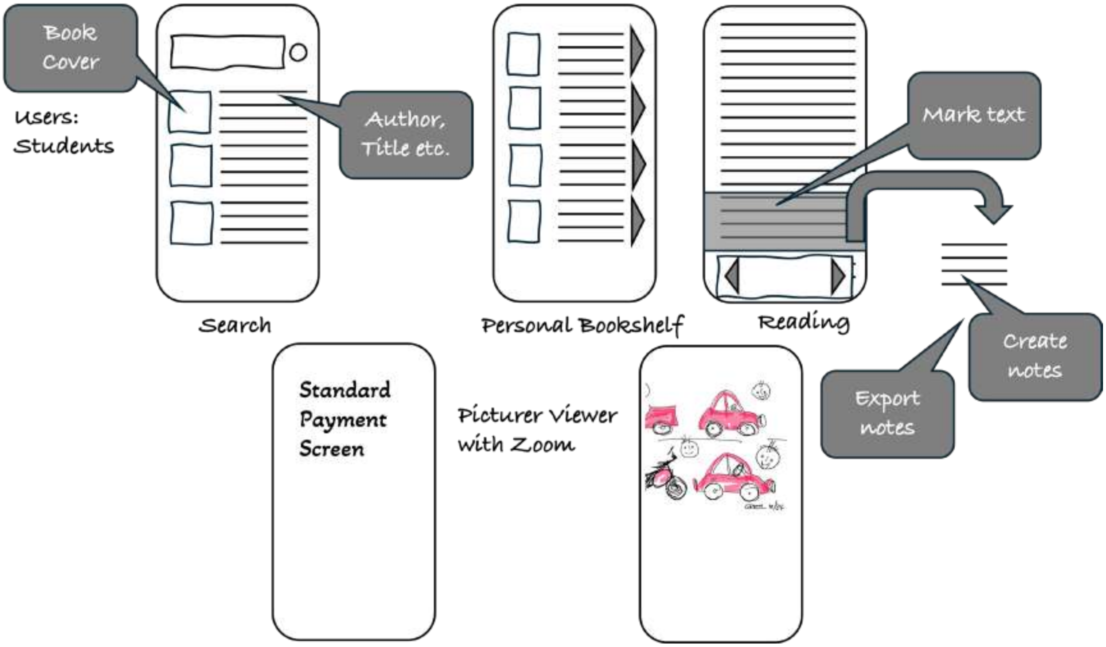


***Figure 2.1**: Sketch of the online library*

***Source**: Sketch.png*

Membership is required to access the reading material. Interested users must register and pay a membership fee to become members of the library.

Moreover, the application will automatically save the user's reading position, ensuring they can pick up exactly where they left off during their last reading session.

Using these first ideas, a discussion can start on how a user's flow throughout the application can look like. If you want to discuss your idea in more detail, you can do that with friends. You can create a paper prototype, as a **Portable Document Format** (**PDF**), where someone can see the user flow roughly as shown in the following figures:

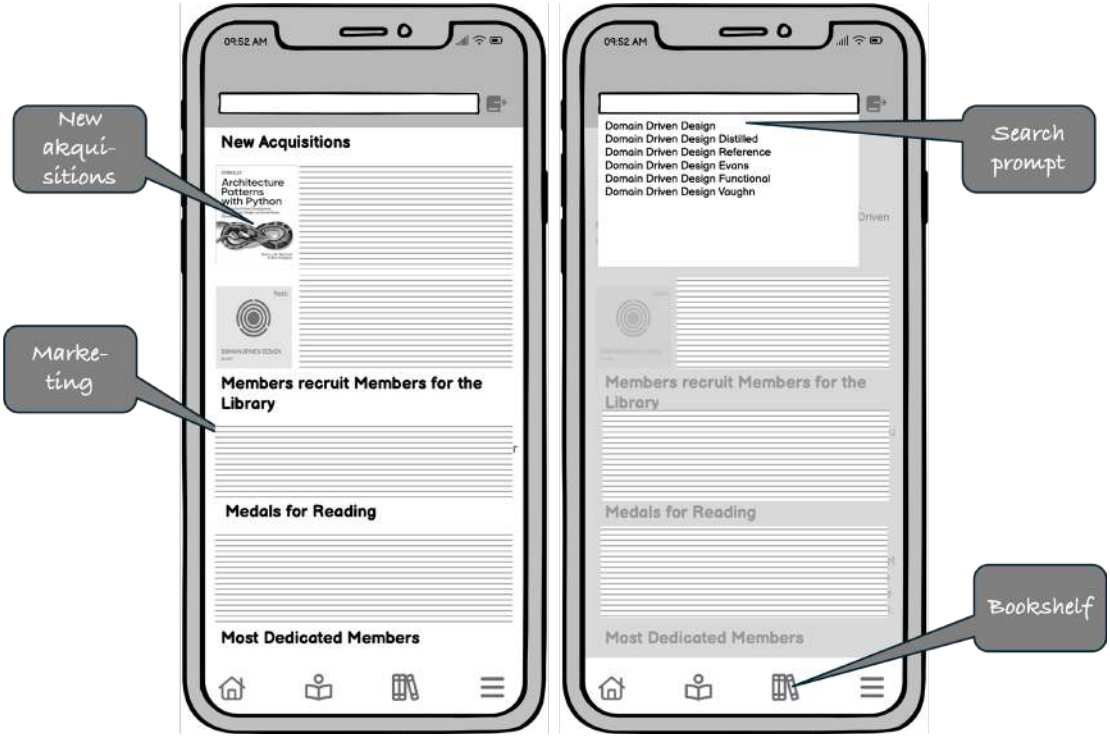

New akquisitions
Marketing
09:52 AM
New Acquisitions
Architecture Patterns with Python
Members recruit Members for the Library
Medals for Reading
Most Dedicated Members
09:52 AM
Domain Driven Design
Domain Driven Design Distilled
Domain Driven Design Reference
Domain Driven Design Evans
Domain Driven Design Functional
Domain Driven Design Vaughn
Members recruit Members for the Library
Medals for Reading
Most Dedicated Members
Search prompt
Bookshelf

Figure 2.2: Step 1: Paper prototype of the online library

The following figure shows the second step of the online library:

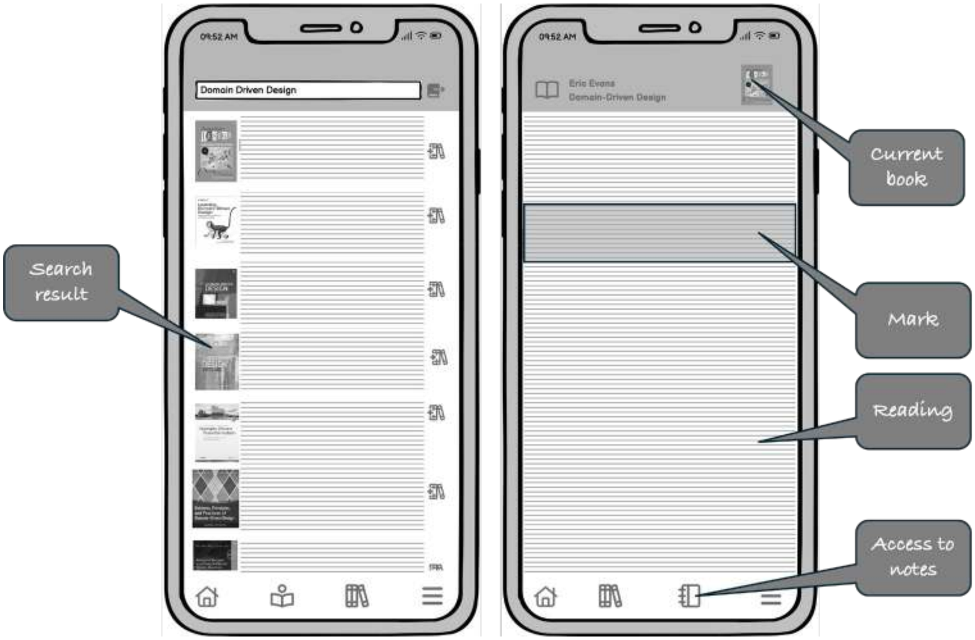


***Figure 2.3:** Step 2: Paper prototype of the online library*

The following figure shows the next step of the online library:

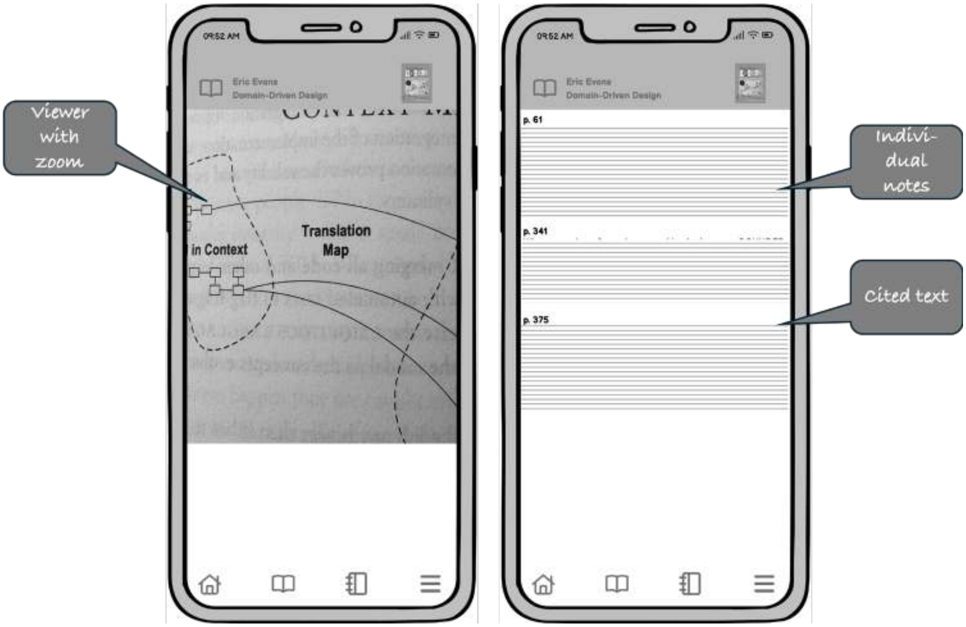


Figure 2.4:

The following figure shows the last step of the online library:

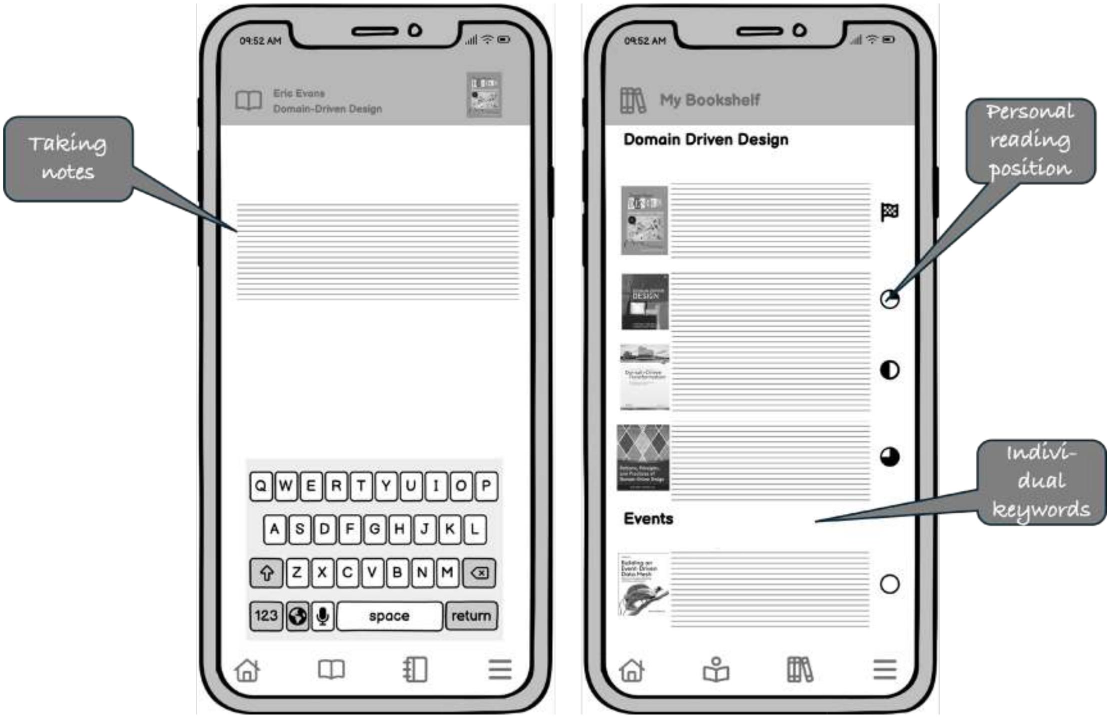


Figure 2.5: *Step 4: Paper prototype of the online library*

Source: *Screenflow1_4.png, Screenflow2_4.png, Screenflow3_4.png, Screenflow4_4.png*

The paper prototype illustrates the key features of the library. On the landing page, users are greeted with the latest library news, including announcements of new book acquisitions.

For searching, users have a single input field at their disposal, with author names, titles, ISBNs, or snippets of the blurb as potential search criteria. As the user types, suggestions pop up, informed by their personal search history and the combined search data of all library members.

Search results are displayed in a list, complete with book covers, allowing users to either commence reading or add books to their personal bookshelf. Users can also arrange their bookshelf using custom labels. A partially filled circle representing the reading position is shown for each book on the shelf, giving a rough idea of how much of the book has been read.

While reading, users can highlight text passages and make notes. Highlighting a passage automatically opens a notes page, though users can also access the notes area directly to view and edit their annotations.

In case the user encounters an image within the text, they can tap on it to view it separately quickly return to their reading exactly at the point where the image is located in the text.

Using these documented thoughts, a business plan can be sketched out.

Business plan model

Usually, it is quite difficult to write a business plan. Using a business model canvas, as introduced by *Strategyzer 2023*[1], makes such a difficult task more tangible.

The *Strategyzer* canvas facilitates workshop formats, promoting a unified vision of the business's core objectives. *Figure 2.6* presents the business model canvas for our online library:

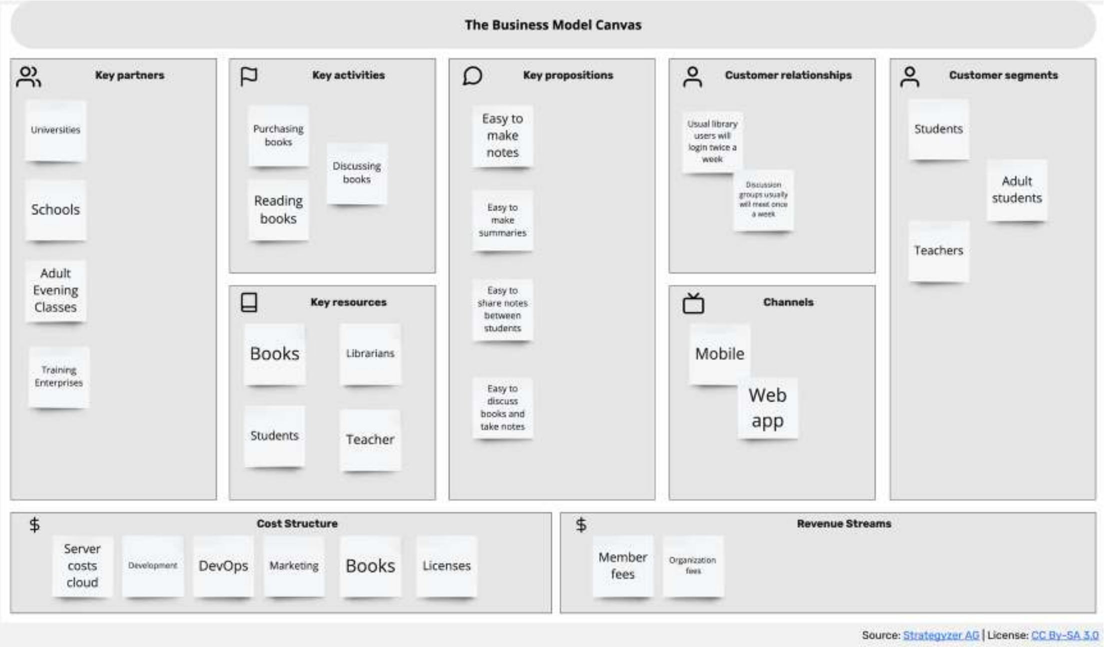


Figure 2.6: *Business Model Canvas of the online library after Strategyzer 2023*

Source: *BusinessModelCanvas.jpg*

The following points cover the questions which are needed to be answered when creating a business plan:

- **Key partners**: Essential to gaining a competitive edge, the canvas identifies necessary alliances. For our online library, forging partnerships with universities, schools, evening adult education programs, and training organizations is crucial, as they can provide endorsements and referrals.

---

[1] Strategyzer. (2023). *The Business Model Canvas*. Retrieved March 29, 2024 from **https://www.strategyzer.com/library/the-business-model-canvas**

- **Key activities**: To meet our library users' requirements, we must procure books or licenses for digitally published media. Reading is, unsurprisingly, the primary activity within a library. However, to cater to students effectively, we must also support discussion and study groups, which are vital for knowledge exchange.
- **Key resources**: The foundation of our business lies in both tangible materials and intellectual expertise. This includes not only books but also the librarians and educators who recommend readings and library services. Additionally, students and alumni serve as ambassadors, further promoting our library.
- **Key propositions**: Surpassing basic customer needs, our value propositions aim to enhance user satisfaction. In the context of our library, this translates to simplifying the process of taking notes and creating summaries, as well as sharing these with other students. Ensuring easy access to books, notes, and summaries for student discussions is also paramount.
- **Customer relationships**: The frequency of customer interactions with the business is a key indicator of relationship strength. The more frequently a user engages with the online library, the better. Consequently, we must implement robust engagement programs. Typically, a regular library user visits twice a week, while study groups convene once a week.
- **Channels**: Determining how to connect with our customers is crucial. For the online library, we will provide access through both a mobile application and a web platform, ensuring convenience and flexibility.
- **Customer segments**: Understanding who our customers are is foundational to business success. Our online library serves a diverse clientele, including students, adult learners, and educators.
- **Cost structure**: Discussions around the cost structure are essential. We need to determine our budget for various expenses, such as development, production, marketing, material, and salaries. Notably, product development is central, referring to the creation of the online library application. Production costs include DevOps and cloud server expenses. Additionally, we need to account for marketing expenditure, purchase of books, and compensation for specialized staff, notably experienced librarians.
- **Revenue streams**: Understanding potential revenue streams is critical for financial planning. Our library aims to generate income through membership fees and charges to organizations wishing to provide the library services to their students.

Capability map

To plan the development of our online library, we need to know what our core competencies are and how we want to implement them. As a first step, we collect all necessary functionalities or better business capabilities we need. Refer to *Figure 2.7* to understand the capabilities of an online library:

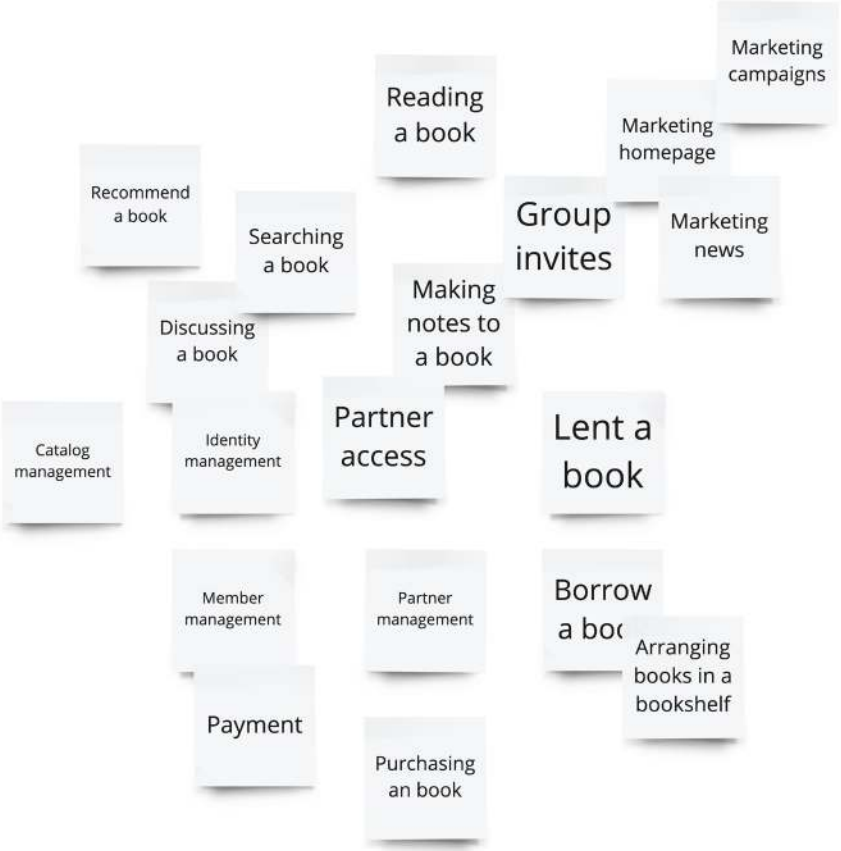


***Figure 2.7**: Capabilities of the online library unsorted*

***Source**: CapabilitiesUnsorted.jpg*

The capabilities can be used to structure the business and the application to be implemented. As the second step, the capabilities are sorted along the user journey of the paper prototype. Certain user interface elements could be missing. Those can be represented by sticky notes.

Now, you can add user groups as swim lanes and assign the capabilities according to the user group. The result is shown in *Figure 2.8* as a capability map:

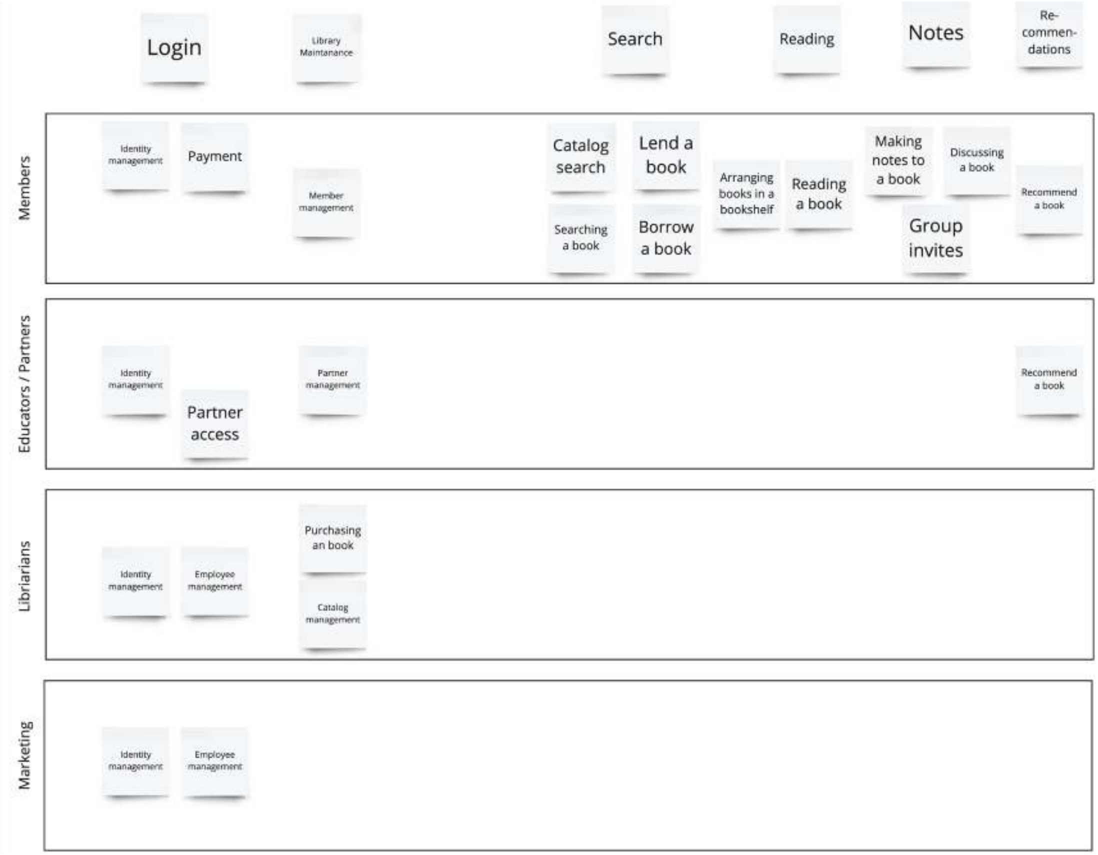


***Figure 2.8**: Capability map of the online library*

***Source**: CapabilityMap.jpg*

The capabilities *Identity Management*, *Member Management*, *Payment*, *Partner Management*, and *Employee Management* are collected below the *Login* functionality. Even though they contain the login functionality, and also cover the onboarding process for the different user groups, they are collected below login. This is because the Login functionality only works when those capabilities are implemented. During the discussions and the creation of the capability map, a new capability was found called Employee management.

For the capability referring to the user group *Librarians,* the purchasing of books is relevant. An according functionality was added to the capability map. The capabilities *Purchasing a book* and *Catalog management* are sorted below *Library maintenance* functionality for the user group of librarians.

The user group *Members* is the most important one for the library. The capabilities *Catalog search*, *Searching a book*, *Lend a book*, *Borrow a book*, *Arranging books in a bookshelf*, *Reading a* *book*, *Making notes to a book*, *Discussing a book*, and *Group invites* are sorted in the *Members swim lane* along the user journey.

For partners and educators, the capability *Recommend a book* is relevant, even though it might be relevant for members as well.

Capability maps are a well-known tool to structure and analyze enterprise architecture as introduced by *Blair, Lail, and Marshall*[2]. Capability maps can be created top-down—as shown in our example—or bottom-up:

- **Top-down**: The top-down capability mapping approach begins by identifying the 20 to 30 highest-level business capabilities, each of which can be decomposed into more detailed levels.
- **Bottom-up**: Business capabilities can be defined from within different parts of the business und built from the bottom up. This approach needs more time.

The bases of the capability map are:

- **Organizational structure**: Shown as horizontal swim lines in our example.
- **Business model**: The capabilities were derived from the business plan canvas.
- **Financial plans**: Shown in the business model canvas with cost structure and revenue streams.

Using the current strategic approach—we want to serve our readers as best as we can. We can identify the core, supportive, and generic domains of our library. The terms domain, core domain, supportive domain, and generic domain were introduced by *Eric Evans*[3].

A **domain** is defined as everything a company is doing. For our library, it includes lending books, purchasing books, or even providing access to the library functionalities via **Application Programming Interfaces** (**APIs**).

Subdomains are contained in the domain of an enterprise. Subdomains can be core domains, supporting domains, or generic domains.

Core domains contain the capabilities, which differentiate the according enterprise from other competitors and define the business of the company. The core domain of an insurance company is to insure people against predefined risks. The core domain of a car manufacturer is to build cars. For our online library, it means to lend books to members so that they can read those books. *Vernon* described it as a part of the business domain that is of primary importance to the success of the organization[4].

---

[2] Blair, A., Lail, J. B., & Marshall, S. (2022). *Business Capabilities, Version 2.* Berkshire: The Open Group.

[3] Evans, E. (2004). *Domain-Driven Design Tackling Complexity in the Heart of Software.* Upper Saddle River, NJ: Addison-Wesley.

[4] Vernon, V. (2013). *Implementing Domain-Driven Design.*

Capabilities and domains

Figure 2.9 shows the relations between terms we used already—an enterprise can contains several core domains, supportive domains, and generic domains:

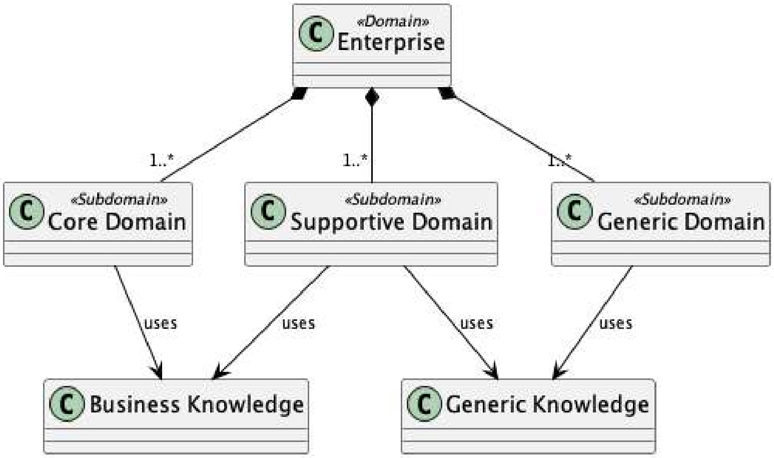


Figure 2.9: *Subdomains of an enterprise*

Source: *DomainDefinition.png*

Core domains, supportive domains, and generic domains are contained in an enterprise. They define the business a company is doing. The core domains use business knowledge of the area the enterprise is doing business in. Generic domains are necessary to do the business of the enterprise, but they do not require business knowledge. Supportive domains are part of the enterprise as well. They need business knowledge as well as generic knowledge.

With the definitions of core domain, supportive domain, and generic domain in mind, we can define which capability belongs to which domain.

When we examine our library's capabilities, we notice that the login features are mostly generic, since login functions are similar across most applications. Everything related to reading a book falls within the core domains. Marketing and the management of members and partners, encompass both business and generic features. This is why we categorize them as supportive domains. The areas are accordingly marked in *Figure 2.10*:

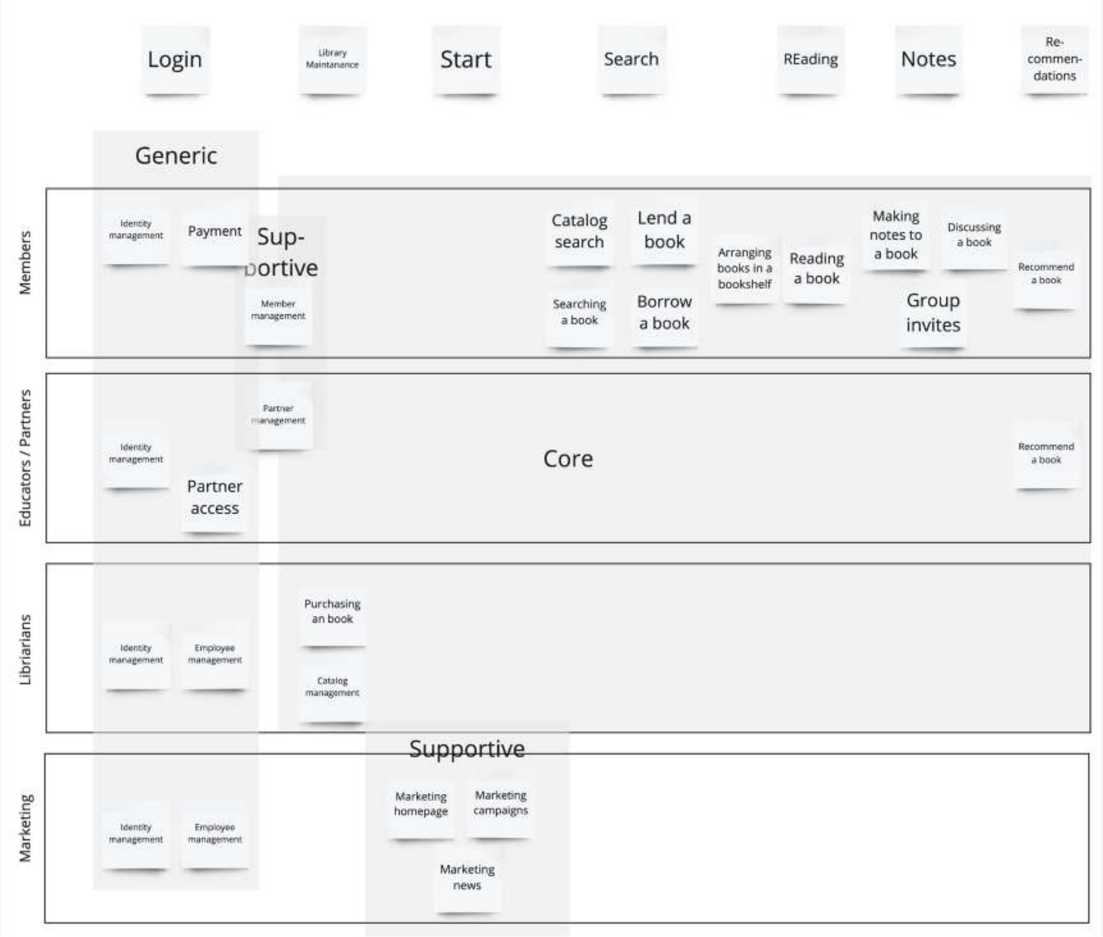


Figure 2.10*: Capability map with subdomain types*

Source*: CapabilityMapWithKindOfDomains.jpg*

One rectangle indicates the generic domains, while the other rectangle signifies the core domains, and the third rectangles represent the supportive domains. The capabilities can be sorted into these different types of subdomains. Each subdomain gets a name indicating its primary functionality.

Now, we can sort out which core domains, supportive domains, generic domains we need. The subdomains are shown in *Figure 2.11*:

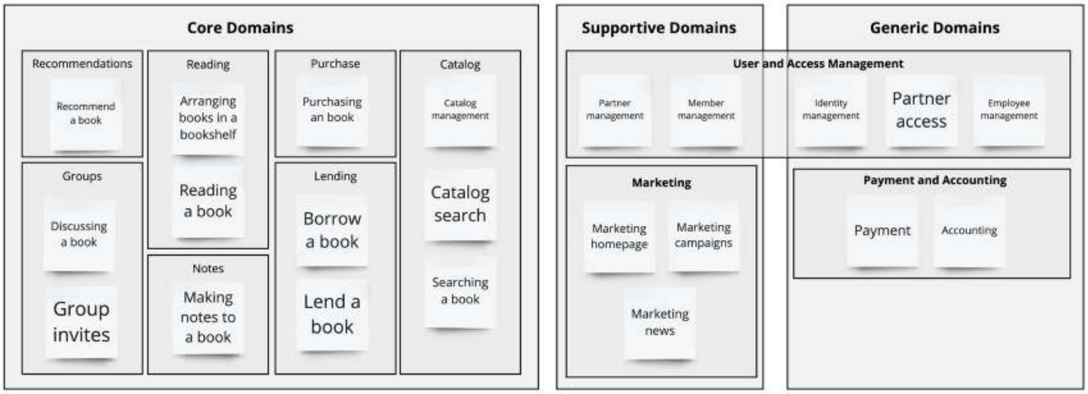


Figure 2.11: *Capabilities sorted by domains*

Source: *CapabilitiesInDomains.jpg*

The analysis provides the following core domains:

- Recommendations
- Groups
- Reading
- Notes
- Purchase
- Lending
- Catalog

Under the umbrella of *User and Access Management*, both supportive and generic capabilities coexist. Such coexistence is quite common for a user and access management, where internal employees as well as external users need to be onboarded. External users require different onboarding processes than internal ones. Consequently, *Partner Management* and *Member Management* serve as supportive capabilities.

Regarding financial functions, *Payment* and *Accounting* are deemed generic capabilities. Although, the capability *Accounting* was not initially specified, its necessity became apparent during the classification of the *Payment* capability.

For the development of our library application, we must prioritize our core domains. We must concentrate on our customers' fundamental needs, adhering to the **Minimum Viable Product** (**MVP**) model outlined in *Chapter 1, Introduction to Domain-Driven Design.*

Therefore, we concentrate on the member capabilities: *Searching a book*, *Borrow/Lend a book*, *Arranging books in a bookshelf*, *Reading a book*, and *Making notes*. Additionally, we would need for an MVP: *Member management* and *Identity management*, *Payment*, and *Accounting*.

The capabilities will be further detailed and how it can lead to implementable architecture is discussed in the following chapters of strategic design:

- *Chapter 4, Bounded Contexts and Domains*
- *Chapter 6, Event Storming*
- *Chapter 7, Context Map*

Conclusion

In this chapter, an online library was introduced which will serve as example throughout the entire book. First the library was roughly sketched and more details were added using a paper prototype. Based on capabilities, the core domains, supportive domains, and generic domains of the library could be detected.

In the next chapter, we will start with the strategic design of our library and see why strategic design is important.

Points to remember

- Business ideas can be a sketch on a piece of paper.
- To get more engagement on an idea, paper prototypes are helpful.
- Based on the paper prototype, a business model canvas can be created to help understand the business ideas and the connected value streams.
- The business model canvas gives a first idea about the necessary business capabilities. Those can be collected.
- Collected business capabilities can be sorted in core domains, generic domains, and supportive domains.
- Core domains are the first to be implemented in an application.

Multiple choice questions

1. **What can help to make a business idea more tangible?**
    a. A paper prototype
    b. A paper flyer
    c. A detailed business plan with Excel tables
    d. A completed implementation of an application

2. **How can a business plan be outlined for discussions?**
    a. Using a large Excel sheet
    b. Using unsorted sticky notes on a whiteboard
    c. Using a business plan canvas
    d. Using a filled-out business plan templated downloaded from the internet

3. **Which kind of subdomains do we know?**
    a. Marketing, accounting, and production
    b. Core domains, supportive domains, and generic domains
    c. Subdomains, domains, and bounded contexts
    d. Classes, objects, and methods

Answers

1. **a.**
2. **c.**
3. **b.**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 3

Why Strategic Design

Introduction

Strategic design, a key element of **Domain-Driven Design** (**DDD**), involves the identification of bounded contexts, the establishment of a ubiquitous language, and the explicit clarification of inter-context dependencies. Bounded contexts do more than delineate the applicability and coherence of domain models; they also demarcate the scope of services within a service architecture. This delineation inherently informs the division of responsibilities among the development teams, facilitating a structure where teams are aligned with distinct segments of the system they are implementing.

Structure

In this chapter, we will cover the following topics:

- Influence of software architecture in a system
- Team independence
- Conway's Law
- Better software

Objectives

This chapter explores the critical role of strategic design within DDD and its profound impact on software quality. You will discover the significance of aligning development teams with the architecture of the intended software and explore how strategic design principles of DDD effectively guide the division of team responsibilities. Additionally, you will gain insights into the ways in which DDD strategic design enhances not only the overall quality but also the security posture of your software system.

> **However beautiful the strategy, you should occasionally look at the results.**
>
> *~Sir Winston Churchill*

Influence of software architecture in a system

When someone sets out to create a software product, they consider value propositions and potential key customers, as illustrated in *Chapter 2, Introduction to the Example Online Library*. However, throughout the entire lifecycle of a software product, its architecture exerts a profound influence. Strategic design addresses these critical questions to construct software that is efficient to develop, deploy, operate, and maintain.

Software design is commonly perceived as the process of crafting classes and methods and delineating the relationships among them. Indeed, this forms a foundational component of software design and architecture, yet our scope must also extend to more expansive considerations. It is at these broader vistas that macro architecture comes into play. Beyond simply making conventional decisions regarding service communication, such as choosing protocols, macro architecture is tasked with defining the very delineation of services themselves. It must ensure that services are self-contained, guaranteeing the consistency of models within a service and their usability solely within that service's context. Identifying the optimal delineation of services is thus crucial.

Strategic design is concerned with the overarching view. It encompasses discerning the challenges a business must confront, pinpointing the domains where it can secure competitive edges, understanding the dynamics between various teams, and synthesizing these insights to determine the most appropriate software to be developed[1].

Team independence during development plays a pivotal role. This autonomy continues to be beneficial during the operational phase, enabling more effective software management.

---

[1] Loughney, W. (2022, July 05). *What you need to know - Strategic Design*. Retrieved from Thoughtworks: **https://www.thoughtworks.com/insights/blog/evolutionary-architecture/domain-driven-design-in-10-minutes-part-one**, Retrieved April 2024

Team independence

When you envision a software project, you likely think of teams engaged in various tasks: coding, designing user interfaces, testing the software, and liaising with customers. Collaboration is key among these individuals. Imagine a project with 50 people, it is a hive of activity where everyone plays a role.

Suppose in this scenario, each person needs to communicate directly with every other colleague to coordinate and reach a consensus. This would necessitate a staggering 1,225 meetings in total. With each participant needing to attend 49 meetings, it becomes clear that such a volume of meetings is impractical within a single week, even if meetings were the only task at hand. It results in an unmanageable synchronization network as illustrated in *Figure 3.1*:

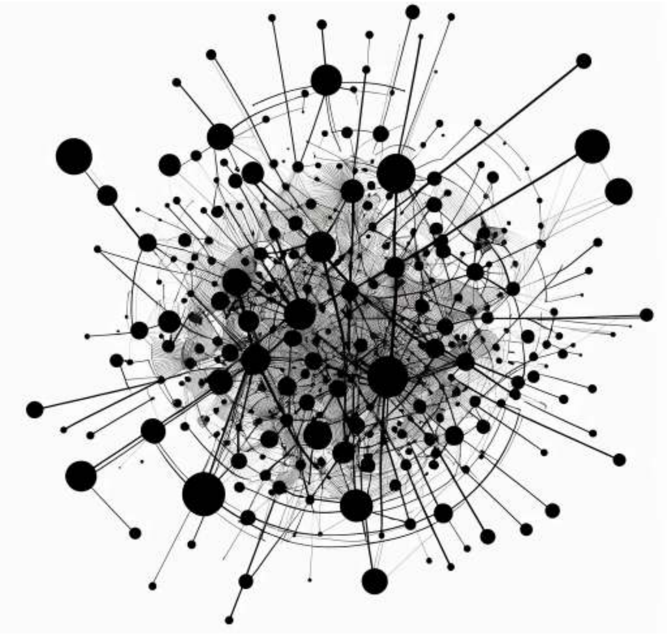


Figure 3.1*: Ball of synchronization hell*

Source*: EachOneToEachOne.png*

To streamline our process, we need to organize specialized groups responsible for synchronization: database developers, business logic developers, user interface experts, testers, sales, and project management. By doing this, we can reduce our meetings to just 30 per week. This amount appears manageable within a standard 40-hour workweek, allotting 30 minutes for each meeting. However, since these meetings may require more than one representative per team, and the outcomes need to be communicated back to their respective teams, the time spent away from productive work becomes substantial. The challenge is to escape this synchronization trap. The goal is to restructure our processes in such a way that extensive synchronization becomes obsolete.

Traditionally in software development, roles were segmented into distinct specialties: database, business logic, and user interface, as shown in *Figure 3.2*:

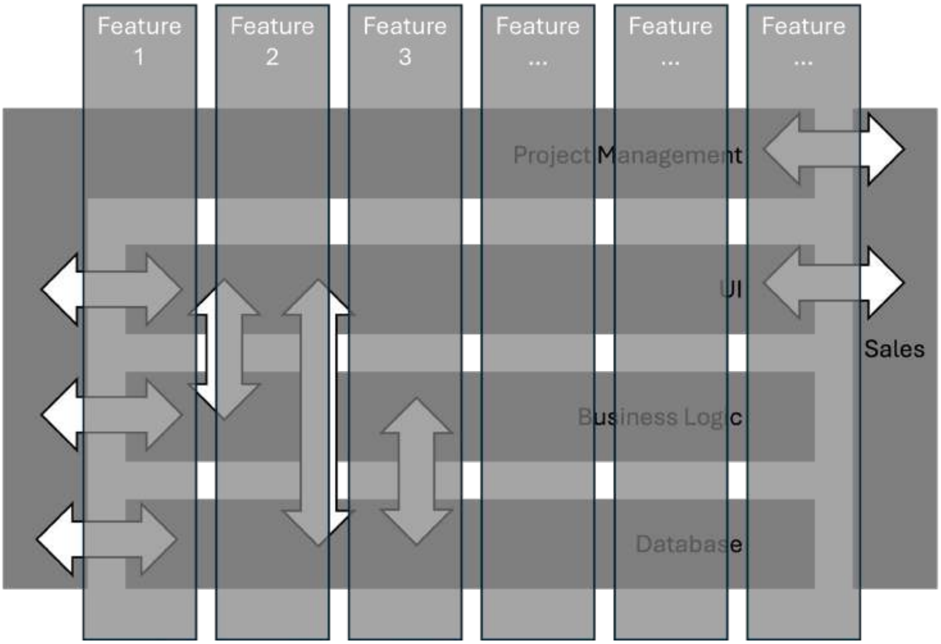


Figure 3.2*: Communication in a classical project setup*

Source*: Classic.png*

This segmentation within organizational structures was commonplace until the advent of the agility movement, which significantly restructured collaborative methods. Despite this paradigm shift, one may still observe such compartmentalized frameworks within organizations as of the current year, 2024. The introduction of a new feature invariably requires the harmonization of all three development tiers, in response to the intricately linked alterations. Consequently, the established workflow that enables teams to generate valuable outputs is disrupted with each addition of a feature, while the imperative to meet delivery deadlines persists. As a result, there is a disproportionate workload close to the delivery date, deemed excessively burdensome, in stark contrast to the initial stages where the workload is relatively minimal, largely due to requisite consultations with project management, sales, and other developmental divisions.

Instead of structuring teams according to technical know-how or activities, organize teams according to business capabilities[2]. Development teams need to be structured around business capabilities and functions, enabling them to operate independently. Consequently, meetings should be confined within teams—for example, daily stand-ups—without the need for extensive cross-team synchronization. External synchronization has

---

[2] Eckstein, J. (2010). Distributed and Dispersed. In D. Smite, & P. J. Agerfalk, *Agility Across Time and Space*. Springer.

become an infrequent requirement. Those feature teams can act independently and take over responsibility, as shown in *Figure 3.3*:

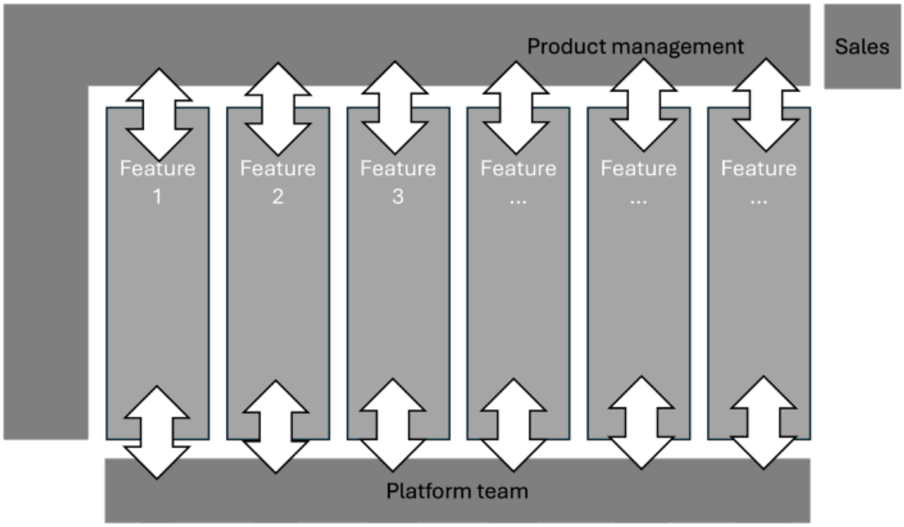


***Figure 3.3**: Feature team*

***Source**: FeatureTeams.png*

To optimize team performance and ensure that teams can operate independently with well-defined responsibilities, it is crucial to minimize dependencies and reduce wait times between teams[3].

Typically, these feature teams are supported by platform teams, as illustrated in *Figure 3.3.* The composition of these platform teams is determined by the technologies the project uses. For further information on platform teams and their roles in the development process, you can refer to *Team Topologies* by *Skeleton and Pais*.

Achieving a high level of autonomy requires identifying features that are inherently independent. Such independence can be effectively established through the strategic part of DDD principles.

Conway's Law

Conway's Law, established in 1968[4], posits that the communication structures of organizations will mirror the designs of the systems they create. This principle implies that the feature teams we establish should not only be independent but also organized around the essential capabilities needed to produce their respective products.

---

[3] Skeleton, M., & Pais, M. (2019). *Team Topologies*. Portland, OR: IT Revolution.

[4] Conway, M. E. (1968, April). How do committees invent?

We have already established subdomains for our online library, but it remains unclear whether they are independent or whether the proposed capabilities can be implemented by a team. The proposed development team structure includes:

- Discussions and group management
- Recommendations and content management
- Reader and bookshelf management
- Notes
- Purchase
- Lending management
- Catalog management
- User and access management
- Payment

If this team structure proves effective, we will need to conduct further analysis in the process of domain-driven strategic design.

Better software

Optimal software offers a seamless and delightful user experience, free from barriers to accessibility. Such software must satisfy the expectations of business stakeholders and resonate with the target user group. These critical aspects are encapsulated by universal defined quality requirements. Employing DDD is pivotal in aligning the software's functionality with these requirements.

Furthermore, in today's digital landscape, cybersecurity is paramount. It is essential that software systems and the data they contain are fortified against unauthorized access. The level of security should be calibrated to the sensitivity of the data. While public information should be readily accessible to its intended audience, personal and especially health-related data demand the highest levels of security to prevent breaches and ensure that only those explicitly authorized may gain access.

Software quality

When independent teams develop software, the quality of the software generally improves. However, what a better software constitutes is unknown. In this context, better software is synonymous with high-quality software, which adheres to the quality attributes defined in ISO/IEC 25010[5]. These attributes include functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, portability, and safety, and they must be met throughout the software development process. The process itself

---

[5] (ISO 25010, 2022)

presents various challenges, as outlined in the *IEEE Software Quality Fundamentals*[6]. These challenges can be effectively addressed using DDD principles. Next, let us explore the specific obstacles encountered in software development and discuss how DDD principles offer targeted solutions to these challenges:

- **Difficulty in clearly defining requirements**: When software requirements are unclear, incomplete, or subject to various interpretations, it challenges the team's ability to deliver a product that aligns with stakeholders' expectations. Such vagueness in the initial stages can propagate errors and defects further into the development cycle. To mitigate this risk, implementing domain storytelling is a proven methodology that facilitates the accurate collection of requirements. This approach ensures a comprehensive understanding among all parties involved, particularly the business stakeholders. Moreover, domain storytelling empowers business experts to effectively evaluate and confirm the comprehensiveness and precision of the requirements, endorsing a solid foundation for the development process.

- **Maintaining effective communication with stakeholders**: Software development is a dynamic and progressive field, where each step in the system's implementation offers stakeholders enhanced insight. To ensure the iterative development of a software product is successful, continuous collaboration is imperative. To bridge the communication gap between developers and business professionals, it is crucial to forge a common, ubiquitous language that is understood by all parties involved. The creation of a visual glossary serves as an invaluable tool in crafting this shared language, fostering clear, consistent communication throughout the development process.

- **Architecture and design errors**: Flaws in the fundamental architecture and design can have a domino effect on the actual software development process and the resulting application. Remedying these failures can be extremely costly. It is, therefore, crucial to ensure that our services and the composition of the teams align with the system's requirements. Event storming is an effective technique for identifying the correct boundaries within the system.

- **Non-compliance with current processes and procedures**: Teams tend to take shortcuts when faced with increasingly tight schedules. To prevent this, it is essential to enable teams to operate independently and distribute their workload evenly over time. Keeping Conway’s Law in mind, team structures should be organized to align with the features and services of a software application. To show the teams where the boundaries are, event storming and context mapping is essential.

- **Documentation errors**: Incorrect, outdated, or completely absent documentation hampers a software engineer's ability to identify issues and update the system

---

[6] Software Quality Fundamentals. (2024). Retrieved from IEEE Computer Society: **https://www.computer.org/resources/what-is-software-quality**, Retrieved April 2024

effectively. Employing domain storytelling, event storming, and context mapping can quickly uncover documentation inaccuracies. Furthermore, by using a ubiquitous language documented in a visual glossary, one can efficiently assess the software's suitability.

We saw how DDD can help to build better software. The next section will discuss how DDD can help, to build more secure software.

Security

Secure software is such a critical aspect in software development, that we put security to separate section, even though it is part of ISO 25010[7].

DDD can be instrumental in identifying potential vulnerabilities and attack vectors within a software system. Specifically, strategic design within the DDD framework provides a methodical approach to segregate services that handle highly sensitive data from those designed to manage less sensitive or public information. By categorizing services based on the level of data sensitivity, it enables the implementation of appropriate security measures tailored to the risk profile of each service, thus fortifying the system's overall security posture against breaches and unauthorized access.

Security within a software system encompasses a suite of critical requirements, paramount among which are confidentiality, integrity, and accountability. Effective security strategies ensure that sensitive information remains private, data is accurate and unaltered, and every action can be attributed to a specific, authenticated entity, greatly reducing the risk of vulnerabilities, and enhancing trust in the system's reliability. Let us explore what this means with regard to strategic design in DDD:

- **Confidentiality**: Confidentiality in a software system refers to its capacity to regulate data access exclusively to individuals who are authorized. Utilizing domain storytelling, stakeholders can delineate and understand the various roles involved in the business process and specify the corresponding data objects each role requires access to. Subsequently, system implementations can undergo rigorous validation to ensure they adhere to these requirements. This method not only aligns technical measures with business needs but also facilitates comprehensive testing of confidentiality against the specified business requirements, thereby reinforcing data security.
- **Integrity**: System data integrity encompasses safeguarding against illicit alterations or erasure by unauthorized entities. It is vital to fortify data not only against malevolent cyberattacks but also against inadvertent system malfunctions. The adoption of visual glossaries can significantly bolster this protection by ensuring uniformity in object nomenclature, which in turn reduces the likelihood of coding errors. Additionally, the practice of event storming is an effective technique for

---

[7] *ISO 25010*. (2022). Retrieved from Portal ISO 25000: **https://iso25000.com/index.php/en/iso-25000-standards/iso-25010**, Retrieved April 2024

delineating system boundaries and clarifying the essential requirements for data exchange. By clearly identifying when and where data exchange is necessary, event storming and context mapping aids in minimizing superfluous data transfers, thereby decreasing the risk of integrity breaches, and enhancing overall system security.

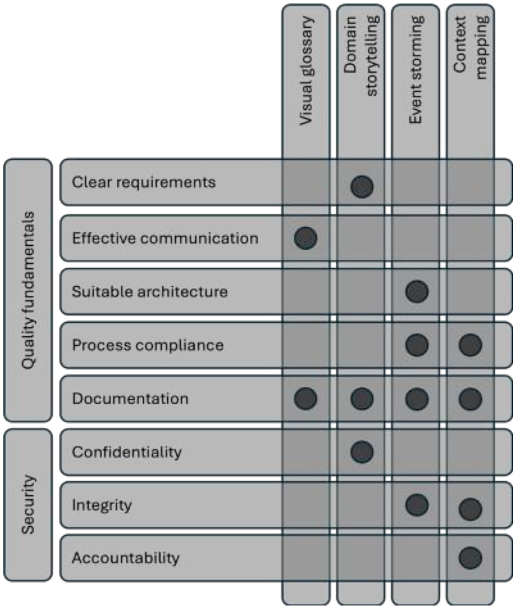
- **Accountability**: In software systems designed with DDD, every action executed by an entity can be singularly attributed to that entity, ensuring clear accountability. DDD emphasizes the creation of distinct bounded contexts, with each context having unequivocal data ownership, which is fundamental for establishing individual responsibility. Particularly, the technique of context mapping within DDD is instrumental in ensuring this accountability. It provides a blueprint for understanding and managing the relationships between different parts of the system, thereby safeguarding that actions taken within one context are transparent and can be traced back reliably to their source, reinforcing the overall integrity of the system.

Other software quality attributes and security aspects like authenticity and non-repudiation will be discussed in the tactical design part of this book (*Chapters 7-10*).

We could see that software quality aspects are supported by DDD.

Software quality aspects and DDD

It is evident that DDD can play a crucial role in producing software of superior quality. The aspects that are particularly enhanced by DDD are delineated in *Figure 3.4*:



		Visual glossary	Domain storytelling	Event storming	Context mapping
Quality fundamentals	Clear requirements		●		
	Effective communication	●			
	Suitable architecture			●	
	Process compliance			●	●
	Documentation	●	●	●	●
Security	Confidentiality		●		
	Integrity			●	●
	Accountability				●

***Figure 3.4**: Supporting of software quality and security aspects by DDD*

Source

Domain storytelling serves as a powerful technique to derive precise and unambiguous requirements, facilitating a solid foundation for project clarity. A visual glossary enhances this clarity by promoting effective communication across teams, ensuring that all stakeholders share a common understanding of key terms and concepts. Employing context mapping enables the development of an architecture that is both appropriate and robust, tailored to the unique demands of the business domain.

Moreover, event storming and context mapping together provide a framework that bolsters process compliance, aligning system behaviors with business workflow and policy requirements. The insights garnered from each of these methodologies contribute to a comprehensive and effective documentation strategy, serving as a valuable reference throughout the software lifecycle.

Additionally, these methodologies collectively reinforce security measures. By outlining clear boundaries, ownership, and data flows, they help identify potential security risks and ensure that safeguarding mechanisms are incorporated into the system's design from the ground up.

The methodologies of the strategic design: visual glossary, event storming, context mapping, and domain storytelling will be discussed in the upcoming chapters (*Chapters 5-7*).

The following chapter delves into the critical concepts of bounded context and domain, elucidating their significance at every stage of the software development lifecycle. It offers an in-depth examination of how these foundational principles of DDD inform the structuring, development, and maintenance of complex software systems, and why they are indispensable for achieving strategic alignment between technical implementation and business objectives.

Conclusion

The chapter outlines the significance of strategic design within the macro architecture of software systems. It emphasizes that strategic design is vital not only for ensuring autonomy among teams but also for exerting a substantial impact on the overall quality of the software. Additionally, the chapter provides a succinct overview of DDD methodologies, highlighting their influence on specific quality and security aspects of the software.

The next chapter will dive into the corner stones of DDD: domain, subdomain, bounded context in great detail.

Points to remember

- Strategic design encompasses the overarching architectural framework that shapes a software system.

- Team independence is essential to construct software systems within budgetary constraints while maintaining quality standards.
- Conway’s Law asserts that the organizational structure of a team will mirror the design of the software they produce, necessitating that teams be organized in alignment with the intended software architecture.
- Adhering to strategic design principles from DDD is critical to achieve and uphold the qualitative benchmarks of a software system.
- Integrating strategic design reinforces a software system's security features, ensuring robust protection against potential threats.

Multiple choice questions

1. **Which part of architecture refers to strategic design?**
    a. Both—Macro architecture and microarchitecture
    b. None of those
    c. Macro architecture
    d. Microarchitecture

2. **Which software quality aspects are supported by strategic design?**
    a. Performance, usability, and resistance
    b. Learnability and capacity
    c. Time behavior, resource utilization, and capacity
    d. Clear requirements, effective communication, suitable architecture, process compliance, effective documentation

3. **Does DDD’s strategic design support security aspects?**
    a. No
    b. Yes
    c. Only tactical design supports security
    d. Strategic design only supports integrity

Answers

1. **c.**
2. **d.**
3. **b.**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 4

Bounded Context and Domain

Introduction

The chapter introduces the most important terms in **Domain-Driven Design** (**DDD**): domain, subdomain and bounded context. Related to the business of an enterprise, we differentiate core domains, supportive, and generic domains. We deep dive into the relationship of subdomain and bounded context and discuss typical architectural patterns which are derived out of it.

Structure

In this chapter, we will explore:

- Domains
- Subdomains
- Bounded context and aggregate
- Patterns in strategic design

Objectives

In this chapter, you will learn the critical role of domains and subdomains in enterprise structures. You will learn how to use Wardley Maps to distinguish core, supportive, and generic domains. The relationship between subdomains and bounded context and the belonging ubiquitous language are discussed.

Moving forward in the chapter, you will learn about typical service pattern: Shared Kernel, Conformist, Anti-Corruption Layer, and Open Host Service. Moreover, you will learn how they support your design goals in a project.

> **Science is a differential equation. Religion is a boundary condition.**
>
> *~Alan Turing*

Domains

The term **domain** indeed spans a variety of contexts with different meanings pertinent to each field. The concept of *public domain* refers to creative works that are not protected by intellectual property laws, such as copyright, patents, or trade secrets. This means that these works are freely available for the public to use without permission or payment to copyright holders. In the field of molecular biology, a *protein domain* is a part of a protein's structure that can fold and function independently of the rest of the protein chain. Each domain typically has a specific role, such as binding to other molecules or catalyzing a chemical reaction. An *internet domain* denotes a unique name that identifies a realm of administrative autonomy, authority, or control on the internet, primarily used in URLs to identify web pages. For instance, in the URL **bpbonline.com** is the domain name that represents the digital identity of an organization or entity's web presence.

When speaking of a domain in the context of software engineering, it refers to the scope of the subject matter that the software addresses, which is the specialized area of knowledge or interest that the software is intended to serve, manage, or automate. The domain shapes the software's features, structure, and user interface to meet the specific needs and requirements of its field of application. It can range from narrow, like a software for a dental clinic management, to broad, like a **Customer Relationship Management** (**CRM**) system designed for various industries[1].

A domain encompasses all activities conducted by an enterprise[2]. More precisely, the scope of a domain may be confined to the internal operations of a single company, though it has the potential to extend beyond that[3]. This concept applies universally, regardless of the size of the enterprise. Therefore, for an insurance company, the domain naturally centers around insuring individuals or property. An IT consultancy firm's domain lies in offering advice to clients on developing and operating IT solutions. Likewise, the domain of a publisher focuses on the publication of books. The enumeration of domains across different industries is vast and diverse.

---

[1] Bjoner, D. (2006). Software Engineering 3. Springer.

[2] Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley.

[3] Kelly, S., & Tolvanen, J.-P. (2008). Domain-Specific Modeling: Enabling Full Code Generation. Hoboken, NJ: John Wiley & Sons, Inc.

The term domain predates 2003, which is when *Evans* notably brought prominence to the concept, emphasizing that models are pertinent only within specific confines.

Evans later defined a domain as, *A sphere of knowledge, influence, or activity*.[4] Applied to the context of a company, the domain effectively captures the entirety of the company's engagements.

Subdomains

Evans introduced the concept of subdomains in 2003, which delineate specific areas of knowledge within a company that are ancillary to its main business activities. While certain operations are universal to most companies, such as employee recruitment, others may be associated with the business but not central to it, such as document archiving for an insurance company.

Subdomains integral to a company's main business are known as **core domains**. These core domains significantly impact the company's operations and success. For instance, an insurance company may have core domains, like sales, contract management, and claims processing. In the case of a library, core domains might include the acquisition of books and the management of book loans. Core domains should be implemented by steady stream teams with a high business knowledge.[5]

Subdomains that are commonly utilized across different companies are known as **generic domains**. Typical examples of such generic domains include user and access management, which can be extensively standardized. Developers can use libraries or even **Platform as a Services** (**PaaS**) via highly standardized **Application Programming Interfaces (APIs)**. As a result, software that supports generic domains is usually available for purchase on the commercial market.

Subdomains that embody both general utility and specific business-related characteristics are termed **supportive subdomains**. For example, archiving systems or document management systems fall into this category. While the fundamental requirements for a standard archiving system are largely uniform, the actual documents stored within require company-specific management, particularly when addressing specific business needs, such as retention periods and the categorization of necessary metadata. Those systems should be bought on the market and internally highly customized.

To find the correct category for already identified capabilities, we used in the *Chapter 3, Introduction to Online Library*, a first approach on how far or tight the according capability is to the business. However, that is only one possible approach. Another approach to do such strategic decisions is **Wardley Mapping**.

---

[4] Evans, E. (2016). Domain-Driven Design Reference. Domain Language, Inc.

Wardley Mapping

To strategically determine which capabilities or subdomains should be developed in-house versus those that can be purchased externally, Wardley Mapping[6] is an invaluable tool. This methodology enables organizations to formulate and navigate their strategic choices. Developed by *Simon Wardley*, a British entrepreneur, the technique draws inspiration from the military strategies of Chinese general *Sun Tzu*[7] and merges these principles with the **Observe, Orient, Decide, Act** (**OODA**) loop conceptualized by *Boyd*[8], thereby generating a novel approach.

Since its introduction in 2005, Wardley Mapping has evolved into a widely recognized standard and has created a broad community of practitioners[9]. The maps themselves are instrumental in guiding decisions about the procurement, customization, or internal development of certain capabilities.

The creation of a Wardley Map is supported by using a specific canvas, as recommended by *Mosior*[10]. This aids in the systematic and effective application of the methodology.

Purpose

In the first step to create the Wardley Map, purpose needs to be formulated. Using the example of the online library, the purpose is:

- Enable students to easily read the necessary literature.
- Enable students to discuss the read literature.
- Enable teachers to propose literature to students.

Scope

In the second step, the scope of the Wardley Map needs to be defined:

- High-level overview of the highest-level strategic priorities.

---

[6] Wardley, S. (2022). Wardley Maps. Simon Wardley.

[7] Gaglieardi, G. (1998-2021). Sun Tzu's Five Elements. Retrieved April 2024, from Science of Strategy: **https://scienceofstrategy.org/main/content/sun-tzus-five-elements**

[8] Luft, A. (2020, March 17). The OODA Loop and the Half-Beat. Retrieved April 2024, from The Strategy Bridge: **https://thestrategybridge.org/the-bridge/2020/3/17/the-ooda-loop-and-the-half-beat**

[9] Tune, N., & Perrin, J.-G. (2024). Architecture Modernization: Socio-technical alignment of software, strategy, and structure. Shelter Island, NY: Manning Publications Co.

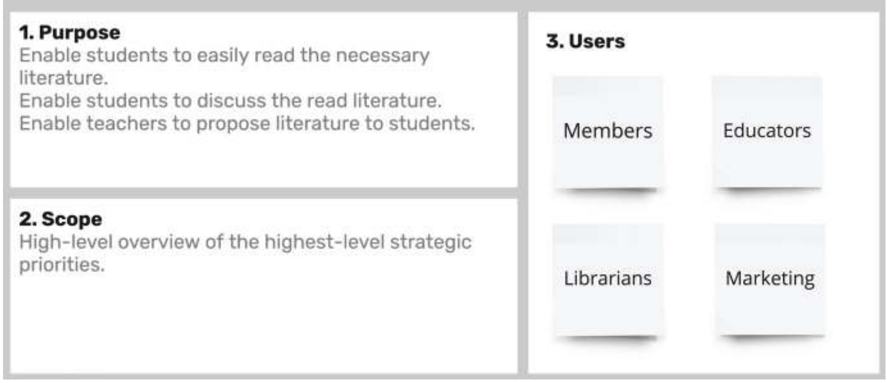
[10] Mosior, B. (2018, May). Understand context and diminish risk: How to build your first Wardley Map. Retrieved April 2024, from Miro:

Users

In the third step, the typical users need to be listed. Out of the example of the online library, the users would be:

- Members
- Educators
- Librarians
- Marketing

The result of the first three steps is shown in *Figure 4.1*:



1. Purpose
Enable students to easily read the necessary literature.
Enable students to discuss the read literature.
Enable teachers to propose literature to students.

2. Scope
High-level overview of the highest-level strategic priorities.

3. Users

Figure 4.1: *First three steps of Wardley Mapping*

Source: *WardleyMapStep1to3.jpg*

User needs

The fourth step contains the user needs for each user group. Students want to read, librarians want to manage book etc. The result of those discussions is shown in *Figure 4.2*:

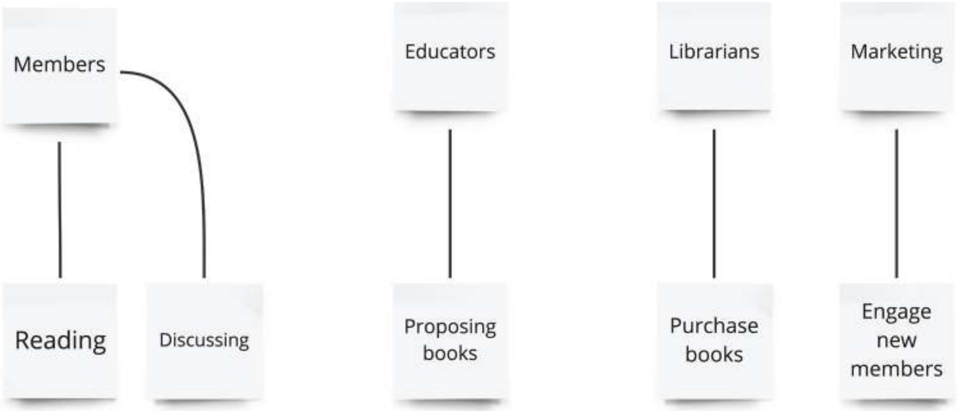


Figure 4.2: *Result of Wardley Mapping*

Source: *WardleyMapStep4.jpg*

Value chain

In the fifth step, the value chain needs to be evaluated. To do so, the users and their needs are copied to the area. Beneath this, catalog the network of components required to satisfy these needs. The term **components** are used broadly here, encompassing anything from actual technical services to a bundle of services, or even entirely virtual solutions. Thus, a detailed debate over the definition of component is not necessary. Initially, leverage the identified capabilities alongside a specific technical solution.

To focus the discussion, this text will primarily consider members and educators since they contribute financially to the library through membership or organizational fees. The needs of internal stakeholders, such as librarians and the marketing team, will be explored separately in an additional value chain model. The result of the discussion is shown in *Figure 4.3*:

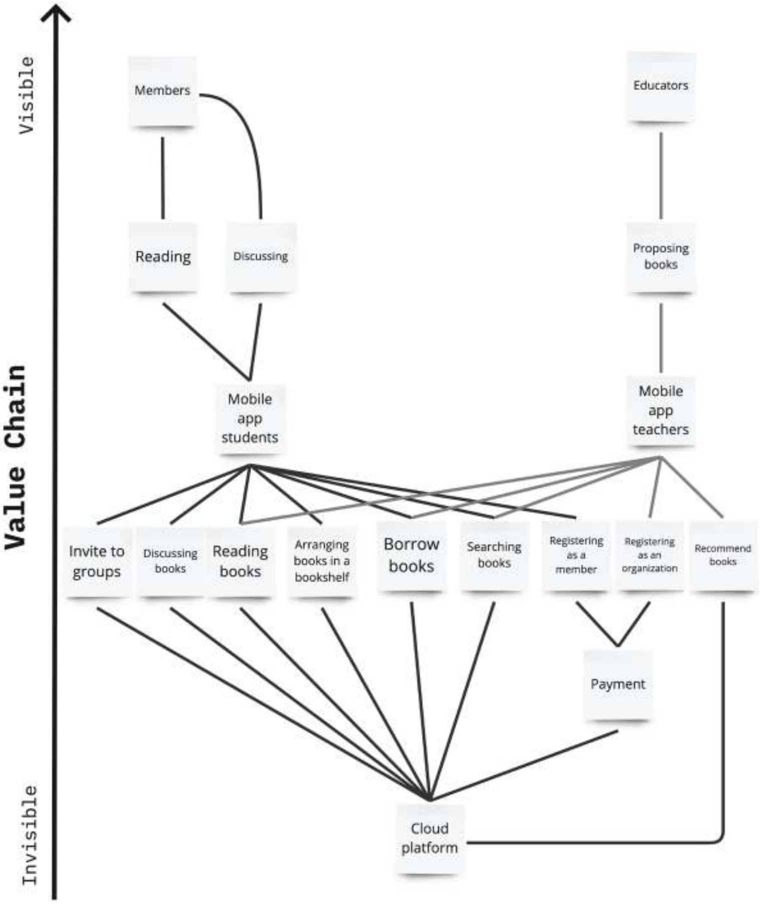


***Figure 4.3**: Wardley Map value chain*

***Source**: WardleyMapStep5.jpg*

The value chain contains the users and the capabilities of the software and its components. The visibility of the components decreases from upper level to the bottom. The lowest components are not visible to the users.

Members and educators use a mobile app which is specialized to the according group. Especially students want to invite other students to learning groups and discuss together book, they read. Educators want to read books and propose those to their following students.

Both user groups need to be registered—either as a member of the library or as an organization. The registration process uses the payment capability. All capabilities should be implemented on a cloud platform.

Librarians and marketeers use an internal web application, to get their needs **Purchasing books** and **Engage new members** fulfilled. The according value chain is shown in *Figure 4.4*:

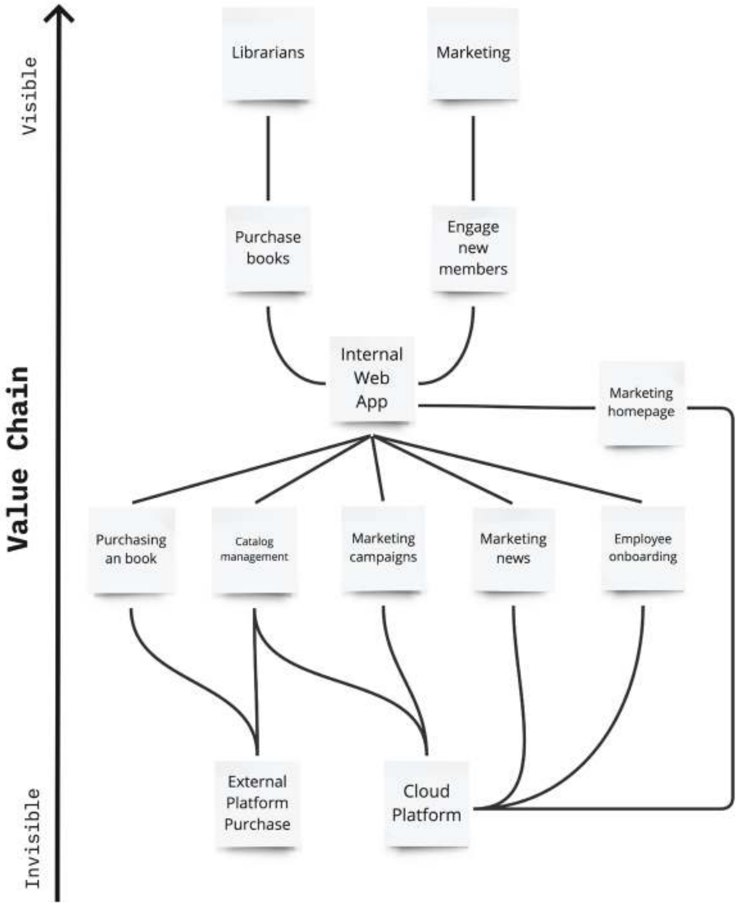


***Figure 4.4**: Value chain of internal users*

***Source**: ValueChainInternals.jpg*

Librians can purchase books via the internal web application. Marketeers can manage the homepage and campaigns via the web application. All components of the internal users use the cloud platform in the same manner as the external users. Additionally, the purchasing of books and the catalog management components use an external purchasing platform, to get prices, page numbers, and other book describing properties easily via an API.

Finally, the according Wardley Map components need to be categorized in their evolution stage to get the Map.

Final step in mapping

The Wardley Map differentiated components along their evolution stage and their market potential. The according stages are called: Genesis, Custom, Product (+rental), and Commodity (+utility).

- **Genesis**: means an undefined market, where failures are highly possible but tolerated. User's perception is highly different and is shaped by confusion, excitement, and surprise. One expects from such a market a high future worth. The market is constantly changing and unstable.
- **Custom**: means a forming market, where failures are moderate and unsurprising, but users are disappointed by them. The market is seen as leading and emerging. One expects from products in such a market profit and **return of investment** (**ROI**). In the market, one can learn from others and testing.
- **Growing**: means a growing market which does not tolerate failures. The focus is on constant improvement. Products in such a market are percept as common and users are disappointed when they are not available. Products are profitable and are differentiated by their feature sets.
- **Commodity**: means a market which is matured. Products are characterized by high efficiency and failures occur seldom. Users notice products as standard and to be expected. Profit can be created by high volume with accepting reduced margins.

The resulting map for the external users is shown in *Figure 4.5*:

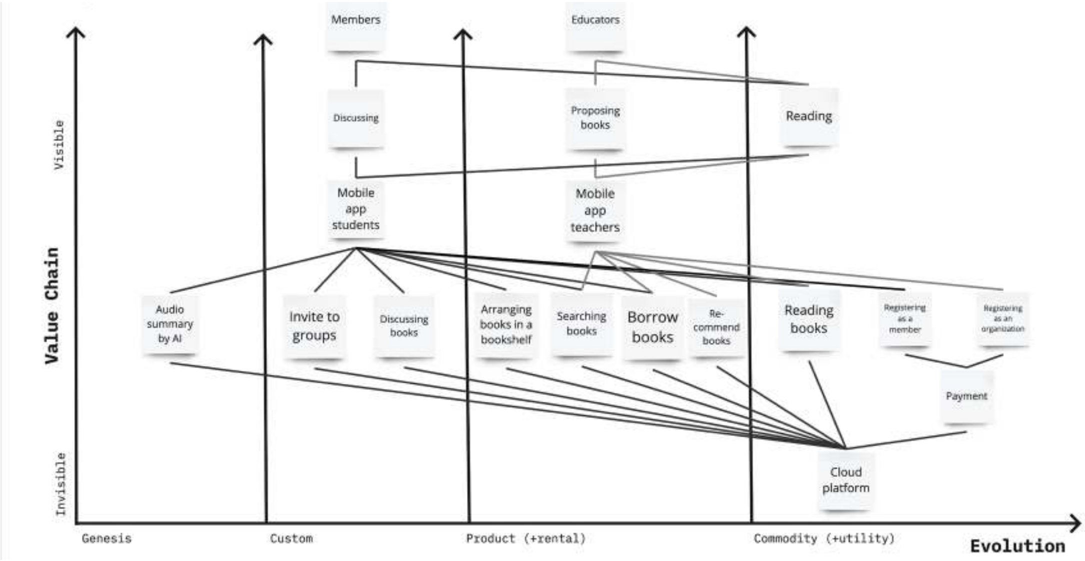


Figure 4.5*: Wardley Map value chain with market evolution*

Source*: WardleyMapStep6.jpg*

In the process of analyzing the competitive landscape with a Wardley Map, it is essential to identify key components that can create a market advantage. Such components may include workgroups in which students collaborate, typically categorized in the custom stage of evolution. However, there is always the opportunity for the emergence of novel components that can captivate and draw in audiences seeking innovation. One example of such an innovative component could be the provision of AI-generated audio book summaries. This unique offering could satisfy the growing demand for accessible and succinct educational content. The according component is sorted in the genesis evolution stage.

The same discussion needs to be done for the internal users. The result is shown in *Figure 4.6:*

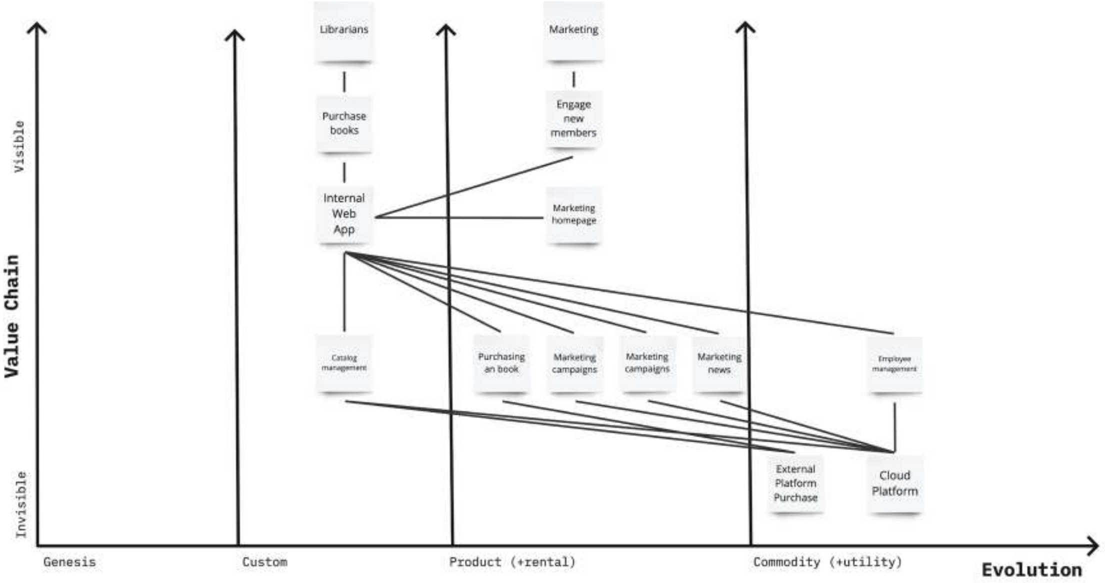
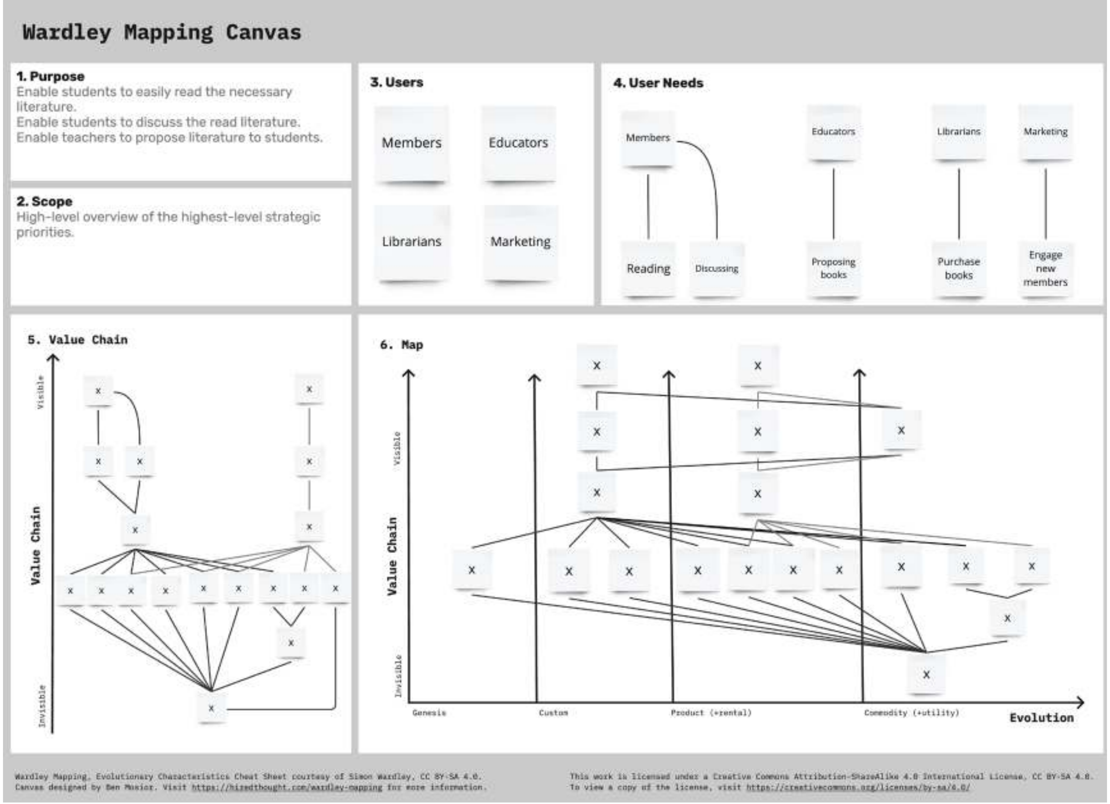


Figure 4.6: *Wardley Map for internal users*

Source: *WardleyMapInternals.jpg*

For internal users, Catalog management is still seen on the Custom stage, whereas Purchasing books and the according marketing components are seen in the product stage. The employee management capability is seen as commodity.

The resulting Wardley Map as a principle is shown in *Figure* 4.7. For the details in the Value Chain and Map areas, please refer to *Figure 4.3*, *Figure 4.4*, *Figure 4.5*, and *Figure 4.6*:



Enable students to easily read the necessary literature.
Enable students to discuss the read literature.
Enable teachers to propose literature to students.

High-level overview of the highest-level strategic priorities.

Wardley Mapping, Evolutionary Characteristics Cheat Sheet courtesy of Simon Wardley, CC BY-SA 4.0.
Canvas designed by Ben Mosior. Visit https://hiredthought.com/wardley-mapping for more information.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License, CC BY-SA 4.0.
To view a copy of the license, visit https://creativecommons.org/licenses/by-sa/4.0/

***Figure 4.7**: Complete Wardley Map for zooming in please refer to the single results presented before*[11]

***Source**: WardleyMapResult.jpg*

With the analysis provided by Wardley Map, the core, supportive, and generic domains as shown in the capability matrix can be further distilled.

Distilling core, supportive, and generic domains

With the analysis of the capabilities and further the analysis of user's needs and belonging capabilities, we can distinguish the capabilities and their subdomains.

---

[11] Wardley Map created using Miro: Wardley Mapping, Evolutionary Characteristics Cheat Sheet courtesy of Simon Wardley, CC BY-SA 4.0, Canvas designed by Ben Mosior, https://hiredthought.com/wardley-mapping

Out of the competition, differentiation capabilities in the evolution stages of genesis and custom should be in a core domain. Capabilities in the evolution stage of product, should be in supportive domains, and capabilities in commodity should be in generic domains as shown in *Figure 4.8*:

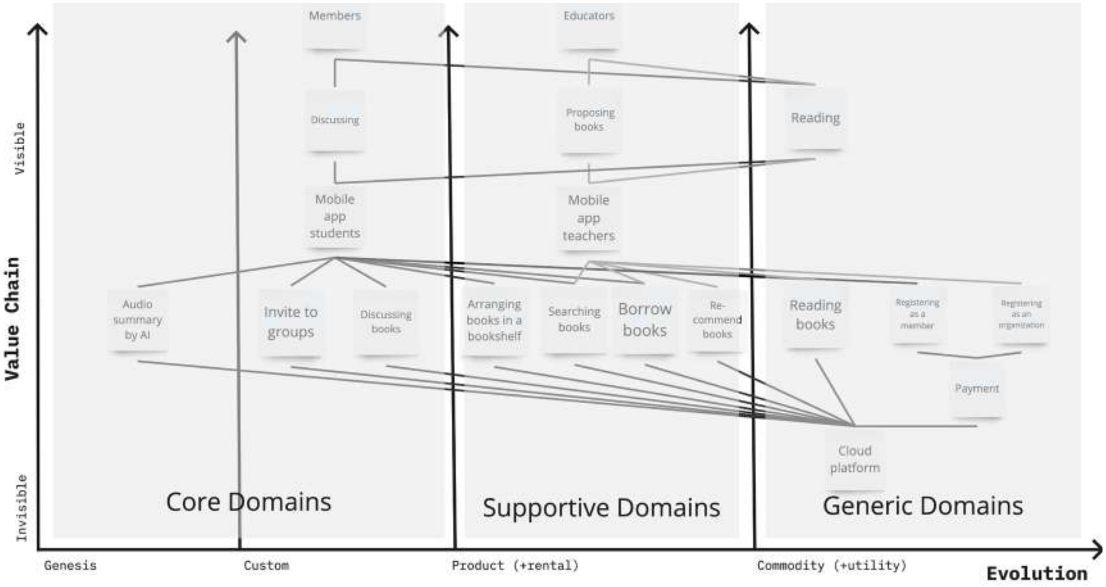


***Figure 4.8**: Subdomains and capabilities/components in Wardley Map*

***Source**: WardleyMapAndSubDomains.jpg]*

Initially categorized as a core domain, the Reading capability has been reclassified as a commodity through the analysis with a Wardley Map. Given its fundamental importance, any failure in this service would be unacceptable to users. Consequently, it has been allocated to a generic domain, where it can be reliably sourced from the market with reduced risk of failure.

While the *Recommendation* feature provided by educators was designated to a supportive domain by the analysis of the Wardley Map, the centrality of student group discussions and the value of recommendations, whether from educators or peers, underpin its core functionality. As a result, the *Recommendation* capability remains within the core domain to reflect its strategic significance.

Although purchasing books is deemed a supportive domain—given that such software is readily available and customizable from the market—catalog maintenance and search are essential to the core domain. A swift and seamless search capability is critical; this is not just a matter of the search engine technology, which is indeed a commodity, but more importantly, it depends on the diligent curation and upkeep by librarians. Thus, the software needs to be finely tailored to librarians' specific requirements.

Lastly, while the ease of borrowing books is a necessary feature, it does not constitute a core function and, accordingly, can be categorized as a supportive domain.

The changed capability map is shown in *Figure 4.9*. The changes to the capability map in *Chapter 2, Introduction to the Example Online Library*, are shown by an exclamation mark:

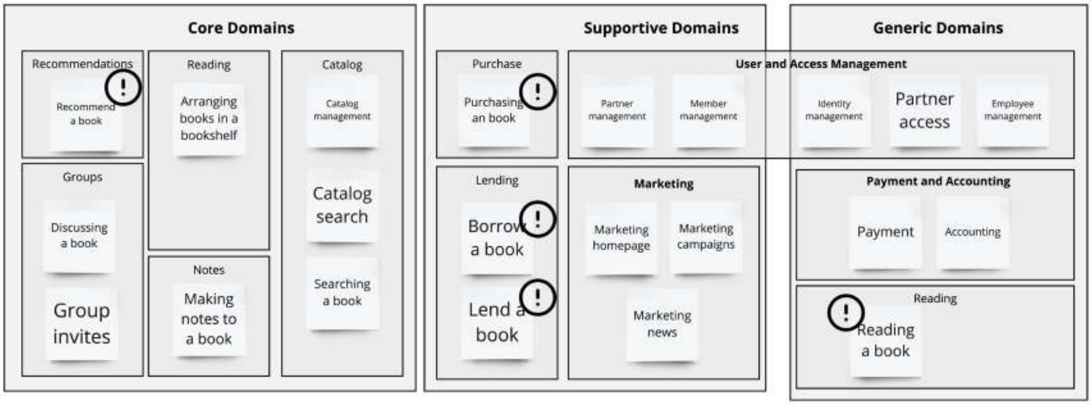


***Figure 4.9**: Changed capability map after analysis with Wardley Map*

***Source**: ChangedCapabilityMap.jpg*

Table 4.1 encapsulates a concise synopsis of the capabilities inherent to the online library, each aligned with its respective phase of maturity according to the Wardley Map and the distilled subdomain category:

Capability	Evolution stage	Subdomain category	Note
Recommend a book	Product	Core	The library's offerings are inherently enriched by the indispensable act of curating a collection that is not only pedagogically substantive but also meticulously aligned with the overarching educational goals.
Discussing a book	Custom	Core	Unique selling point
Group invites	Custom	Core	Unique selling point
Arranging books in the bookshelf	Custom	Core	Unique selling point
Audio summary by AI	Genesis	Core	This one of the library's unique selling points focuses on technical resources, particularly curated to resonate with the interests and academic needs of technologically adept students.

Capability	Evolution stage	Subdomain category	Note
Making notes to a book	Custom	Core	Unique selling point
Catalog management	Custom	Core	Critical component in the delivery of library services Functioning as the foundational framework that enables efficient organization, retrieval, and distribution of information resources
Catalog search	Custom	Core	The procurement of standard library services, while anticipated and essential, lacks the element of innovation that excites users. However, outsourcing these services to market suppliers significantly increases the library's dependency on external providers.
Searching a book	Custom	Core	As said before—fundamental service of a library
Purchasing a book	Product	Supportive	Even though, it belongs to the core competencies of a librarian to purchase books, the necessary software only needs to support them. Even more corresponding platforms, which need to be integrated, are on the commodity stage.
Borrow a book / Lending a book	Product	Supportive	Book lending services serve as a supportive aspect of the library's operations, offering flexibility through customization and enhancement with third-party products.
Partner management / Member management	Commodity	Supportive	Although member and partner management can be facilitated using available market solutions, there is an imperative for significant customization to ensure that the library's unique services are effectively and distinctively conveyed to its members and partners.
Marketing	Product	Supportive	Market products can be used for marketing activities like campaigns, but they need to fulfill the specifics of the online library.
Identity management	Commodity	Generic	Several products are available on the market. No crucial customization necessary.

Capability	Evolution stage	Subdomain category	Note
Partner access	Commodity	Generic	Several products are available on the market. No crucial customization necessary.
Employee management	Commodity	Generic	Several products are available on the market. No crucial customization necessary.
Payment	Commodity	Generic	Several service providers are available on the market. No customization necessary.
Accounting	Commodity	Generic	Several products are available on the market. No crucial customization necessary.
Reading books	Commodity	Generic	Is expected by users – must be a highly available, well-perceived product from the market

***Table 4.1**: Capabilities in evolution stage and subdomain type*

Out of the shown capabilities and subdomains, the bounded contexts need to be distilled. To the bounded context belong aggregates. We will be discussing both in the next section.

Bounded context and aggregate

When we talk about strategic design in DDD, we need to discuss the term bounded context. Bounded context is the central pattern in DDD. It describes the borders where a model is valid[12].

Evans defines a bounded context as: A description of a boundary (typically a subsystem, or the work of a particular team) within which a particular model is defined and applicable.[13]

This means that, to allow teams to work independently, they need to be supported by a bounded context. Only in such a way, teams can work independently. For the moment, we will assume that the provided capability model is sufficient.

When examining a model, it is crucial to define the appropriate aggregates within the relevant context. Let us consider, a model for a book is developed within a library system. The approach must differ based on whether the model deals with searching for books, acquiring new books, or facilitating the reading of books. Each scenario represents a parallel model with its own distinct definition.

These models adopt a ubiquitous language that is consistently applied within a bounded context. While the ubiquitous language may not extend across the entire company, it becomes omnipresent within its designated scope, as it serves as a common dialect for

---

[12] Fowler, M. (2014, January 15). Bounded Context. Retrieved April 2024, from Martin Fowler: **https://martinfowler.com/bliki/BoundedContext.html**

both business experts and IT specialists. This shared linguistic framework is essential from the initial concept stages, through interface design, and down to the code level. Moreover, the aggregates within the bounded context are named in harmony with this ubiquitous language to maintain coherence and support communication.

An aggregate is closely connected to a bounded context—a domain within which it has a specific and valid meaning. It is not valid outside of its bounded context. Thus, it is essential to clearly identify the context when discussing the model. Terms that are specific to a context may not retain their meaning across different contexts. Given these subtle, yet significant distinctions, clarifying the context is a vital step to prevent misunderstandings that could lead to resource-intensive mistakes during the implementation phase.

We will delve into the nuances of how models differ from one context to another in *Chapter 5, Domain Storytelling*. In that chapter, we will examine the use of visual glossaries as a strategic instrument to clarify these nuances and facilitate unambiguous communication.

Within a bounded context, there may be a few aggregates present. However, it is common for a single aggregate to serve as the representative entity or primary model of that context. For instance, in the context of purchasing, this central aggregate might be book, or in the context of catalog maintenance, it could be catalog. As such, establishing a lead aggregate for each bounded context is considered a good practice.

Moreover, aggregates may encompass subordinate aggregates. Take, for example, a catalog, which may include numerous catalog entries. Despite this hierarchical composition, the catalog remains the emblematic aggregate that symbolizes the given bounded context.

With the bounded context in mind, we can discuss the patterns used in strategic design.

Patterns in strategic design

The strategic design of DDD delivers a couple of typical patterns which point to service and team organization. In the following, the patterns Shared Kernel, Conformist, Anti-Corruption Layer, and Open-Host Service and their implications to team topologies will be discussed in the next section.

Shared Kernel

The Shared Kernel[14] is a design pattern that facilitates collaboration among multiple teams by establishing a common foundation in their work. The according model is shown in *Figure 4.10*:

---

[14] Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley.

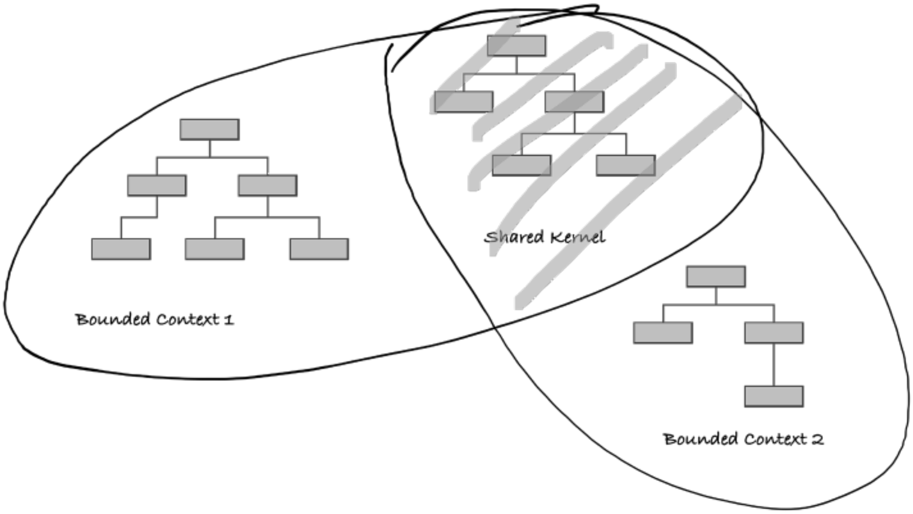


Figure 4.10: *Shared Kernel*

Source: *SharedKernel.png*

In the absence of coordination, teams working on comparable models may be compelled to invest considerable effort into developing translation layers that bridge one model in a bounded context to another in a distinct bounded context. It is this reliance on translation layers that contributes to significant implementation demands and fosters unnecessary interdependencies among systems[15]. Furthermore, these layers have the tendency to introduce accidental complexities into the overall architecture, complicating the system beyond its essential functional requirements.

The better choice would be to implement a common model to avoid those translation layers. That model is called **Shared Kernel**. The common part is defined between the different models. Anyhow any changes at such a model need to be synchronized between the teams. Using a Shared Kernel should only be done when other possible solutions are not worthwhile.

Sometimes a Shared Kernel is seen as an anti-pattern in software architecture. Even if it is not really an anti-pattern, it needs good reason to use it. The teams using the Shared Kernel need to synchronize closely and most probably cannot deploy the services independently from each other.

---

[15] Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley. pp. 354-360

In accordance with the team topologies framework proposed by *Skelton* and *Pais*[16], stream-aligned teams are designed to deliver value continuously and should ideally operate independently to avoid bottlenecks caused by inter-team dependencies. When a Shared Kernel is necessary, enabling teams are better positioned to provide and maintain it, ensuring that stream-aligned teams can access these shared resources without sacrificing their delivery cadence. However, care must be taken to ensure that enabling teams can provide rapid support to maintain the flow of the stream-aligned teams' work. While stream-aligned teams have the capability to contribute to the shared model managed by enabling teams, such involvement should be carefully balanced to prevent a decline in their own delivery velocity.

Conformist

To avoid those tight coupling between teams, one can try to have a common model, but only one team defines the model, and the other team is using it. Such a strategy is called **Conformist**[17] as shown in *Figure 4.11*:

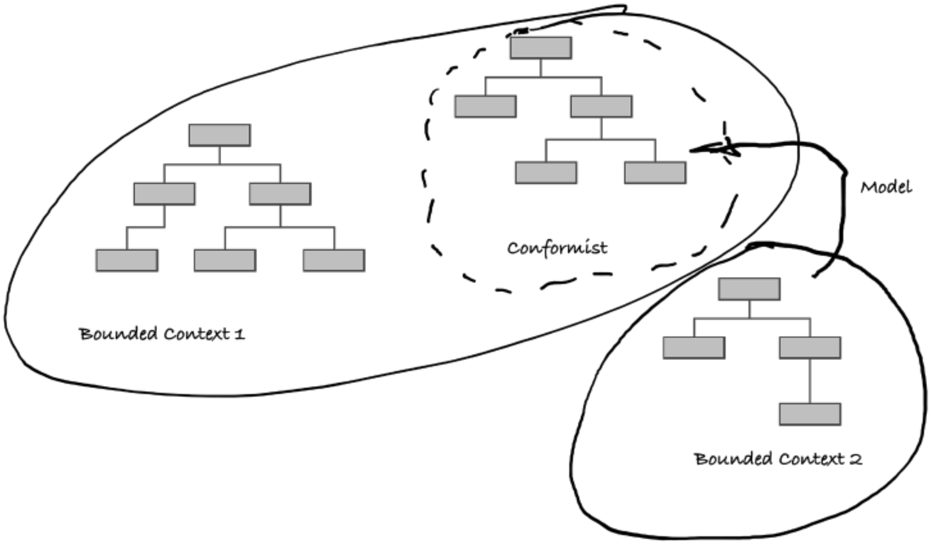


***Figure 4.11**: Conformist*

***Source**: Conformist.png*

The teams are still coupled. Changes in the conform model, need to be adapted by the team using it. Independent deployments are not possible in most of the cases.

The enabling team's model can also be applied effectively within the Conformist pattern. However, as the enabling team is solely responsible for delivery, scaling this model to

---

[16] Skeleton, M., & Pais, M. (2019). *Team Topologies*. Portland, OR: IT Revolution.

[17] Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley. pp. 361-363

meet the needs of stream-aligned teams demands meticulous attention to ensure seamless integration without compromising the delivery process or team dynamics.

When dealing with entirely external or legacy systems, the Conformist and Shared Kernel patterns are not feasible. Under these circumstances, it becomes essential to implement an Anti-Corruption Layer to facilitate the interaction with these systems while preserving the integrity of the new system's design and data structures.

Anti-Corruption Layer

An Anti-Corruption Layer[18] is crucial when an implementation team requires data from a system beyond their control. Frequently, the implementation of a new system necessitates tapping into the data and services of pre-existing legacy systems, which often operate on poorly structured interfaces with outdated protocols. By employing an Anti-Corruption Layer, developers can safeguard the new system against inheriting the obsolescence and inefficiencies of these legacy systems, ensuring a cleaner and more robust integration.

The approach of an Anti-Corruption Layer is shown in *Figure 4.12*:

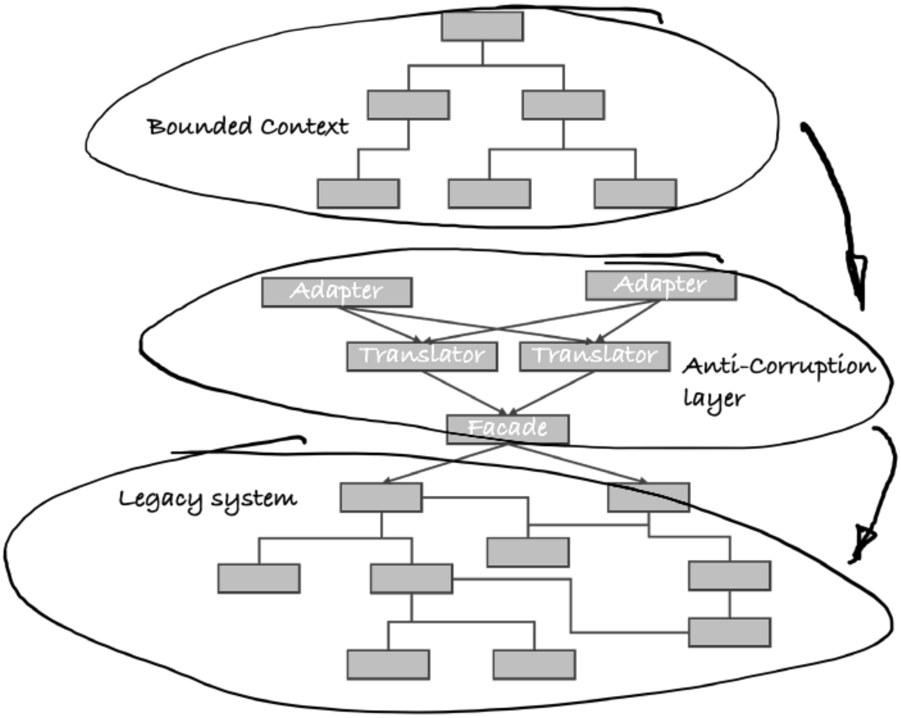


***Figure 4.12**: Anti-Corruption-Layer*

***Source**: AntiCorruptionLayer.png*

---

[18] Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley. pp. 364-370

An Anti-Corruption Layer employs adapter patterns to encapsulate the communication between systems, using translators to convert and align data formats, and a facade pattern to simplify interactions with the legacy system. The underlying design patterns employed within this pattern are explained in the foundational work by *Gamma*, *Helm*, and *Johnson*[19].

An adapter serves as an intermediary that enables a client to interact with a service using a different protocol. It can convert incoming messages into semantically equivalent messages compatible with the service's protocol. A distinct adapter is created for each client service to match the specific service interface. This adapter converts client requests into a format that the facade can understand, effectively concealing the complexities of the underlying legacy system. As implied by its name, the translator component simplifies the complex model of the legacy system, ensuring that it integrates seamlessly with the new system's tailored model.

We will discuss the approach of an Anti-Corruption Layer deeply in *Chapter 14, Usage of DDD in a Brownfield Project*.

In employing DDD to craft services, it is crucial to empower teams to develop their services independently, with a ubiquitous language. The Open Host Service pattern is particularly beneficial for these autonomous teams, as it facilitates the creation and maintenance of well-defined interfaces that ensure seamless interaction and integration with external systems and services.

Open Host Service

An Open Host Service[20] is a service which lives independently, which can be developed and deployed independently. The services communicate via well-defined interfaces as shown in *Figure 4.13*:

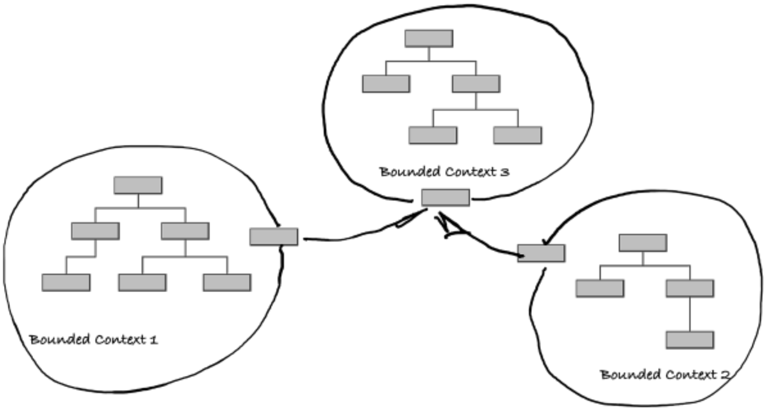


***Figure 4.13**: Open Host Services*

***Source**: OpenHostService.png*

---

[19] Gamma, E., Helm, R., & Johnson, R. (1997). Design Patterns. Elements of Reusable Object-Oriented Software. Prentice Hall.

[20] Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley. p. 374

The services define a protocol that gives access to them by other subsystems. The protocol needs to be open, so that even for the moment unknown services can access it to make future integration easy.

Via the interfaces the internal model becomes public. The language used becomes a published language[21]. Such a language must be well documented, which can be done via standardized API definition languages like OpenAPI[22], AsyncAPI[23], Apache Thrift[24], or even Protocol Buffer[25]. We will discuss them later in *Chapter 10, Exposing aggregates via RESTful APIs* and *Chapter 11, Exposing domain events*.

Open Host Services are the most elegant and useful services when it comes to modern software architectures. They are implemented by stream-aligned teams, which are self-responsible and independent from other teams.

Conclusion

This chapter highlighted the critical role of domains within enterprise structures and detailed the process of distilling subdomains utilizing Wardley Maps. We emphasized the significance of bounded contexts and the adoption of a ubiquitous language. Additionally, we explored architectural patterns, like Shared Kernel, Conformist, Anti-Corruption Layer, and Open Host Service, which are instrumental in implementing bounded contexts.

Moving forward, the subsequent chapter will explore the next phase of our journey. We will employ domain storytelling, a collaborative workshop method, to visualize and construct the system's narrative, ensuring a cohesive design aligned with the project's goals.

Points to remember

- A domain is a sphere of knowledge, influence, or activity.
- Subdomains delineate specific areas of knowledge within a company that are ancillary to its main business activities.
- Subdomains can be core domains essential to the business of an enterprise.
- Subdomains can be generic domains, which can be used cross-enterprise, and which do not have specific business characteristics.

---

[21] Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley. p.375-380

[22] OpenAPI. (2022). The world's most widely used API description standard. Retrieved July 2024, from OpenAPIs: **https://www.openapis.org/**

[23] AsyncAPI. (2024, January). Building the future of event-driven architecture (EDA=. Retrieved July 2024, from AsyncAPI: **https://www.asyncapi.com/en**

[24] Apache Software Foundation. (2024). Apache Thrift TM. Retrieved July 2024, from Apache Thrift TM: **https://thrift.apache.org/**

- Subdomains can be supportive domains, which combine generic as well as business characteristics.
- Wardley Maps can be used to distinguish components necessary for user needs in genesis, custom, product, and commodity evolution stages.
- Components in genesis and custom stages should be part of the core domains because the deliver market differentiation to other competitors.
- Components in the product area should be handled in a supportive domain.
- Components in the commodity area should be handled as generic because they do not deliver any differentiation.
- Bounded contexts deliver a border where a model is valid, and a ubiquitous language is defined.
- A Shared Kernel is a pattern where two bounded contexts share a model – the Shared Kernel. It produces a tight coupling between those bounded contexts.
- A Conformist is a pattern where one bounded context defines a common model for a second bounded context. It produces a tight coupling between both bounded contexts, but less synchronization effort than the Shared Kernel.
- An Anti-Corruption Layer is necessary when new services should be decoupled from existing legacy systems. An Anti-Corruption Layer uses the patterns adapter, facade, and translator.
- An Open Host Service is an independent service, which can be implemented and deployed by one team. Usually, an Open Host Services should be the target of a DDD process.

Multiple choice questions

1. **What is the content of a domain?**
    a. A single model as part of a bounded context
    b. A sphere of knowledge, influence, or activity
    c. A model designed to serve a single purpose
    d. A topic in chemistry and biologic without meaning in software development

2. **What are types of subdomains?**
    a. Supportive, generic, and core
    b. Supportive, core, and business
    c. Business and non-business
    d. Generic and specific

3. **What are Wardley Maps for?**
    a. To find a treasure
    b. To find answers to tactical questions
    c. To give guidance for a game
    d. To get help for strategic decisions

4. **In which subdomain type should be components out of Custom evolution stage of a Wardley Map?**
    a. Supportive domain
    b. Generic domain
    c. Core domain
    d. Could be in each subdomain type, it only depends on the CEO

5. **When you have two teams which can work closely together and which need to handle parts of their models on common, which pattern of service implementation you would choose?**
    a. Shared Kernel
    b. Conformist
    c. Anti-Corruption Layer
    d. Open Host Service

6. **When you design a completely new service, which pattern would be target of your design?**
    a. Shared Kernel
    b. Conformist
    c. Anti-Corruption Layer
    d. Open Host Service

7. **When you need to create new services, which need to access data of a legacy system, which pattern would you implement to decouple the legacy system from the new to be implemented system?**
    a. Shared Kernel
    b. Conformist
    c. Anti-Corruption Layer
    d. Open Host Service

Answers

1. **b.**
2. **a.**
3. **d.**
4. **c**
5. **a.**
6. **d.**
7. **c.**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 5
Domain Storytelling

Introduction

Domain storytelling is a collaborative modeling technique that allows business experts and IT specialists to work together. It crafts a common understanding and ubiquitous language between people with different backgrounds. The chapter introduces the methodologies of domain storytelling and visual glossary as workshop formats.

Structure

In this chapter, we will explore:

- Target of domain storytelling
- Workshop setup
- Domain storytelling for a current system and the intended status
- Visual glossary

Objectives

In this chapter, the reader will learn how to use domain storytelling to create a common understanding between business experts and IT specialists. Furthermore, the reader will learn how to gather business requirements in a visualized way. The workshop format of domain storytelling is highly interactive, and the results can be used as a model for a system to be implemented. The workshop methodology is accompanied by the format of a visual glossary, which captures the terms of the experts and sets them in relation to each other using strong verbs. The reader will learn how to use a visual glossary to establish a ubiquitous language people with different backgrounds use.

> **A picture is a poem without words.**
>
> *-Horace*

Target of domain storytelling

Domain storytelling is a collaborative modeling technique[1]. It allows business experts and IT specialists to work together. The primary purpose is to create a common understanding and language between people of different backgrounds.

To tell stories is a deep human capability. Using stories for communication purposes is our human desire since our first activities in caves[2]. Furthermore, our ancestors wanted to visualize their stories and painted them on the cave walls. To tell a story is neither childish nor archaic, it is part of our humanity.

Storytelling has long been recognized as an effective strategy in business, particularly in marketing, where it has been utilized for decades. The most impactful advertisements often employ brief narratives that enhance the allure of a product.

Given the success of this method, its application in more technical fields, such as software development, should be considered as well. However, introducing storytelling in this context presents a notable challenge: how to make it resonate with IT specialists accustomed to logical and model-driven descriptions of software systems.

IT professionals, including architects and developers, typically conceptualize a software system through diagrams using boxes and arrows—a visual representation of the system's structure. Words and narratives do not always align with their preferred method of understanding.

To bridge this gap, business experts and IT specialists must find a way to merge the art of storytelling with technical modeling. Moreover, the IT specialists need to understand the domain they are working for—and the business specialists as well.

It is the developer's (mis)understanding, not the expert knowledge, that gets released in production[3].

---

[1] Hofer, S., & Schwentner, Henning. (2021). Domain Storytelling: A Collaborative, Visual, and Agile Way to Build Domain-Driven Software. Pearson International.

[2] Yong, E. (2017, December 5). The Desirability of Storytellers. Retrieved May 2024, from The Atlantik: **https://www.theatlantic.com/science/archive/2017/12/the-origins-of-storytelling/547502/**

[3] Brandolini, A. (2016, November 20). Optimized for what. Retrieved May 2024, from Slideshare: **https://www.slideshare.net/ziobrando/optimized-for-what**

To tell a story and listen to it helps in the following ways:[4]

- Understanding the domain
- Finding a common language between the business experts and the IT specialists
- Overcoming misunderstandings
- Clarifying software requirements
- Implementing better software
- Structuring and architecting the software
- Designing digital business processes

To effectively integrate the storytelling strengths of business experts with the modeling expertise of IT professionals, the following collaborative approach is recommended:

- Encourage business experts to share their narratives, conveying requirements and visions for the project through compelling stories.
- Have IT specialists interpret these narratives and express their understanding through models, which could include diagrams or other forms of technical representation.

This interactive process not only fosters a more profound understanding but also ensures that both perspectives are synthesized into a cohesive plan that aligns business objectives with technical feasibility.

Storytelling

To make the approach more understandable, let us tell a story and try to model it. The story should be about handling tasks stored in an inbox.

The business expert tells their first sentence: A clerk creates a task.

So, our picture should contain the following aspects:

- An actor who performs something: a clerk.
- An action which is performed by the actor: creates.
- An object to which the action is directed: the task.

In the visualization, the actor is represented by an icon accompanied by the appropriate label, and the object is similarly depicted with its icon and name. An arrow connects the two, signifying the interaction between them. The action is assigned a numeric identifier, as it constitutes an element of a narrative sequence. The progression of these numbers

---

[4] Hofer, S., & Schwentner, Henning. (2021). Domain Storytelling: A Collaborative, Visual, and Agile

denotes the flow of events within the story. The according sentence visualized is shown in *Figure 5.1*:

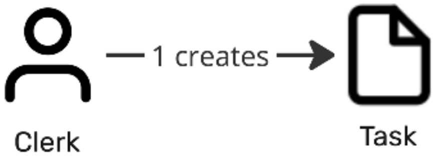


Figure 5.1*: First sentence of the story*

Let us see how our story goes further. The expert tells their second sentence: After the task is stored in the inbox, it is assigned to a clerk. Such a sentence cannot be drawn in the manner of actor-action-object. It even contains two actions: storing and assignment. Therefore, the moderator first needs to draw the storing action. However, the given sentence does not contain an actor. The moderator asks for it. The expert answers with the system. As the system is too unspecific, the moderator asks for a name for the system and gets task manager. So, the second sentence is: The task manager stores the task in the inbox.

Now, our sentence does not only contain the actor, the action, and the object. The sentence contains additionally an adverbial. The adverbial can be shown by an arrow containing the corresponding prepositions without a number. The second sentence enhancing the first one is shown in *Figure 5.2*:

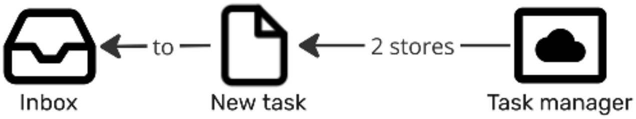


Figure 5.2*: Second sentence*

Source*: SecondSentence.jpg*

The second sentence features a technical system in the role of an actor, an acceptable yet cautious choice since narratives ought to reflect the business perspective. Personifying technical systems can obscure the user-centric focus of the story. Nonetheless, there are exceptions where the primary users are the technical systems themselves. Determining whether to employ this approach should be a decision made by business experts rather than technicians to ensure alignment with business objectives and user needs.

In the third sentence, the actor is not clearly identified. The initial narrative states, *..., it is assigned to a clerk*, with *it* referring to the task. However, we need to specify who is responsible for task assignment and the task's origin. To clarify, the business expert must provide additional details: firstly, the clerk chooses the task from the inbox, and secondly, the clerk then assigns the task, either to themselves or to a colleague. This elaboration ensures a complete understanding of the process.

The next two sentences are as follows:

- The clerk selects a task from the inbox.
- The clerk assigns the task to a clerk.

The visualization is shown in *Figure 5.3*. With this visualization, the business expert can ensure that they were understood correctly:

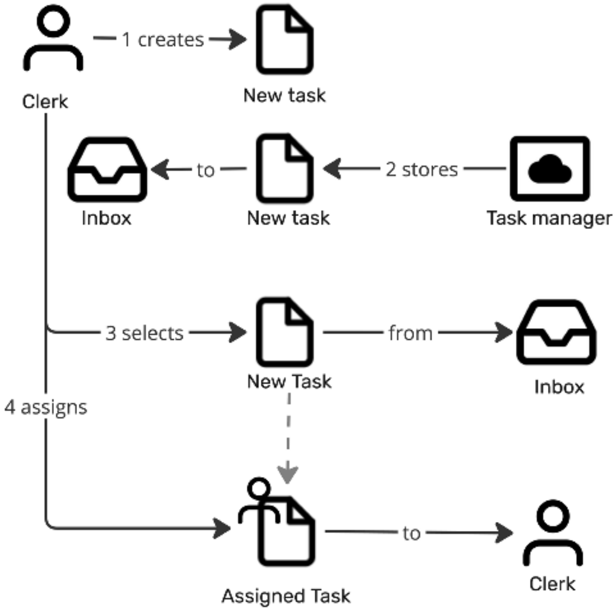


***Figure 5.3**: Third and fourth sentence of the domain story*

***Source**: ThirdFourthSenctence.jpg*

The narrative progresses with, *The clerk selects a task assigned to them and begins working on it*. This prompts a set of queries about the location where the clerk selects their assigned tasks. The necessity for distinct assigned and in-progress status indicators. It is important to remember that during the workshop, attendees have the opportunity to pose questions to both the business expert and the moderator.

Addressing the first question, it is clarified that clerks can choose tasks specifically allotted to them from their personalized inboxes. As for the second query, the answer is affirmative—having specific statuses, like assigned and in progress, helps track the task's journey more accurately. With these clarifications, the next step is to refine the icons representing these distinct task statuses, enhancing clarity and workflow visualization.

As a result of this discussion, we have two more sentences:

- The clerk selects a task from their individual inbox.
- The clerk works on their task.

The visualization of the story is shown in *Figure 5.4*:

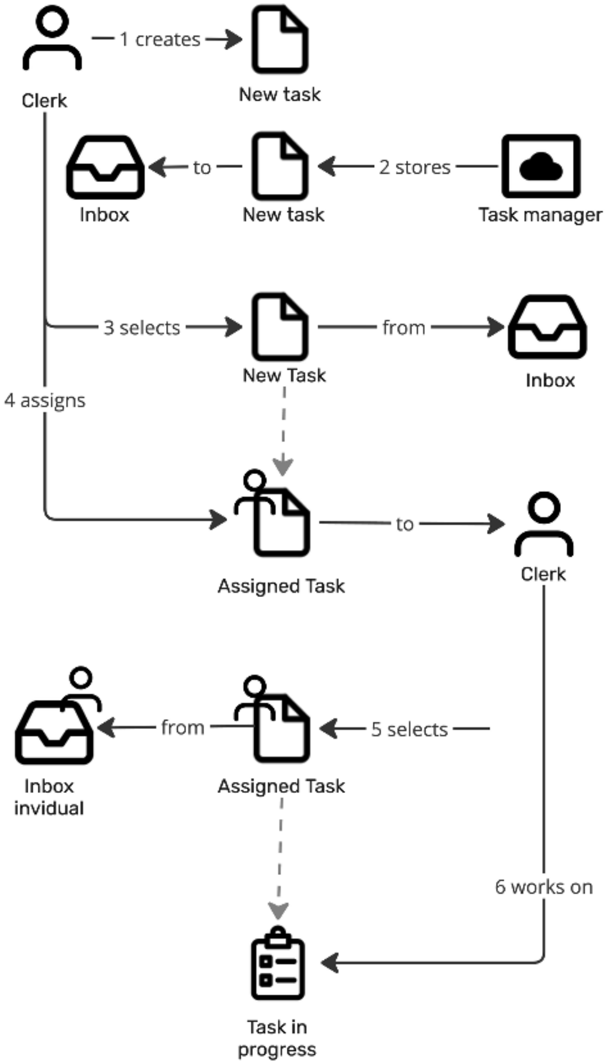


***Figure 5.4**: Sentence 5 and 6 of the domain story*

***Source**: FifthSixthSentence.jpg*

The visualization in *Figure 5.4* shows not only the sentences. It contains the task status as different icons as well—new task, assigned task, and task in progress. The status transitions are shown with grey dashed arrows so that they can be differentiated from the sentences.

The business expert tries their seventh sentence: The clerk finishes their task.

The sentence can be visualized without further questions about it. The result is shown in *Figure 5.5*:

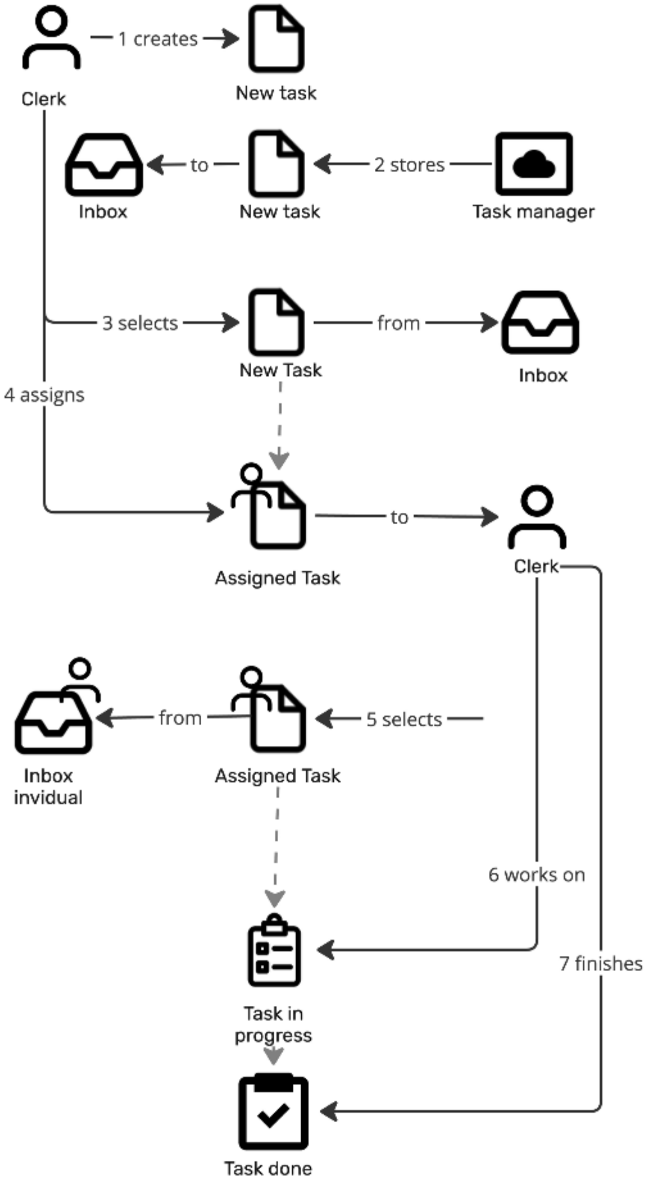


***Figure 5.5**: Completed domain story of the task management*

***Source**: DomainStoryTaskManagement.jpg*

The story is finished, and the business expert can be sure, they were understood. The workshop attendees and the moderator are sure that they understood the business expert.

Now the workshop attendees covered one story of one expert. The next expert wants to enhance the story, that the clerk can reject the task assigned and that such a rejected task is send back to the inbox.

The moderator can enhance the already existing story, as shown in *Figure 5.6*:

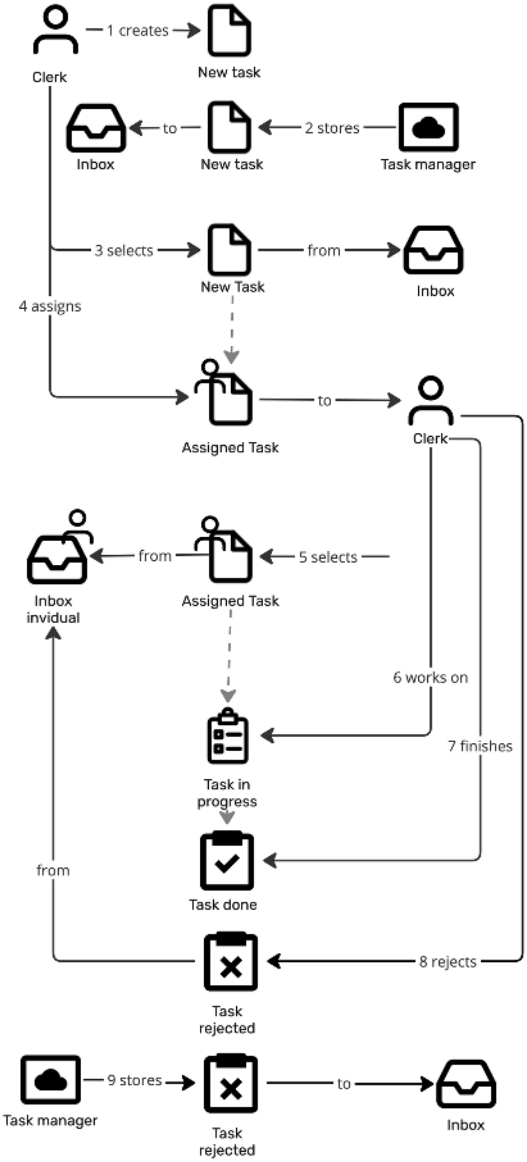


Figure 5.6*: Domain story enhanced*

Source*: DomainStoryTaskManagementEnhanced.jpg*

When several business experts can tell their stories, it is possible to consolidate and generalize them to get a comprehensive picture of the relevant business requirements.

Visual language of domain storytelling

Let us summarize our storytelling. As we used actors in the stories, we use typical icons to symbolize the actors, as shown in *Figure 5.7*:

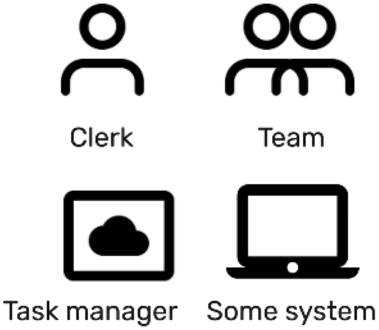


***Figure 5.7**: Typical icons for actors*

***Source**: Actors.jpg*

These icons need to be selected carefully so that they can be understood by others as well. There are great sources to find icons on the internet. The icons used here are provided by Iconfinder[5].

It is noteworthy that individuals in the process are not identified by their personal names but are referred to by their functional roles. Similarly, technical systems are designated by their standard names. If there is any systems that is yet to be actualized, participants of the workshop are encouraged to assign these systems a descriptive or informal name for ease of reference during discussions.

Common objects used in those workshops include files, documents, and an inbox (as demonstrated in the example). To avoid confusion, it is essential for all workshop attendees to have a mutual understanding of the iconography being used. The workshop would be less effective if participants are unclear about what the icons are intended to represent.

Additionally, the objects can be linked with arrows to depict transitions in business status. To clarify the distinct meanings of the used arrows, utilizing varying colors is advisable.

Actions are marked by arrows labeled with the name of the action, for example, assigns. Additionally, they are marked with a number that indicates the sequence of the story.

Annotations can be made via sticky notes or texts marked as annotations, for example, by using a legend box.

Each domain story should get a good name. For example, Task management for librarians. Certain assumptions should be marked as those by text boxes or sticky notes.

---

[5] **https://www.iconfinder.com/**

Groups can cluster activities, such as when they are to be repeated or when they are optional. Groups can be used to indicate subdomains or organizational boundaries.

The completed domain story with groups and caption is shown in *Figure 5.8*:

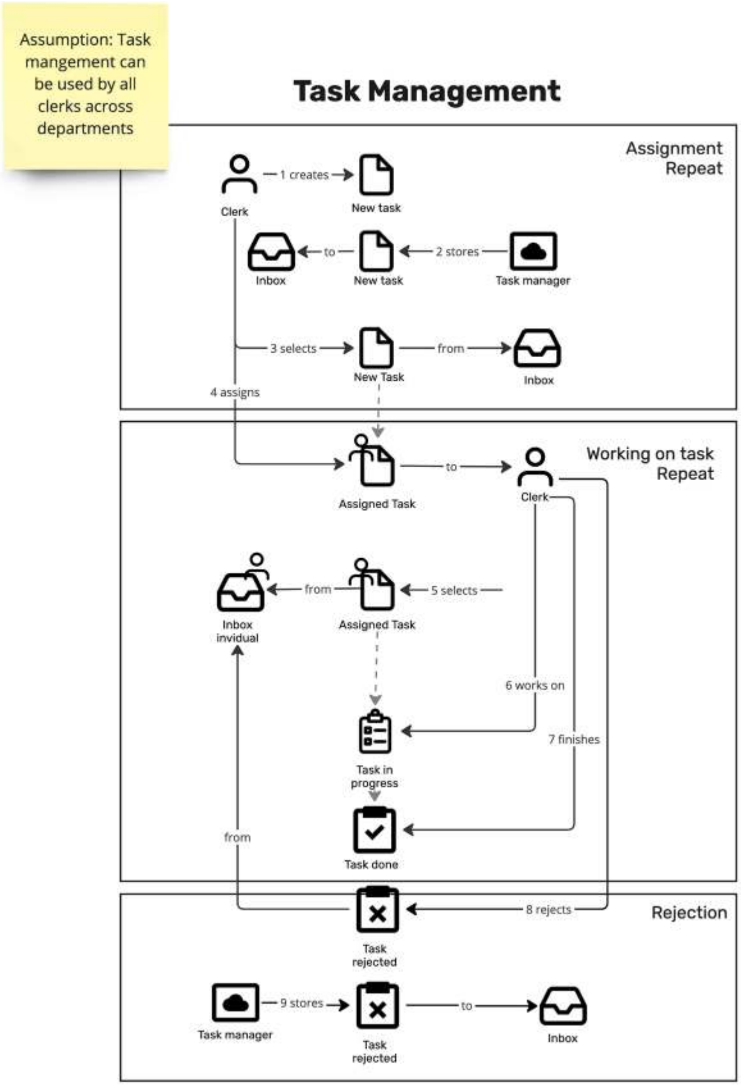


***Figure 5.8**: Completed domain story with caption, groups, and documented assumptions*

***Source**: DomainStoryComplete.jpg*

The shown language omits any conditionals. This choice is deliberate to streamline discussions within the workshop setting[6]. Debating various alternatives often consumes

---

[6] Hofer, S., & Schwentner, Henning. (2021). Domain Storytelling: A Collaborative, Visual, and Agile Way to Build Domain-Driven Software. Pearson International., p. 24

more time than focusing on the intended solution, which can significantly slow down the workshop's progress. Furthermore, incorporating conditionals complicates the language, making it less intuitive and, as a result, less effective for a workshop environment. To address those alternatives, we can use scenario approaches.

Workshop setup

Domain storytelling is a workshop format. It cannot be prepared and then discussed with the workshop attendees. The interaction of the workshop attendees is needed to get the unambiguous language and the necessary common understanding.

Workshops can be conducted both remotely and in person. Typically, in-person workshops tend to be more effective, particularly when participants are meeting for the first time. This format fosters direct interaction and engagement among attendees.

For a successful workshop, the setup should facilitate open discussion among participants. For instance, during an in-person workshop, seating should be arranged to allow attendees to see and engage with one another easily. In a remote setting, all participants should be encouraged to keep their cameras on to maintain active participation and a sense of connection.

It is crucial to focus on the main discussion and minimize side conversations, which can detract from the workshop's value. In a remote workshop, for example, using the chat function for separate discussions while the main conversation is directed by the moderator should be discouraged. This ensures that all participants contribute to and benefit from the collective dialog.

Anyhow, the workshops need to be setup with the right attendees to be successful.

Workshop attendees

Workshops should be done with workshop attendees of all stakeholder groups: IT specialists, business experts, UX experts, etc. As more diverse the workshop is set up, more perspectives are coming in, and it will be more successful.

The workshop should not be done with less than five attendees and no more than 15. With less than five attendees, the discussed perspectives are less diverse to get good results. With more than 15 attendees, the discussions can become tiresome, and the quality of the results decreases.

Different tools can be used to capture the story, such as whiteboards, modeling tools, or even manual drawings.

Whiteboards

The examples provided in this book have been designed using *Miro*[7], which is an online collaborative whiteboard platform. Alternative tools, such as *Conceptboard*[8] and *Mural*[9], also exist and offer comparable functionalities. With the wide array of options available on the market, it is important for workshop participants to choose a tool that aligns best with their specific needs.

Often, organizations have a preferred whiteboard tool that they endorse for use. Workshop attendees are generally encouraged to utilize the tool that their company supports to ensure consistency and integration with existing workflows.

To streamline the workshop preparation process, it is advisable to proactively copy the necessary icons onto the whiteboard area before the session begins. This will eliminate the need to search for them during the workshop, allowing for a smoother and more efficient flow. Additionally, it is recommended that captions be prepared for the stories that will be discussed to ensure that the focus remains on the topic at hand and to prevent any unnecessary and time-consuming debates about the scope of the workshop.

Modeling tools

There are few modeling tools available on the market that support domain storytelling. However, two are recommended.

Foremost among the available tools is Egon.io, the development of which was facilitated by the pioneers of domain storytelling, *Hofer and Schwentner*. For a practical illustration of its utility, refer to *Figure 5.9*:

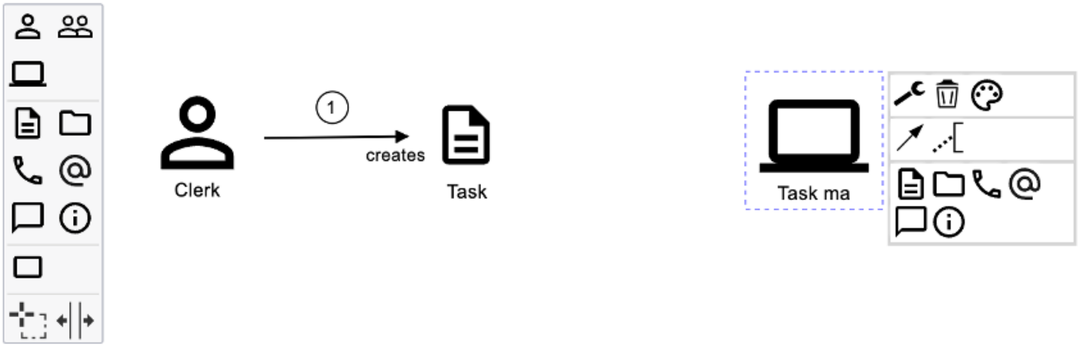


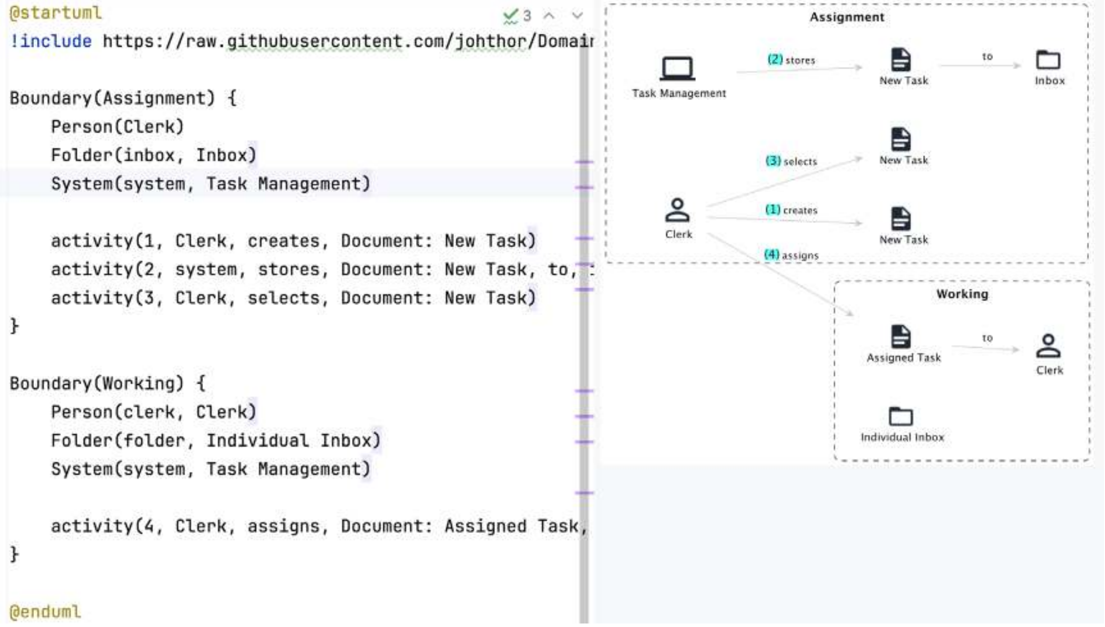
***Figure 5.9**: Using Egon.io to create domain stories*

***Source**: egonio.png*

---

[7] **https://www.miro.com**
[8] **https://conceptboard.com/**
[9] **https://www.mural.co/**

Another good possibility would be to use an extension of PlantUML for domain stories provided by *Thorn*[10]. PlantUML[11] is a markup language in which standard **Unified Modelling Language** (**UML**) diagrams, like class diagrams, can be described. This language was enhanced in a way that makes it possible to describe domain stories. The code and the rendered picture as a sample are shown in *Figure 5.10:*



```
@startuml
!include https://raw.githubusercontent.com/johthor/Domai

Boundary(Assignment) {
    Person(Clerk)
    Folder(inbox, Inbox)
    System(system, Task Management)

    activity(1, Clerk, creates, Document: New Task)
    activity(2, system, stores, Document: New Task, to,
    activity(3, Clerk, selects, Document: New Task)
}

Boundary(Working) {
    Person(clerk, Clerk)
    Folder(folder, Individual Inbox)
    System(system, Task Management)

    activity(4, Clerk, assigns, Document: Assigned Task,
}

@enduml
```

Figure 5.10: *Domain storytelling using PlantUML*

Source: *plantuml.png*

The drawing can be done physically on a whiteboard as well. The icons can be drawn on prepared sticky notes, and during the workshop, the activities are drawn as shown in *Figure 5.11*:

---

[10] Thorn, J. (2024, April 22). DomainStory-PlantUML. Retrieved May 2024, from **https://github.com**: **https://github.com/johthor/DomainStory-PlantUML**

[11] **https://plantuml.com/**

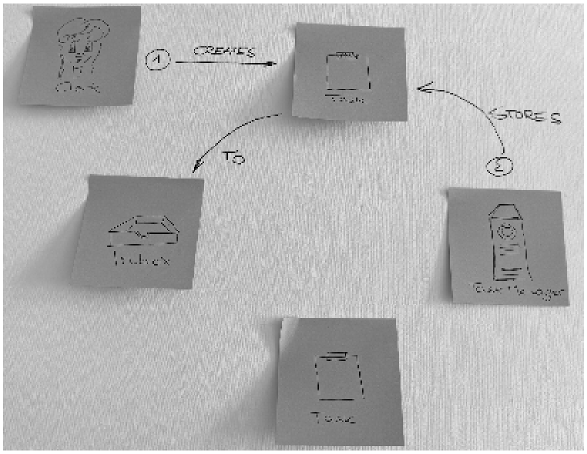


***Figure 5.11**: Domain story as whiteboard drawing*

***Source**: WhiteboardDrawing.png*

It is crucial for the moderator to be proficient in the chosen tool to ensure a seamless workshop experience, avoiding any unnecessary pauses that could inconvenience the attendees awaiting the completion of the drawings.

The moderator and their colleagues should prepare the icons necessary. There might be a couple of specific icons, like user or document, and there might be a couple of new icons specific to the domain to be discussed.

Workshop length

For optimal effectiveness, domain storytelling workshops, while invaluable for creating a shared understanding of the domain, can be intensive endeavors for the moderator. To maintain focus and energy levels, it is imperative to manage the duration of these sessions strategically. While it is crucial not to curtail discussions prematurely, as dialog is central to the workshop's objectives, the sessions should be as concise as possible without compromising their effectiveness.

For systems of lesser complexity, such as the discussed task management, a half-day workshop may suffice. However, for more complex systems like the online library, it is advisable to allocate at least half a day for each anticipated subdomain to ensure thorough exploration and understanding.

Breaks between the subdomain discussions are essential to facilitate information processing and retention. Yet, these pauses should be carefully scheduled so as not to exceed a fortnight, ensuring continuity and engagement among participants.

Furthermore, while our discussion has so far focused on the application of domain storytelling for systems in the development phase, it is worth noting that this methodology is equally applicable to existing systems. It serves as a powerful tool for analyzing and improving upon current structures by illuminating how they can be adapted or enhanced to better serve the domain's needs.

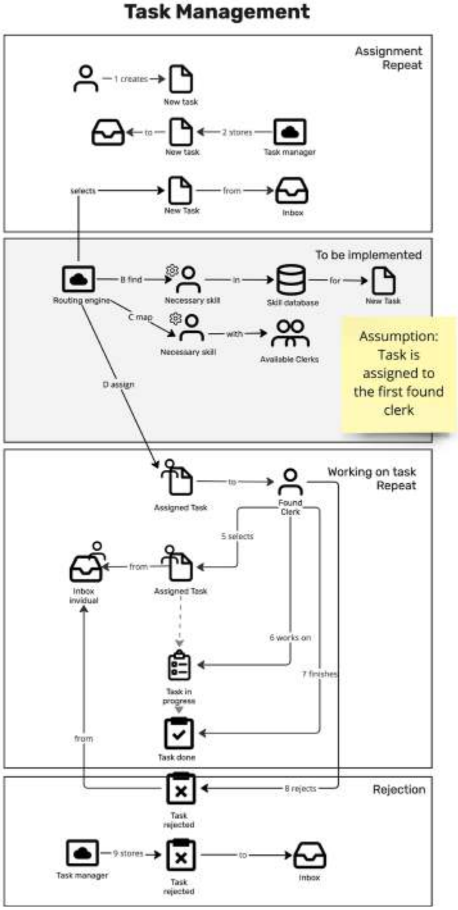
Enhancing domain storytelling for existing systems

Imagine a system in which a certain feature should be enhanced, when we take our task management system from the beginning and assume that it is already implemented. Now it should be enhanced that the tasks are not routed manually anymore.

We can take the previously prepared story for task management.

In a workshop, the expert should tell the story of how the current system works to set a common ground.

Upon establishing a shared understanding, the new feature to be implemented can be described. This new feature should be visually distinguished either using different colors or by employing varied arrow styles. Additionally, enhancements can be represented by introducing new groups, as depicted in the following figure:



Assumption: Task is assigned to the first found clerk

Figure 5.12: Domain storytelling for existing systems

***Source**: DomainStoryIntended.jpg*

The new part is shown as a grey box indicating a group, which should enhance the existing system. The according sentences are numbered by characters rather than numbers to indicate the enhancement. The additional sentences are as follows:

- The automatic routing engine selects a task from the inbox.
- The automatic routing engine finds the necessary skills in the skill database for the new task.
- The automatic routing engine maps the necessary skills with the available clerks.
- The automatic routing engine assigns the new task to the found clerk.

The example demonstrates that domain storytelling, as a workshop format, can effectively articulate even the refinements to established systems. Let us now explore whether this method can also successfully convey the intricacies of a more complex system, such as an online library.

Domain storytelling of the online library

The domain story of the online library is not one story. It is constructed of multiple stories around the already detected subdomains.

Purchase of books

The following domain story describes the purchasing of books. *Figure 5.13* shows the purchasing of books:

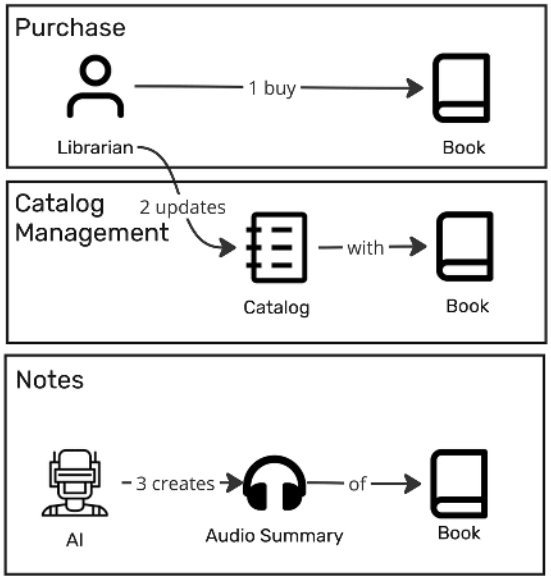


***Figure 5.13**: Domain story for purchasing books*

***Source**: Purchase.jpg*

The purchasing of books does not only contain the purchasing subdomain. It contains the subdomains of catalog management and audio summary as well. The story contains the following sentences:

- **Subdomain purchase**: The librarian buys a book.
- **Subdomain catalog management**: The librarian updates the catalog with the book.
- **Subdomain notes**: The AI creates an audio summary of the book.

The next complex to be discussed is membership management.

Membership management

The membership management must include the registration and the payment of member fees. An example of the membership management story is shown in *Figure 5.14*:

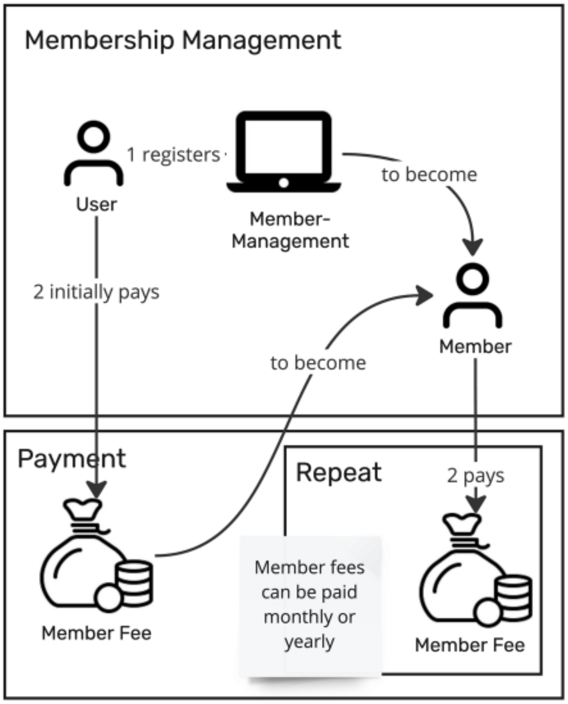


***Figure 5.14**: Domain story membership management*

***Source**: MembershipManagement.jpg*

The story covers the intertwined subdomains of membership management and payment. Payment options are flexible, allowing for either monthly or annual contributions. This detail is visually represented by a box marked Repeat, further clarified with an adjacent sticky note specifying the frequency.

The story contains the following sentences:

- **Subdomain membership management**: A user registers at member management to become a member.
- **Subdomain payment**: A user initially pays the member fee to become a member. A member pays the member fee.

After these preliminaries of buying books and membership management, the workshop attendees can discuss the subdomains affected by lending and reading books.

Reading

Although we saw that reading is a generic domain, there are other capabilities that are worth describing in more detail. The reading story is shown in *Figure 5.15*:

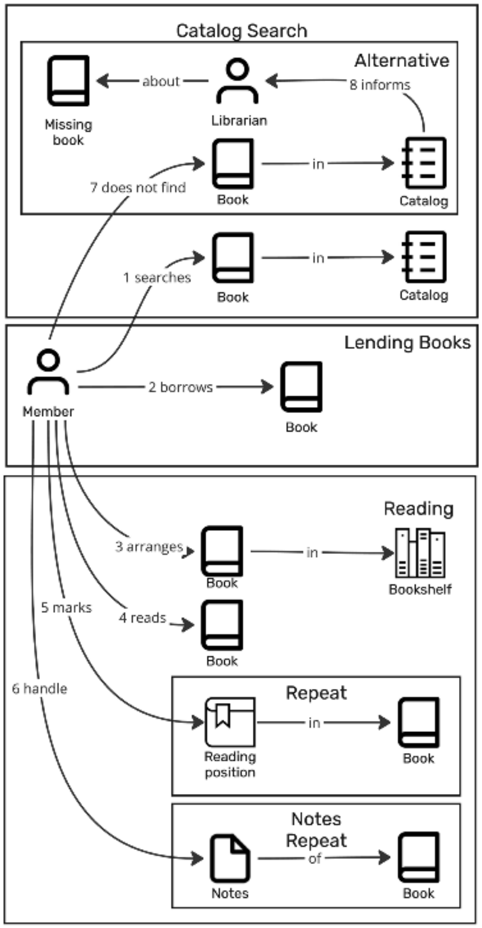


***Figure 5.15**: Domain story reading*

***Source**: Reading.jpg*

Understandably, members frequent a library with the intent to read books, whether for leisure, study, or research. For those engaged in more than casual reading, organizing books on a shelf, bookmarking their place, and taking notes become essential practices to enhance their reading experience. The story progresses as follows:

- **Subdomain catalog management**: A member searches for a book in the catalog.
- **Subdomain lending books**: A member borrows the book.
- **Subdomain reading**: The member arranges books in his bookshelf. The member reads the book. The member marks reading positions in the book.
- **Subdomain notes**: The member handles (creates, edits, deletes) notes of the books.
- **Subdomain catalog management**: A member does not find a book in the catalog. The catalog informs the librarian about the missing book.

The subdomain catalog management appears a second time. It is not uncommon that subdomains reoccur multiple times in a story, as the stories are crafted to focus on user experience and the business processes rather than being strictly bound to subdomains or even technical systems.

The sentences five and six are marked with an additional box marked repeat, indicating that members have the capability to create multiple bookmarks within a single book and can make numerous notes for any given text.

An alternative was found during the discussions. When a book could not be found in the catalog, the librarian is informed about it.

Members of the library not only want to read books, but they also want to discuss books in studying or reading groups.

Discussion

When a library member wishes to engage in discussions about books they have read, they often seek out fellow students as conversational partners. Similarly, even those who read for pleasure may come across a book that sparks their interest in discussing it with others.

While reading or study groups provide a forum for such discussions, they are not formally organized by the library. Instead, these groups naturally emerge through the initiative and collaboration of the members themselves. The domain story for discussion groups is shown in *Figure 5.16*:

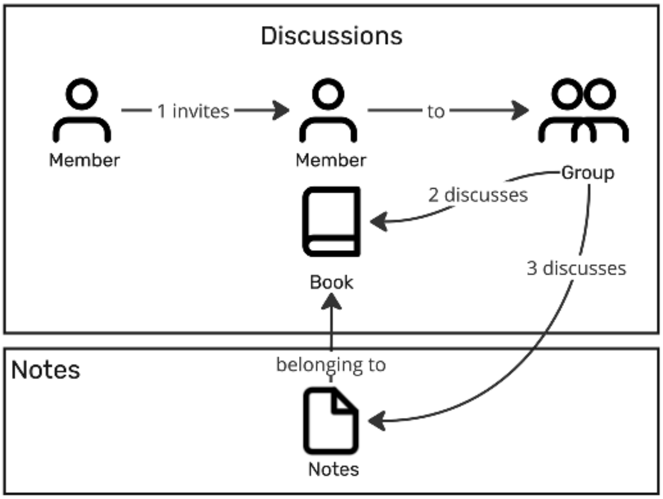


***Figure 5.16**: Subdomains discussions and recommendations*

***Source**: Discussion.jpg*

The domain story goes as follows:

- **Subdomain discussions**: A member invites another member to a group. The group discusses a book. The group discusses notes belonging to the book.

The story intersects again with subdomains.

Members have the capacity to form groups with their peers for the purpose of discussing books. These discussions can extend beyond the books themselves to include an exchange of the notes associated with the books taken by members of the group.

Students usually need guidance on which literature is of help for their studies. Educators can give those recommendations.

Recommendations

Recommendations are usually given by educators. Even fellow students can give recommendations, but it is not the core of the domain story. Hence, it is not visualized. The domain story is shown in *Figure 5.17*:

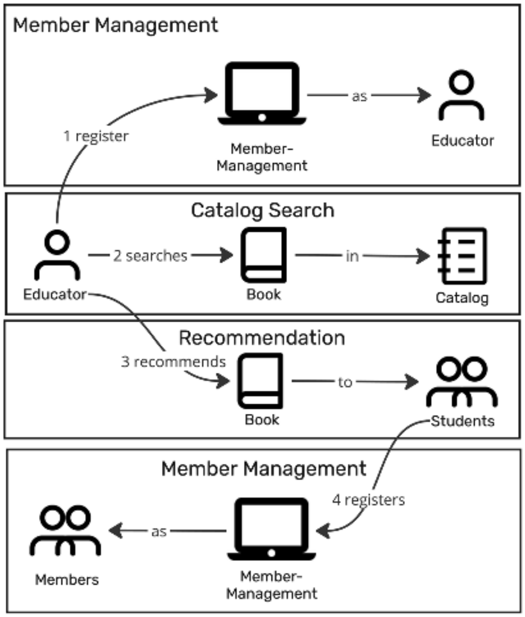


***Figure 5.17**: Domain story recommendations*

***Source**: Recommendation.jpg*

The domain story contains the following sentences:

- **Subdomain member management**: An educator registers as an educator.
- **Subdomain catalog search**: An educator searches for a book in the catalog.
- **Subdomain recommendation**: The educator recommends a book to students.
- **Subdomain member management**: Students register at member management as members.

When an educator signs up as a member, the payment subdomain is not activated, signifying that educators are exempt from paying membership fees. Through their role as influencers, educators can encourage students to join the library by recommending books, thereby fostering a growing community of library members.

With the domain stories told, a ubiquitous language in the according subdomains evolved. It is essential for the success of a project to document that evolved language. To do so, the workshop attendees can create visual glossaries along the domain storytelling.

Visual glossary

A visual glossary serves as a strategic tool for providing clarity on the terminology employed within a specific bounded context. It compiles key terms and charts their interconnections with strong verbs[12]. During the domain storytelling workshops, the moderator can facilitate the capture of emerging terms, which can then be collectively scrutinized by attendees to better understand their interrelations. It is beneficial to undertake this consolidation at the conclusion of each session, as it offers participants an opportunity to review and crystallize the session's work, reinforcing their understanding and contributions. The consolidated result of those discussions is shown in *Figure 5.18*.

The relationships between the terms are labeled with strong verbs like becomes or reads. In such a way, it is more expressive than a classic domain model using class or object diagrams of UML. However, the expressiveness of class or object diagrams is defined even by cardinalities. Those can be taken over in visual glossaries as well, e.g. with an **0..*** between bookshelf and book.

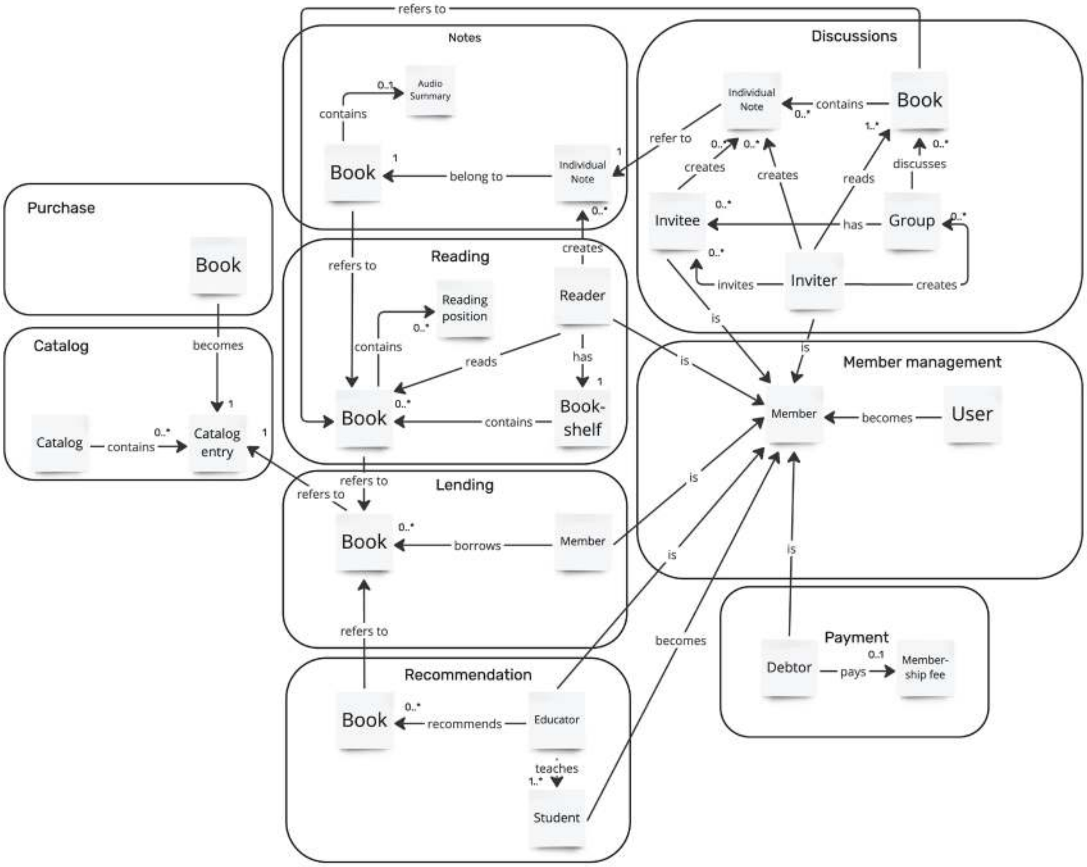


***Figure 5.18**: Visual glossary of the online library*

***Source**: VisualGlossary.jpg*

---

[12] Zörner, S. (2015). Softwarearchitekturen dokumentieren und kommunizieren, Entwürfe, Entscheidungen und Lösungen nachvollziehbar und wirkungsvoll festhalten.

In the subsequent sections, the visual glossary is delineated, with each subdomain detailed individually. Unlike domain stories where subdomains may reoccur to illustrate dynamic interactions, in a visual glossary, each subdomain is represented just once. This distinction arises because a visual glossary is a static model that provides a snapshot of terms and concepts, while a domain story is a dynamic model that captures the flow and interactions of users and systems over time. The subdomains contain the following entities and aggregates:

- **Subdomain purchase**: It contains a book.
- **Subdomain catalog**: The catalog subdomain contains obviously the catalog. The catalog itself contains several catalog entries. A catalog entry is created out of a purchased book.
- **Subdomain notes**: Individual notes can be created by a reader. They belong to one book. A book can contain an audio summary. The book refers to the book read by a reader.
- **Subdomain reading**: The reader can read several books, and they have exactly one individual bookshelf. The bookshelf can contain several books. The book read can contain several reading positions. The book refers to the book borrowed by the reader. A reader is a member of the library.
- **Subdomain lending**: A member can borrow several books. The book borrowed refers to a catalog entry. A member is a member of the library.
- **Subdomain recommendation**: An educator can recommend books to the students they teach. An educator needs to teach at least one student. Students of an educator can become members of the library.
- **Subdomain discussions**: An inviter who is a member of the library can invite invitees who are members of the library as well to a discussion group. The inviter creates the group. The group discusses books, which refer to the books read by the inviter. The books contain individual notes created by the inviters as well as by the invitees. This individual note refers to the notes done for a read book.
- **Subdomain member management**: A user, independent of its role, can become a member of the library.
- **Subdomain payment**: A debtor who is a member of the library pays their membership fee.

The visual glossary is shown in *Figure 5.18*. It looks like a context map, but still the bounded contexts are assumed. They need to be detailed and consolidated using event storming and will be discussed later in *Chapter 6, Event Storming*.

Recommendations for a domain storytelling workshop

Domain storytelling is a great possibility for modeling a domain in workshops. It can structure the domain discussed and give deep insights into a project to be implemented. There are a couple of pitfalls that need to be avoided. Read the following recommendations carefully and apply them in your next workshop:

- **Do a workshop with diverse attendees**: Domain storytelling proves particularly valuable when workshop attendees bring a diverse range of experiences to the table. Typically, it is advantageous for the result when both business experts and IT specialists from the relevant domain participate in the workshops to ensure a comprehensive understanding and effective collaboration. Ideally the number of attendees is no less than five and no more than 15.

- **Do an architect as a moderator**: Software architects are often well-suited to facilitate these workshops due to their strong communication and visualization skills. When appointing a moderator, it is advantageous to choose an individual with prior experience in leading workshops, even if their experience does not specifically pertain to domain storytelling sessions. This background will ensure that they possess the necessary facilitation expertise to guide the workshop effectively.

- **Do not prepare stories**: To achieve optimal outcomes from domain storytelling workshops, it is essential to allow the stories to emerge organically during the sessions themselves. Pre-conceived stories may impede the authentic and in-depth conversations necessary for crafting meaningful domain narratives. It is through these collaborative and explorative discussions that truly valuable domain stories are shaped, reflecting the collective insights and experiences of the participants.

- **Do preparation with the business experts**: For domain storytelling workshops to be successful, it is beneficial if business experts come prepared to share actual experiences and narratives instead of merely presenting a list of requirements. This narrative approach ensures that the focus remains user-centric, capturing the authentic context and challenges encountered by users. Encouraging business experts to prep with concrete stories from their domain fosters a more engaging and insightful dialog, enhancing the overall value of the workshop.

- **Do plan the workshop sessions wisely**: When preparing the workshop, plan half a day for each anticipated subdomain that needs to be discussed. Plan the workshops using Wardley maps for priorities.

- **Do use an object in each sentence**: It is recommended to use a new icon for an object in each sentence. Reusing the objects might lead to overlapping and not understandable arrows. Moreover, underlying meanings like status changes of the objects are hidden in the reused icons.

- **Do not separate modeler and moderator**: While it may seem practical to assign the roles of moderator and modeler to two separate individuals, the author's experience suggests that a more effective approach is to unify these roles in a single person. However, for beginners, juggling both responsibilities can be challenging. In such cases, it is preferable to pair a novice moderator with a seasoned modeler. This allows the beginner to concentrate on guiding the discussion while the experienced modeler handles the intricacies of modeling. As the moderator gains confidence and expertise, they often express a desire to undertake modeling tasks themselves, gradually transitioning to manage both roles concurrently.

- **Do not generalize too early**: People in the software industry, among others, frequently tend to oversimplify tasks by assuming similarities between different functionalities or features—*Thing A is nearly identical to thing B*. This mindset can lead to critical misunderstandings and can escalate the complexity and effort required for implementation.

  To prevent misunderstandings and preserve the integrity of each expert's insights, it is essential to let business experts express their perspectives using their own vocabulary. Motivate each expert to articulate their experiences using terms they are comfortable with. After collecting the stories, thoughtfully integrate the narratives and compile the used terms into a visual glossary. This glossary serves as a foundation for harmonizing the vocabulary throughout the stories, leading to a clear and consistent language that everyone can understand. It is important to avoid premature consolidation of terms; instead, wait until the visual glossary is fully developed. This approach ensures that the unique details and nuances of every expert’s input are maintained and the ubiquitous language is evolved.

With a good workshop result, as a domain story and the visual glossary, the core domains can be detailed using event storming.

Conclusion

Domain storytelling is a powerful approach for crafting narratives within a domain and visually documenting business requirements. This methodology encourages the establishment of a ubiquitous language, bridging the gap between business experts and IT specialists toward a common understanding.

Employing straightforward and accessible grammar, domain stories effectively steer complex discussions, ensuring all participants remain aligned. While the grammar and associated iconography are intentionally simple to facilitate clear communication, there are instances where prioritizing comprehension over strict adherence to predefined rules may yield better collaborative outcomes. Flexibility in this method allows for nuanced interpretation and alignment with the unique context of each discussion.

Integrating domain stories with visual glossaries empowers stakeholders to define bounded contexts and subdomains with greater precision. This combination lays a strong foundation for subsequent modeling activities and implementation processes.

Moving forward, event storming is an integral part of the modeling journey, a topic that will be elaborated upon in the following chapter.

Points to remember

- Domain storytelling helps in:
    - understanding the domain
    - finding a common language between the business experts and the IT specialists
    - overcoming misunderstandings
    - clarifying software requirements
    - implementing better software
    - structuring and architecting the software
    - designing digital business processes
- Domain stories are visualized narratives of business experts.
- Domain stories contain several sentences containing actors, sequenced actions, and objects. They can be enhanced by one or more adverbials.
- The chosen tool for the visualization needs to serve the understanding. The moderator needs to be proficient with it.
- A visual glossary serves as strategic tool to clarify the terminology and to create a ubiquitous language.
- A visual glossary is more expressive than other static models by the usage of strong verbs.
- Domain storytelling workshops should be setup with experts of different expertise in business and IT.
- Architects should moderator and model in the workshops.
- Domain stories cannot be prepared. A workshop should always start with a blank canvas.
- Early generalizing jeopardizes the understanding and leads to more effort in implementation. Visual glossaries can be used to avoid a too early generalization.

Multiple choice questions

1. **For which problems in software development is domain storytelling an excellent tool?**
    a. Understanding the domain
    b. Implementation of specific math problems
    c. Make the operation of complex software in the cloud better
    d. Helping the support for faster answers
2. **What is a domain story?**
    a. A story of a business expert written in English.
    b. A story of an IT specialists written in Python.
    c. A visualized story of business experts.
    d. A narrative of UX experts visualized a specialized tool.
3. **How many sentences does a domain story contain?**
    a. A domain story does not contain any sentences only bullet points.
    b. A domain story contains only one sentence.
    c. A domain story contains one or three in maximum sentences.
    d. A domain story can contain several sentences.
4. **Which tools can be selected for visualizing?**
    a. Only high-end drawing tools like Adobe Illustrator can be used for visualization.
    b. Tools can be quite any virtual or physical drawing tool. It needs to support the needs of the workshop attendees and the moderator.
    c. Drawing tools are not necessary for a domain storytelling. It can be written with a standard text editor.
    d. Domain stories can only be drawn with pencils on paper.
5. **What is the best tool to create a ubiquitous language between business experts and IT specialists?**
    a. A complex UML class diagram.
    b. An easy-to -understand description on the web page.
    c. A visual glossary created in domain storytelling workshop.
    d. A long table across domains with all terms defined.

6. **How should domain storytelling workshops be prepared?**
    a. Domain stories cannot be prepared, they should start with a blank canvas.
    b. Domain storytelling workshops should be prepared with full visualized stories.
    c. Domain storytelling workshops should be prepared by a created visual glossary.
    d. Domain storytelling workshops should be prepared with a clean room and flowers.

Answers

1. **a.**
2. **c.**
3. **d.**
4. **b.**
5. **c.**
6. **a.**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 6

Event Storming

Introduction

Event storming is another workshop format of **Domain-Driven Design** (**DDD**). It concentrates on the domain events along a business process. These domain events are enhanced by commands triggering the events, aggregates and read-only data, and roles and processes performing the triggering command. It is used to define bounded contexts along the discussed business processes.

Structure

This chapter covers the following topics:

- Target of an event storming workshop
- Setup of the workshop
- Event storming in detail
- Collection of events
- Consolidation of events
- Finding the commands
- Roles and processes
- Finding aggregates and views

- Finding bounded contexts
- Event storming of the online library
- Bounded context canvas
- Guidelines in event storming workshops

Objectives

This chapter covers the DDD workshop format event storming. You will learn how to setup an event storming workshop, how to find events and aggregates. The chapter additionally covers how to define and document bounded context.

> **You learn to know a pilot in a storm.**
>
> *~Lucius Annaeus Seneca*

Target of an event storming workshop

Event storming is a workshop format invented by *Alberto Brandolini*[1]. It combines the idea of brainstorming and process modelling.

Brainstorming is a creativity technique. The term was published by *Osborn* in 1953[2]. *Osborn* said that there are two principles which contribute to *ideational efficacy*, which are mentioned as follows:

- Defer judgement
- Reach for quality

To avoid judgement, each idea on a workshop needs to be heard. To reach quality each idea needs to be elaborated and together either to be accepted or to be rejected.

Brainstorming

Brainstorming encourages workshop attendees to leave standard thinking behind and to think in a fresh and unusual manner. There are four basic rules which help to create successful workshops[3]:

- Focus on quantity, not quality (quantity will lead to quality later)

---

[1] Brandolini, A. (2023). EventStorming. Retrieved 04 2024, from EventStorming: **https://www.eventstorming.com/**

[2] Osborn, A. F. (1953). Applied imagination: principles and procedures of creative thinking. New York: Charles Scribner's Sons.

[3] Miller, B. C. (2012). Quick brainstorming activities for busy managers: 50 exercises to spark your

- Withhold evaluation in a first step
- Encourage wild outlandish ideas
- Combine or build ideas from others

Brainstorming is an essential tool for generating new ideas and requirements, finding innovative solutions to complex problems, and supporting conceptual design by creating metaphors or reimagining traditional methods. It can also be used to explore uncharted design territories and foster social cohesion within teams[4]. This last point is particularly important in software architecture, where there is often a gap between the technical architecture and business objectives. By utilizing brainstorming like workshops, teams can bridge this divide and work together more effectively to achieve their goals.

With the help of event storming as a kind of brainstorming, it is possible to create a common understanding and deep insights into the domain to be evaluated.

Event storming

The target of an event storming workshop is to get a deep insight into the domain discussed and to know the corresponding bounded contexts and the necessary aggregates.

It models the business processes using domain events. Domain events are events which happen along the business process. The event storming process follows the following steps:

1. Collect events
2. Consolidate events
3. Find commands
4. Find roles and processes
5. Find aggregates and views
6. Find bounded contexts

These steps will be evaluated in detail in the following sections, but before that we will discuss the setup of the workshop.

Setup of the workshop

An event storming workshop can be setup almost in the same manner as the previously discussed domain storytelling workshops. Ideally, an event storming workshop follows a domain storytelling workshop and is done by the same attendees. Even the moderator should not change. It is most advisable that a seasoned architect moderates the sessions.

---

[4] Wilson, C. (2013). Brainstorming and Beyond: A User-Centered Design Method. Oxford: Morgan Kaufmann.

Scheduling

For simple subdomains, like the previously discussed task management, you can plan one day. For more complex subdomains, like the reading subdomain of the online library, you need at least two days. It is advisable to schedule two days for the workshops for more complex discussions. The sessions can be done in half days as well—which means four sessions for complex discussions. The pause between the session should not be longer than one week.

Subdomains can be discussed with larger pause between the sessions, but is should not more than four weeks.

When the event storming workshops are done in combination with domain storytelling sessions, the workshops should be done as closely as possible. The pause between the domain storytelling workshop and the event storming workshop should be no longer than one week.

The activity can be conducted both remotely and in person. However, similar to most workshops, in-person sessions are typically more efficient than remote ones. The tooling is comparable to those for domain storytelling.

Tooling

To guide remote workshops, digital whiteboarding tools, as described in *Chapter 5, Domain Storytelling*, are necessary. Options include *Miro*[5] (as utilized in this book), *Conceptboard*[6], and *Mural*[7]. Additionally, there are tools specifically designed for brainstorming, such as *FigJam*[8], as well as specialized options for event storming, like *Prooph Board*[9] or *Modellution*[10]. The selection of the tool should be guided by two key considerations: first, both the moderator and attendees should find the tool user-friendly and comfortable to work with; second, the tool must comply with the software usage policies of the company hosting the workshop.

When conducting workshops in person, it is essential to have the standard equipment on hand to facilitate an effective session. This includes an adequate supply of markers and sticky notes, as well as boards or flip charts for displaying and organizing ideas.

Regardless of whether the workshop is conducted in person or remotely, it is crucial to ensure that all necessary materials are readily accessible to attendees. For workshops using a digital whiteboard, the board should be set up with pre-prepared sticky notes, eliminating the need for participants to search for their own. In an in-person setting using

---

[5] **https://www.miro.com**
[6] **https://conceptboard.com/**
[7] **https://www.mural.co/**
[8] **https://www.figma.com/figjam/**
[9] **https://prooph-board.com/**
[10] **https://www.modellution.com/**

physical notes, markers and sticky notes should be conveniently placed and available for immediate use by participants. By doing so, the workshop can run smoothly, allowing attendees to fully engage in the collaborative process without any disruptions.

In the following the structure of an event storming workshop is discussed, starting with the collection of events.

Event storming in detail

As already pointed out, event storming is a workshop format, which follows certain rules. The process of event storming workshop is shown in *Figure 6.1*:

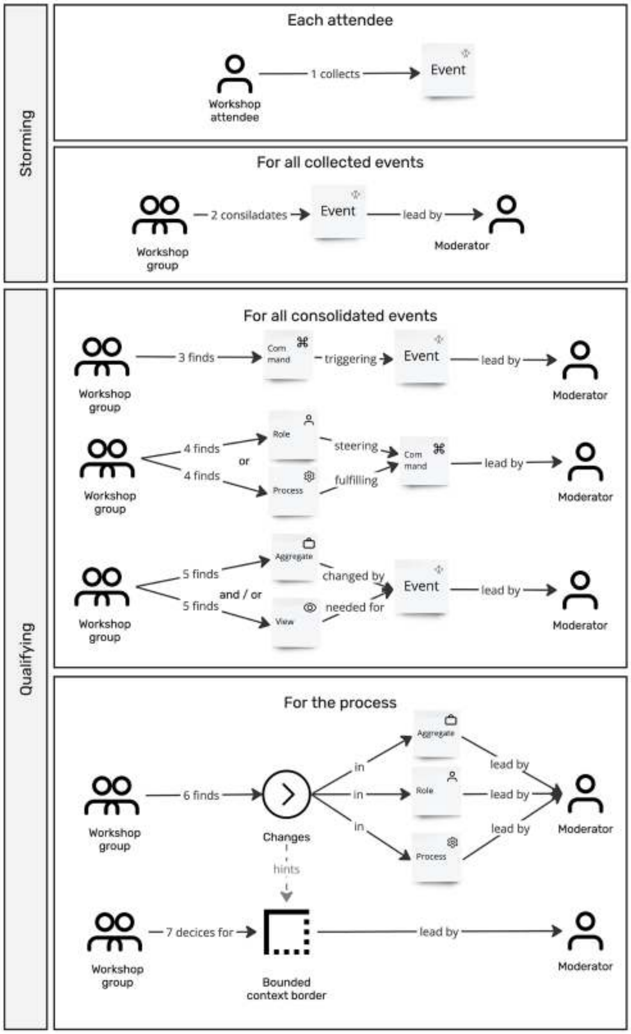


Figure 6.1: *Event storming process*

Source: *ProcessEventStorming.jpg*

The steps of the process will be discussed as follows using the simple task management example introduced already in *Chapter 5, Domain Storytelling*:

1. All attendees collect the domain events along the business process to be discussed
2. The workshop group consolidates the events moderated
3. The workshop group finds the commands which trigger the events lead by the moderator
4. The workshop group finds the roles or processes which steer or fulfill the command again lead by the moderator
5. The workshop group finds the views and/or aggregates which are necessary for fulfilling the command or which are changed by the event again moderated
6. The workshop attendees try to find changes in aggregates or processes respectively roles moderated
7. The workshop attendees decide together if a change is a border of a bounded context or not lead by the moderator

The preceding seven points will be discussed in extensive detail in the next sections, starting with the collection of events.

Collection of events

As the first step in an event storming workshop, the domain events along the business process are collected. Events are always something that happen already, and which happen to a certain object. Therefore, events are named with an object name and with verbs in past tense, e.g., *user declined*. The verbs should be as expressive as possible. A verb like *approved* is more expressive than *created*, and *declined* is more expressive than *deleted*. Obviously, the verbs *created*, *updated*, and *deleted* are helpful and necessary, but the workshop group should always discuss, if they can find a more expressive verb.

The order of events in time are shown from left to right. Events which can be done in parallel, or which are exclusive, are ordered vertically. Sometimes, certain loops need to be shown to indicate them arrows can be used.

The workshop attendees collect all domain events along the business process to be discussed. The result of such a collection is shown in *Figure 6.2*:

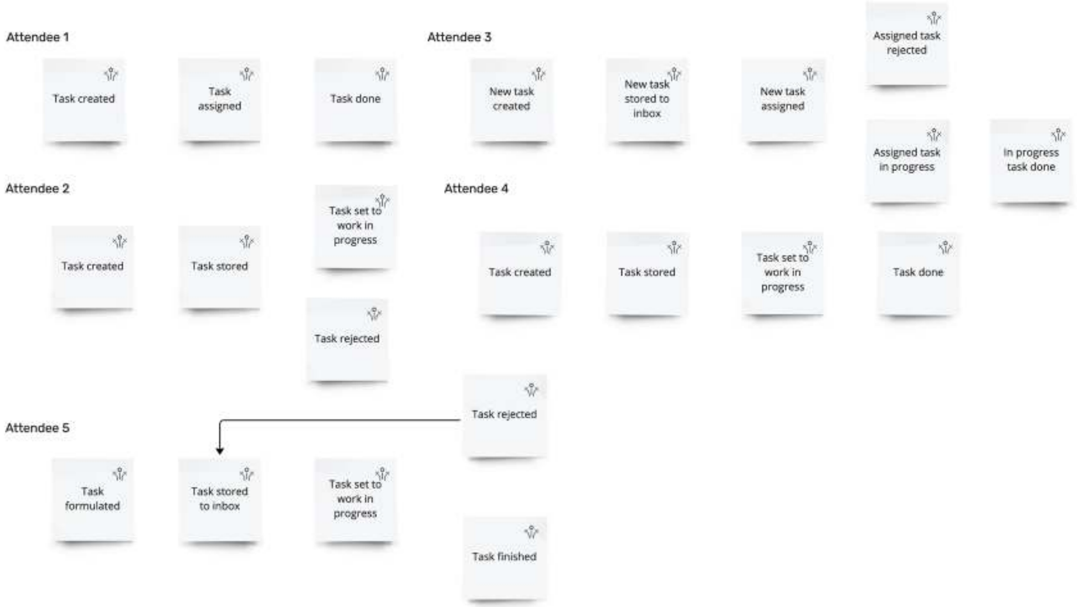


***Figure 6.2**: Collection of events*

***Source**: EventsCollected.jpg*

The figure shows that each workshop attendee finds different events and even gives the found event different names. That is needed in the storming phase of the workshop to steer different ideas without judging them.

Usually, events are shown by orange sticky notes, which are marked with a fireworks icon in the upper right corner.

The events collected by the attendees are now consolidated by the attendees lead by the moderator.

Consolidation of events

The moderator takes the events of an attendee and let them discuss by the workshop group. Then the moderator takes the next group of events, and the workshop group compares them with the already discussed events. They discuss if the event taken is already in the line of events already discussed or if it is a new event. If it is a new event, it needs to be sorted in the correct order of events. If it is an already existing event with a different naming, the group needs to decide which naming is better or event to find a third naming which describes the event best.

In the first step the moderator takes the events from attendee one, as shown in *Figure 6.3*:

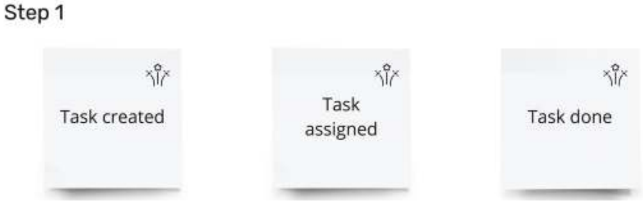


Figure 6.3*: Consolidation of events step 1*

Source*: EventConsolidationStep1.jpg]*

Afterwards, they sort the events from attendee two after the ones from attendee one. As expected, there are commonalties and differences. **Task created** is common. **Task assigned** and **Task stored** are different. The workshop attendees decide that they are all different and decide about the following sequence: **Task created**, **Task stored**, and **Task assigned**. Even the remaining ones are different, and the workshop attendees decide that all can remain. The events, **Task rejected**, and **Task set to work in progress**, are done in parallel. **Task done** is on the same level as **Task set to work in progress**, because a rejected task cannot be done. The result of this discussion is shown in *Figure 6 4*:

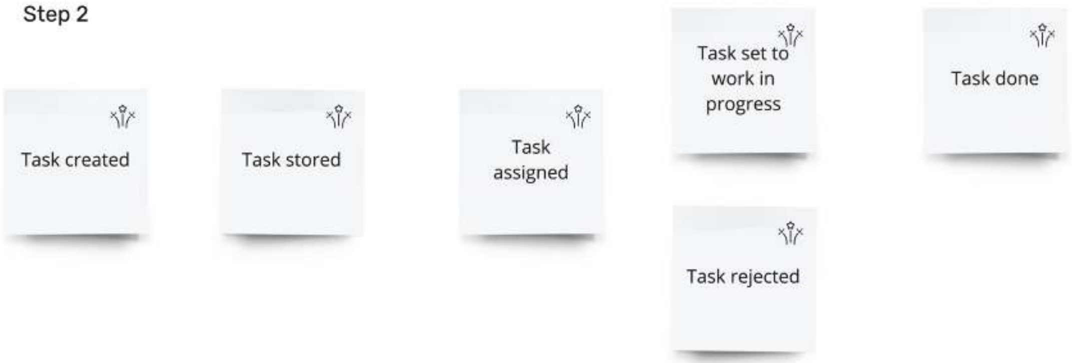


Figure 6 4*: Consulting of events step 2*

Source*: EventConsolidationStep2.jpg*

The discussion goes further with the events of attendee three. Attendee three has almost the same events as already found. However, they take the names of the task status as event names. The workshop group decides against those names and stick to the former names. As an exception, they renamed **Task set to work in progress** to **Task set to in progress**, because it is better to understand.

The events of attendee four are already in the stream and the workshop group decides to leave the found events as they are.

The events of attendee five are almost in the stream. Except the event **Task done** is named as **Task finished**. The workshop attendees decide to stick to **Task done**. A second difference is an arrow between **Task rejected** and **Task stored to inbox,** indicating that rejected tasks are given back to the inbox. The workshop attendees decide to stick still to **Task stored**, but to take over the arrow.

That is the last step of the consolidation, and the result is shown in *Figure 6.5*:

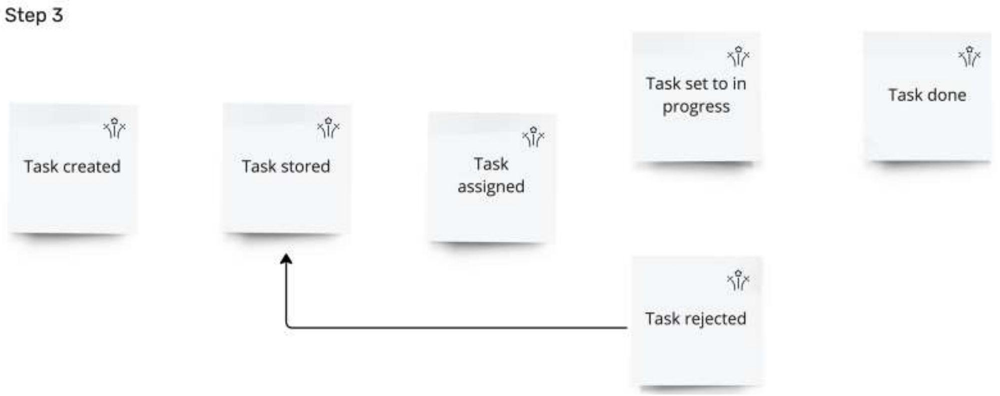


***Figure 6.5**: Result of the event consolidation step*

***Source**: EventConsolidationStep3.jpg*

In the next step, the workshop attendees need to find the commands which trigger the events.

Finding the commands

In the next step, the workshop attendees need to find the commands for each event. The moderator shows the event and asks for the appropriate command triggering the event.

A command is a verb in imperative usually accompanied by an object. A command is called just before the event happens. Therefore, it is located left to the belonging event. The result is shown *Figure 6.6* with the resulting command: event | command | event series:

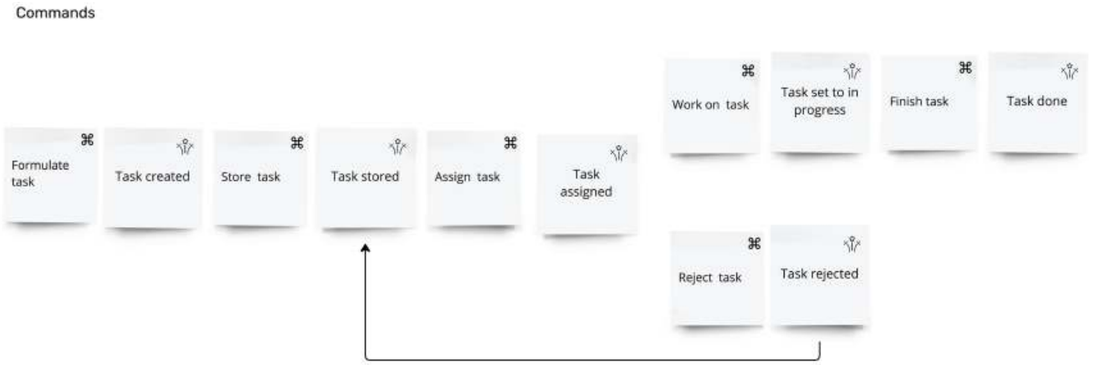


***Figure 6.6**: Results of the command discussion*

***Source**: CommandsLarge.jpg*

Usually, commands are shown in blue sticky notes. Here, they are marked with the command icon ⌘.

As one can see, the verbs used are expressive, but usually, they are equal to the verbs in the event or at least synonyms to them. They do not give more information than the events themselves. Therefore, the author prefers the commands in the left bottom corner of the event with a smaller sticky note, as shown in *Figure 6.7*:

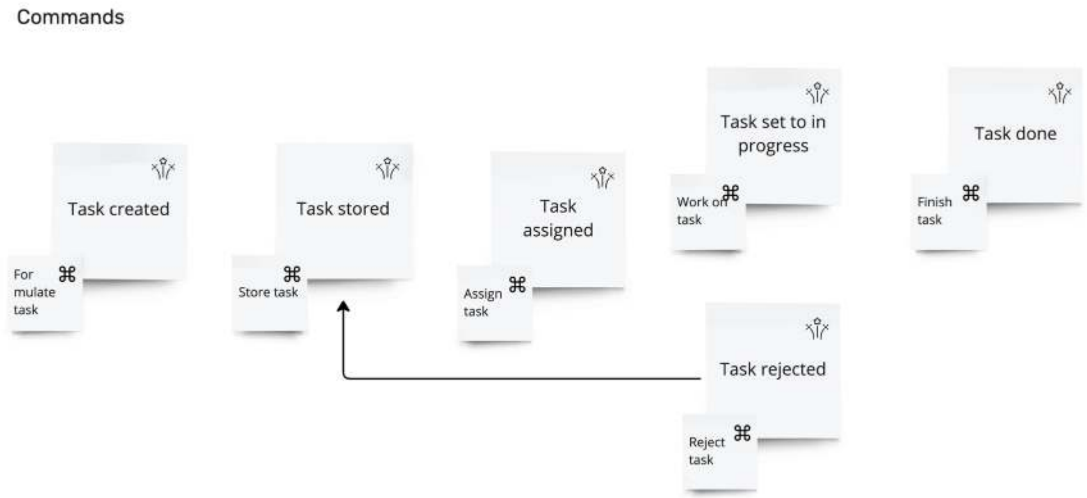


***Figure 6.7**: Events and commands*

***Source**: Commands.jpg*

In the next step the workshop attendees need to find the roles steering the commands or the automatic processes fulfilling the commands.

Roles and processes

Again, the moderator goes over the events and asks for the role of steering the event or the automatic process of fulfilling the event. When a domain storytelling was done before the event storming, the roles and processes were already pointed out as actors in the domain storytelling. The result of this discussion is shown in *Figure 6.8*:

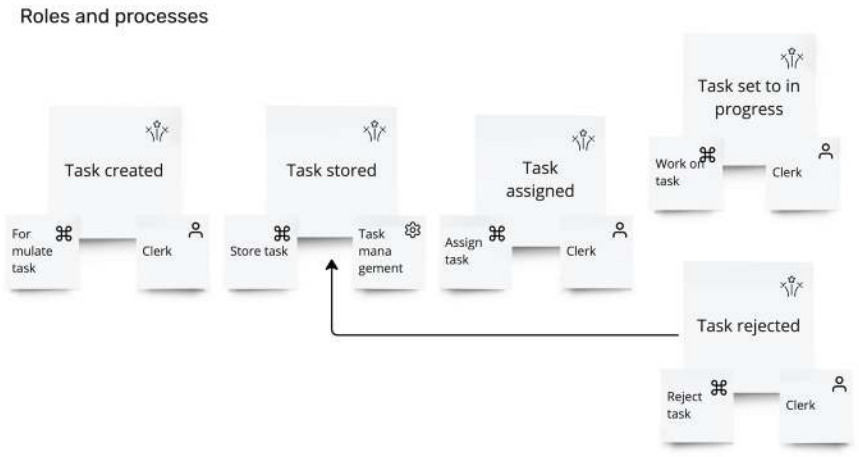


***Figure 6.8**: Events enhanced by roles and processes*

***Source**: RolesAndProcesses.jpg*

The human roles or the automatic processes are ordered in the bottom right corner of the event. The roles are marked with a tiny human sign, whereas the automatic processes get the steering wheel icon.

Usually, roles are marked with dark yellow sticky notes, and the automatic processes with purple ones.

This step can seamlessly be integrated with the prior one, which involves identifying commands, can streamline the process and add depth to the understanding. The moderator can probe further by inquiring, *Who initiates this command?* This prompts the addition of the appropriate role or process to the event.

When the attendees come up with two roles or two processes or even a process and a role for one event, the event should be split. It is a sign that events happen in parallel.

The subsequent step involves thoroughly identifying the objects essential for the corresponding event.

Finding aggregates and views

Again, the moderator asks for each event, the object that is changed—which indicates the aggregate, and which additional data are needed—which indicates a view. The result of this discussion is shown in *Figure 6.9*:

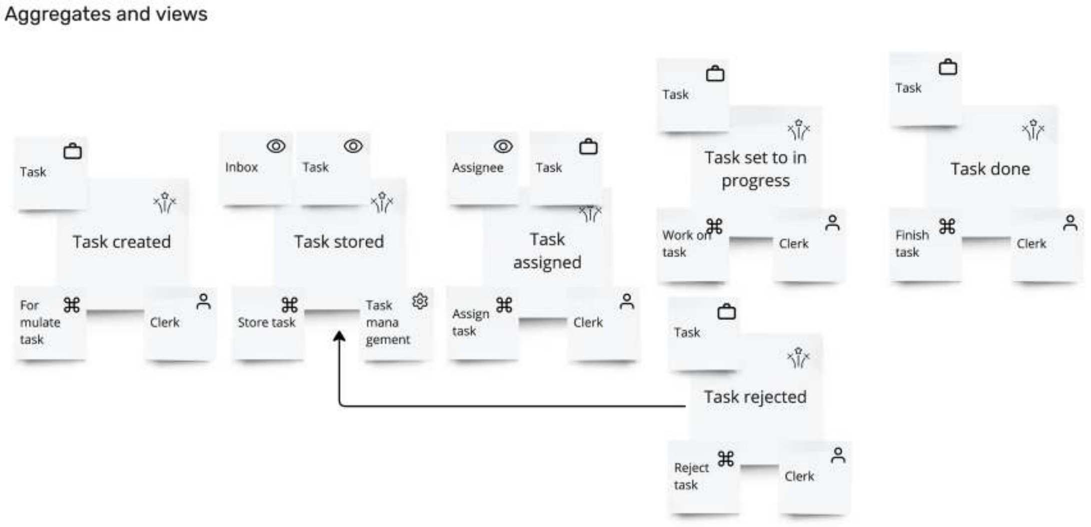


***Figure 6.9**: Events enhanced with views and aggregates*

***Source**: AggregatesAndViews.jpg*

Due to this simple example, the aggregate is almost at every event **Task**. At the event **Task stored**, the task serves only as view because the task itself is not changed during storing. At that event, even a second *read mode*l or *view* is necessary the **Inbox**. For the event **Task assigned**, the relevant assignee serves as view.

The aggregate is usually shown with a dark yellow sticky note. Here, it is marked with a suitcase icon. The views, usually light green, are marked with the eye sign.

In the next step of the workshop, the bounded contexts need to be found. The first step is to find hints for a border of a bounded context. With those hints the workshop attendees can decide if it is a border or not.

Finding bounded contexts

The event stream for Task Management shows only a few changes in aggregates or roles. The aggregates change from **Task created** to **Task stored** and from **Task stored** to **Task assigned**. However, the aggregate only changes because of the change of **Task** from an aggregate to a view and vice versa.

The roles change from **Clerk** to **Task management** and back for the same events. However, the process **Task management** points out to the corresponding bounded context. The workshop attendees decide that no bounded context border can be found here. So, all found events belong to the same bounded context **Task management**.

The bounded contexts are shown with large circles around the events belonging to them. They are named with an expressive name. Additionally, the changes are indicated using arrows. These arrows do not need to appear in a workshop result. They are only used in this book to visualize the discussion. The result of this discussion is shown in *Figure 6.10:*

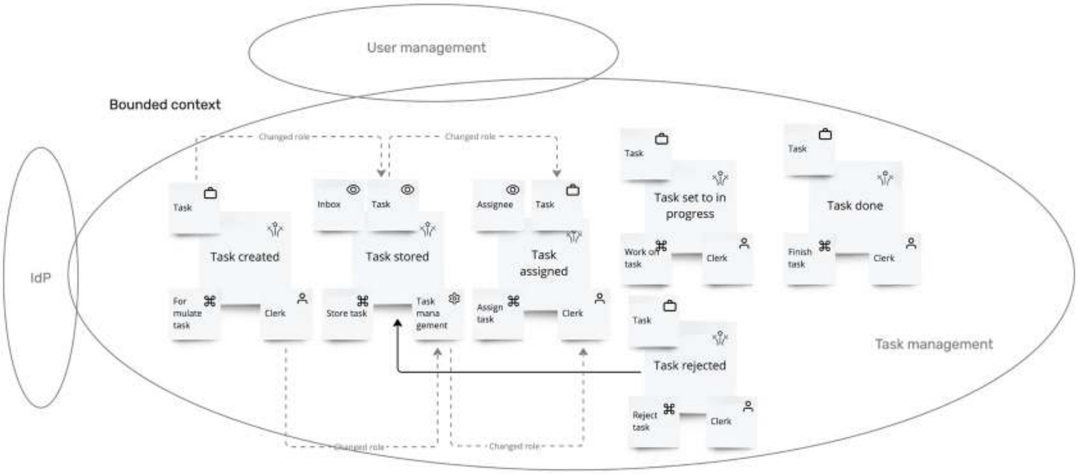


Figure 6.10*: Bounded contexts of task management*

Source*: BoundedContextTaskManagement.jpg*

During the discussion, two more bounded contexts appear, even though no events were detected for them. First, the workshop attendees detect that the users need to log in. That means an **Identity Provider** (**IdP**) is necessary. Additionally, assignees are required. Possible entries are delivered as views to the event **Task assigned** by the bounded context **User management**. The attendees name bounded contexts. Names with the aggregate and an expressive verb in a noun form are recommended, e.g., **Task management**.

With the bounded context detected, the context map can be created. The context map will be discussed using the example of the online library to show the advantages of using a more complex example than the shown task management.

Event storming of the online library

In the following sections, we will be introduced to the results of the online library capabilities. To do so, the consolidated events, the commands, the roles respective to the processes, and the views and aggregates are discussed for the corresponding bounded contexts detected during the domain storytelling workshops.

Purchase

The purchase capability allows librarians to purchase books, as already discussed in *Chapter 5, Domain Storytelling*. The books are entered into the catalog by the librarians and an artificial intelligence creates audio summaries for each book.

The catalog update can be done in parallel to the creation of the audio summary, as shown in the consolidated events in *Figure 6.11*:

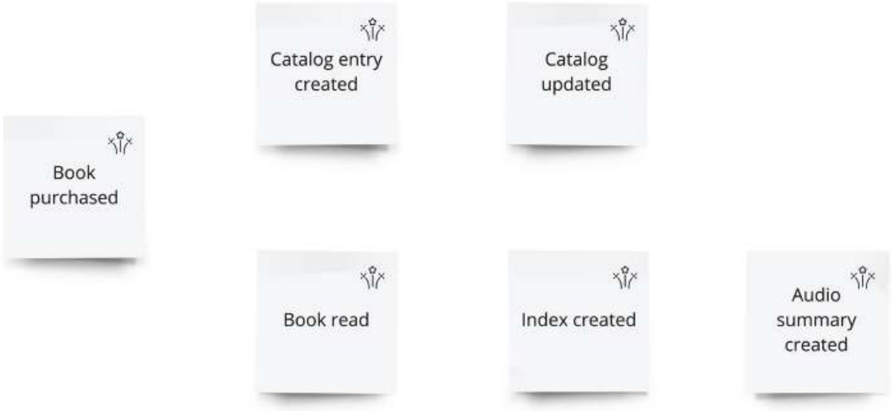


Figure 6.11*: Events consolidated for purchasing*

Source*: PurchaseEvents.jpg*

The enhancement of the events by commands and roles respective processes is shown in *Figure 6.12*. The found events are **Book purchased**, **Catalog entry created**, **Catalog updated**, **Book read**, **Index created**, and **Audio summary created**.

In the following step of the event storming *Enhance events*, the events are enhanced by commands, roles respective processes, views, and aggregates. The corresponding result is shown in *Figure 6.12*:

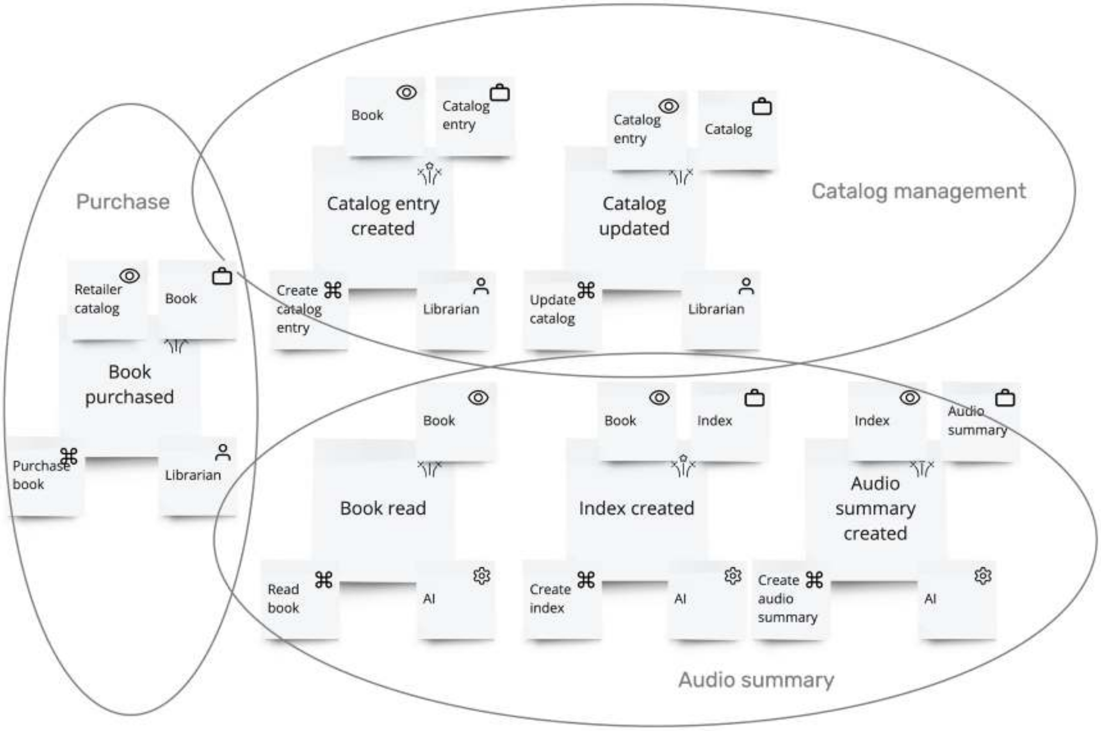


Figure 6.12: *Events and bounded contexts of purchasing*

Source: *PurchaseboundedContext.jpg*

There is a change of aggregate from **Book purchased** with the aggregate **Book** to **Catalog entry created** with **Catalog entry**, and an additional change to **Book read**. Therefore, three bounded contexts are detected: **Purchase**, **Catalog management**, and **Audio summary**. They correspond to the founded bounded context of domain storytelling.

The next capability that is discussed is Member management.

Member management

The member management contains the functionality to register as a library member. The found events are shown in *Figure 6.13*:

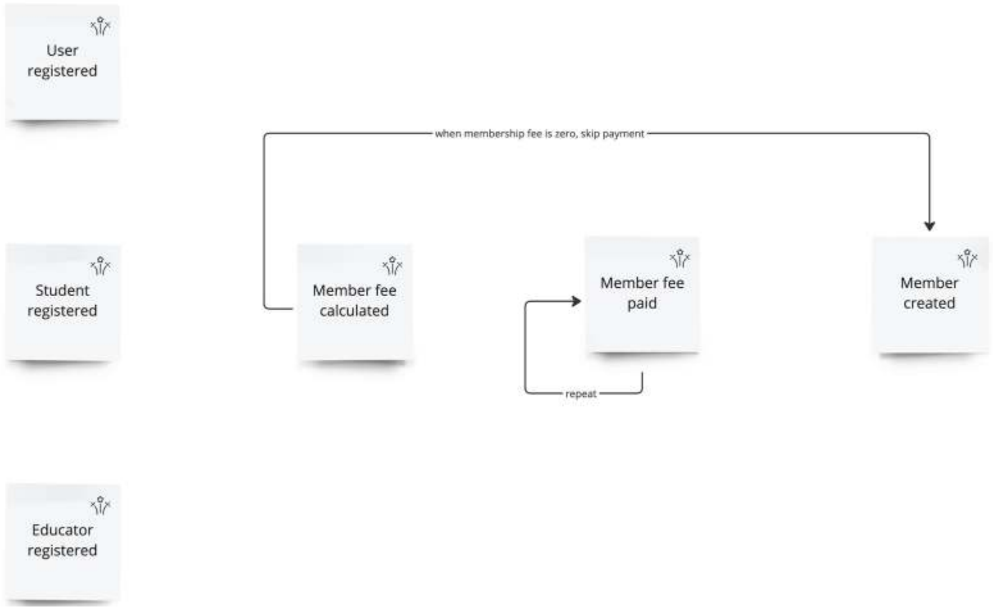


***Figure 6.13**: Events found for member management*

***Source**: MemberManagementEvents.jpg*

The events for the member registration contain different registrations for users, students, and educators.

The member fee calculation is a new aspect that needs to be discussed in more detail. The membership fee is paid. The **repeat** arrow indicates that the membership is settled based on a payment plan.

When the initial payment is done, a new member can be created. The payment step might be skipped if no membership fee needs to be paid as it is planned for educators.

The events can be enhanced, and the bounded contexts can be defined. The result is shown in *Figure 6.14*:

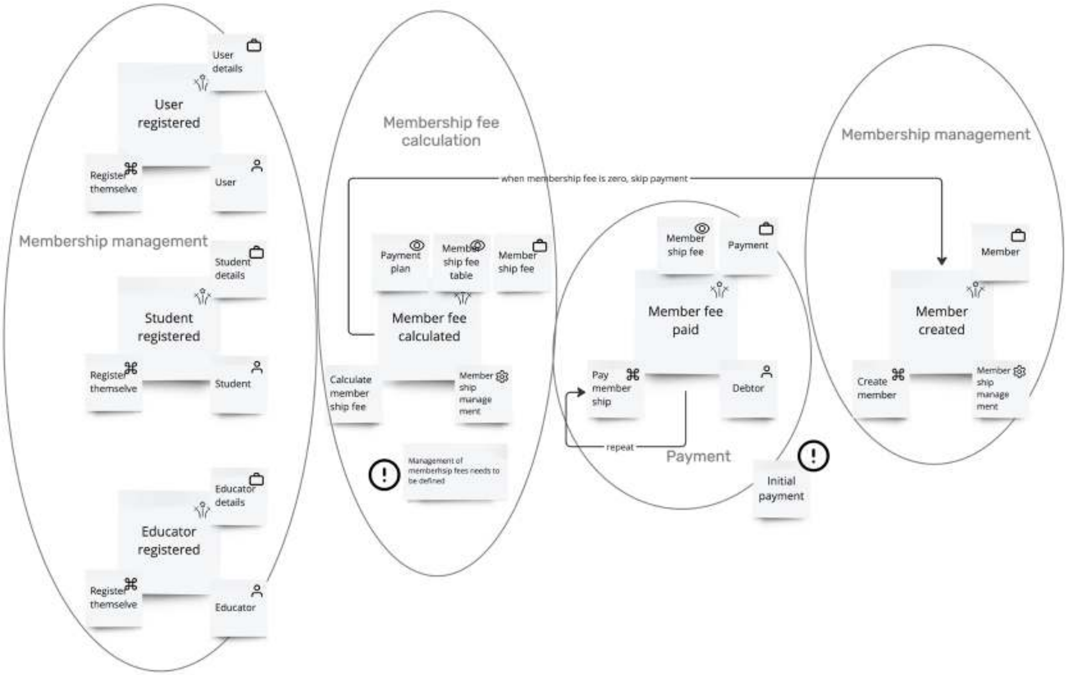


Figure 6.14: *Result of the event storming workshop Member management*

Source: *MemberManagementBoundedContext.jpg*

The registration events cover various aggregates, including **User details**, **Student details**, and **Educator details**. This distinction is made during the event storming workshop, reflecting the different handling of membership fees for students and educators or unspecific users. To accurately classify their roles, educators and students are required to provide appropriate evidence.

The membership fee structure should vary depending on the role of the member (e.g., student, educator) and the chosen payment schedule (e.g., monthly, annually). Additionally, we need to consider if the initial payment should differ from subsequent scheduled payments. Implementing a dynamic membership fee calculation system is an added capability and requires further discussion. The relevant questions are highlighted with exclamation marks (!) for emphasis. In an event storming workshop, these notes can be marked in red for better visibility.

The bounded contexts are found between the **Member fee calculated** and the **Member fee paid**, and between **Membership fee paid** and **Member created**. The bounded context **Membership management** appears twice. Comparable to domain storytelling, event storming concentrates on the business process rather than the business objects. Therefore, bounded context as model boundaries can appear multiple times along the process.

In the next step, the Membership calculation is discussed.

Membership calculation

The necessity for an additional capability, membership calculation**,** is detected during the event storming of member management. It is specific to the library and belongs to the core domains. The result of the discussion is shown in *Figure 6.15*:

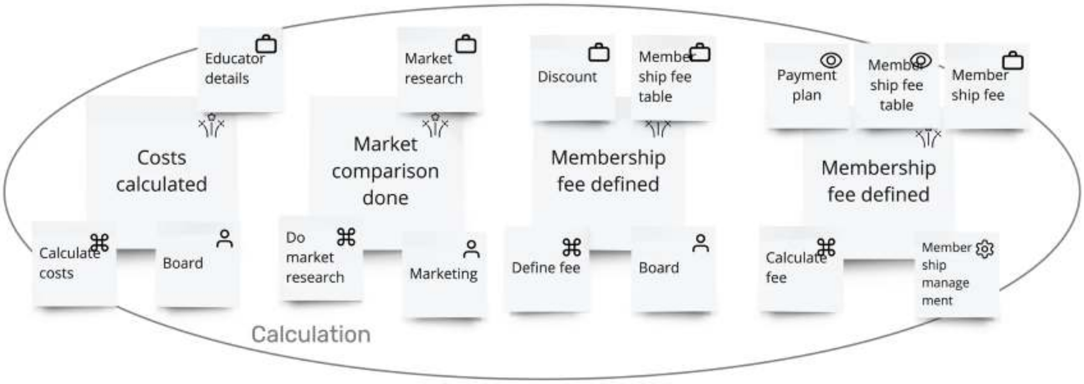


Figure 6.15*: Event storming result of membership fee calculation*

Source*: MembershipFeeCalculationBoundedContext.jpg*

The costs of the library are calculated by the board, and market research is done accordingly. Based on the calculation and the given market including the competition situation, the membership fee can be defined by the board. The resulting table can be used to calculate the membership fee based on the role and the payment plan selected for the member to be.

The membership calculation was not discussed during the domain storytelling workshops; therefore, the visual glossary needs to be enhanced by the calculation part.

Visual glossary enhancement

The result of the enhancement of the visual glossary by the membership fee calculation is shown in *Figure 6.16*:

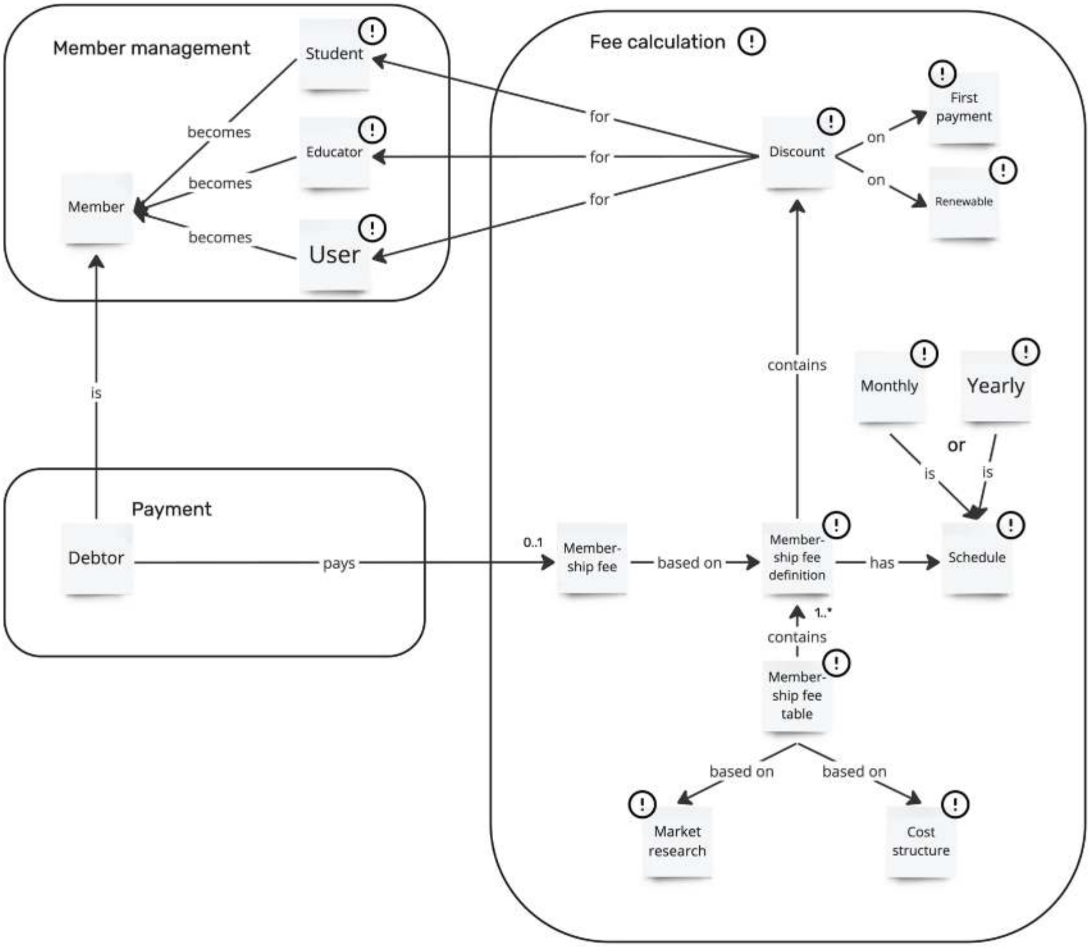


Figure 6.16*: Visual glossary of membership management*

Source*: MembershipFeeCalculationVisualGlossary.jpg*

The changes in comparison to the visual glossary based on the domain storytelling workshops are marked with an exclamation mark. The corresponding changes are listed in the following points:

- **Member management**: Member management differentiates between users, educators, and students when they become members.
- **Fee calculation (new capability)**: The fee calculation contains the membership fee to be paid based on a membership fee definition in a corresponding table. The definition of a fee amount depends on its schedule, e.g., monthly or annually, and the calculation of an appropriate discount. The discount can be given on the initial amount as well as on the scheduled payment. The discount is given based on the role of the member, e.g., educator, and is up to 100%.

The reading capability and the belonging bounded contexts will be discussed in the next section.

Reading

The reading part of the discussion contains not only the reading itself, but it also contains the search for the book in the catalog and the arranging of books in the bookshelf. The first part of the events in reading process is shown in *Figure 6.17*:

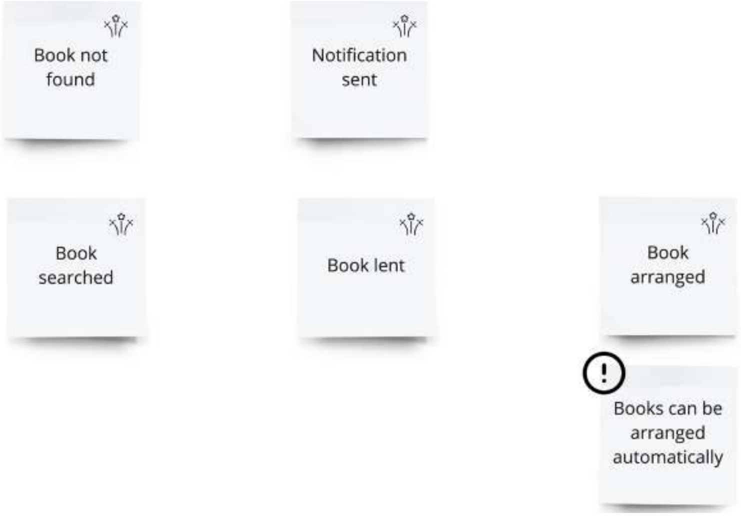


***Figure 6.17**: Events of the first part of the reading process*

***Source**: Reading1Events.jpg*

First, a book is searched for. When the book is not found, a corresponding notification needs to be sent. When the book is found, it can be lent, and the books can be arranged in the bookshelf. In addition to the arrangement of books by the members, as discussed in the domain story, the books can be arranged automatically at convenience for the members.

The enhanced events including the defined bounded contexts are shown in *Figure 6.18*:

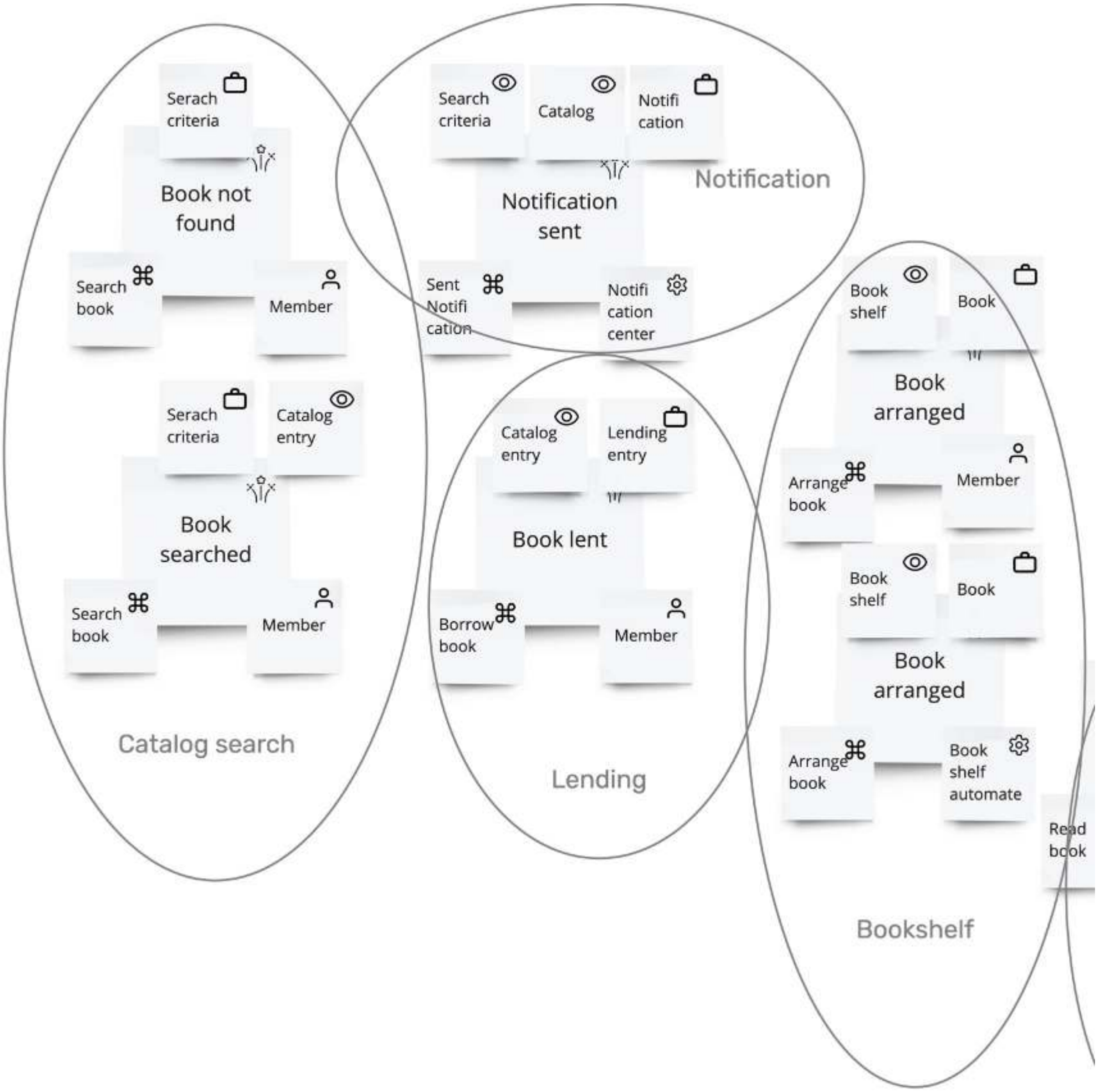


***Figure 6.18**: Result of the event storming for the reading part inclusive bounded contexts*

***Source**: Reading1BoundedContext.jpg*

The member of the library can search for a book using their search criteria, which serve as aggregate in the search. When the book is found according to the catalog entry, it can be used further. When the book is not found, the search criteria are used to create a notification sent to the librarian.

Once the book is found in the catalog, the member can borrow it, and a corresponding lending entry is created. Subsequently, the book is automatically placed on the member's bookshelf as the most recently borrowed item. The member also has the option to manually different roles and respective automatic processes are necessary for the command **Arrange book**.

The second part of the reading discussion contains the reading itself and the possibility of making notes. The result is shown in *Figure 6.19*:

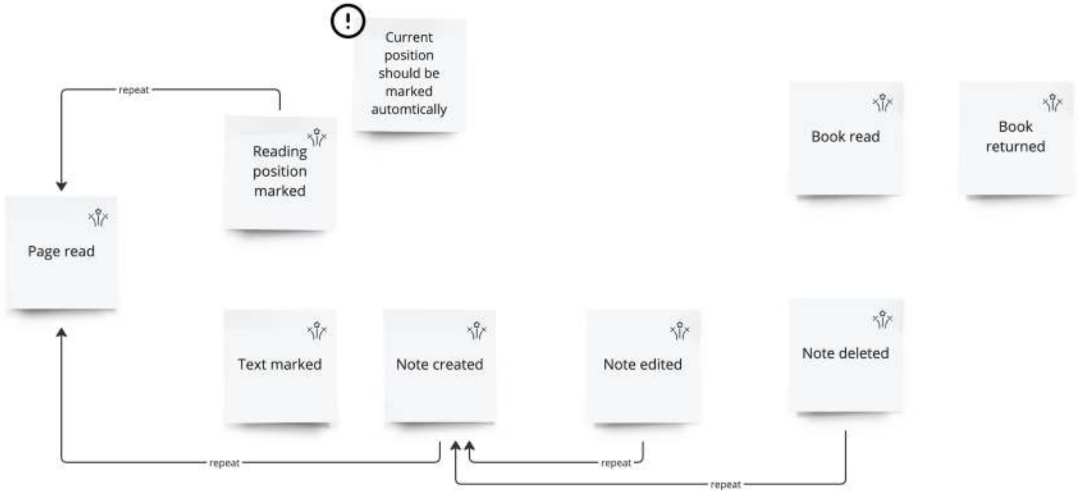


***Figure 6.19**: Events of the second part of reading*

***Source**: Reading2Events.jpg*

A book can be read page by page. During the discussion, it was detected that an event **Book read** is not enough, therefore, an additional event **Page read** is necessary. The reading position can be marked as discussed in the domain storytelling, but it should be possible for the reading position to be marked automatically as well.

Additionally, text can be marked, and notes can be created and edited. When the book is completed, the book can be returned.

The enhancement of the events and the defined bounded context are shown in *Figure 6.20*:

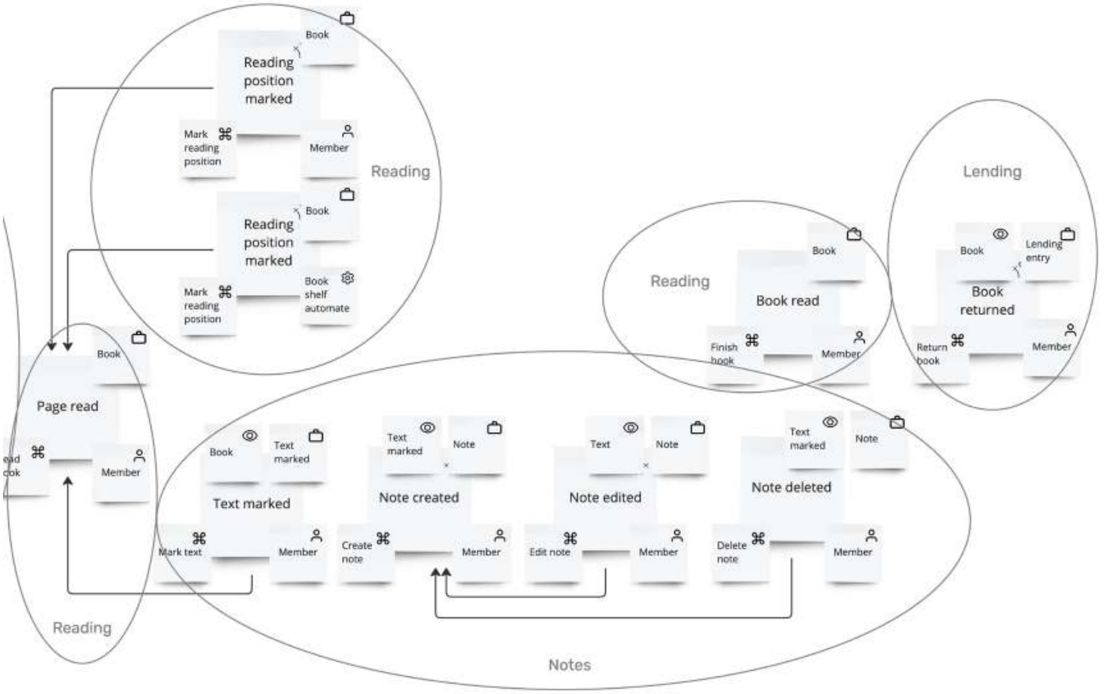


Figure 6.20: Bounded contexts of the second reading part

***Source:** Reading2BoundedContext.jpg*

The events for marking the reading position are separated into a manual and an automatic event.

The bounded context **Reading** appears several times along with the bounded context **Lending**.

To create notes, the text must be highlighted. This functionality was not covered during the domain storytelling, so the visual glossary needs to be updated to include it.

Visual glossary enhancement

The enhancement is shown in *Figure 6.21*:

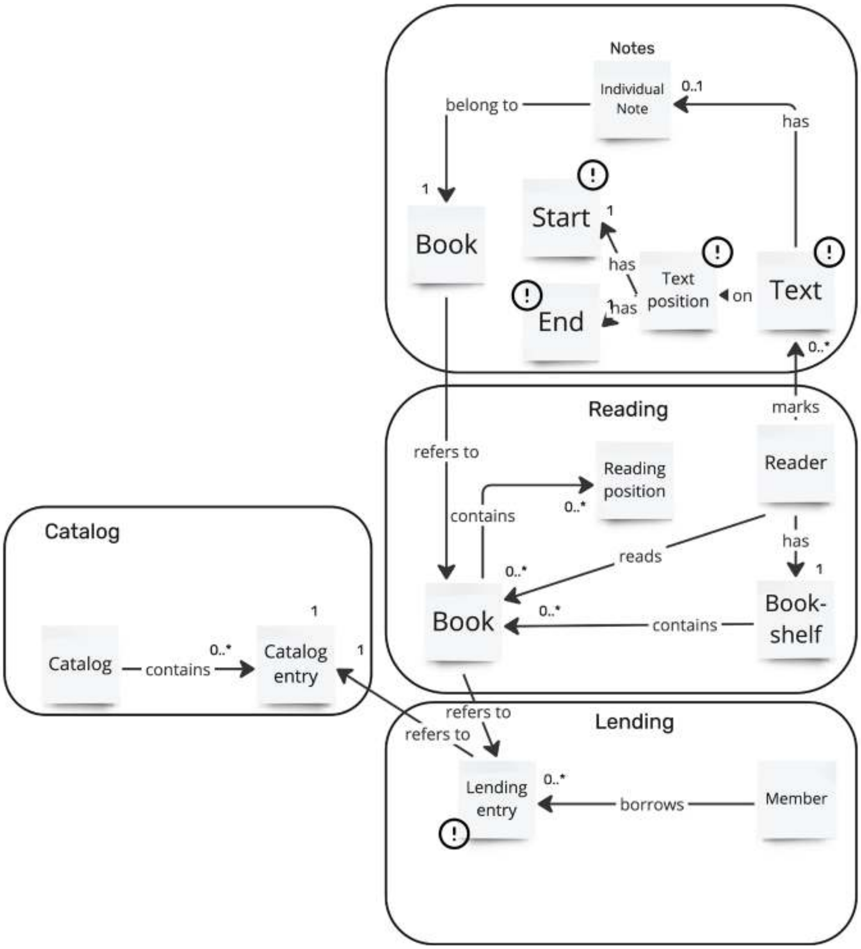


***Figure 6.21**: Enhanced visual glossary reading*

***Source**: ReadingVisualGlossary.jpg*

The changes to the visual glossary of the domain storytelling are highlighted with exclamation marks.

The further two capabilities are as follows:

- **Notes**: The reader can mark several text passages in the book read. Those text passages contain a start and an end position. Belonging to those text passages, a member can create notes.
- **Lending**: When a member borrows a book, a lending entry is created. The lending entry refers to the catalog entry describing the book.

The basic reading capability is considered a commodity, as discussed in *Chapter 4, Bounded Context and Domain*. However, several key propositions need careful discussion in the upcoming implementation steps. Specifically, the bookshelf arrangements and the note-taking features require special attention.

The next part of the event storming workshop covers the discussions.

Discussions

Discussions are done by members of the library, whereas one member initiates the discussion group. A member of a discussion group has access to all notes created by other group members related to the book being discussed.

The belonging events are shown in *Figure 6.22:*

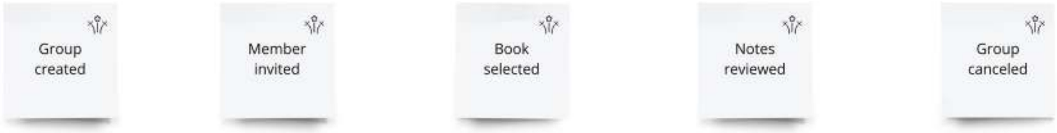


***Figure 6.22**: Events detected for the capability discussion*

***Source**: DiscussionEvents.jpg*

A group needs to be created so that a member can be invited to it. A book needs to be selected so that the notes can be accessed by all member of the group.

It needs to be discussed how member can be invited and how notes are reviewed. The according result is shown in *Figure 6.23*:

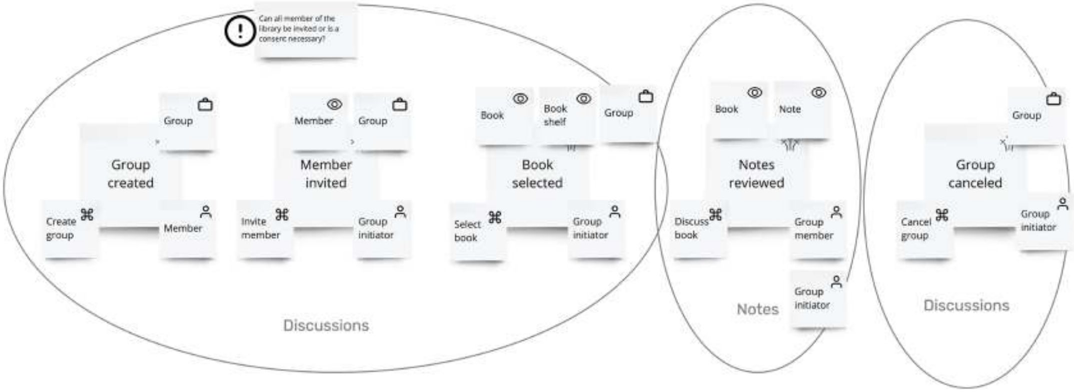


***Figure 6.23**: Bounded contexts found during events storming for the capability discussion*

***Source**: DiscussionBoundedContext.jpg*

A member can create a group and, as the group initiator, invite other members to join. It remains to be decided whether a member can view all other library members in order to invite them. It is assumed that obtaining corresponding consent from the members will be necessary. The method for obtaining this consent will need to be discussed during the implementation stage.

Group members must decide which book they wish to discuss. Upon selecting the book, the related notes will be made available for reading by all group members.

If the group is no longer needed, the group initiator has the authority to cancel it.

The relationship of members, group initiators, and invited members was not discussed during the domain storytelling. The belonging visual glossary needs to be enhanced.

Visual glossary enhancement

A member of the library can create a group and invite other members to it. The corresponding enhancement is shown in *Figure 6.24*:

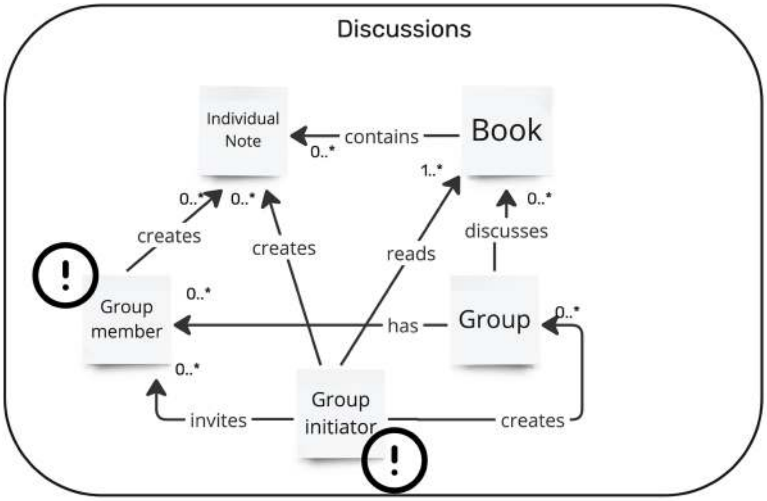


***Figure 6.24**: Enhancement of the Discussions boundary*

***Source**: DiscussionVisualGlossary.jpg*

The changes to the original visual glossary of the domain storytelling are highlighted with exclamation marks. The changes to the initial visual glossary created during the domain storytelling is explained in the next paragraph:

- **Discussions**: A member of the library as a group initiator can create a group and invite other members to the group.

The visual glossary guides the discussion on when members become group members—whether it is upon receiving the invitation or only after accepting it. During the event storming workshop, attendees decide that members must accept the invitation first. The corresponding status figure is shown in the following *Figure 6.25*:

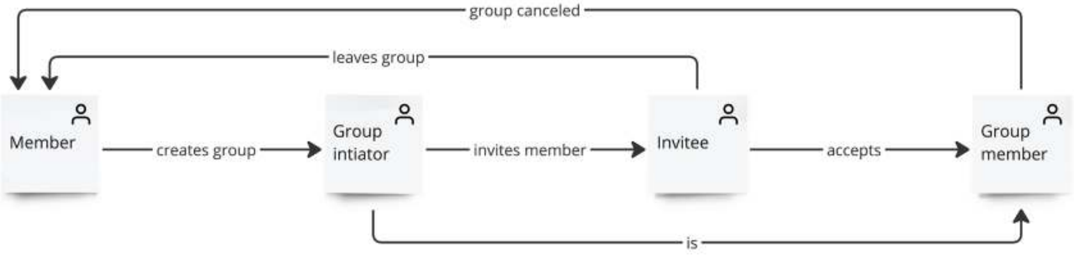


***Figure 6.25**: Discussion group members state diagram*

***Source**: DiscussionMemberState.jpg*

A member becomes a group initiator upon creating a group and is automatically included as a group member. The group initiator can invite other library members, who are designated as invitees upon receiving the invitation. Once an invitee accepts the invitation, they become a group member. If the group is canceled, all group members, including the group initiator, revert to their status as regular library members.

The last capability to be handled in the event storming sessions is the recommendation.

Recommendation

The found events for recommendation are shown in *Figure 6.25*:

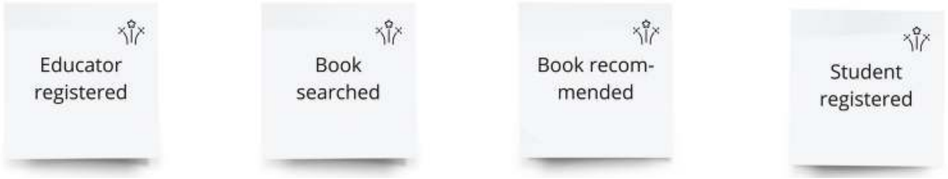


***Figure 6.26**: Events for recommendation capability*

***Source**: RecommendationEvents.jpg*

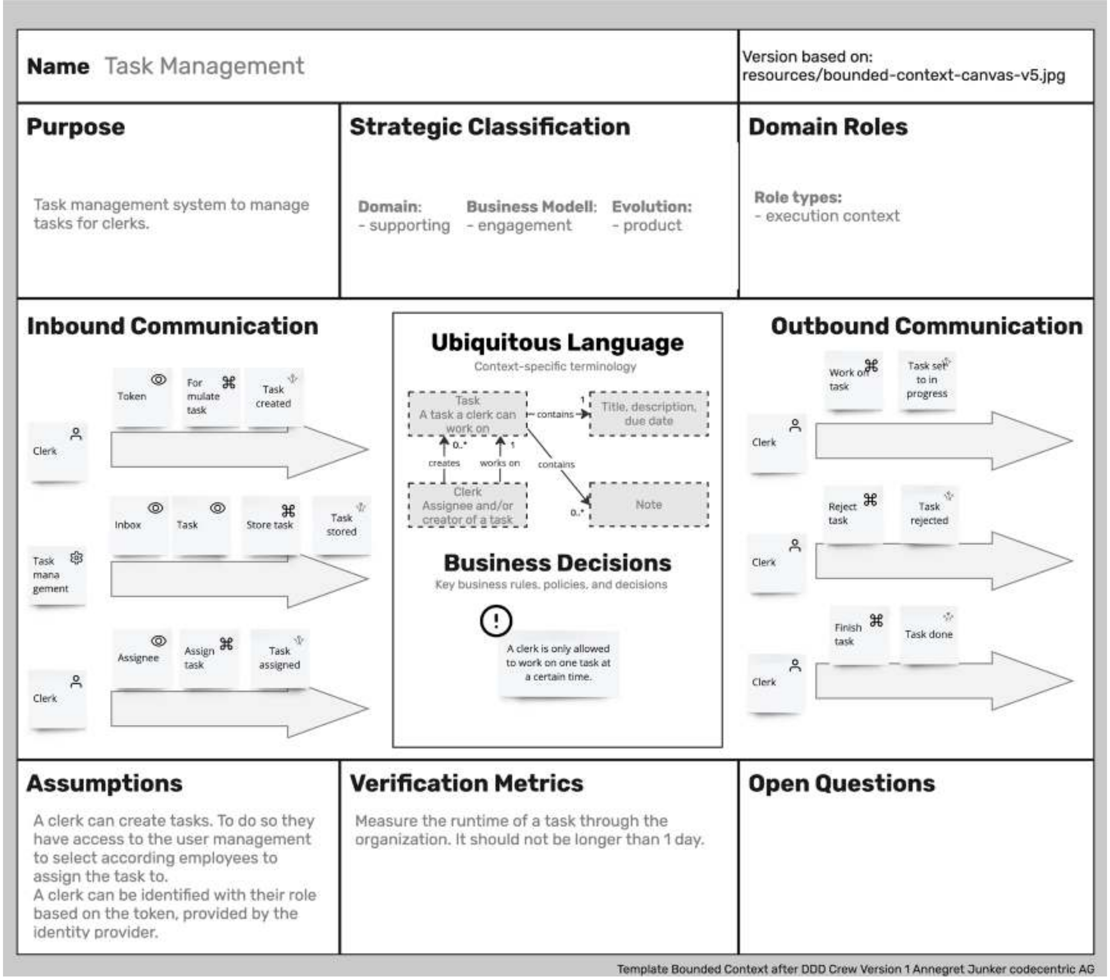
The events identified for recommendation—**Educator registered**, **Book searched**, **Book recommended**, and **Student registered**—have already been discussed in the contexts of **Reading** or **Member management**. Therefore, further discussion is not necessary at this time. A separate bounded context for recommendation is not required. However, it may be beneficial in the future to allow students to download book lists without having to search for each individual book on reading lists.

The found bounded contexts in the event storming can be documented in the bounded context canvas, which will be explained in the next section.

Bounded context canvas

For each of the bounded contexts, a canvas can be created. The canvas was created by the DDD crew and it is published on GitHub[11].

In this section, we will explain the usage of a bounded context canvas based on the task management example. The canvases for the online library are available via GitHub. The bounded context for task management is shown in *Figure 6.27*:



Name Task Management

Version based on: resources/bounded-context-canvas-v5.jpg

Purpose

Task management system to manage tasks for clerks.

Strategic Classification

Domain: - supporting

Business Modell: - engagement

Evolution: - product

Domain Roles

Role types: - execution context

Inbound Communication

Ubiquitous Language

Context-specific terminology

Task
A task a clerk can work on
contains
1
Title, description, due date
0..*
creates
1
works on
contains
Clerk
Assignee and/or creator of a task
0..*
Note

Business Decisions

Key business rules, policies, and decisions

A clerk is only allowed to work on one task at a certain time.

Outbound Communication

Assumptions

A clerk can create tasks. To do so they have access to the user management to select according employees to assign the task to.
A clerk can be identified with their role based on the token, provided by the identity provider.

Verification Metrics

Measure the runtime of a task through the organization. It should not be longer than 1 day.

Open Questions

Template Bounded Context after DDD Crew Version 1 Annegret Junker codecentric AG

***Figure 6.27**: Bounded context canvas task management*

***Source**: BoundedContextCanvasTaskManagement.jpg*

---

[11] Baas, K., Lindhard, K., Plöd, M., & Sanglan-Charlier, M. (2024, May 20). The Bounded Context Canvas. Retrieved May 2024, from GitHub: **https://github.com/ddd-crew/bounded-context-canvas/tree/master**

The single sections of the canvas are explained as follows:

- **Name**: It contains the name of the bounded context: Task management.
- **Purpose**: Formulates the purpose of the bounded context in a business point of view.
- **Strategic classification**: The strategic classification contains the properties to identify how much important the bounded context for the according company is. The properties were discussed in *Chapter 4, Domain and bounded context*.

  The domain can be core, supportive, or generic.

  The business model means that if revenue can be generated or if the bounded context is unavailable, it is a risk. The business function of the bounded context can be categorized in the following points:

  - Revenue generator means a user will pay for the functionality.
  - Engagement creator means that users will pay for it, but do not pay for it.
  - Compliance enforcer will protect the company's reputation and existence.

  The evolution means the evolution stages of Wardley maps, as discussed in *Chapter 4, Domain and Bounded Context*.

- **Domain roles**: Domain roles are archetypes of the domain as described by *Brandolini*[12] and *Wirfs-Brock*[13]:
  - **Analysis**: means that a high volume of data is analyzed
  - **Draft**: means that the context needs further modeling
  - **Execution**: means that the context enforces the workflows of the company
  - **Gateway**: means that the context has the role of a gateway to or from the company
- **Inbound communication**: Inbound communication contains the messages that need to be received by the bounded context. We will discuss the messages in detail in the next *Chapter 7, Context Map*. The messages are defined by the trigger command, the role or the process performing the command, and the resulting event.
- **Outbound communication**: The outbound communication contains the outgoing messages in the same form as the inbound communication. We will discuss both variations in the next chapter.

---

[12] Martraire, C. (2012, September 18). Collaborative Construction by Alberto Brandolini - An archetype of Bounded Contexts. Retrieved May 2024, from Medium: **https://medium.com/@cyrillemartraire/collaborative-construction-by-alberto-brandolini-an-archetype-of-bounded-contexts-bea640bbb5b**

[13] Wirf-Brock. (2006). A Brief Tour of Responsibility-Driven Design. Retrieved May 2024, from Wirfs-Brock:

- **Ubiquitous language**: The ubiquitous language section contains the definition of the language used in the context. The according visual glossary can be used to define the terms.
- **Business decisions**: Business decisions need to be documented in a clear way. The middle section can be used for it. For example, for task management, it is decided that a clerk should not be allowed to work on multiple tasks in parallel.
- **Assumptions**: Assumptions need to be discussed and documented carefully. Especially those assumptions where the workshop attendees have different opinions. For example, the role of clerk should be provided by the identity provider via a token. However, it is uncertain if this is feasible. Therefore, it is documented as an assumption.
- **Verification metrics**: During the domain storytelling and event storming workshops, how the model can be proven needs to be discussed. The metrics need to be defined accordingly.
- **Open questions**: If there are still open questions, they can be collected in this section.

The bounded context canvas can help in documenting the bounded context in a standardized way and helps people who did not attend the workshops to understand the function of the bounded context afterwards. Let us summarize the regulations of event storming workshops.

Guidelines in event storming workshops

Event storming workshops are a great way to detect and define the boundaries of contexts. Moreover, finding domain events along the business process helps to understand the business and define a suitable architecture. Anyhow, event storming workshops should be prepared and performed carefully. The points in the next paragraphs will guarantee a successful event storming workshop:

- **Do not change the workshop attendees**: Workshop attendees should not be changed along a workshop series. When the event storming workshops follow domain storytelling workshops, it is advised not to change the attendees. The moderator can change when the domain storytelling workshops were not moderated by an architect.
- **Moderator needs to be an architect**: An event storming workshop should be moderated by an experienced architect, who can give hints about suitable aggregates and views.
- **Do not criticize found events before consolidation**: Found events by the attendees should not be criticized by the moderator or even by other attendees. Only the

- **Do communicate a standard color coding or iconography before starting**: The events should adhere to a specific iconography and color-coding system proposed by *Brandolini*[14]. The suggested color scheme is as follows:
    - Orange for events
    - Light yellow for aggregates
    - Green for views
    - Blue for commands
    - Dark yellow for roles
    - Pink for external processes

While the processes described in this chapter are not inherently external, they are treated as such automatically.

Additionally, *Brandolini* proposed the following:

- White for user interfaces
- Purple for conditions

Note: These categories were not covered in the chapter as the author finds them rarely necessary based on her experience.

- **Communicate the place of the notes in relation to the even**t: Additionally, the placement of notes should be standardized. The following arrangement is suggested (oriented on *Brandolini*):
    - Aggregates should be placed at the upper right corner of the event.
    - Views should be positioned to the left of the aggregate.
    - Commands should be placed at the lower left corner of the event.
    - Roles or processes should be located at the lower right corner of the event.

While some place commands between events, this approach often adds little additional information. Therefore, commands should be smaller than the events.

---

[14] Brandolini, A. (2023). EventStorming. Retrieved 04 2024, from EventStorming: **https://www.eventstorming.com/**

The arrangement is shown in *Figure 6.28*:

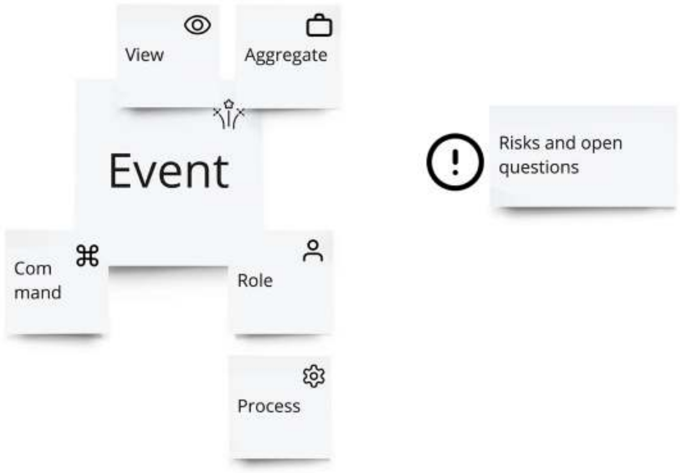


***Figure 6.28**: Arrangement of the different notes to an event*

- **Document open questions and risks**: When certain questions cannot be resolved during the discussion, the moderator should document them using appropriately marked sticky notes. The rule is to document all discussions, even if they do not solve the immediate problem. This practice ensures that all points are recorded, which can prevent repetitive discussions in the future.

Event storming is a great tool to create appropriate architecture and to prepare sustainable and modern architecture.

Conclusion

This chapter introduces event storming as a workshop format to find bounded contexts and domain event along a business process. Event storming workshops can be held without previous workshops, or the can held subsequent to domain storytelling workshop.

The collaborative nature of these workshops allows people with quite different experiences to attend even though the workshops should be moderated by a seasoned architect.

Event storming workshops result in the definition of bounded contexts. Those are the base of context maps, which will be discussed in the next chapter.

In the next chapter, context maps are used to show the dependencies between bounded contexts which were defined during the event storming workshop. Additionally, the data flow between the bounded contexts can be defined, as it is explained in the next chapter.

Points to remember

- Event storming is a workshop format which can be used to find domain events and bounded context.
- Event storming bases on the brainstorming method and defers judgement and reaches for quality throughout the process.
- Event storming processes the following steps:
  - Collect events
  - Consolidate events
  - Find commands
  - Find roles and processes
  - Find aggregates and views
  - Find bounded context
- Schedule an event storming workshop for at least one day. For more complex problems two days are necessary.
- If the event storming workshop were done after a domain storytelling workshop, the already existing visual glossaries can be enhanced. If no visual glossaries exist, they need to be created based on the discussions of the event storming workshop.

Multiple choice questions

1. **Select the correct statement**
   a. Event storming workshops judge events found by attendees. False events are not used for further discussions.
   b. Event storming workshops do not judge found events during the storming phase.
   c. Event storming workshops can only be attended by domain experts.
   d. Event storming workshops should be done in nature and not in closed rooms.
2. **Select the correct order of steps in an event storming workshop**
   a. Collect events, enhance events by aggregates, enhance events by roles
   b. Collect events, consolidate events, find commands
   c. Collect events, consolidate events, find commands, enhance events by roles and processes, enhance events by aggregates and views, find bounded context
   d. Collect events, consolidate events, find bounded contexts, find aggregates, find commands

3. **How long would an event storming workshop take for middle-complex problem?**
    a. 2 hours
    b. 2 days
    c. 2 months
    d. 2 years

4. **Should a visual glossary created in a domain storytelling workshop preceding the event storming workshop be enhanced by a event storming workshop?**
    a. No, never
    b. Yes, in every case
    c. Yes, when necessary
    d. Yes, but only when the domain storytelling workshop were not successful

5. **Should a found bounded context be named?**
    a. No, it should only be numbered
    b. No, it should get only the number of found aggregates
    c. Yes, it should get the name of the aggregate
    d. Yes, it should get the name the attendees can agree upon

6. **Is the following statement correct: A bounded context canvas can be used to document a bounded context in a standardized way.**
    a. Correct
    b. Incorrect
    c. Correct only for bounded contexts in the security domain
    d. Incorrect, but they can be used by user interface designers

Answers

1. **b**
2. **c**
3. **b**
4. **c**
5. **d**
6. **a**

CHAPTER 7

Context Map

Introduction

The derived bounded contexts from an event storming or domain storytelling workshop can be summarized in a context map.

A context map gives an overview of the bounded contexts in one or more subdomains. It facilitates the team dependencies between the bounded contexts and the belonging services. Moreover, a context map can be used to deduce the data flows synchronously and asynchronously between the bounded contexts.

Structure

In this chapter, we will cover the following topics:

- Context map
- Creation of a context map
- Domain events and REST interfaces
- API definitions
- Context map and API definition

Objectives

This chapter covers the DDD overview tool called **context map**. You will learn how to create a context map using the results of an event storming workshop. Moreover, you will learn how to derive the team dependencies from it and how to define **application programming interfaces (APIs)** using the context map.

> **Do not quench your inspiration and your imagination; do not become the slave of your model.**
>
> *~Vincent van Gogh*

Context map

First and foremost, a context map is used to visualize the dependencies between bounded contexts.

The context map was first introduced by *Eric Evans*[1], showing communication patterns between teams. As discussed in *Chapter 4, Bounded Context and Doma*in, a context map shows different patterns of dependencies, for example: shared kernel, open host service, anti-corruption layer etc., and if dependencies are upstream or downstream. A sample of such a context map is shown in *Figure 7.1*:

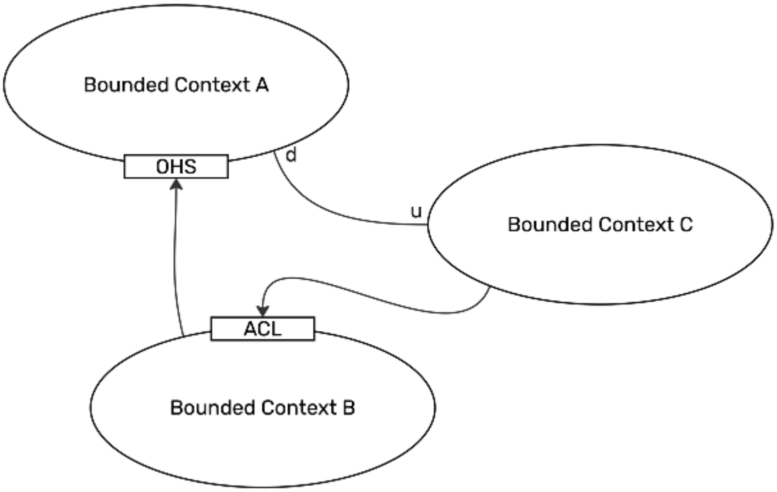


Figure 7.1*: General principle of a context map*

Source*: ContextMapGeneral.jpg*

---

[1] Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley., pp. 344 ff

Figure 7.1 shows three different bounded contexts. Bounded context A is an **Open Host Service (OHS)**, whereas bounded context B can be accessed via an **Anti-corruption Layer** (**ACL**). Bounded context C and bounded context A depend on each other. Bounded context A has a downstream dependency to bounded C. The according dependency is shown by the characters d and u. Using this mapping, the found bounded contexts of the event storming can be analyzed and the dependencies can be formulated.

According to the DDD Crew, a context map describes the contact between bounded contexts and teams with a collection of patterns[2]. The context map patterns describe a variety of perspectives like service provisioning, model propagation or governance aspects. This diversity of perspectives enables you to get a holistic overview of the team and bounded context relationships.

The team dependencies can be mutual when both teams cannot deliver independently. An upstream-downstream dependency appears when changes by the downstream team do not have a significant influence on the upstream team, but changes by the upstream team have a significant influence on the downstream team. The downstream team cannot deliver without collaboration with the upstream team, whereas the upstream team might deliver independently. Usually, free teams will be the best choice to deliver functionality because they can deliver independently.

The context patterns OHS, conformist, anti-corruption layer, and shared kernel were discussed in *Chapter 4, Bounded Context and Domains*.

The partnership pattern appears when two teams can deliver independently. However, they need a tight synchronization of their activities[3]. Bounded contexts do have the relationship of customer and supplier, which is explained by the pattern name.

Two bounded contexts might even depend on each other but need a common language. The sharing of a common language between two bounded contexts is called **published language** (**PL**)[4]. Two bounded contexts might even have no significant relationship to each other – that is called **separate ways**[5].

In modern software systems, it is common to encounter mixed models and inconsistent boundaries, often leading to what is termed a **Big Ball of Mud** (**BBoM**). Rather than allowing this disorganized structure to infiltrate well-maintained boundaries, it is crucial to describe and encapsulate it with clearly defined borders[6].

---

[2] Gunia, K., & Tune, N. (2023, October 9). Context Mapping. Retrieved June 2024, from DDD Crew: **https://github.com/ddd-crew/context-mapping**

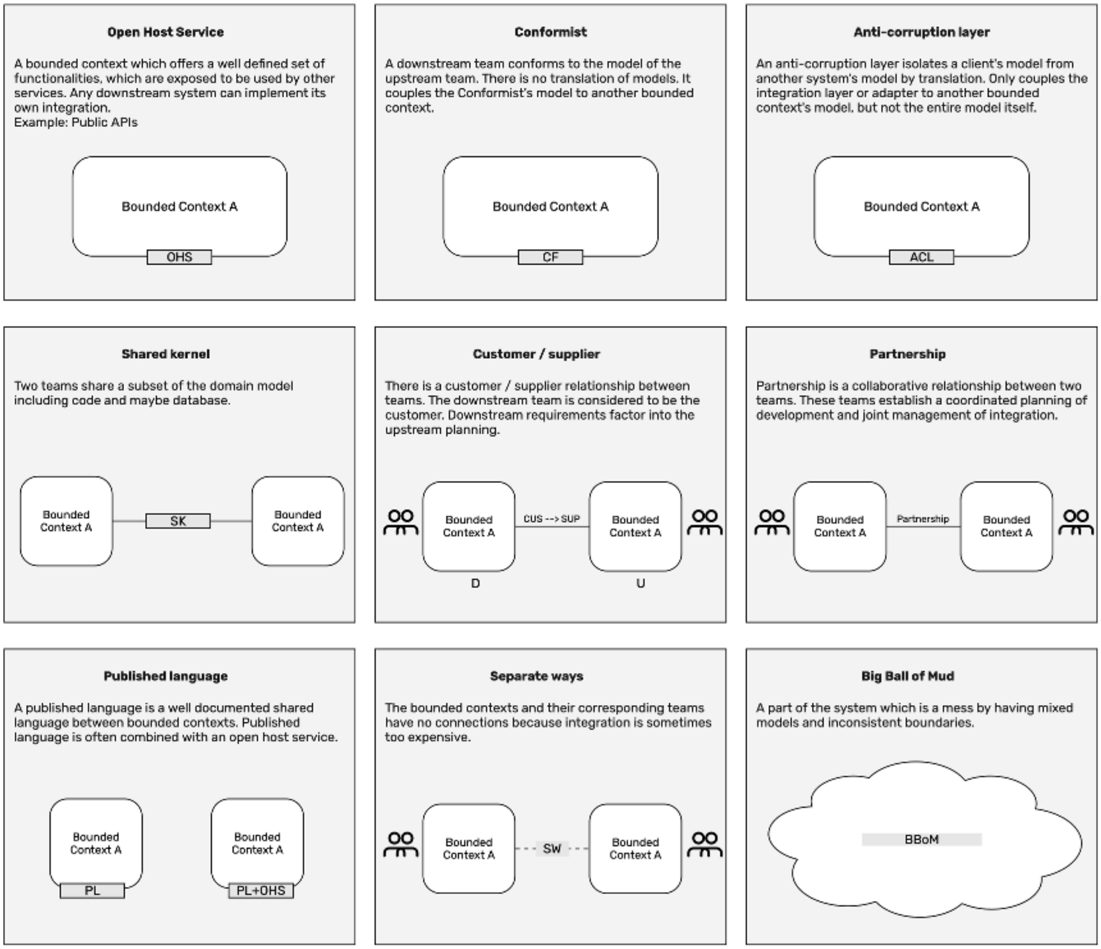
[3] Evans, E. (2016). Domain-Driven Design Reference. Domain Language, Inc.

[4] Evans, E. (2016). Domain-Driven Design Reference. Domain Language, Inc.

[5] Evans, E. (2016). Domain-Driven Design Reference. Domain Language, Inc.

[6] Gunia, K., & Tune, N. (2023, October 9). Context Mapping. Retrieved June 2024, from DDD Crew: **https://github.com/ddd-crew/context-mapping**

An overview of those team dependencies and communication patterns is shown in *Figure 7.2* and *Figure 7.3*:



Open Host Service

A bounded context which offers a well defined set of functionalities, which are exposed to be used by other services. Any downstream system can implement its own integration.
Example: Public APIs

Conformist

A downstream team conforms to the model of the upstream team. There is no translation of models. It couples the Conformist's model to another bounded context.

Anti-corruption layer

An anti-corruption layer isolates a client's model from another system's model by translation. Only couples the integration layer or adapter to another bounded context's model, but not the entire model itself.

Shared kernel

Two teams share a subset of the domain model including code and maybe database.

Customer / supplier

There is a customer / supplier relationship between teams. The downstream team is considered to be the customer. Downstream requirements factor into the upstream planning.

Partnership

Partnership is a collaborative relationship between two teams. These teams establish a coordinated planning of development and joint management of integration.

Published language

A published language is a well documented shared language between bounded contexts. Published language is often combined with an open host service.

Separate ways

The bounded contexts and their corresponding teams have no connections because integration is sometimes too expensive.

Big Ball of Mud

A part of the system which is a mess by having mixed models and inconsistent boundaries.

***Figure 7.2**: Overview of context map communication patterns after* DDD Crew[7]
***Source**: ContextMapSheet1.jpg, ContextMapSheet2.jpg, ContextMapSheet3.jpg*

---

[7] Gunia, K., & Tune, N. (2023, October 9). Context Mapping. Retrieved June 2024, from DDD Crew: https://github.com/ddd-crew/context-mapping

The following figure shows the team dependencies after DDD crew:

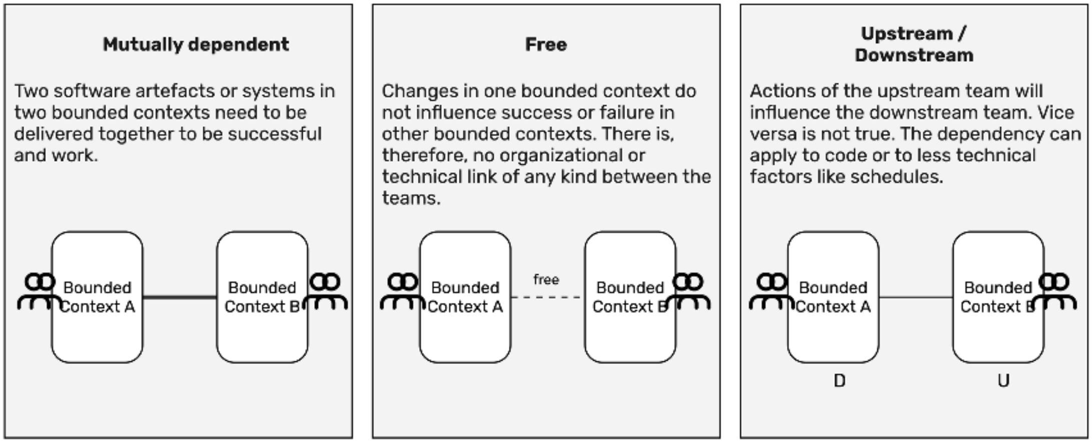

Mutually dependent
Two software artefacts or systems in two bounded contexts need to be delivered together to be successful and work.
Bounded Context A
Bounded Context B
Free
Changes in one bounded context do not influence success or failure in other bounded contexts. There is, therefore, no organizational or technical link of any kind between the teams.
Bounded Context A
free
Bounded Context B
Upstream / Downstream
Actions of the upstream team will influence the downstream team. Vice versa is not true. The dependency can apply to code or to less technical factors like schedules.
Bounded Context A
Bounded Context B
D
U

Figure 7.3: *Team dependencies after DDD Crew*[8]

Source: *TeamDependencies.jpg*

Let us apply those communication patterns and team dependencies to the task management already introduced in *Chapter 5, Domain Storytelling,* and *Chapter 6, Event Storming*.

Creation of a context map

Based on the discussed event storming, a context map can be easily created by following the given steps:

1. First, draw a rectangle for each identified bounded context. If a bounded context appears multiple times during the stream of events, it is drawn only once.
2. Next, place the identified events into their corresponding rectangles.
3. Finally, copy each found aggregate into the appropriate rectangle representing its bounded context, ensuring each aggregate is included only once.

---

[8] Gunia, K., & Tune, N. (2023, October 9). Context Mapping. Retrieved June 2024, from DDD Crew: **https://github.com/ddd-crew/context-mapping**

The result is shown in *Figure 7.4*:

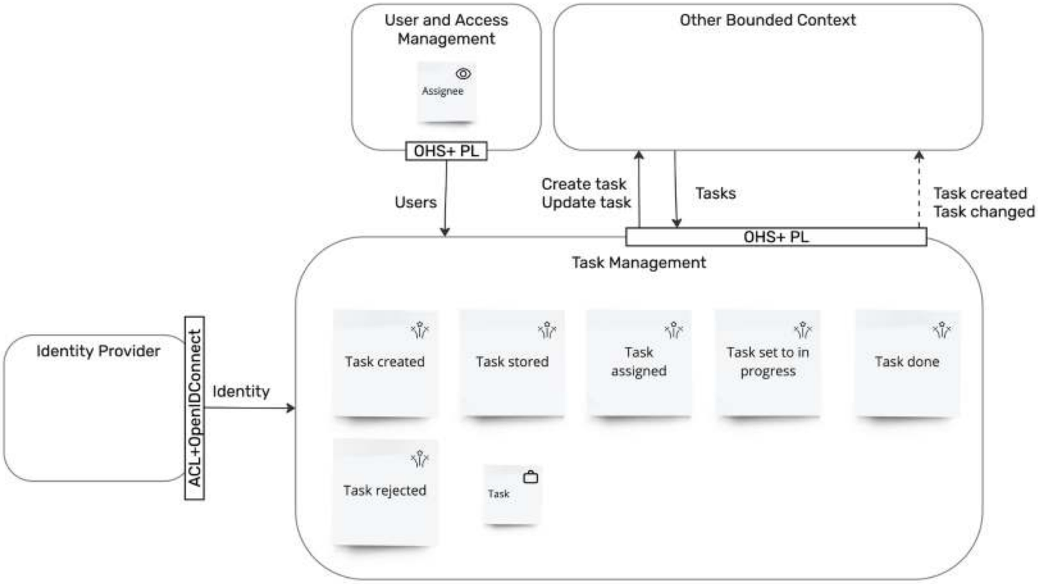


Figure 7.4: *Context map task management*

Source: *ContextMapTaskManagement.jpg*

The task management service is an OHS, which publishes its language via REST interfaces.

Let us assume the identity provider is an external, generic service purchased from a third-party provider. To ensure the security of the internal infrastructure, an anti-corruption layer is implemented in front of the identity provider. This layer acts as a protective barrier, preventing potential external dependencies from corrupting the internal model. The identity provider leverages the OpenID Connect[9] standard interface for secure and standardized authentication as a well-defined PL.

The user and access management service is an open host service that communicates with task management and other related services. It serves as a typical supportive service due to the generic nature of user management, in contrast to access management, which is tailored to specific business needs.

Domain events and REST interfaces

When we discuss bounded contexts in context maps, we not only discuss the team dependencies and the nature of the service. We discuss the communication patterns and the data to be exchanged as well.

---

[9] OpenID Foundation. (2024). What is OpenID Connect. Retrieved June 2024, from openid: What is OpenID Connect

The data exchange can be done synchronously via REST interfaces, **gRemote Procedure Call** (**gPRC**), or **Simple Object Access Protocol** (**SOAP**) or asynchronously via domain events. In the next sections, we will shortly discuss what synchronous and asynchronous communication means. We assume a synchronous communication via REST.

Synchronous communication

Synchronous communication occurs when a client and server exchange information through an established connection. It is like a phone call; the server must answer the call, whereas the client initiates the communication. A double-pointed arrow shows this principle in *Figure 7.5*.

Representational State Transfer (**REST**) is today's de facto standard for synchronous communication. It means that instead of accessing objects directly, e.g., via database calls, the state of a business object is transferred to the client as an object. Such objects are called **resources in REST**, which are transferred or manipulated using standard HTTP verbs: POST to create a resource, PUT to manipulate a resource, DELETE to delete a resource, or GET to read a resource. We will discuss the design of REST interfaces later in *Chapter 11, Exposing Aggregates via RESTful APIs* in more detail. For now, it is enough to know that we publish the bounded context language via REST for synchronous communication.

Asynchronous communication

Asynchronous communication allows a client to read a message regardless of whether it is available when the server initiates the communication. It is similar to an advertisement, where the client can choose whether to read and respond to it or not. Usually, the server sends its message to a broker. The broker either forwards the message to the client or stores it for the client to fetch. However, asynchronous communication can be established even without a broker. Because the communication is not answered directly, the arrows in *Figure 7.5* for asynchronous communication point only in one direction.

The domain events we found earlier, like *Task created*, can be published asynchronously.

Both principles are shown in *Figure 7.5*:

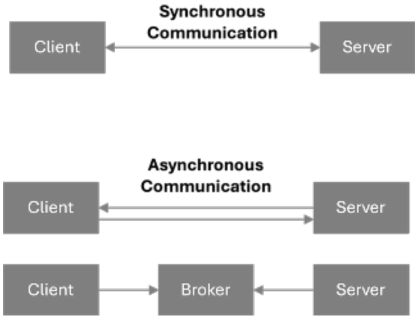


***Figure 7.5**: Principles of synchronous and asynchronous communication*

***Source**: PrincipleSynchAsych.png*

West-east and north-south communication

Asynchronous communication is commonly employed in server-to-server interactions within a microservice architecture. This approach ensures a high degree of decoupling between services, enabling them to operate independently and enhancing system resilience and scalability.

In contrast, synchronous communication is typically necessary when a user interface interacts with a server. This type of communication provides immediate feedback to the user and ensures real-time data exchange.

In architectural diagrams, user frontends are usually depicted at the top, with service-to-service interactions flowing from left to right. Consequently, communication between the frontend and services is referred to as north-south communication, while service-to-service communication is termed **east-west communication**[10]. The corresponding principle is illustrated in *Figure 7.6*:

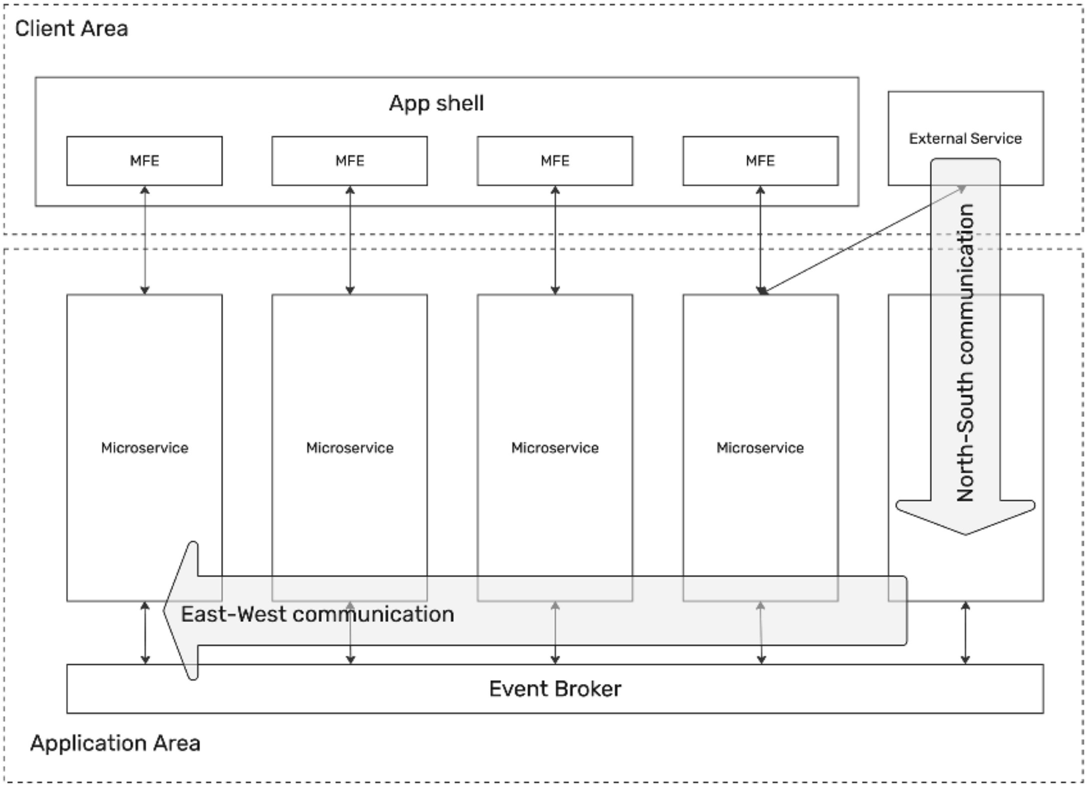


***Figure 7.6**: North-south and west-east-communication principle*

***Source**: WestEastNorthSouth.png*

---

[10] Junker, A. (2023, May 16). Eventgetriebene Integrationsarchitekturen. Retrieved June 2024, from Informatik aktuell: **https://www.informatik-aktuell.de/entwicklung/methoden/eventgetriebene-integrationsarchitekturen.html**

A user interface accesses services that represent different bounded contexts. Typically, these bounded contexts provide micro-frontends embedded within the corresponding user interface. Here, a **micro-frontend** (**MFE**) refers to the portion of the user interface provided by a specific bounded context service. Similarly, external services can access these bounded contexts via REST interfaces. These interactions are referred to as **north-south communication**.

The bounded context services themselves communicate asynchronously with each other through a broker. It is considered good practice for the bounded contexts to publish certain business state changes, such as *book purchased*, asynchronously via the broker. This type of communication is known as **east-west communication**.

Applying communication patterns in context map

Let us use our task management example again to discuss communication patterns.

The identity management service provides the user's identity, such as through a token, to the task management service. This communication must be synchronous because the user needs to log in first before accessing the task management service. The synchronous nature of the communication is represented by a straight arrow, as shown in *Figure 7.4*.

The user and access management service synchronously provides user information to the task management service, as it is essential to deliver the information the moment a user wants to assign a task to a specific individual. This is shown by straight arrows pointing toward the data flow.

When another bounded context needs to access the task management service, it can interact with the aggregate via REST endpoints for reading, creating, and updating. However, there may be instances where the service needs to be asynchronously notified of task changes. This asynchronous communication is represented by a dashed arrow pointing in the direction of the data flow.

The explained communication patterns and interface principles can be applied to the online library.

API definitions

In the following section, we look at the bounded contexts of the online library found in *Chapter 6, Event Storming*. We start with the purchase area.

Purchase

The context map of the purchase area is shown in *Figure 7.7*:

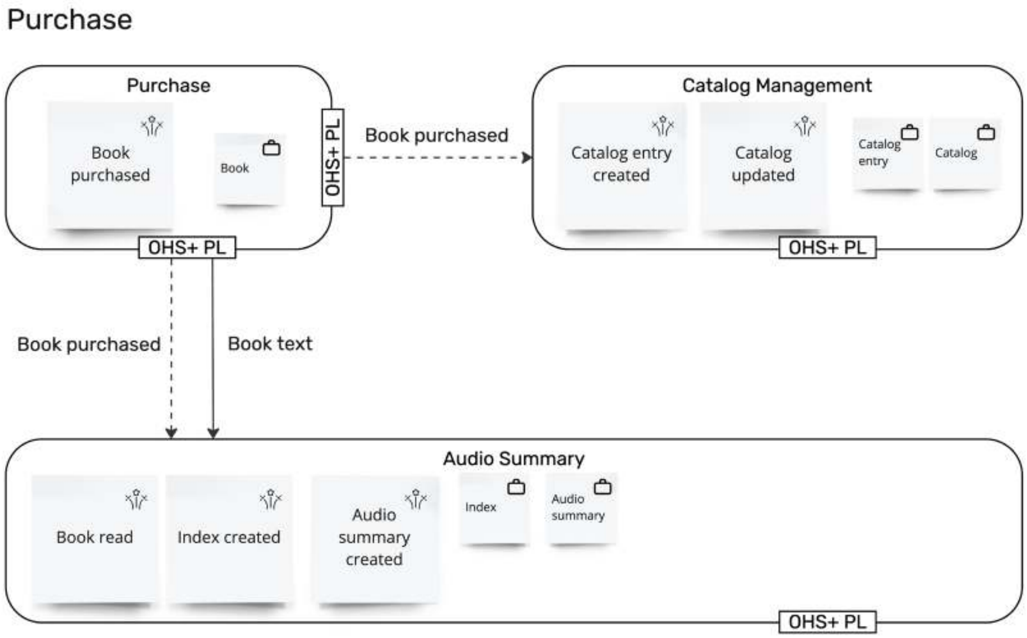


Figure 7.7: *Context map purchase*

Source: *ContexMapPurchase.jpg*

The bounded contexts purchase, catalog management, and audio summary are each OHS with a corresponding PL.

The purchase service publishes a domain event *Book purchased* which is consumed by the services catalog management and audio summary. Both services can react to the event: catalog management will create a catalog entry whereas audio summary creates the corresponding summary.

The audio summary service retrieves the book text, which forms the basis for the summary, synchronously and incrementally from the purchase section. This method is more efficient than embedding the entire text in an event and storing it on both the broker and within the audio summary service.

Next, let us have a look at the member management.

Member management

The context map for the member management is shown in *Figure 7.8*:

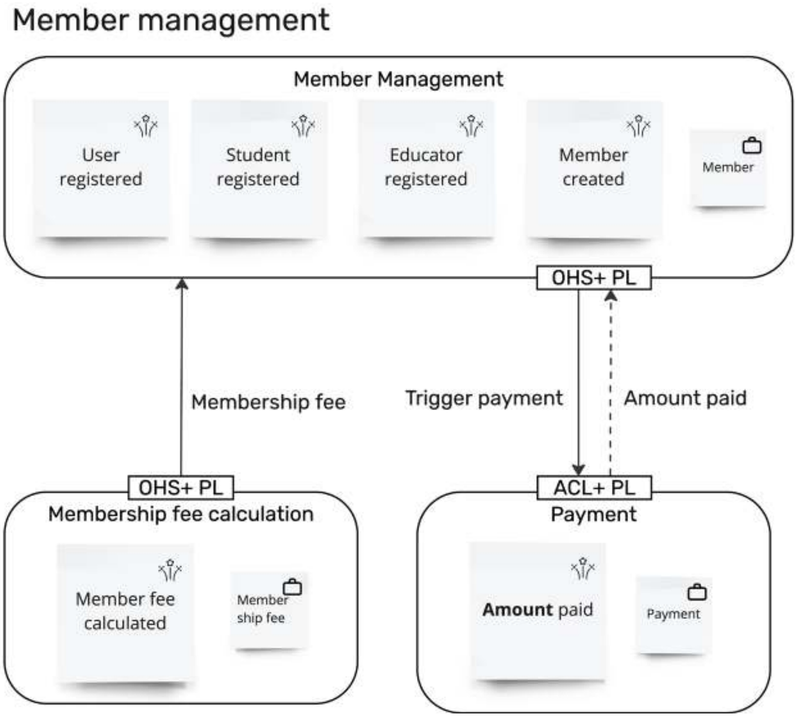


***Figure 7.8**: Context map member management*

***Source**: ContextMapMemberManagement.jpg*

The services member management and membership fee calculation are OHS. Whereas the payment service is an external service which is decoupled from the other services via an ACL. Usually, ACLs are implemented by the organization using an external service. Specialized integration teams can do so.

All three services use the communication pattern PL.

The membership fee calculation provides the membership fee to the member management service via a synchronous call.

The payment service provides the *amount paid* event both synchronously and asynchronously. The initial payment must be processed synchronously to ensure that the member onboarding process is paused until the payment is completed. The corresponding domain event has been renamed from *membership fee paid* as discussed in *Chapter 6*, *Event Storming* to *amount paid*. This change reflects the fact that the payment service is an external service that can be used for various types of payments.

Amount paid is also provided asynchronously. For recurring payments, such as monthly payments, there is no need to trigger the payment process each time; the payment can be scheduled following the initial payment. The member management system can then be informed asynchronously about the success or failure of subsequent payments.

In the next section, we want to discuss the context map of the reading area.

Reading

The context map of the reading area is shown in *Figure 7.9* and *Figure 7.10*:

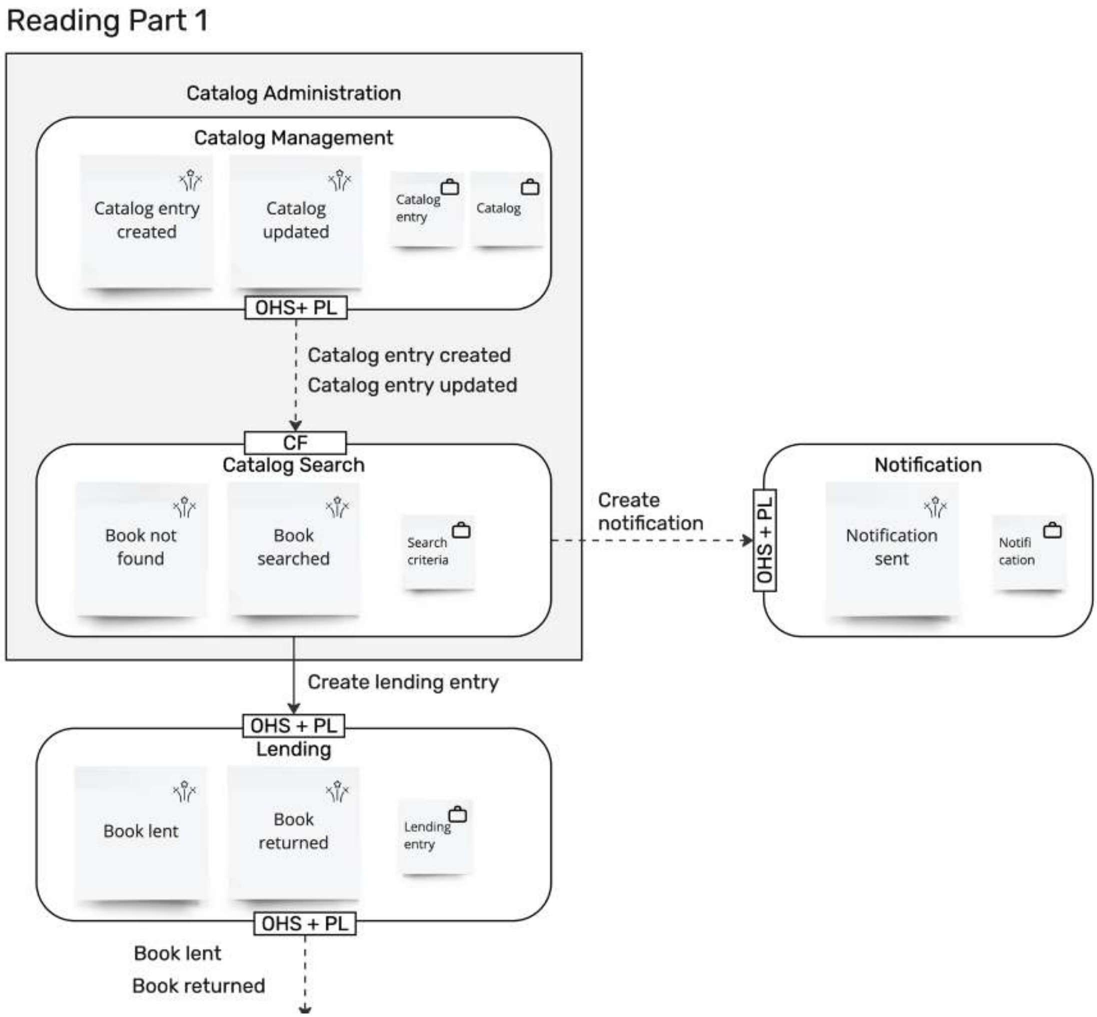


Figure 7.9: *Context map reading*

Source: *ReadingPart1.jpg*

The following figure shows the second part of the context map reading:

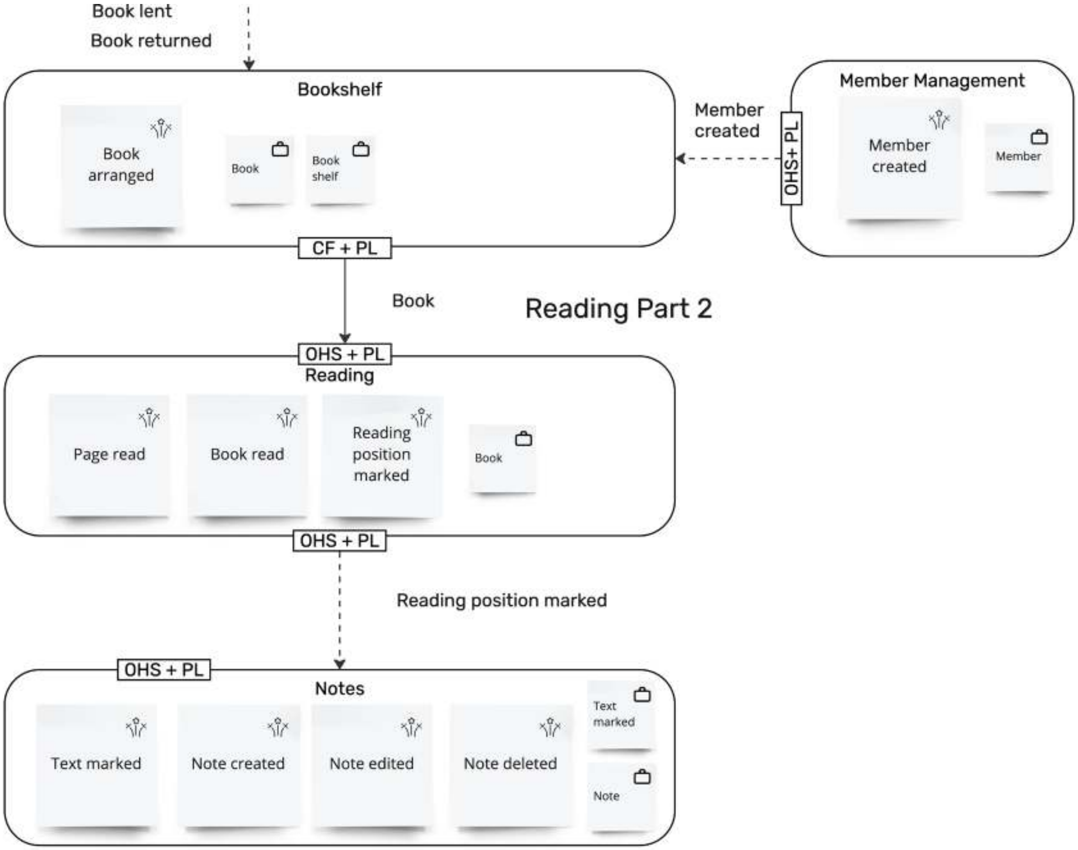


Figure 7.10*: Context map reading part 2*

Source*: ReadingPart2.jpg*

The reading area contains several services.

As library members need the ability to search for books, the catalog search functionality is essential in the reading area. However, the catalog search is closely tied to the catalog management system. Therefore, it is advisable to implement the catalog search as a conformist to the catalog management service.

The catalog management, discussed earlier in the purchase area, publishes domain events such as *catalog entry created* and *catalog entry updated*. The catalog search service can listen to these events and update its search index accordingly.

Given their tight dependency, the catalog search service should act as a **conformist** (**CF**) to the catalog management service. Both services can be developed by a single team and collectively managed under catalog administration.

Catalog management and catalog search follow the **Command Query Responsibility Segregation** (**CQRS**)[11] pattern, which will be discussed in detail in *Chapter 11, Exposing Domain Events*.

The notification service is an open host service with a corresponding PL. It consumes commands like *create notification* and reacts upon receiving them.

The lending service can also be implemented as an open host service. It offers a synchronous interface to create lending entries. These entries are created when a library member successfully searches for a book and confirms their intent to borrow it.

The lending service publishes domain events such as *book lent* and *book returned*, enabling the bookshelf service to consume these events and update the corresponding member's bookshelf accordingly. In addition to handling domain events from the lending service, it consumes domain events from the member management service. When a new member is created, the bookshelf service automatically creates a bookshelf for that member reacting on the domain event *member created*.

The bookshelf service acts as a conformist to the reading service because the book representation, including features such as marking reading positions, is defined by the reading service. The bookshelf service provides its book information to the reading service synchronously e.g. when a member opens a book. The reading service functions as an open host service, publishing its data via domain events, which are then consumed by the notes service.

For example, when a reading position is marked in a book, the reading service publishes a domain event called *position marked*. The notes service, also an open host service, consumes this event and can react by creating a note for the corresponding position in the book.

Finally, we will discuss the area of discussions.

Discussions

The following *Figure 7.11* illustrates the context map of the discussions area:

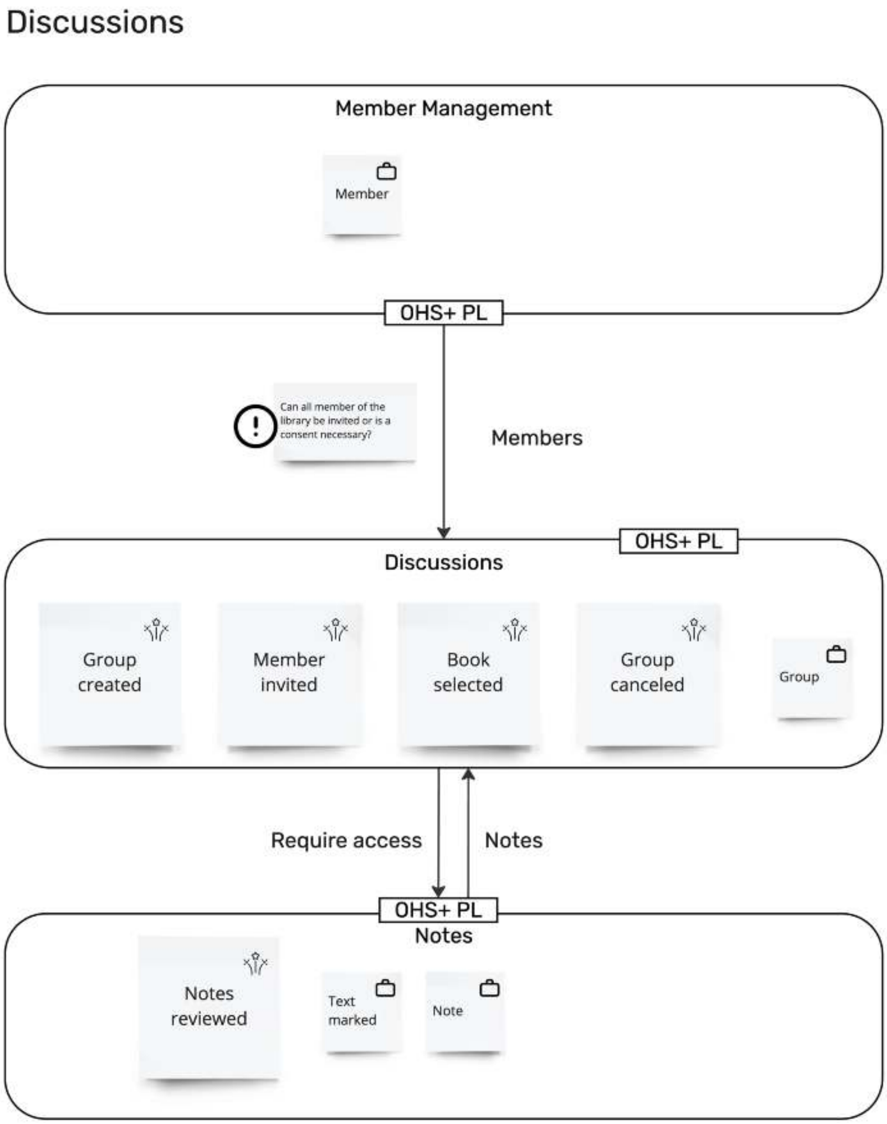


Figure 7.11*: Context map discussions*

Source*: ContextMapDiscussions.jpg*

The discussions area synchronously consumes data from the member management system, enabling group initiators to invite other library members to join a group. Group members can synchronously request access to the notes of other group members. The notes service provides these notes synchronously to the discussions area, facilitating their discussion within the group.

In the following, we will discuss the context map and the belonging API definitions.

Context map and API definition

The entire context map of the online library can be divided into librarian, reader, and member activities. *Figure 7.12* shows the librarian's activities:

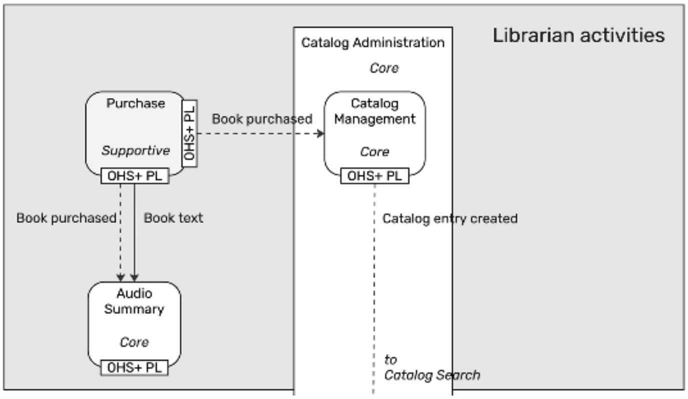


Figure 7.12*: Context map part of the librarian activities*

Source*: Context map librarian*

Figure 7. 13 shows the reader's activities. The bounded context catalog administration covers both librarian and reader activities:

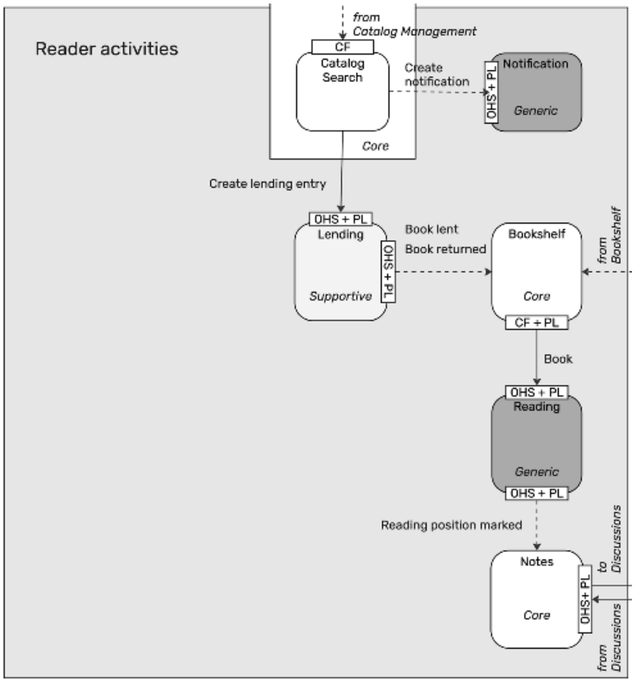


Figure 7. 13*: Context map part of reader's activities*

Source*: ContextMapReader.jpg*

The third part of the activities refers to members, as shown in *Figure 7.14*:

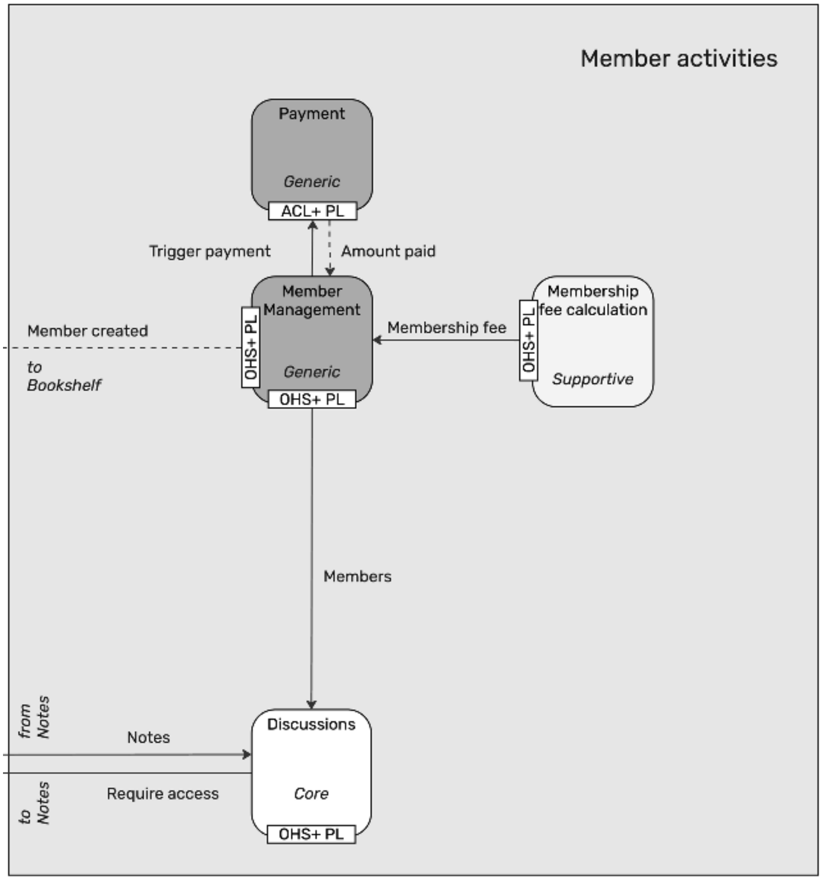


***Figure 7.14**: Context map part of member's activities*

***Source**: ContextMapMember.jpg*

The context maps of the online library give a good high-level overview of how the services of the library should be implemented. The services that belong to a core subdomain are marked with a white background. Services that belong to a supportive subdomain are marked in light grey, whereas the services in the generic subdomains are marked in grey.

The services are highly decoupled, which is visible only due to the following factors:

- Member management and payment
- Member management and membership fee calculation
- Notes and discussions
- Bookshelf and reading
- Purchase and audio summary

They have synchronous communication necessities. All other service communication can be done asynchronously.

One critical point might appear. The bookshelf should follow the reading service as conformist. However, the reading service is a generic component and should be purchased on the market. Which means, that the implementation of the bookshelf might need an anti-corruption layer to guarantee independence from the provider of the reading service. We will discuss that point in more detail in the tactical design when we discuss the API definitions of the single services.

Conclusion

A context map is an invaluable tool for analyzing business requirements and designing modern software architectures. It effectively bridges the tight coupling between business requirements, represented by bounded contexts, and technical design, represented by asynchronous and synchronous APIs. This makes context maps ideal for facilitating collaborative software architecture design between business experts and technical teams.

Even more, a context map gives an overview of how implementing teams can work together. Those dependencies can easily be recognized by business and IT specialists. In such a way, team tailoring can be adapted corresponding to the business needs of the project.

The bounded contexts within the context map are derived from the event storming analysis. The domain events used for data flows are explicitly taken from this analysis. However, the resources for the PL are deduced from the identified aggregates. These API designs are initially outlined as data flows between the identified bounded contexts. In subsequent stages, these designs need to be further detailed through tactical design within each bounded context, as will be explained in *Chapter 10, Exposing Aggregates via RESTful APIs*, and *Chapter 11, Exposing Domain Events*.

In the next chapter, we will give an overview of the strategic part of Domain-Driven Design.

Points to remember

- A context map is used to visualize team dependencies.
- A context map can be used to show synchronous and asynchronous APIs and the corresponding data flow.
- A context map can be derived from the result of an event storming workshop:
    1. Draw a rectangle for each found bounded context.
    2. Copy the belonging events from the event storming into the corresponding

    bounded contexts.

    3. Copy the aggregates once into the bounded context.
    4. Discuss the data flows and team dependencies between the bounded contexts using the results of the event storming workshops.

- When a data flow is recognized, it should be discussed if it is a synchronous or asynchronous data flow. Usually, an asynchronous data flow should be preferred to guarantee a high level of decoupling of the services. If a user interaction is involved, a synchronous communication is necessary.
- Aggregates which belong to a certain bounded context can be published via REST interfaces.
- Business state changes of aggregates can be published via domain events. Those events were found already in the event storming.

Multiple choice questions

1. **What is a context map for?**
    a. Visualize team dependencies
    b. Visualize one bounded context
    c. Visualize one subdomain
    d. Visualizing implementation places
2. **What is NOT a communication pattern of a context map?**
    a. Open host service
    b. Calculation kernel
    c. Published language
    d. Conformist
3. **Which geometrical figure represents a bounded context in a context map?**
    a. A rectangle
    b. An ellipse
    c. Both
    d. Does not matter—it only needs to be understandable
4. **What is comparable with a synchronous communication in your day-to-day live?**
    a. A TV show

   b. A podcast
   c. A phone call
   d. An advertisement

5. **What is comparable with an asynchronous communication in your day-to-day live?**
   a. A TV show
   b. A podcast
   c. A phone call
   d. An advertisement

6. **Which part of communication follows the metaphor of west-east-communication?**
   a. User interface to service communication
   b. Service-to-service communication
   c. Client to cloud communication
   d. Cross-cloud communication

Answers

1. **a**
2. **b**
3. **d**
4. **c**
5. **d**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 8

Overview of Strategic Design

Introduction

Strategic design is a great tool to define the service structure of applications. It helps to understand the business better and to establish a ubiquitous language for the implementation teams. It needs to be implemented in a sustainable way if we want to reap all the benefits of it. This chapter explains why strategic design is important and gives tips how to introduce it to a company.

Structure

In this chapter, we will cover the following topics:

- Overview of strategic design
- Significance of strategic design
- Relationship of bounded context and team structure
- Tips and tricks for applying strategic design

Objectives

This chapter covers the importance of strategic design in an enterprise and in projects.

It explains the tight coupling of software architecture and organizational structure. Moreover, you will get tips and tricks how to introduce **Domain-Driven Design** (**DDD**) in your organization and how to establish a corresponding governance process.

> **You have to be fast on your feet and adaptive or else a strategy is useless.**
>
> *-Charles de Gaulle*

Overview of strategic design

When examining the strategic design of DDD, it is clear that the design process is highly collaborative. *Figure 8.1* provides an overview of the methodologies involved in strategic design:

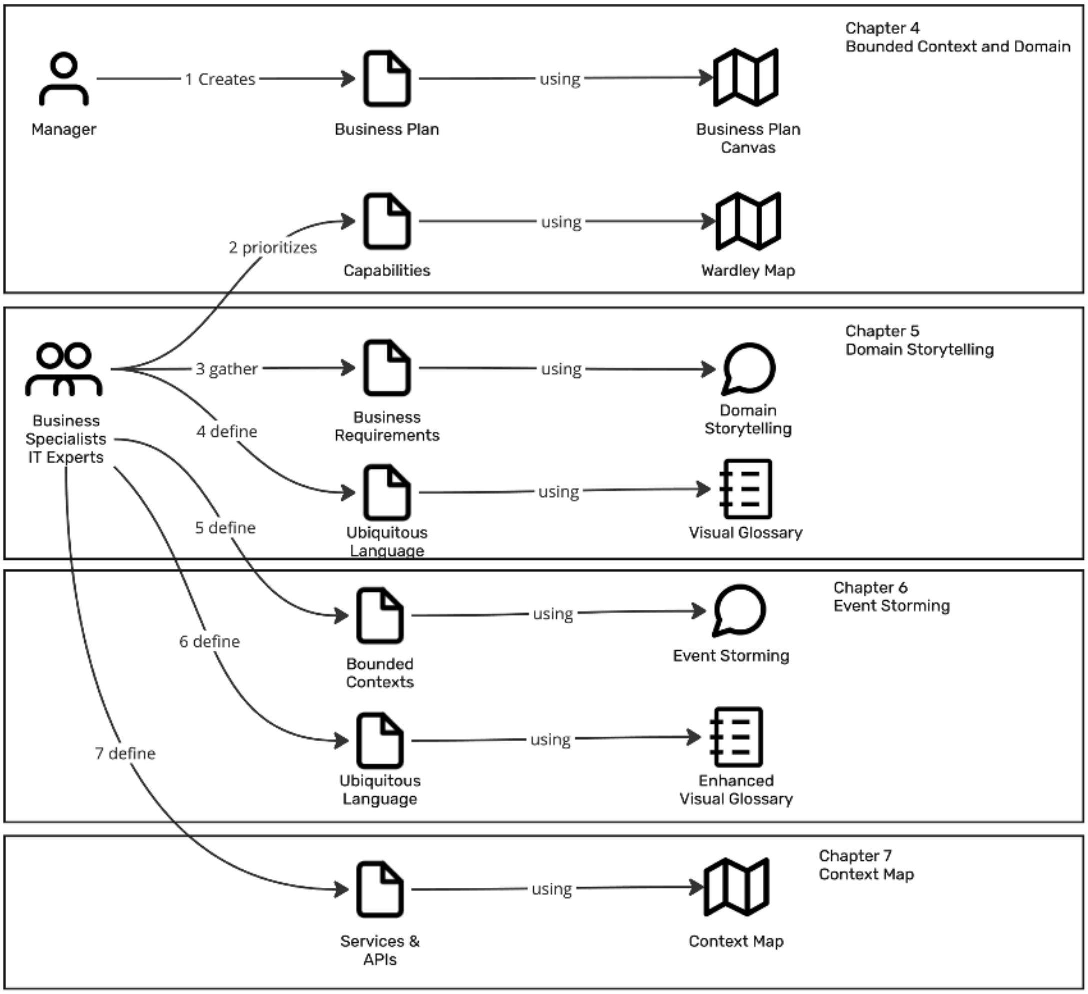


***Figure 8.1**: Overview about the strategic part of DDD*

***Source**: OverviewStrategicDesign.jpg*

Let us have a deeper look into the steps of the process shown in *Figure 8.1:*

1. A manager creates a business plan using Business Plan Canvas[1], as introduced in *Chapter 2, Introduction to the Example Online Library*.

   The manager defines the important business capabilities and the key partners and resources using a Business Plan Canvas. They define the cost structure and the revenue streams.

2. Business specialists and IT experts together prioritize the capabilities defined in the Business Plan Canvas using a Wardley Map[2] as described in *Chapter 4, Bounded Context and Domain*.

   Experts and specialists categorize discovered business capabilities based on their significance to key users—such as library readers—considering their specific needs. They identify the necessary components and determine their visibility for the users. More crucially, they measure the market maturity of these components.

   For components at the forefront of technology, they chart their progression from genesis to custom, which requires in-house implementation. They then oversee the transition to a market-available product, which can be customized as needed, and finally to a commodity that can be purchased ready-to-use. This comprehensive evaluation is pivotal in ensuring an efficient delivery of service that effectively meets user demands.

3. Business specialists and IT experts gather business requirements using the workshop format of domain storytelling[3], as described in *Chapter 5, Domain Storytelling*.

   In a domain storytelling workshop, a team of experts and specialists collaboratively map out the user journey using visually expressive language. The selected grammar of this language evolves around an actor performing actions on a work item, making the process easy to comprehend. However, this grammar mainly serves as a tool to facilitate the understanding of a business specialist's perspective and is not restrictive. If a different grammar makes the visual story more comprehensible, it can certainly be used, emphasizing the flexibility of this process.

4. Business specialists and IT experts define the ubiquitous language using the technique of a visual glossary[4] as described as well in *Chapter 5*.

---

[1] Strategyzer. (2023). The Business Model Canvas. Retrieved January 2024, from Strategyzer: **https://www.strategyzer.com/library/the-business-model-canvas**

[2] Wardley, S. (2022). Wardley Maps. Simon Wardley.

[3] Hofer, S., & Schwentner, Henning. (2021). Domain Storytelling: A Collaborative, Visual, and Agile Way to Build Domain-Driven Software. Pearson International.

[4] Zörner, S. (2015). Softwarearchitekturen dokumentieren und kommunizieren, Entwürfe, Entscheidungen und Lösungen nachvollziehbar und wirkungsvoll festhalten.

Defining a ubiquitous language is a key task in the entire design process. It not only bridges the understanding between business specialists and IT experts but also proves instrumental in marketing the functionality to external stakeholders via APIs. This language penetrates all layers of development, from the implementation of services and clients to the collation of business requirements.

A visual glossary is a valuable tool for creating and documenting this language. It is characterized by specific terminology and the relationships between those terms using distinct, strong verbs. This precise linguistic framework is crucial for effective communication and comprehension across all project stages and roles.

5. Business specialists and IT experts define the bounded contexts of the application using the workshop format event storming[5] as described in *Chapter 6, Event Storming*.

   Event storming is an excellent technique for delineating the bounded contexts within a business process. It highlights the key event when a business object typically changes its status in a meaningful way—for instance, when an anonymous user becomes a member of a library.

   Throughout this business process, varying aggregates are used, and different people in various roles or separate systems interact. These shifts help shape the borders of the bounded contexts, enabling a better understanding of the process flow.

6. Business specialists and IT experts enhance the ubiquitous language as described in *Chapter 6* as well.

   During event storming workshops, the pre-established ubiquitous language can be further refined. For example, a book can be described in more depth with attributes such as the title, author, and editor. These attributes can then be implemented across APIs, object models, and database structures, enhancing their understandability.

7. Business specialists and IT experts define the service structure of the application and the necessary APIs using the technique of a context map[6] as described in *Chapter 7, Context Map*.

   A context map illustrates the interdependencies among teams working on separate bounded contexts, with usually one team assigned to one bounded context. Teams that implement open host services can execute their work with considerable independence. They communicate externally via a published language rooted in the already defined and documented ubiquitous language.

---

[5] Brandolini, A. (2023). EventStorming. Retrieved 04 2024, from EventStorming: **https://www.eventstorming.com/**

> This language is distributed through APIs, which can be implemented either as asynchronous or synchronous, offering flexibility based on the needs of the operation. This enhances the process of distinguishing and understanding the interactions and relationships among various contexts and teams.

At first glance, the entire process may seem overly complex and costly. However, it is important to remember that this process is built on workshop formats that unite business experts and IT specialists. In more siloed approaches, misunderstandings and misinterpretations often occur, leading to unsuccessful and costly projects. Therefore, strategic design is of prime importance. The essence of its value lies in preemptively addressing potential communication gaps, thereby enhancing project success and cost efficiency.

Moreover, the entire process can be done step by step for certain capabilities or capability groups prioritized in Wardley Maps and relies on feedback loops, as shown in *Figure 8.2*:

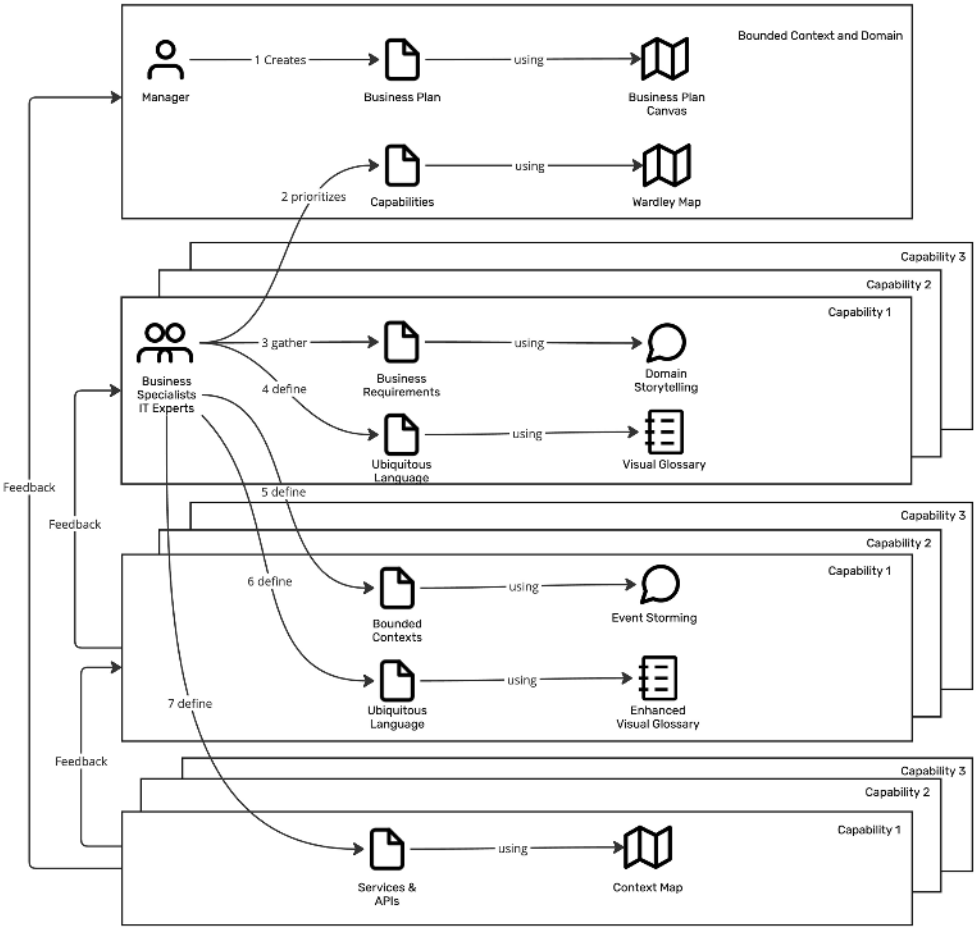


***Figure 8.2**: Strategic design with feedback loops*

Source

Such an iterative process can be applied to the creation of new products as well as to the re-engineering of existing application environments. It is highly important to conduct, as we will see in the next section.

Significance of strategic design

Strategic design is relevant for development because it defines the structure of not only one application but also the structure of several applications and how they work together.

Development over the years

When examining an application environment in recent years, we encounter a diverse range of applications. These can range from modern microservice architectures utilizing artificial intelligence to monolithic legacy applications built with, for example **Programming Language One** (**PL/I**)[7].

Such heterogeneous application environments are typical in today's digital landscape. However, these systems need to undergo modernization that aligns with the business needs of the respective company. Both the technology and its business implementation in software must progress simultaneously. It is crucial to ensure that business operations are not stopped until the software has been modernized and brought up to contemporary standards.

Therefore, we need to prioritize our work. We need to determine where modernization is important and valuable. Strategic design gives the answer to those questions.

An according process model could look like the following steps:

1. Analyze current business using domain storytelling
2. Determine bounded contexts
3. Analyze market maturity using Wardley Map, define commodities and modernization areas
4. Analyze the modernization areas using event storming
5. Create a context map of the new landscape
6. Define APIs

Using such a process allows the evolution of the landscape step by step without interrupting the business. An example of a modernization project will be given in *Chapter 14, Usage of DDD in the Brownfield*.

In such a way, the strategy is set for the next couple of years. Strategic design needs to be established as a continuous process to support an evolving architecture.

However, strategic design is not only important for custom development but also for purchasing products as well.

Development and purchase

Prioritizing the work does not only define the implementation work in the company. It defines the necessary purchases of software as well. When capabilities are seen as a product or commodity, corresponding software needs to be purchased.

The software needs to be weighted by pure technical properties like the following:

- It is available as managed software in the cloud.
- It provides APIs to access the functionalities of other applications.
- It follows standards.

However, the software needs to be weighted by more business perspective besides the functionality provided, such as:

- Whether the size of the company corresponds to the purchasing company
- If support in the local language is guaranteed
- If support can be guaranteed outside of business times
- Whether the software supports barrier-free access

Clearly, the relevant characteristics heavily rely on the nature of the business and the unique features of the respective company. For instance, an insurance company's requirements will differ from those of a gaming or digital health company.

Strategic design is instrumental in identifying the correct inquiries for software acquisitions. Techniques like Wardley Maps and domain storytelling aid in understanding not only the business requirements for the software but also the technical integration prerequisites for the corresponding landscape. By involving these approaches, businesses can ensure a smoother transition and integration of new software into their existing framework.

Finding the structure of applications is one perspective of strategic design, to find the correct team setting is the other perspective.

Define the structure of development

Business Plan Canvases and Wardley Maps can help a company identify its business capabilities. Tools like domain storytelling and event storming facilitate the definition of the bounded context, while a context map visualizes the necessary data exchange and team dependencies.

The bounded context does not just shape the service structure of an application landscape; it also guides the configuration of team structures. Considering *Conway's law*[8], the team structure should be congruent with the intended service structure. Thus, the strategic design not only involves outlining the service structure—with each service implementing a separate bounded context—but also sets a corresponding team structure, wherein one team is responsible for one service.

This methodology means that the cross-functional teams executing a strategic design are not only shaping the software architecture; they are also constructing team structures. These teams should, therefore, be empowered to initiate a changing of team structures in a way that aligns with the intended architecture. This is especially necessary for re-engineering of existing application landscapes.

The relationship of team structures and bounded contexts is discussed in the next section.

Relationship of bounded context and team structure

Teams that work in an application environment depend on each other. They need to deliver results so that other teams can go further in the implementation. We can differentiate between different types of teams, as we will discuss them in the next sections.

Open host service

Teams implementing open host services[9], as discussed in *Chapter 4, Bounded Context and Domain*, operate as independently from other teams as possible. They can implement their features without being reliant on the progress of other teams. They use APIs and a published language, which aligns with their ubiquitous language, to make the functionality of their service accessible.

These open host service teams adhere to the concept of stream-aligned teams in team topologies[10], further emphasizing their independence and efficiency. This extends their capability to progress seamlessly without external interdependencies.

Anti-corruption layer

An anti-corruption layer[11], as discussed in *Chapter 4, Bounded Context and Domain*, becomes necessary when there is a need to segregate legacy software from modern implementations within an application landscape. This layer ensures that the models of the newly designed services remain uncorrupted by legacy models. The legacy services exist behind a façade,

---

[8] Conway, M. E. (1968, April). How do committees invent? Datamation magazine.

[9] Evans, E. (2003). DDD: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley.

[10] Skeleton, M., & Pais, M. (2019). Team Topologies. Portland, OR: IT Revolution.

enabling the modern services to utilize a well-architected model while the legacy model stays concealed behind the façade.

A platform team[12], which supports the steady aligned teams, can provide this anti-corruption layer. This allows teams to maintain the integrity of their modern services while seamlessly interacting with legacy structures.

Conformist

A team that adopts the definitions from another team without modifications is referred to as a conformist[13]. This team integrates another team's model in its entirety. This approach is commonly executed by stream aligned teams that embrace standardized models, such as OpenID Connect[14] for authentication purposes, ensuring consistency and interoperability across different parts of the organization.

Big Ball of Mud

A *Big Ball of Mud*[15] arises when the models of individual bounded contexts become entangled, leading to interdependencies among implementing teams. This entanglement greatly increases implementation times due to the need for synchronization and the associated delays. To prevent this chaotic situation, it is essential to apply DDD principles within an organization.

These principles encourage the definition of clear team structures and the separation of models, enhancing team autonomy and minimizing dependencies. The necessary structures and separations to avoid a big ball of mud are outlined in *Figure 8.3*:

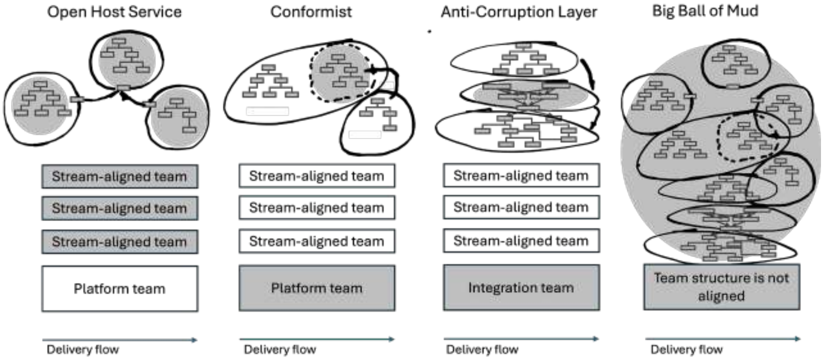


Figure 8.3*: Teams and structures*
Source*: TeamsAndStructures.png*

---

[12] Skeleton, M., & Pais, M. (2019). Team Topologies. Portland, OR: IT Revolution.

[13] Evans, E. (2003). DDD: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley.

[14] OpenID Foundation. (2024). What is OpenID Connect. Retrieved June 2024, from openid: What is OpenID Connect

[15] Baas, K., Lindhard, K., Plöd, M., & Sanglan-Charlier, M. (2024, May 20). The Bounded Context Canvas. Retrieved May 2024, from GitHub: **https://github.com/ddd-crew/bounded-context-canvas/tree/master**

The next section discusses how DDD can be applied in an organization.

Tips and tricks for applying strategic design

In the following sections, some points are discussed on how the strategic part of DDD can be implemented in an organization. We will discuss how to define the business capabilities, how to train the teams, how to implement a consequent API-first strategy, and how to govern the process. Additionally, we will discuss how to build up a supporting team structure to follow the principles of Conway's law.

Definition of business capabilities

The first stage in the strategic aspect of DDD involves delineating the business capabilities of a company. Such capabilities are usually defined at least indirectly—expressed as either business functions or fields. However, they are often not usable in their current state to be mapped to subdomains or enclosed contexts. That said, they make a solid starting point.

These defined capabilities serve as a useful tool during discussions with management and business experts, helping you outline the specific capabilities. Using these, bounded contexts can initially be established, correlating one-to-one with the outlined business capabilities.

Further, tools like Wardley Maps (Wardley, 2022) can be employed to prioritize the discovered capabilities. Once prioritized, the bounded contexts can be detailed more systematically using methods such as domain storytelling and event storming.

Training for the teams

For successful implementation of DDD, it is imperative to educate both the business and application teams. While external training can be used, it is more effective to conduct internal sessions using real-life examples specific to the company's challenges.

These sessions should cover fundamental concepts such as domain and subdomain, as well as bounded context and aggregate. More advanced training should include the application of context maps for technically focused attendees and the use of Wardley Maps for participants with a business-centric perspective.

API-first strategy

For practical application of DDD within a company, individual development teams must function independently. Data can be shared between teams through APIs, keeping the implementation hidden. To uphold this independence, APIs should be defined at the earliest stage possible during the implementation process, an approach known as API-first strategy.

The teams collaboratively define these APIs using the context map. This collaboration can determine the nature of the API—whether it be synchronous or asynchronous, as well as the types of data that need to be exchanged. Based on these carefully defined parameters, the teams can then independently work on their particular bounded contexts.

Governance process

Results of workshops and corresponding architectural decisions need to be documented. The documentation can be exported and be part of team repositories. Later, decisions and the technical design needs to be aligned to the strategic design documented. Obviously the governance process needs to be establish as a iterative process, where the decisions are aligned with the current business requirements and are reviewed constantly.

The according process and the belonging artifacts are shown in *Figure 8.4*:

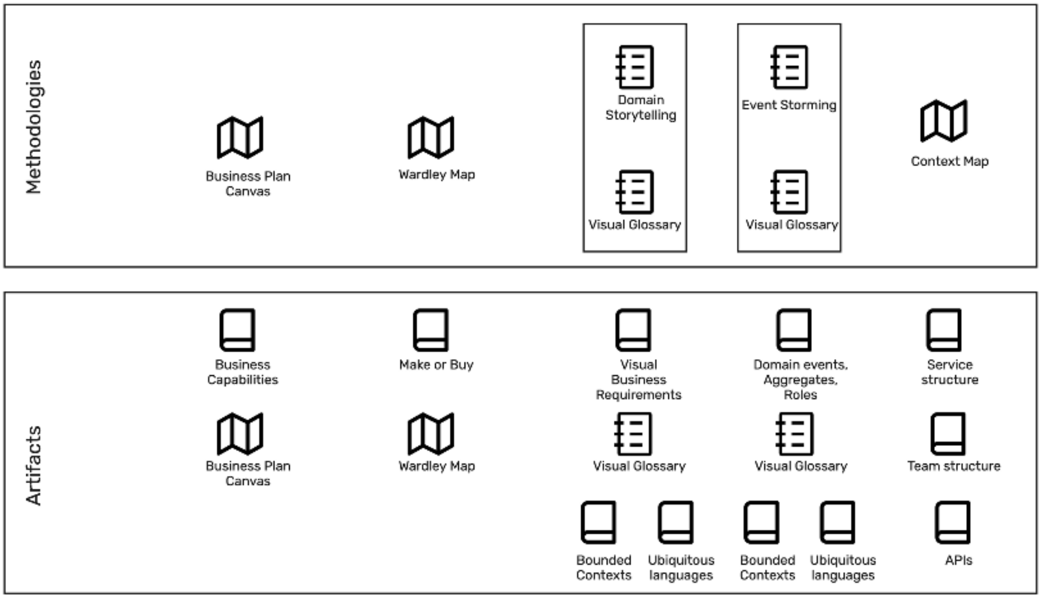


***Figure 8.4**: Governance of architectural decisions*

***Source**: Decisions.jpg*

Which artifacts are created using which methodology is shown in *Table 8.1*. Additionally, the predecessor steps and successor steps of the design flow are listed:

Methodology	Artifacts	Form	Predecessor	Successor	Note
Business Plan Canvas[16]	Business capabilities	Table with explanations		Wardley Map Domain storytelling	If the make or buy decision is clear, it can be followed directly by the domain storytelling.
	Business Plan Canvas	Exported image			
Wardley Map[17]	Make or buy decisions	Table with explanations	Business Plan Canvas	Domain storytelling Event storming	A Wardley Map can be done even after the domain storytelling in case the business requirements need to be better understood. Event storming can follow directly in case the user flows are clear.
	Wardley Map	Exported image, Table			
Domain storytelling[18] and visual glossary[19]	Visual business requirements	Exported picture and table	Business Plan Canvas Wardley Map	Event storming Context map	Can be left out when business requirements are clear. Event storming can be left out when domain events and aggregates are clear.
	Bounded contexts	Table			
	Visual glossary and ubiquitous language	Exported picture and table			

---

[16] Strategyzer. (2023). The Business Model Canvas. Retrieved January 2024, from Strategyzer: https://www.strategyzer.com/library/the-business-model-canvas

[17] Wardley, S. (2022). Wardley Maps. Simon Wardley.

[18] Hofer, S., & Schwentner, Henning. (2021). Domain Storytelling: A Collaborative, Visual, and Agile Way to Build Domain-Driven Software. Pearson International.

[19] Zörner, S. (2015). Softwarearchitekturen dokumentieren und kommunizieren, Entwürfe, Entschei-

Methodology	Artifacts	Form	Predecessor	Successor	Note
Event storming[20] and visual glossary[21]	Domain events, aggregates, and roles	Exported picture and table	Business Plan Canvas Wardley Map Domain storytelling	Context map	Can be left out when domain events and aggregates are clear
	Visual glossary and ubiquitous language	Exported picture and table			
	Bounded context	Table			
Context map[22]	Service structure	Exported picture	Domain storytelling Event storming		Definition out of bounded contexts either out of domain storytelling or event storming
	Team structure	Table			
	APIs	Exported picture			

Table 8.1*: Overview of methodologies for strategic design and the artifacts produced*

Implementing a governance process that checks the delivered artifacts and the belonging steps to check decisions made allows a sustainable and agile DDD process.

Team structure

Using the service structure defined in a context map, the team structure needs to be defined. The teams should work as open host service teams as independently as possible.

The teams develop the belonging business capability inside of their bounded context. They use tactical DDD which will be discussed in the next part of this book.

Conclusion

This chapter gives an overview of the strategic part of DDD. It shows the process of designing a software application, from defining business capabilities through gathering business requirements using domain storytelling to creating the application and APIs

---

[20] Brandolini, A. (2023). EventStorming. Retrieved 04 2024, from EventStorming: https://www.eventstorming.com/

[21] Zörner, S. (2015). Softwarearchitekturen dokumentieren und kommunizieren, Entwürfe, Entscheidungen und Lösungen nachvollziehbar und wirkungsvoll festhalten. München: Carl Hanser Verlag.

using event storming and context mapping. It explains which step of the overall process and which artifacts need to be produced using workshops and an overall collaborative approach.

It shows how a strategic design can be implemented in an organization and gives hints on how to apply it successfully.

In the next chapter, we will introduce the tactical part of DDD. We will discuss the advantages of a model-driven approach. Furthermore, we will discuss the design patterns introduced by DDD.

Points to remember

- Strategic design is important to find structures for newly built applications or to improve the structure of existing applications.
- Strategic design needs to be established as a continuous process to support an evolving architecture.
- Strategic design helps to make the decision between make or buy.
- Strategic design helps to align the organizational structure and the technical design.
- Workshop formats support the collaborative design and make the design sustainable.
- Workshop formats even help to establish an agile architectural style.
- DDD needs training to be introduced in a company.
- To establish DDD, it must be integrated into the development process and follow the established governance processes.

Multiple choice questions

1. **Can strategic design only be applied to new applications?**
    a. Yes
    b. No
    c. Depends on the business
    d. Depends on the local regulations

2. **Does strategic design help make or buy decisions?**
    a. No
    b. Only if the management wants to

   c. Only for certain business
   d. Yes

3. **Is strategic design combined with the team structure of the implementing teams?**
   a. No
   b. Only for insurance companies
   c. Yes
   d. Only for international companies with more than 5,000 employees

4. **Does strategic design need a governance?**
   a. No
   b. Only if the pictures are done physically
   c. Only if the pictures are done digitally
   d. Yes

Answers

1. **b**
2. **d**
3. **c**
4. **d**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 9

Introduction to Tactical Design

Introduction

This chapter will discuss tactical design and the other side of **Domain-Driven Design** (**DDD**). Developers should apply the principles in their daily professional lives. However, business analysts and product owners should also understand the principles and concepts of tactical design to discuss the designs with developers.

Structure

In this chapter, we will discuss the following topics:

- Tactical design
- Tactical design for modeling
- Model-driven design
- Further development of the ubiquitous language
- Patterns provided by DDD

Objectives

You will learn which pattern you can use in tactical DDD. You learn which patterns are provided, how they fit in the overall picture of DDD, and how they depend on each other.

> **I usually solve problems by letting them devour me.**
>
> *-Franz Kafka*

Tactical design

The tactical design approach in domain-driven development focuses on building a software application within a pre-determined bounded context. *Chapters 3* through *8* previously comprehensively explored the strategic design concept, which guides the identification of ideal bounded contexts and aids in customizing the alignment of corresponding services.

Now, our focus shifts towards a deeper exploration of designing objects within a bounded context and their subsequent publication. This involves a concentrated effort on modeling within a specific bounded context. A model is essential to representing business requirements within a bounded context and logically aligning services.

Tactical design for modeling

Twenty years ago, when *Evans* wrote his famous work, the world of software development was quite different from today. Architects created large models printed on walls without writing any line of code. Those developed models were given to the developers in another department for implementation. Even though the models were correct, they were not implementable. Quickly, the developers changed dependencies and associations to create storable and retrievable objects.

On the other hand, a model that does not reflect the business is of no use and no value. A model needs to reflect the domain it is intended for and be implementable. A model that tries to reflect the entire world of an enterprise cannot be implemented because all those dependencies and associations block completely any meaningful database or object design. Therefore, the model needs to reflect the bounded context as a functional unit for which it is valid.

Today, we use agile principles and corresponding frameworks such as Scrum[1] and Kanban[2]. As we saw in *Chapter 4, Bounded Context and Domain*, teams are organized around bounded

---

[1] Welcome to the home of Scrum. (2024). Retrieved August 2024, from Scrum.org The home of Scrum: **https://www.scrum.org/**

[2] Agile glossary Kanban. (2024). Retrieved August 2024, from Agile Alliance: **https://www.agilealliance.org/glossary/kanban/**

contexts. Such a large mode upfront is no longer suitable and does not work. Based on *Evans's* results, teams can develop their models inside a bounded context and establish their ubiquitous language belonging to it.

If a model is to be valid for a bounded context, ubiquitous language must be used. As we have seen before, ubiquitous language can be developed in workshops between business experts and IT specialists. However, the implementing teams need to build a model in a model-driven design.

Model-driven design

Even in 2003, when *Evans* published his famous book, large upfront models were not helpful. With the development of agile methodologies, the **Big Model Upfront** (**BMUF**) became an anti-pattern[3]. Therefore, we initially envision the business requirements using domain storytelling and the architecture using event storming.

Evans defined model-driven design as valuable when it covers a reduced part of the problem—the bounded context. The chapters covering strategic design showed how we can split the problem—the domain—into digestible parts. Anyhow, a bounded context needs some design itself. Let us look at the parts *Evans* described as necessary in such a model-driven design.

Note: Model-driven design is not an iconic term. It is used in several contexts. For example, the Open Management Group uses model-driven architecture as part of their framework. It means developing models that can be directly used for generating code. That is not what we mean here[4]. We want to point out how to architect a bounded context in a team without too much modeling upfront.

Today, self-responsible teams must define whether they want **Unified Modeling Language** (**UML**)[5] models or a C4 model[6] to describe their bounded context. However, the model and the code must reflect the business.

Evans already described the essential parts of such team discussions in tactical design, as shown in *Figure 9.1*. The author adapted the parts to current methods in software architecture. The changed parts are marked in grey.

---

[3] Ambler, S. (2022). Big modeling up front (BMUF) anti-pattern. Retrieved August 2024, from Agile modeling: **https://agilemodeling.com/essays/bmuf.htm**

[4] MDA - The architecture of choice for a changing world. (2024). Retrieved August 2024, from OMG: **https://www.omg.org/mda/**

[5] UML 25. (2024). Retrieved August 2024, from UML: **https://www.uml.org/index.html**

[6] Brown, S. (2024). The C4 model for visualizing software architecture. Retrieved August 2024, from C4 Model: **https://c4model.com/**

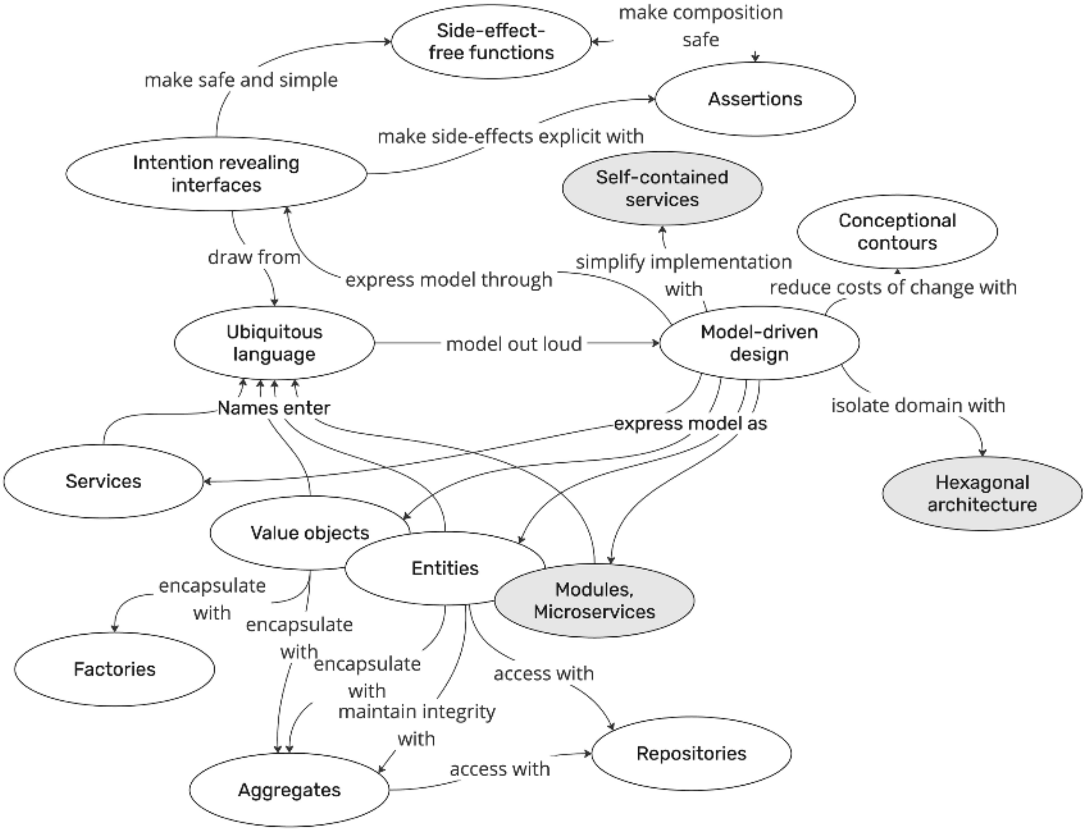


***Figure 9.1**: Parts of tactical design by Evans adapted by the author*

***Source**: ModelingParts.jpg*

Let us explore the parts of modeling further:

- **Side-effect-free function**: When one calls a function, one expects a particular behavior indicated by the function's name and probably even by specific standards. So, changing data when one calls a get in **Hypertext Transport Protocol** (**HTTP**) is a lousy style. We will dive deeper into this topic when we discuss how to expose **application programming interfaces** (**APIs**).

  Side-effect-free functions are necessary to make intention-revealing interfaces possible.

  The function contains specific assertions, such as HTTP verbs, that allow it to make compositions safe. When there are side effects, those assertions make them explicit.

- **Assertions**: Assertions can be used to ensure the quality of compositions. They make the side effects of functions explicit[7]. For example, an assertion could be that changes can only be made to an object's entire state. Therefore, only put is provided in a web interface. The HTTP verb patch is not offered.

- **Intention-revealing interfaces**: Interfaces should always show what they are intended for. To do so, ubiquitous language must be used, including where they are drawn from. Moreover, an interface providing a read function for a particular aggregate must not return other information. An interface defined to change one aggregate must not change another aggregate if it is only referenced.

  Intention-revealing interfaces are expressed by a model-driven design.

- **Self-contained services**: Self-contained services are services that can live for themselves. They can fulfill a business function and can be scaled easily. In *Evans'*[8] origin, it was a single class, and even though a self-contained service is much more than just a class, it has a comparable meaning. Self-contained services allow simplifying the model implementation. A self-contained service does not need a BUMF. The team implementing it can model and architect it.

- **Conceptional contours**: Conceptional contours are defined by model-driven design. Aggregates, entities, value objects, and other patterns express them. We will discuss them in *Chapter 10, Aggerate, Entity, and Value Object*.

- **Ubiquitous language**: Ubiquitous language is the cornerstone of the model. Without expressive language, an expressive model and, with it, a successful implementation is not possible.

  It is used to create intention-revealing interfaces. The ubiquitous language expresses the model out loud. A model without expressive language is not possible. Names throughout the entire implementation, from modules and packages down to value objects, find their way into the ubiquitous language and need to be understood by IT specialists and business experts.

- **Model-driven design**: Model-driven design covers tactical design inside a team responsible for a bounded context. The team needs to develop the ubiquitous language further and apply it to all design parts. The model is expressed in services, entities, aggregates, value objects, and microservices.

- **Services**: Services are usually part of the strategic design, and a team is responsible for a logical service covering a business function like *lending*. However, a business function might need to be split because of quality requirements like performance. In such a case, the ubiquitous language must also be found in the service names, as we already saw it naming the logical services in the event storming part.

- **Value objects**: Value objects encapsulate non-referenceable parts of an aggregate or an entity. We will discuss them in more detail in the next chapter. They need to be encapsulated by aggregates and factories.

- **Entities**: Entities are referenceable objects encapsulated by an aggregate and a factory. Aggregates maintain their integrity. We will discuss them in more detail in the next chapter.
- **Modules and microservices**: Modules and microservices structure a model. Modules can be implemented, e.g., as packages, whereas microservices are helpful when a platform team provides infrastructure components. Those microservices are usually implemented as side cars so that the steady-delivery team does not need to take care of them. The team can use corresponding standard images delivered by the platform team. However, a team must carefully discuss how to structure the logical service and when to use microservice[9] or strategic monolith[10] patterns.
- **Hexagonal architecture**: A hexagonal architecture can isolate the domain model[11]. That architecture approach states that the model is the core of a service. The hexagonal architecture defines input and output ports, which sentinel the domain model of the service by adapters. The business logic surrounds the model. The domain model is the core of the hexagon and is secured. The principle is shown in *Figure 9.2*:

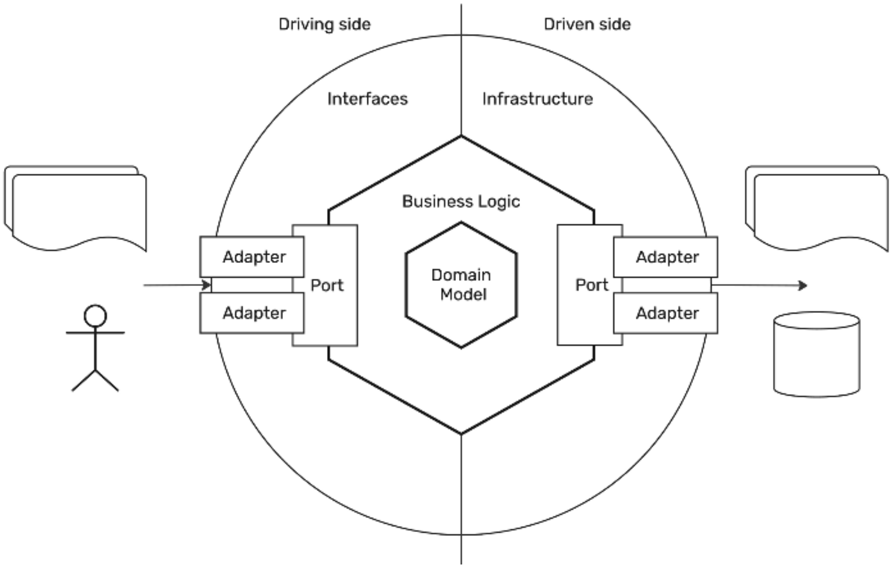


***Figure 9.2**: Model of a hexagonal architecture*

***Source**: HexagonalArchitecture.jpg*

---

[9] Richardson, C. (2019). Microservices Patterns. Shelter Island: Manning.

[10] Vernon, V. (2022). Strategic monoliths and microservices: Driving innovation using purposeful architecture. Addison-Wesley Publishing.

[11] Cockburn, A., & Garrido de Paz, J. (2024). Hexagonal Architecture Explained. Human and Technology Inc.

Note: Hexagonal architecture is a significant topic worth a separate book. We mentioned it here because it helps isolate and secure the model.

For Evans, the isolation was taken over by the typical layered architecture pattern, which was usual in the early 2000s. We changed it to hexagonal architecture because the layer model led to poorly structured monoliths. Patterns like hexagonal architecture are better suited for flexible and maintainable architectures.

- **Factories**: Factories encapsulate entities and aggregates. We will discuss them in greater detail in *Chapter 10, Aggerate, Entity, and Value Object*.
- **Aggregates**: Aggregates are the cornerstone of a bounded context. They encapsulate entities and value objects. They represent the objects of a bounded context. We will discuss them in detail in the next chapter.
- **Repositories**: The repository pattern allows access to the domain model representation in a storage. It ensures that specific database structures do not spoil the domain model of the bounded context. We will discuss this in more detail in the next chapter.

As we saw in tactical design, we must combine ubiquitous language with an implementable model. As discussed in the next section, the ubiquitous language needs to be developed further.

Further development of the ubiquitous language

Using domain storytelling (as discussed in *Chapter 5*) and event storming (detailed in *Chapter 6*), we gained in-depth knowledge of the problem area. This understanding was established and shared between business experts and IT professionals. Remarkably, we also forged a common language between these two sectors. This ubiquitous language of the bounded contexts was established using a visual glossary, further bridging communication gaps.

Our next step involves creating a model for each bounded context. The model will be a shared reference for business experts and IT professionals. This process is known as **model-driven design**.

Following the actual value of DDD, model-driven design is also collaborative. Whereas strategic design focuses on the business process and business capabilities, tactical design focuses on the bounded context. Creating the model will enhance the already established language, making it more detailed, but it will not be changed completely. To model successfully, business experts and IT specialists must work together.

There are best practices to evolve the ubiquitous language during the first implementation of a service and its maintenance period based on *Millet* and *Tune*[12] and the author's experience. Those best practices for shaping the ubiquitous language are listed as follows:

- **Develop the visual glossary**: The visual glossary created in the domain storytelling and event storming workshops needs to be evolved and maintained. The visual glossary is valid for one bounded context. Other models from other bounded contexts should not conquer the model. The language needs to be clean and precise. The collaboration of business experts and IT specialists helps to make the ubiquitous language stable and understandable.
- **Ensure consistency**: When using a term in code that is not part of the language of the business experts, the term needs to be checked with them. Either something is missing in the visual glossary, and the term needs to be added, or the term is simply wrong in this bounded context, and another term needs to be used.
- **Avoid weak words**: Weak words are less expressive than more specific words. Words like service, policy, or manager are unspecific. Library catalog management is more wordy but even more expressive and, insofar, better to understand. A user can be anyone and is less expressive than a library member, which better expresses a person's situation using a library application.
- **Avoid words with a different meaning in the domain**: Names like *adapter, factory*, or *monolith* refer to specific software architecture patterns. To use them in ubiquitous language would mean to use words outside of their standard context. Developers understand different things, and they are probably business experts. Therefore, names like *voltage adapter, chip factory*, or *silicon monocrystal* are more expressive and better understood by business experts and IT specialists.
- **Discuss code with business experts**: Engaging business experts in code discussions may initially seem unproductive. However, it is beneficial. Business experts can identify the ubiquitous language used and understand how their concepts are mirrored in the code. This is particularly important for reviewing API definitions collaboratively with business experts and IT specialists.
- **Use the language everywhere**: The ubiquitous language should be consistently visible throughout namespaces, classes, properties, database tables, etc. Consistently using the ubiquitous language inside the bounded context helps recognize the impact of changes and facilitates the quick onboarding of new team members.
- **Evolve the language**: The ubiquitous language will evolve throughout the development of the bounded contexts. Review your names and refactor when necessary. The business grows, and in the same manner, your language will evolve and become more precise.

Besides the ubiquitous language, DDD was first seen only as a collection of tactical patterns. As we saw in the chapters around strategic design in the first part of this book, DDD is collaboration and communication.

Patterns provided by DDD

Tactical patterns were first introduced by *Evans*[13]. They were later sharpened and rendered by *Millet and Tune*[14]. The according patterns enhanced by the author are shown in the following *Figure 9.3*:

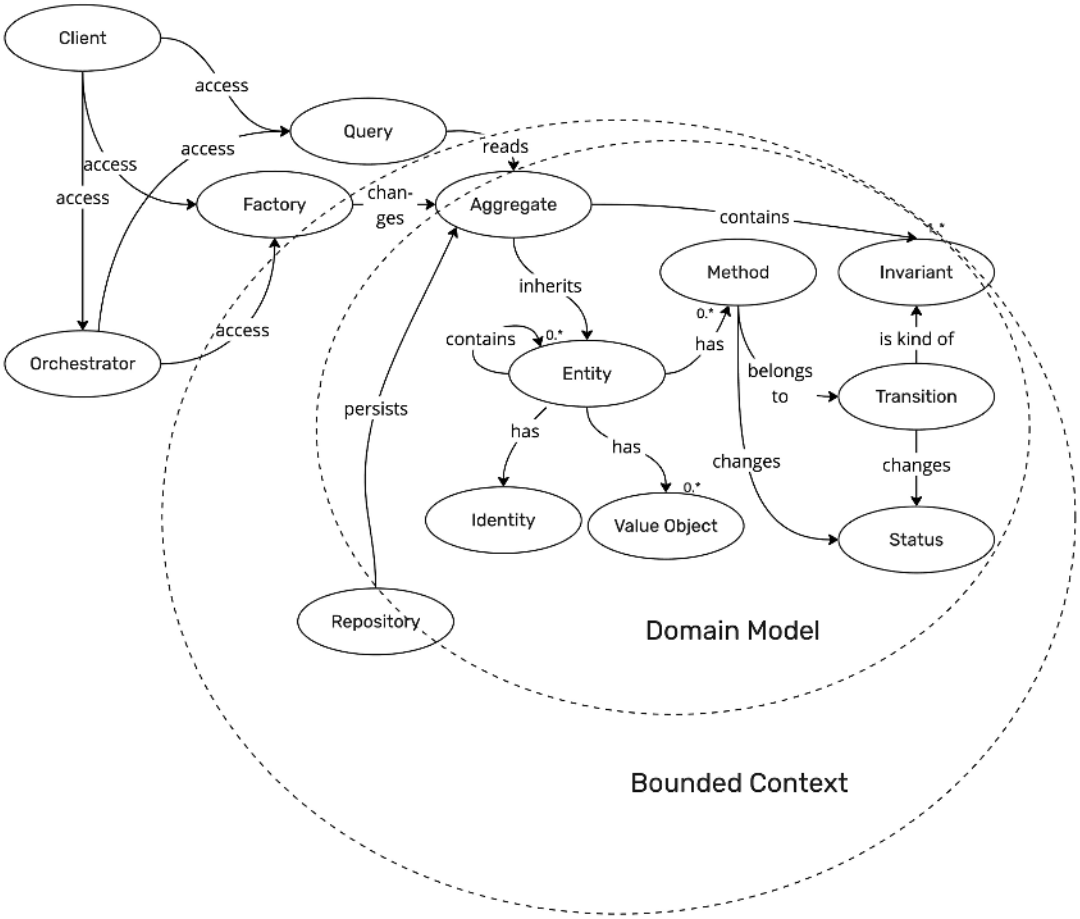


***Figure 9.3**: Pattern used for tactical DDD*

***Source**: PatternsInDdd.png*

---

[13] Evans, E. (2003). DDD: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley.

The tactical patterns *Aggregate*, *Entity*, and *Value Object* belong to the domain model of a bounded context. A repository pattern persists the data of the domain model in a database. The patterns Orchestrator and Client are used outside of the bounded context. Query and Factory patterns publish the language of the bounded context. All those patterns will be discussed in the next chapter.

Conclusion

Tactical design allows a team responsible for a bounded context to model their solution inside the context. They can design corresponding aggregates and decide how to structure modules or services.

A deep understanding of the business requirements is necessary to successfully model the bounded context using modern design patterns like hexagonal architecture and DDD patterns.

The next chapter will discuss designing a bounded context and encapsulating the domain.

Points to remember

- Tactical design provides a couple of patterns that can be used to model a bounded context.
- Patterns introduced by DDD are aggregates, entities, value objects, factories, and repositories.
- Queries, clients, and orchestrations can enhance the classic patterns.
- Today, hexagonal architectures isolate the domain model from external influences.

Multiple choice questions

1. **Which pattern used in the DDD most essentially represents a bounded context?**
    a. An entity
    b. An aggregate
    c. A value object
    d. A repository

2. **How can tactical design benefit from a model?**
    a. A model can be implemented by the team implementing the bounded context.
    b. The model can be developed upfront by architecture teams.

   c. The model allows governance over all implementing teams.
   d. The model expresses exclusively the business requirements.

3. **Is it sensible to create a big upfront model for an entire application?**
   a. Yes, because all involved can see where they need to implement the model
   b. Yes, because a project manager can plan the dependencies
   c. No, because the interdependencies need to be handled between the teams by interfaces defined exclusively by technicians
   d. No, because the team implementing a bounded context is responsible for the tactical design

Answers

1. **b**
2. **a**
3. **d**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 10

Aggregate, Entity, and Value Object

Introduction

This chapter discusses the patterns of Domain-Driven Design. Patterns help us understand the underlying structure of code and design successful, well-structured code.

It discusses the bounded context patterns aggregate, entity, and value object. Furthermore, it discusses the pattern repository for persisting objects and the factory for creating objects.

To access aggregates, the pattern orchestrator and choreography are discussed as well.

Structure

In this chapter, we will cover the following topics:

- Patterns provided by DDD
- Aggregate, entity, and value object
- Example model of the online library
- Repository
- Factories and orchestrators

Objectives

In this chapter, you will learn how to determine aggregates and entities and how to use the value object pattern to be meaningful. You will understand how to integrate the DDD and microservice architecture patterns and learn how to access aggregates externally.

> **…while the individual man is an insoluble puzzle, in the aggregate, he becomes a mathematical certainty.**
>
> *-Arthur Conan Doyle*

Patterns provided by DDD

In the previous chapter, we first saw the pattern introduced by Domain-Driven Design. *Evans* already introduced the patterns aggregate, entity, and value object, which we will discuss in detail in this chapter. He also introduced the pattern repository.

Later, patterns to access a bounded context were introduced. We will discuss the patterns of the orchestrator, choreography, and client.

As tactical design patterns of microservice architecture play a role in service design, similar to the original patterns of DDD, we will discuss the CQRS pattern as it is one of the most important service pattern.

Aggregate, entity, and value object

Evans defined the main concepts of domain-driven tactical design as aggregates, entities, and value objects[1]. We will discuss them in the following sections.

Aggregates

An aggregate maintains the entities of a bounded context. It serves as an anchor point of the bounded context. As the name suggests, an aggregate object aggregates the different objects of a bounded context. One aggregate object aggregates even other aggregates, like a car aggregates chassis and motor, which are aggregates themselves. *Evans* called them root aggregates. Meanwhile, we have experienced that this subtle difference does not play such a role. Therefore, we call the essence of a bounded context aggregate.

An aggregate is usually a class in object-oriented languages—that aggregates other entities and value objects. An aggregate is a cluster of associated objects treated as an atomic unit for data changes. In this way, consistency inside a bounded context can be ensured[2].

---

[1] Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley.

[2] Millet, S., & Tune, N. (2015). Patterns, Principles, and Practices of Domain-Driven Design. Indianapolis, IN: Wrox.

Internal entities of a bounded context can only be accessed by an aggregate. So, no object outside of the bounded context can hold a reference to the internal entities—only via an aggregate reference can internal entities be accessed[3].

This principle is shown in the example of task management in *Figure 10.1*:

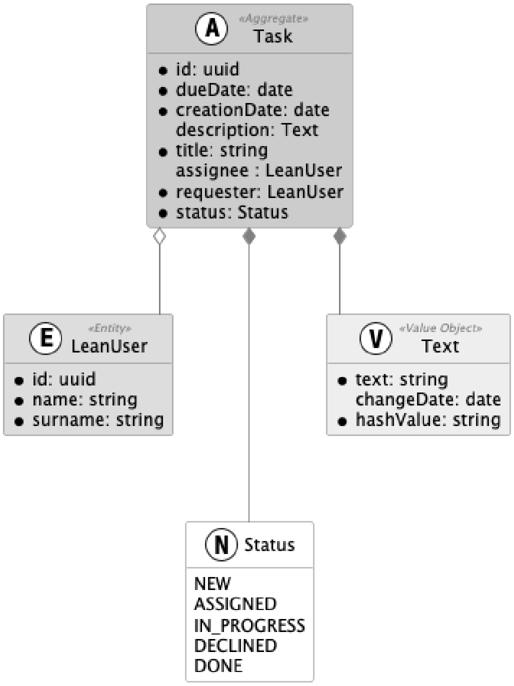


***Figure 10.1**: Class diagram of the bounded context task management*[4]

***Source**: TaskManagement.png*

The aggregate of task management is Task.

Task contains the value object description of type Text. The description can only be accessed by the task itself.

Note: A task description could also be a simple string. However, it was modeled as a value object to show how a value object could be used.

The aggregate contains an invariant. Invariants are business rules applied to the aggregate. The rules do not change even if the aggregate changes. For example, a task needs a valid description with a hash value. Changes to the description outside the task are not allowed.

---

[3] Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley.

[4] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter10-AggregateEntityAndValueObject/TacticalDesign/TaskManagement/TaskManagement.puml**

Therefore, the description is modeled as a value object. The rule *do not change the description outside of the task* is an invariant.

The state model with the belonging transitions is a valuable invariant. For example, the task management state model can be derived as an invariant from the domain events found in the corresponding event storming.

The invariant of a task is its status model, shown in *Figure 10.2*:

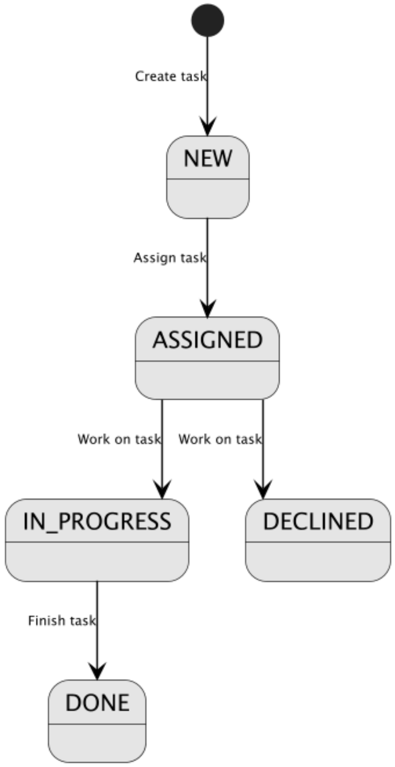


Figure 10.2*: State model of task management as an invariant of aggregate Task*[5]

Source*: TaskMangementStateModel.png*

Entity

An entity is a referenceable object inside a bounded context that cannot be accessed from outside. Entities can be used to reference aggregates foreign to the current bounded context. In such a case, they should be read only because they are not directly accessible. They can be accessed only via the aggregate. The responsible bounded context offers appropriate interfaces for them.

---

[5] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter10-AggregateEntityAndValueO-**

It can be used by an aggregate or by other entities. *Figure 10.1* shows the relationships between those.

For example, the assignee and requestor inside the Task aggregate are typical examples of entities used in an aggregate. They belong to the bounded context *User Management* and can be accessed by the aggregate LeanUser belonging to the bounded context.

The class Task contains the class LeanUser as an entity. LeanUser represents the data for the assignee and requester as part of the Task. LeanUser and Task can be referenced and contain an identifier: id. However, LeanUser is an entity in *Task Management* and can only be accessed via Task as the aggregate of *Task Management*.

The requestor and the assignee are modeled as *lean objects*. Usually, one does not need all the properties of an object foreign to its bounded context. Sometimes, just the identifier is sufficient. However, one usually needs a couple of properties, e.g., the user's name and given name, to display them conveniently in the user interface. The responsible bounded context can provide those properties iconic to the entity as a *lean object*.

Value object

A value object is an object without its own identity. So, it cannot be referenced by itself. It can only be accessed by the entity to which it belongs. They do not have their lifecycle.

They belong to the entities of the bounded context. Typical value objects are addresses belonging to a customer. From a consistency point of view, one needs to guarantee that a customer always has a valid address. The aggregate Customer can easily ensure that. It cannot be guaranteed by the aggregate Address of a bounded context *Address Management* because the business rules belong to the bounded context of *Customer Management*. Therefore, it is advisable to model the customer address as a value object without an identifier. So, it is possible to control changes at addresses by the aggregate and guarantee consistency.

In *Task Management*, we modeled Description as a value object. It can be accessed via Task but does not have its lifecycle.

Example model of the online library

The online library's model can easily be derived from the event storming results presented in *Chapter 7, Event Storming*.

First, we discuss the context of Catalog Management and Lending. The domain model is shown in *Figure 10.3*:

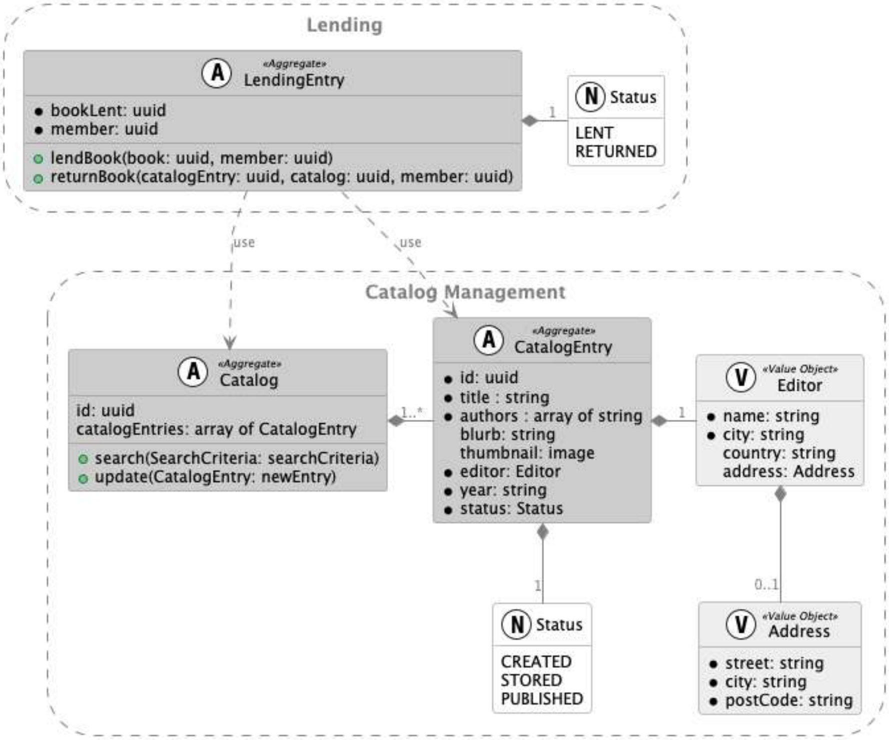


Figure 10.3*: Domain model of the library for contexts Lending and Catalog Management*[6]

Source*: LibraryCatalogManagementAndLending.png*

The aggregates and value objects were discussed in the event storming workshops. The status models can also be addressed during these or even during the domain storytelling workshops.

The bounded context *Catalog Management* contains the aggregates Catalog and CatalogEntry. CatalogEntry contains the value object Editor, whereas Editor contains the value object Address. Catalog offers the method search with searchCriteria and updates with a newEntry.

The bounding context *Lending* contains the LendingEntry as aggregate. LendingEntry includes the book of the bounded context *Catalog Management* and member of the bounded context *Member Management* as references.

---

[6] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter10-AggregateEntityAndValueO-**

Second, we show the result for the Reading and the Notes context in *Figure 10.3*:

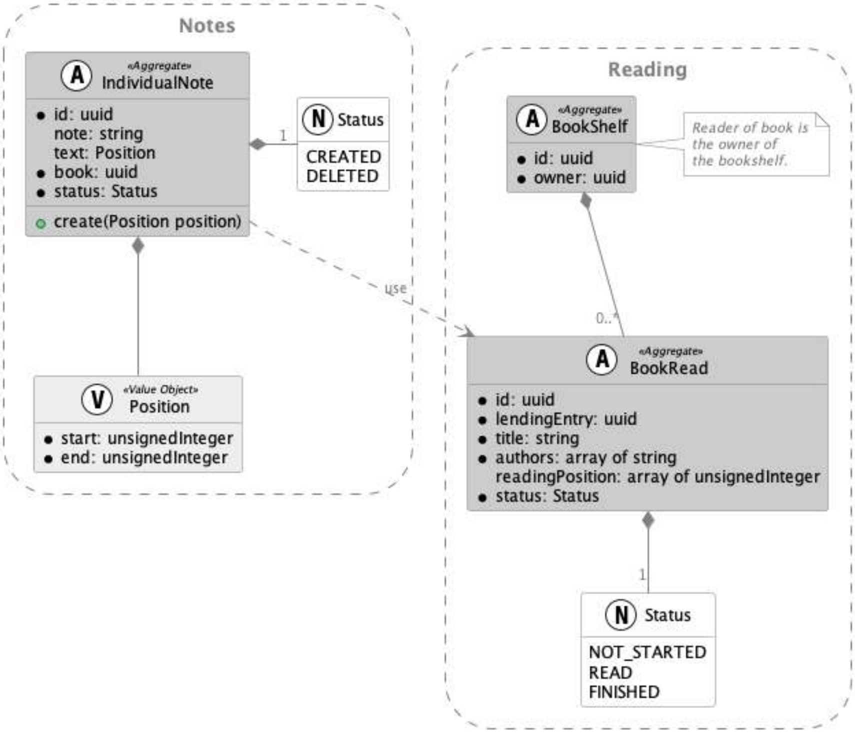


***Figure 10.4**: Domain model of the contexts Notes and Reading*[7]

Source: LibraryReadingAnd Notes.png

Reading as bounded context contains the aggregate BookRead. It is named BookRead and not only Book to avoid naming conflicts in the published language. With BookRead, not only is the object indicated, but the belonging bounded context *Reading* is also indicated. Additionally, the *Reading* bounded context contains a second aggregate BookShelf. BookShelf contains the books read by the corresponding reader. The reader of a book on the bookshelf is the owner. As discussed during the event storming sessions (see *Chapter 7*), a bookshelf is created when a new library member is onboarded, and insofar as the owner of the bookshelf is defined.

The bounded context *Notes* contains the aggregate Note, which belongs to a particular book on a specific user's bookshelf. To avoid naming conflicts, the note is called IndividualNote. It contains the note's text position in the book as a value object.

The aggregates' invariants, especially the corresponding state model, can be formulated similarly to the domain model.

---

[7] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter10-AggregateEntityAndValueObject/TacticalDesign/Library/LibraryReadingAnd%20Notes.puml**

A dependency from Reading to Lending is also necessary, as shown in *Figure 10.5:*

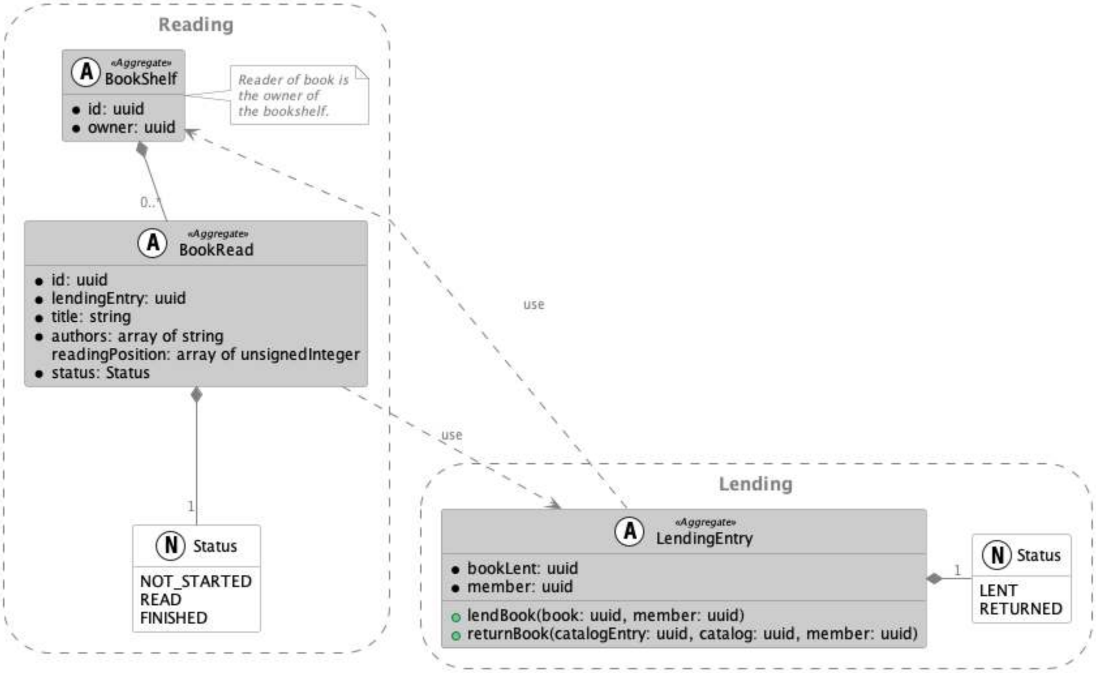


Figure 10.5: *Domain model of the library with Reading and Lending contexts*

Source: *LibraryReadingAndLending.png*

Invariants of the aggregates can be derived from the event storming workshop, as shown in *Figure 10.6*:

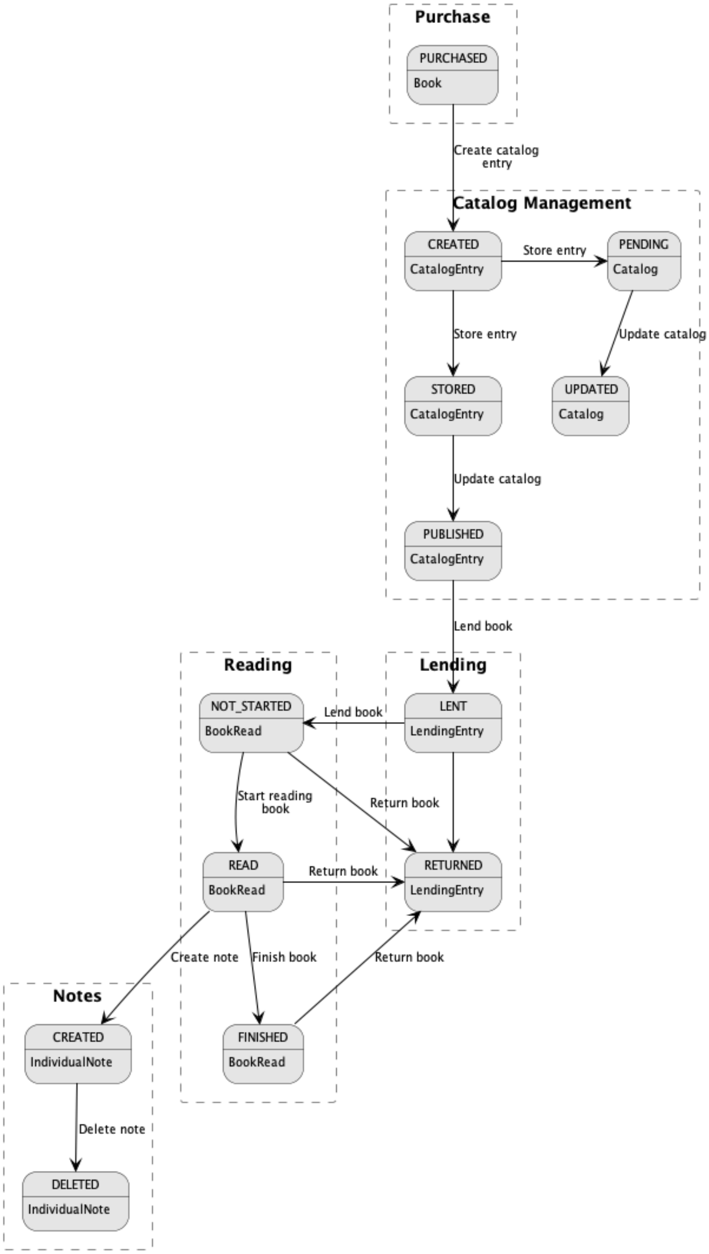


***Figure 10.6**: State diagram of the library based on event storming*[8]

***Source**: LibraryState.png*

---

[8] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter10-AggregateEntityAndValueObject/TacticalDesign/Library/LibraryState.puml**

The according states shown in *Figure 10.4* are explained in the following *Table 10.1*:

Status	**Bounded context**	**Aggregate**	**Transition to**	**Description**
PURCHASED	Purchase	Book	CREATED CatalogEntry	A book was newly purchased and needs to be put into the library catalog.
CREATED	Catalog Management	CatalogEntry	PENDING Catalog	A new catalog entry was created. The update of the catalog is still pending.
			STORED CatalogEntry	The new catalog entry was stored
PENDING		Catalog	UPDATED Catalog	A new catalog entry has been created, but the catalog itself needs to be updated.
UPDATED		Catalog		The catalog was updated. The new purchased book can be found in the catalog.
STORED		CatalogEntry	PUBLISHED CatalogEntry	After a catalog entry was stored, the according entry could be published after the catalog was published.
PUBLISHED		CatalogEntry	LENT LendingEntry	After a catalog entry has been published, the according entry can be found in the catalog.
LENT	Lending	LendingEntry	NOT_STARTED BookRead	When a book can be found, it can be lent by the member of the library. The book is available in the bookshelf, but the reading is not started yet.
			RETURNED LendingEntry	The member of a library can decide when to return a book.
RETURNED		LendingEntry		A member of the library returned a book.

Status	Bounded context	Aggregate	Transition to	Description
NOT_STARTED	Reading	BookRead	READ BookRead	A member of the library started to read a lent book.
			RETURNED LendingEntry	A member of the library can return a book even though they has not started reading it yet.
READ		BookRead	FINISHED BookRead	A member of the library has finished reading the book.
			RETURNED LendingEntry	A member of the library returns the book even if it is not finished yet.
FINISHED		BookRead	RETURNED LendingEntry	A member of the library returns the book after finishing it.
CREATED	Notes	IndividualNote	DELETED IndividualNote	A member of the library can create a note and even delete it afterwards.
DELETED		IndividualNote		A member of the library can delete a note to a book.

Table 10.1: *Status of the online library*

The library's domain model shows that the aggregates, entities, and value objects can easily be derived from domain storytelling and event storming results. It is recommended that the state models be discussed during the workshops with IT specialists and business experts. The status model must reflect the ubiquitous language, like the names for aggregates, entities, and value objects.

Persistence also needs to use the ubiquitous language. The pattern *Repository* is used to persist a domain model of a bounded context in a database. It is discussed in the next section.

Repository

A domain model needs to be persisted. As an aggregate should be handled as an atomic unit, an aggregate should be persisted as a whole object. Persisting single changes should be avoided[9].

So, it is used to decouple the access to the bounded context from the persistence of the domain model. The corresponding principle is shown in *Figure 10.7*:

---

[9] Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley.

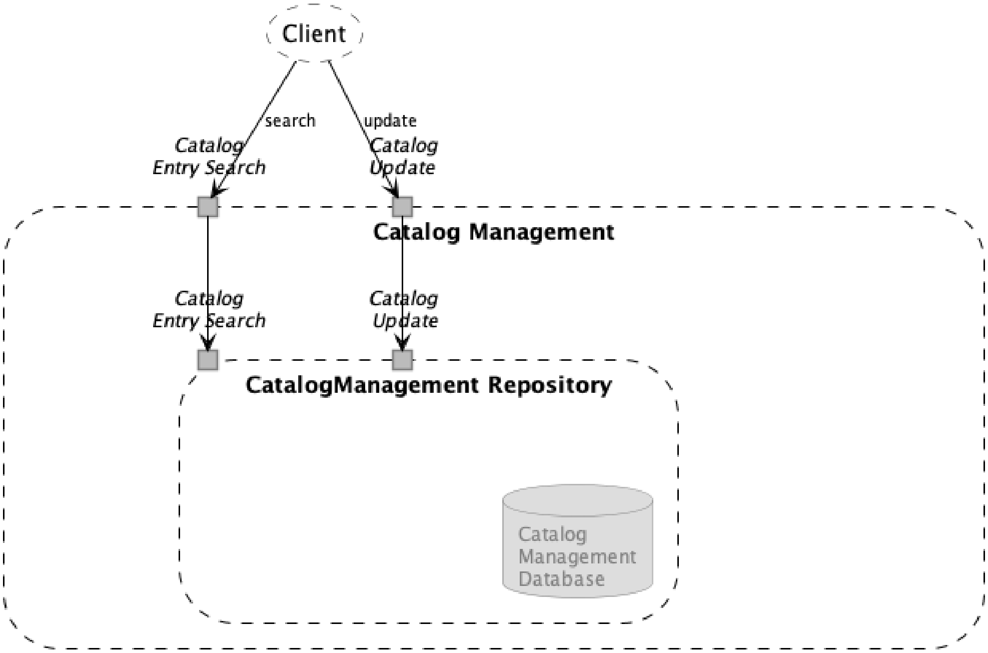


Figure 10.7*: Repository principle of bounded context Catalog Management*[10]

Source*: CatalogManagementRepository.png*

The aggregates Catalog and CatalogEntry can only be accessed by the client via the corresponding interfaces. Even internally, the database can only be accessed through the interfaces provided by the repository.

A repository hides the capabilities of the underlying persistence framework[11]. It separates the technical database model from the business-driven domain model[12].

Evans introduced the patterns aggregate, entity, value object, and repository. In recent years, other patterns become noteworthy because they can be used nicely to access the aggregates in a bounded context. Patterns in the microservice architecture especially need to be looked at more closely. But first, we want to start with the classic factory pattern.

Factories and orchestrators

Factories and orchestrators separate the domain model from other perspectives so that technical demands do not weaken it inside a bounded context. We will first discuss the factory pattern and, afterward, the orchestrator.

---

[10] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter10-AggregateEntityAndValueObject/TacticalDesign/Library/CatalogManagementRepository.puml**

[11] Millet, S., & Tune, N. (2015). Patterns, Principles, and Practices of Domain-Driven Design. Indianapolis, IN: Wrox.

Factories

The gang of four[13] first introduced the factory pattern. They introduced design patterns as reusable elements in object-oriented software design. Factories are used to create domain objects using a complex creation logic, whereas the logic is hidden behind the factory interface.

In the case of the library, a factory needs to be provided for the catalog entries. The corresponding approach is shown in *Figure 10.8*:

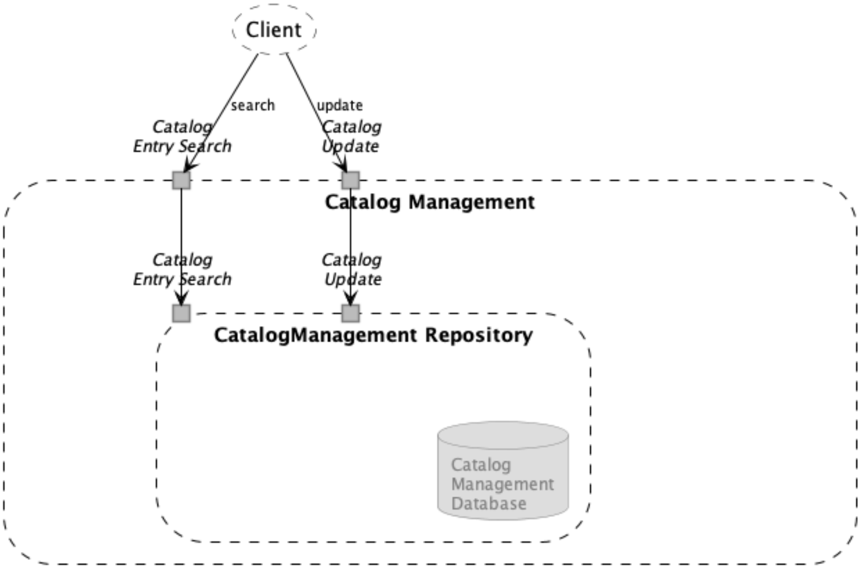


Figure 10.8: *Factory of catalog entry in a CQRS pattern*[14]

Source: *CatalogManagement.png*

The factory provides an interface for creating catalog entries. The catalog entries are created, and an event catalogEntryCreated is produced. The event is consumed by the query part of the catalog management, where a member can search for catalog entries.

The pattern **command query responsibility segregation** (**CQRS**)[15] was already mentioned in *Chapter 3, Why Strategic Design*. It is a typical design pattern in modern microservice environments. The necessity was determined in the event storming—business and not purely technical driven.

A factory is also needed to handle books purchased to produce a corresponding event for the summary bot.

---

[13] Gamma, E., Helm, R., & Johnson, R. (1997). Design Patterns. Elements of Reusable Object-Oriented Software. Prentice Hall.

[14] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter10-AggregateEntityAndValue-Object/TacticalDesign/Library/CatalogManagementRepository.png**

The catalog entry might even be created automatically based on an event produced by *Purchase*. However, the factory interface looks the same as for the librarian's creation of catalog entries.

In the next section, we want to discuss the orchestrator pattern. It is a pattern used to access aggregates outside of a bounded context, but it is important to understand how aggregates can be accessed.

Orchestrator

An orchestrator is needed to orchestrate several APIs (application programming interfaces) along a user journey. Usually, the orchestrator is implemented in the user interface client, and the corresponding state is stored in the client.

However, integrators might provide simple APIs for more complex API journeys. For example, a complex API journey might involve closing an insurance contract. The integrator provides an easy-to-use API that integrates the insurer's multiple APIs. In *Chapter 12*, we will explore those topics in the brownfield project.

In the library example, the client must handle the user journey from searching to lending and reading. An orchestrator on the client side can take over that task. How the client orchestrator can orchestrate the single calls is shown in *Figure 10.9*

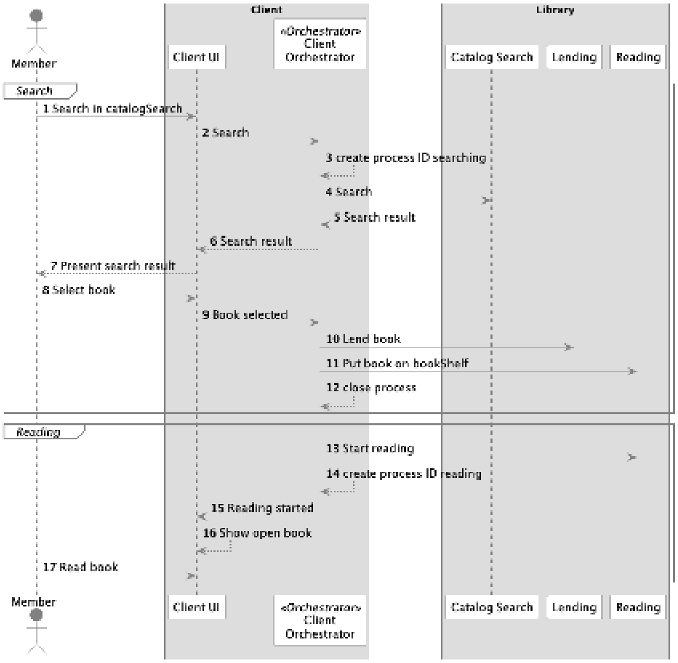


***Figure 10.9**: Orchestrator of search and reading on the client side*[16]

***Source**: Orchestrator.png*

---

[16] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter10-AggregateEntityAndValueObject/TacticalDesign/Library/Orchestrator.puml**

Figure 10.9 shows the searching and reading parts of the user interface as examples; organizing the bookshelf or creating and managing notes would be another example.

In our example, a sequence could look like the following points:

- **Searching**:
    - A user searches through the catalog and gives their search criteria.
    - The client user interface gives the search criteria to the orchestrator on the client side.
    - The orchestrator creates a process presented by an identifier.
    - Sends the search criteria to the library.
    - The Catalog Search component sends back the search result. Whether the orchestrator stores the search criteria as process properties depends on the design of the user interface. For example, it must be stored if the user wants to adapt the search later.
    - The orchestrator returns the result to the user interface.
    - The user interface presents the search result to the member.
    - The member can select the book from the result set they want to read.
    - The selected book is sent to the Lending component, and
    - Afterward, the orchestrator sends it to the Bookshelf component to appear on the bookshelf for the member.
    - The orchestrator can close the search process.
- **Reading**:
    - The orchestrator sends the selected book to the Reading component and
    - Starts the reading process.
    - The reading starts in the user interface and
    - The user interface shows the first page to the member.
    - The member can read the book.

The short and simplified example shows that the context boundaries and their belonging APIs can be kept pure even though they appear as one request to the user. The orchestrator stores the state for the user so that all calls to the component can be stateless. In the next chapter, we will discuss why stateless calls are beneficial. The domain models of the involved bounded contexts must not be weakened by user-friendliness. Moreover, they appear clean to the user interface implementor, whereas the user flow is not disrupted.

Choreography

Another approach would be choreography, where corresponding events are published, and each service reacts independently to those events. Such an when the services publish **micro-frontends** (**MFEs**) combined in a portal framework, which would provide the event queue on the client side. The corresponding pattern is shown in *Figure 10.10*:

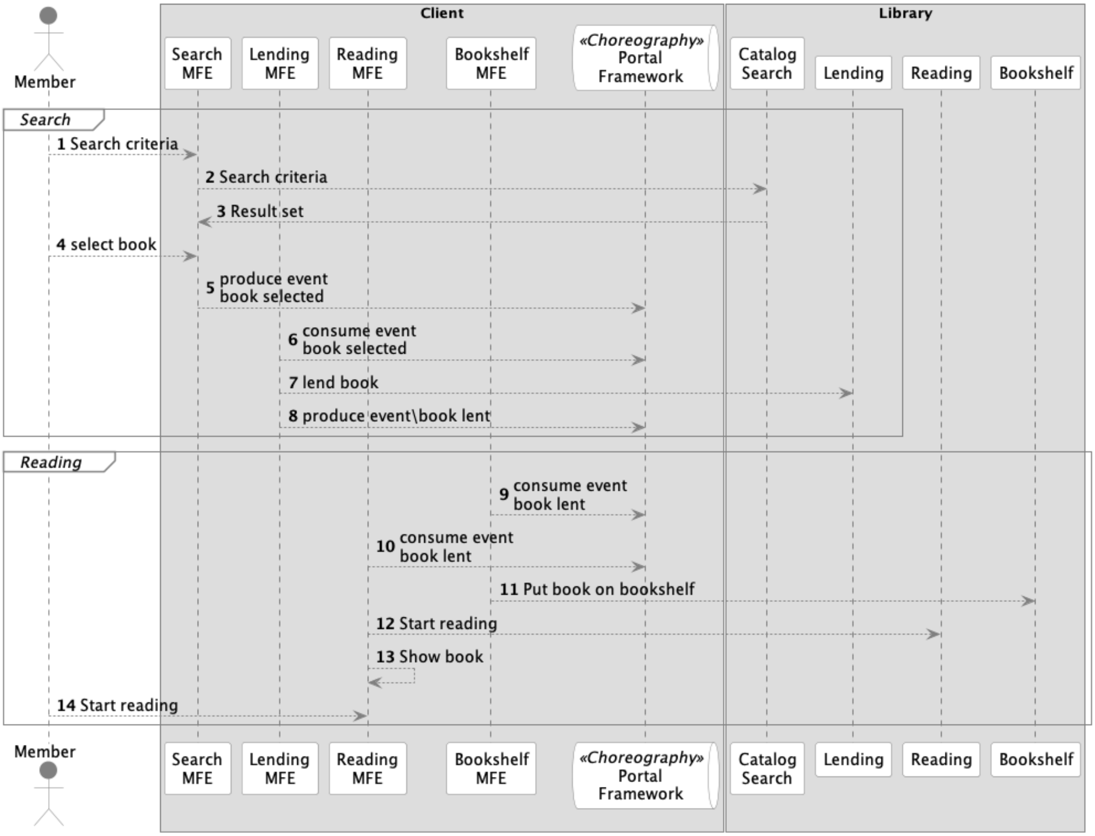


***Figure 10.10**: Choreography of the library*[17]

***Source**: Choreography.png*

As shown in *Figure 10.8*, the user interface components do not depend on each other. They are combined in a portal framework, providing a communication queue between the MFEs. The communication sequences could go as follows:

- **Searching**:
    - A member searches for a book in the *Search MFE*.
    - Search MFE sends the search criteria to the *Catalog Search*.
    - *Catalog Search* returns the result set.
    - The member selects the book they want to read.

---

[17] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter10-AggregateEntityAndValue-Object/TacticalDesign/Library/Choreography.puml**

  - *Search MFE* produces an event *book selected*.
  - *Lending MFE* consumes the event.
  - *Lending MFE* sends the book to the *Lending* component.
  - *Lending MFE* produces event *book lent.*
- **Reading**:
  - Bookshelf MFE consumes the book lent event.
  - Reading MFE consumes the book lent event.
  - Bookshelf MFE puts the book on the bookshelf by calling the Bookshelf MFE.
  - Reading MFE starts the reading by calling the Reading component.
  - The book is shown to the member by the Reading MFE.
  - The member starts reading.

The shown choreography allows more independent components in the user interface. The user state is stored in the queue and can be read independently by the components. However, it requires that the services provide MFEs of their bounded context. The MFEs also need to follow the interface definition of the portal framework. Though it would benefit user interface development, provider-independent standards for those MFEs have yet to be developed.

The shown workflow is not immensely complex, so a choreography would suit it. In more complex scenarios, an orchestrator would be the right choice[18]. Introducing orchestrators on the server side might be the right choice in other scenarios, as we will discuss in the brownfield project (*Chapter 14*).

The implementation teams need the right skills and resources for both scenarios. Choreography supports better future requirements regarding scalability and flexibility, while orchestration might be more suitable for stable systems and better maintainable solutions[19].

---

[18] Bhardwaj, D. (2023, April 11). Orchestration vs. Choreography: How to Pick the Right Integration Pattern. Retrieved August 2024, from Techcommunity Microsoft: **https://techcommunity.microsoft.com/t5/azure-integration-services/orchestration-vs-choreography-how-to-pick-the-right-integration/m-p/3792149**

[19] Bhardwaj, D. (2023, April 11). Orchestration vs. Choreography: How to Pick the Right Integration Pattern. Retrieved August 2024, from Techcommunity Microsoft: **https://techcommunity.microsoft.com/t5/azure-integration-services/orchestration-vs-choreography-how-to-pick-the-right-integration/m-p/3792149**

Conclusion

Tactical design is as essential to successful software products as strategic design. Developers can only depict a business sense by representing the ubiquitous language in the implemented bounded context. Without an understanding of published language, a bounded context stays meaningless. As *Evans* put it, the domain model becomes anemic.

Keep in mind that only a tight collaboration between business experts and IT specialists guarantees success and that the results of the strategic design documented during the domain storytelling and event-storming workshops need to be reflected in the business logic code and the database design.

Whereas we have discussed chiefly the internal design of a bounded context, we will discuss the publishing of the ubiquitous language in APIs in the next chapter.

Points to remember

- An aggregate is the essence of a bounded context's domain model.
- An aggregate contains invariants, usually represented by state models of the business the aggregate represents.
- Entities can be accessed via references. Usually, the objects belong to other bounded contexts, but an appropriate reference is necessary.
- Value objects cannot be referenced. They depend on the lifecycle of the containing object.
- A repository abstracts the persistence layer of the bounded context's domain model. It sentinels the domain model from weak influences of the database schemes.
- Factories are sentinels to the domain model from the outside. Factories abstract complex construction logic to the outside world so that externals to the bounded context do not need to know the creation logic.
- Factories can be used beneficially in CQRS patterns of the microservice world.
- An orchestrator is a pattern outside of the bounded context. It orchestrates the journey along different bounded contexts represented by their APIs.
- A choreography allows different bounded contexts to communicate via events, which increases the application's independence and, insofar, flexibility.

Multiple choice questions

1. **Which pattern used in the Domain-Driven Design most essentially represents a bounded context?**
    a. An entity
    b. An aggregate
    c. A value object
    d. A repository
2. **Which pattern is used to abstract a persistence layer from the bounded context?**
    a. A repository
    b. A factory
    c. An orchestrator
    d. A choreography
3. **When do you use a choreograph pattern at best?**
    a. When the bounded context services provide MFEs, and high flexibility is required.
    b. No specific requirements or flexibility are known when a stable application is required.
    c. A choreography pattern should always be selected independent of software usage.
    d. A choreography pattern should not be used. An orchestrator should be used consistently.

Answers

1. **b**
2. **a**
3. **a**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 11

Exposing Aggregates via APIs

Introduction

In this chapter, we see that aggregates could be accessed via a published language. In the last chapter, *Chapter 10, Aggregates, Entities, and Value Objects*, we understood aggregates. Aggregates maintain the content of the bounded context. Let us see how we can publish aggregates using **application programming interfaces** (**APIs**).

Structure

In this chapter, we will understand the following topics:

- Introduction to RESTful APIs and its specifications
- Specification of the APIs of task management
- APIs for online library

Objectives

In this chapter, you will learn how to publish RESTful APIs using an OpenAPI specification. It will guide you through the process of utilizing HTTP verbs for various business functions and incorporating ubiquitous language to create meaningful APIs.

> **…while the individual man is an insoluble puzzle, in the aggregate, he becomes a mathematical certainty.**
>
> *-Arthur Conan Doyle*

Introduction to RESTful APIs and its specification

In this chapter, we want to publish aggregates of bounded contexts via RESTful APIs. When we discuss RESTful APIs, we need to discuss REST as an architectural style and API.

Application programming interface

An application programming interface is an interface through which a programmer can interact with the application like a user interface, which allows a user to interact with the application using a graphical interface, an application programming interface allows a programmer to interact via code.

The concept of an interface has existed since the first computers, allowing applications to be structured and maintained. *Goldstine and von Neumann* published the idea of *libraries* in 1947[1].

The term API appeared first in 1968 by *Cotton and Greatorex*[2]. The idea describes how an interface and its implementation can be separated. So, it is possible to change the implementation without changing the interface, which does not correspondingly impact the client using the API.

Here, we mean APIs used via a network with an underlying protocol. The appropriate protocol is the **Hypertext Transport Protocol** (**HTTP**). Next, we will discuss **REST** as an API architectural approach that uses HTTP for transport.

REST

REST stands for **Representational State Transfer**. It means that you do not directly access the state of an object, e.g., via a direct method call; instead, you get the state of the object as a representation. The state is transported via HTTP. RESTful APIs are synchronous APIs, which means that a client receives a response to its request in the same session.

---

[1] Goldstine, H., & von Neumann, J. (1947). Planning and Coding of Problems for an Electronic Computing Instrument (Vols. Report on Mathematical and Logical Aspects of an Electronic Computing Instrument, Part II, Volume 1-3). Princeton, New Jersey: Institute for Advanced Study.

[2] Cotton, I. W., & Greatorex, F. S. (1968). Data structures and techniques for remote computer graphics. Fall joint computer conference.

It is comparable to a human conversation. A person speaks to another person who is directly with them. They wait for the person's answer before speaking again.

One way to specify RESTful APIs is to use OpenAPI.

Richardson's maturity model helps to understand RESTful APIs better[3]. The principle is shown in the following *Figure 11.1*:

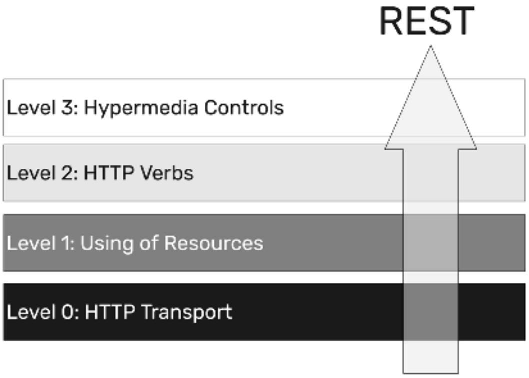


Figure 11.1*: Richardson maturity model*

Source*: RichardsonMaturityModel.jpg*

In level 0, HTTP is only used as the transport layer.

In level 1, resources are used. Resources mean something that can be addressed[4]. The aggregate can be addressed because they are referencable. **Universal Resource Identifiers** (**URIs**) are used to address resources.

In level 3, HTTP methods[5] are used to access resources[6]:

- GET to get an object
- POST to create an object
- PUT to change an object
- DELETE to delete an object.

Because RESTful interfaces use those well-defined verbs, the APIs can be understood well, and a programmer using them knows what to expect.

---

[3] Fowler, M. (2010, March 18). Richardson Maturity Model. Retrieved August 2024, from Martin Fowler: **https://www.martinfowler.com/articles/richardsonMaturityModel.html?ref=philsturgeon.com**

[4] Richardson, L., & Ruby, S. (2007). RESTful Web Services. Cambridge: O'Reilly.

[5] HTTP request methods. (2024). Retrieved August 2024, from MDN web docs: **https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods**

[6] The methods HEAD, CONNECT, OPTIONS, TRACE, and PATCH exist as well. But we will not use

The use of HTTP verbs includes the use of HTTP response codes. So, response code 200 is required for a successful operation, response code 201 is required to create an object, or response code 500 is required when the system is unavailable. Please refer to the website HTTP response codes[7] for further information about HTTP response codes.

On level 3, hypermedia controls are used to refer to other resources. Entities contained in aggregates can be linked, making it convenient for a client developer to access those entities in their original bounded context.

RESTful interfaces need to be specified. Using the standard language OpenAPI to specify.

OpenAPI specification language for RESTful APIs

OpenAPI is a specification standard for defining RESTful APIs[8]. Please be aware that we cannot discuss the entire specification here because we only use it as a tool to define the APIs found in the DDD process.

An OpenAPI specification contains the parts info, servers, paths, and components. An overview of those parts is given in *Figure 11.2*:

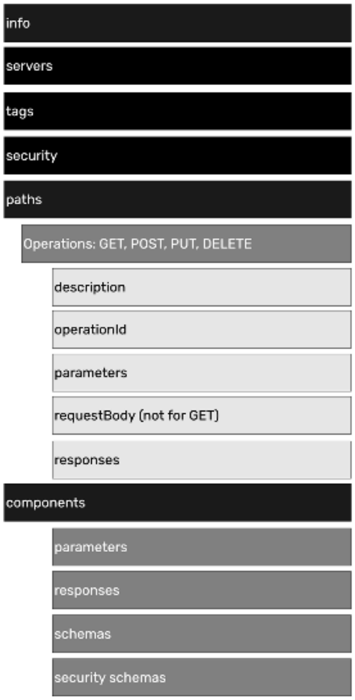


***Figure 11.2**: Structure of an OpenAPI specification*

***Source**: OpenApiStructure.jpg*

---

[7] HTTP response codes. (2024). Retrieved August 2024, from MDN Web Docs: **https://developer.mozilla.org/en-US/docs/Web/HTTP/Status**

[8] (OpenAPI, 2022)

The info block contains essential information about the API itself.

A skeleton, which already contains all the necessary blocks, is provided to facilitate the API specification[9]. An excerpt of the skeleton is given in the following listing:

```
openapi: 3.1.0
info:
  title: Skeleton
  description: A skeleton which can be used to create OpenAPI specifications
  contact:
…
servers:
…
security:
..
tags:
…
paths:
  /resources:
    get:
    …
  /resources/{resourceIdentifier}:
    get:
    …
    put:
    …
    delete:
components:
  parameters:
  …
  responses:
…
  requestBodies:
  …
  schemas:
  …
  securitySchemes:
..
```

Using the task management example, let us discuss individual parts of an OpenAPI definition.

---

[9] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter11-ExposingAggregatesViaApis/RestApis/RestSkeleton.yaml**

Specification of the API of task management

First, we can define the schemas needed on the base of the tactical design.

Schema design

The schema design starts with the design of a task that should be created:

```
TaskToBeCreated:
  description: A task created by a clerk
  type: object
  required:
    - title
    - dueDate
    - requester
  properties:
    title:
      description: A title of the task
      type: string
      minLength: 2
      maxLength: 10
      examples:
        - Create a new book
    description:
      description: A longer description of the task
      type: string
      minLength: 2
      maxLength: 2048
      examples:
        - Please define the outline with the number of pages for each
chapter.
    dueDate:
      description: Date when the task is due
      type: string
      format: date
      example: 2024-08-10
    requester:
      $ref: '#/components/schemas/LeanUser'
```

A task to be created differs from a regular task because it does not have an identifier. However, it has a title, a due date, and a requester. Optionally, the task requester can describe the task in more detail.

The requester is given as **`LeanUser`** – described in the tactical design for convenience:

```
LeanUser:
  description: A user of the system defined in the user management with ID
and name
  type: object
  required:
    - id
  properties:
    id:
      description: Identifier of the user
      type: string
      format: uuid
      examples:
        - c2ca22c8-33e7-424d-ba09-ece8b4fe083f
        - ab3edd94-db81-4012-88d8-674423a4a671
    name:
      description: Family name of the user
      type: string
      minLength: 2
      maxLength: 40
      examples:
        - Doe
        - Smith
    surname:
      description: Given name of the user
      type: string
      minLength: 2
      maxLength: 40
      examples:
        - John
        - Joanna
    link:
      description: Link to the user resource in user management
      type: string
      format: uri
      examples:
        - https://mydomain.com/user-management/users/c2ca22c8-33e7-424d-
ba09-ece8b4fe083f
        - https://mydomain.com/user-management/users/ab3edd94-db81-4012-
88d8-674423a4a671
```

The **`LeanUser`** contains its identifier, name, surname, and a link to user management, where a client can request more information about the corresponding user.

A task needs an identifier, an assignee, and a status in addition to **`TaskToBeCreated`**. Only the additional properties must be defined since a task can inherit from **`TaskToBeCreated`**. The inheritance in OpenAPI is declared by the keyword **`allOf`**:

```
Task:
  description: A task a clerk can fulfill
  type: object
  allOf:
    - $ref: '#/components/schemas/TaskToBeCreated'
  required:
    - id
    - creationDate
  properties:
    id:
      description: Unique identifier for the task
      type: string
      format: uuid
      examples:
        - a07d62e1-5e2e-4e15-b50d-6ed412175443
    assignee:
      $ref: '#/components/schemas/LeanUser'
    creationDate:
      description: Date when the task was created
      type: string
      format: date
      examples:
        - 2024-08-01
    status:
      description: Status of the task
      type: string
      enum:
        - NEW
        - ASSIGNED
        - IN_PROGRESS
        - DECLINED
        - DONE
```

The task's identifier is a **universally unified identifier** (**UUID**)[10], guaranteeing its uniqueness. It is advisable to define all identifiers as UUIDs.

The creation and due dates are defined as strings, as usual in OpenAPI. OpenAPI does not provide specific date or time formats. However, certain formats can be given for those

---

[10] Leach, P., Mealling, M., & Salz, R. (2005, July). RFC 4122—A Universally Unique Identifier (UUID) URN Namespace. Retrieved August 2024, from Network Working Group: **https://datatracker.ietf.org/doc/html/rfc4122**

strings, as they are with the format `date`. For further information, refer to the OpenAPI specification[11].

Status is also a string, but it is an enumerator that limits possible entries. As usual, the enumerator is given in upper cases. The entries were defined in tactical design.

With the schema definition, we can define the resource paths.

Resource paths

First, we want to discuss the methods on an array of tasks. Afterward, we discuss the resource paths that act on a single resource.

Tasks Get

The first resource path usually given is a GET with corresponding search parameters. The resource itself is named in the plural form. Naming resources in plural is a de facto standard on the web, e.g., as required in the guidelines of the company *Zalando*[12].

The search parameters are given as an array of strings. The server implementation allows a full-text search using title and description parameters. Searches based on specific parameters, e.g., requestor, assignee, or due date, are also possible. The search parameters are given as a query in the URL:

https://mycompany.com/task-management/tasks?title=book&title=chapter&dueBefore=2024-08-31&assignee=c2ca22c8-33e7-424d-ba09-ece8b4fe083f

The response in a successful operation is given as an array of tasks. Error conditions are formulated using the corresponding HTTP response codes.

The entire path definition looks like the following:

```
/tasks:
  get:
    description: Using this endpoint a user gets back all task fulfilling
their search criteria
    operationId: getTasksByFilterCriteria
    tags:
      - Task
    security:
      - openIdConnect:
          - resource:read
    parameters:
      - $ref: '#/components/parameters/SearchCriteriasParameter'
```

---

[11] OpenAPI. (2022). The world's most widely used API description standard. Retrieved July 2024, from OpenAPIs: **https://www.openapis.org/**

[12] 7. REST Basics - URLs. (2024, June). Retrieved August 2024, from API Guidelines: **https://opensource.zalando.com/restful-api-guidelines/#urls**

```
      - $ref: '#/components/parameters/AssigneeSearchParameter'
      - $ref: '#/components/parameters/RequesterSearchParameter'
      - $ref: '#/components/parameters/VersionParameter'
    responses:
      '200':
        $ref: '#/components/responses/TaskListResponse'
      '400':
        $ref: '#/components/responses/BadRequestResponse'
      '403':
        $ref: '#/components/responses/ForbiddenResponse'
      '500':
        $ref: '#/components/responses/ServiceNotAvailableResponse'
      default:
        $ref: '#/components/responses/DefaultResponse'
```

In addition to the previously described parameters, **`VersionParameter`** is also required. The version parameter contains the specification version as given in the info block of the specification. To do so, the client can request a particular version of the API, which the server can manage. The server can support different API versions, including breaking changes, without exposing different paths[13].

Next, let us see how to create a task.

Tasks Post

To create a task, a **`POST`** must be sent to the server. The **`POST`** contains a request body containing the **`TaskToBeCreated`** object:

```
/tasks:
…
  post:
    description: Creates a new Task
    operationId: createNewRessource
    tags:
      - Task
    security:
      - openIdConnect:
          - resource:write
    parameters:
      - $ref: '#/components/parameters/VersionParameter'
    requestBody:
    # link to your resource to be created
      $ref: '#/components/requestBodies/TaskToBeCreatedRequest'
```

---

[13] Please be aware that we do not handle pages or something comparable, even though a large result set is to be expected when handling tasks. We want to discuss Domain-Driven Design and use API design as an example of how to use DDD best.

```
    responses:
      '201':
        $ref: '#/components/responses/LinkToTaskResponse'
      '400':
        $ref: '#/components/responses/BadRequestResponse'
      '403':
        $ref: '#/components/responses/ForbiddenResponse'
      '500':
        $ref: '#/components/responses/ServiceNotAvailableResponse'
      default:
        $ref: '#/components/responses/DefaultResponse'
```

The successful operation response to the creation request of a new object is the standard `201` response. The error conditions contain the standards as defined.

Single task Get

The task's identifier can be given as a URL parameter to get a single task:

https://mycompany.com/task-management/tasks/448b5bc-a9bc-4036-8899-c77f781a76d5

The parameter `taskId` can be defined, and in such a way, a single task can be obtained.

```
TaskIdentifierParameter:
  name: taskId
  description: Task identifier
  required: true
  in: path
  schema:
    type: string
    format: uuid
    examples:
      - 448b5bc-a9bc-4036-8899-c77f781a76d5
```

Parameters can be defined in the schemas section parameters. In such a way, they can be reused for different paths.

Single task Delete

Comparably, a task can be deleted. The task is identified by its identifier as a path variable:

```
delete:
  description: Deletes a task identified by its id
  operationId: deleteResourceById
  tags:
    - Task
  security:
    - openIdConnect:
        - resource:admin
  parameters:
    - $ref: '#/components/parameters/TaskIdentifierParameter'
    - $ref: '#/components/parameters/VersionParameter'
  responses:
    '200':
      $ref: '#/components/responses/SuccessfulOperationResponse'
…
```

With task management, we learned how to apply ubiquitous language and tactical design to an API. Now, let us explore the online library further.

APIs for online library

The APIs for the online library need to be created following the visual glossary created during domain storytelling and event storming workshops. Let us review it again (see *Chapter 5, Visual Glossary*), as shown in the following figure:

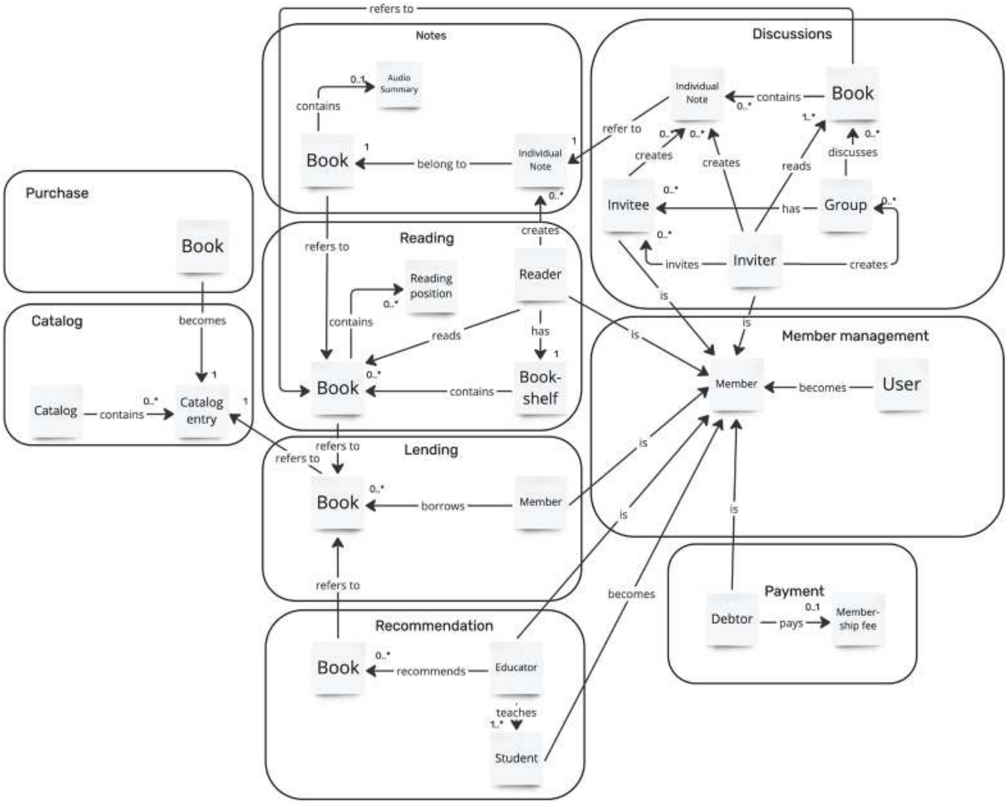


Figure 11.3*: Visual glossary of the online library*

Source*: VisualGlossary.jpg*

We will start with the API of Catalog Management.

API of Catalog Management

The API of *Catalog Management* is comparable to the API of *Task Management* discussed previously.

A catalog entry can be created, changed, or even deleted.

Catalog entries can be searched for by entries such as authors, titles, or even ISBNs (International Standard Book Number). Additionally, books can be searched for by their publishing year. Because OpenAPI does not give a specialized year format, the year format is given as a regular expression:

```
PublishingYear:
  description: The year when a book was published. The year is in the past.
  type: string
  pattern: ^[0-9]{1,4} ?[a,b]?.?c?.?$
  examples:
    - 1964
    - 2023
    - 583 bc
```

The formulation as string allows you to give additional information to the year, like BC or AC, to indicate whether it occurred before or after Christ in the Gregorian calendar.

The full definition of the API can be found in the GitHub project[14].

Next, we want to discuss the bounded context *Member Management* API.

API Member Management

Member Management is also based on the skeleton. However, a couple of specifics need to be considered.

Searching for several members requires transferring personal data. Transferring personal data via URL is open to misuse because malicious people can easily obtain a URL. One solution might be to encrypt the query part of the URL, which requires the exchange of corresponding certificates. This might be a threshold to acquire new library members. Therefore, the search is not done via GET; the search is done via POST. In such a way, the search criteria can be transferred via the request body, which standard **Transport Layer Security** (**TLS**) secures, shown as follows:

---

[14] **https://github.com/Grinseteddy/MasteringDdd/blob/main/RestApis/OnlineLibrary/CatalogManagement/CatalogManagement.yaml**

```
/members/search:
  post:
    description: |
      A member administrator can get all members based on specific search
criteria e.g. their names or email-addresses.
      The search criteria are provided via post to avoid passing sensible
data via a URL.
      Members can be found by name or email address to be invited by other
members. The members can only be found if
      the according flag is set.
    operationId: createSearch
    tags:
      - Member Administration
      - Discussions
    security:
      - openIdConnect:
          - member:admin
    parameters:
      - $ref: '#/components/parameters/VersionParameter'
    requestBody:
      $ref: '#/components/requestBodies/SearchCriteriasRequestBody'
    responses:
      '200':
        $ref: '#/components/responses/MembersResponse'
      '400':
        $ref: '#/components/responses/BadRequestResponse'
      '403':
        $ref: '#/components/responses/ForbiddenResponse'
      '500':
        $ref: '#/components/responses/ServiceNotAvailableResponse'
      default:
        $ref: '#/components/responses/DefaultResponse'
```

Administrators can find members, or they can be found by other members to be invited to discussions. However, other members can find members only if the according flag is true, which means that the corresponding member has explicitly allowed to be found, shown as follows:

```
MemberToBeCreated:
  description: Member to be created, the member will be created after the
payment fee has been paid
  type: object
  required:
    - familyName
    - emailAddress
    - canBeFoundByOthers
  properties:
    familyName:

    …
    givenName:

    …
    emailAddress:

    …
    role:
      description: Role in which the member uses the library
      type: string
      enum:
        - STUDENT
        - TEACHER
        - MEMBER
      default:
        - MEMBER
      examples:
        - STUDENT
    canBeFoundByOthers:
      description: If this flag is true, the member allows to be found by
other members
      type: boolean
      default: false
      examples:
        - false
```

Data privacy is of high quality and needs to be checked carefully with security and business specialists. Specific domain storytelling workshops can help identify potential threads from a business point of view. An example of such an analysis is shown in *Figure 11.4:*

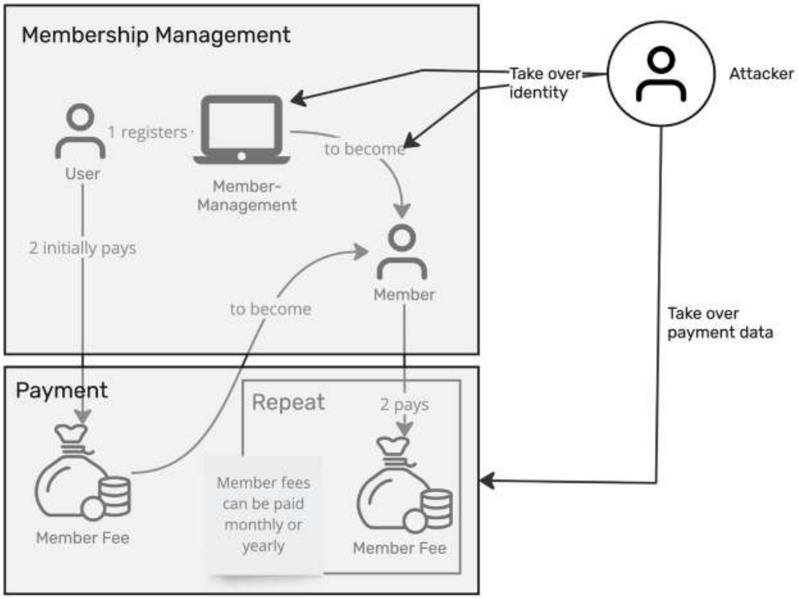


Figure 11.4*: Example of a thread analysis using domain storytelling*[15]

Source*: ThreadAnalysis.jpg*

The analysis shows where potential threads need to be handled. In our case, possible identity theft needs to be avoided. Moreover, the theft of payment data needs to be avoided as well. The payment provider's specialists can manage this thread. In contrast, the online library implementation must prevent the theft of member identities, e.g., by transferring the search data via a request body.

Now, let us see how we can formulate the discussion API.

API of discussions

In the bounded context discussions, a member wants to invite other members. Other members can be found via member management when they can be found. Multiple members can be asked to join the discussion group at once.

The discussion itself contains the invitees, shown as follows:

```
Discussion:
  description: A discussion initiated by a member
  type: object
```

---

[15] Please be aware that we show only an example of such an analysis here. For more information, please refer to specific literature discussing data privacy.

```
  # allows some kind of polymorphism
  allOf:
    - $ref: '#/components/schemas/DiscussionToBeCreated'
  required:
    - id
    - bookTitle
    - initiator
  properties:
    id:
      description: Unique identifier for the resource
      type: string
      format: uuid
      examples:
        - dc975e44-9155-49b3-bbdc-45fd577e536a
    bookTitle:
      description: For convenience reason the book title is stored with the
discussion
      type: string
      minLength: 2
      maxLength: 255
      examples:
        - Domain-Driven Design
    initiator:
      description: Member identifier of the initiator of the discussion
group
      type: string
      format: uuid
      examples:
        - 8103ec7b-63d3-4e55-9677-60478061d00b
    initiatorName:
      description: For convenience the name of the initiator is stored as
well
      type: string
      minLength: 2
      maxLength: 65
      examples:
        - Doe
    invitees:
      $ref: '#/components/schemas/Invitees'

…

Invitees:
  description: An array of invitees
  type: array
  minItems: 0
  maxItems: 20
  items:
    $ref: '#/components/schemas/Invitee'

…

Invitee:
  description: |
    Invitee to a discussion group. The invitee is given by their member
identifier.
    For convenience, the member name is stored as well.
  type: object
  allOf:
    - $ref: '#/components/schemas/InviteeToBeCreated'
  required:
    - id
    - status
  properties:
    id:
      description: Identifier of the invitee
      type: string
      format: uuid
      examples:
        - 2c4365b7-d26b-4ecd-b742-b6e7fe8cdfce
    inviteeName:
      description: For convenience the name of the invitee is stored
      type: string
      minLength: 2
      maxLength: 65
      examples:
        - Smith
    status:
      - PENDING
      - ACCEPTED
    # when an invitee rejects their invitation, they do not appear in the
group anymore
    # therefore a status "DECLINED" does not appear
```

`Invitees` are formulated as an object and not only as an array inside of the discussion to allow the creation of a separate object that only needs to be tested on an empty array and not additionally on null.

A group's invitees are managed as a sub-resource of a discussion. This sub-resource can only be accessed via the aggregate **`Discussion`** and insofar via the resource **`discussions`**, shown as follows:

```
/discussions/{discussionId}/invitees:
  get:
    description: |
      Returns all invitees of a discussion with their invite status
      Invitees can be accessed by all discussion member who have already
accept the invite
…
  post:
    description: |
      The initiator of a discussion can invite other members of the library
to the discussion
      The member are given just by their member identifiers. The initiator
needs to search them first.
    operationId: addInviteesToDiscussionById
    tags:
      - Invitees
    security:
      - openIdConnect:
          - discussion:admin
    parameters:
      - $ref: '#/components/parameters/DiscussionIdentifierParameter'
      - $ref: '#/components/parameters/VersionParameter'
    requestBody:
      $ref: '#/components/requestBodies/InviteesToBeCreatedRequest'
    responses:
      responses:
        '201':
          $ref: '#/components/responses/LinkToDiscussionResponse'
…

schemas:

…

  requestBodies:

    InviteesToBeCreatedRequest:
      description: Invites several member to a discussion
      content:
        application/json:
          schema:
            type: array
            minItems: 1
            maxItems: 20
            items:
              $ref: '#/components/schemas/InviteeToBeCreated'
```

Invitees must accept the invite even if they are found (the flag **`canBeFoundByOthers`** is **`true`**). Therefore, they are stored first in the status **`PENDING`**. Later, a member can change status by accepting or rejecting the invite. Moreover, a member can reject their invitation even after accepting it. It seems like a deletion of a group member. This can be seen in the following code snippet:

```
/discussions/{discussionId}/invitees/{inviteeId}:
  put:
    description:
      An invitee can change status of the invite. They can accept it or
they can reject it.
      An invite can reject their invite even if they have already accepted
it. To reject an accepted invite
      is seen as leaving the discussion group.
  operationId: updateInviteeByIdAndByDiscussionId
  tags:
    - Invitees
  security:
    - openIdConnect:
        - discussion:write
  parameters:
    - $ref: '#/components/parameters/DiscussionIdentifierParameter'
    - $ref: '#/components/parameters/VersionParameter'
    - $ref: '#/components/parameters/InviteeIdentifierParameter'
  requestBody:
    $ref: '#/components/schemas/Invitee'
  responses:
    responses:
      '200':
        $ref: '#/components/responses/LinkToInviteeResponse'

…
  delete:
    description:
      An initiator of a discussion can uninvite an invitee. They will leave
the group independent of their
      invitation status.
    tags:
      - Invitees
…
```

```
    responses:
      '200':
        $ref: '#/components/responses/SuccessfulOperationResponse'
```

The invitee to be created contains only the member identifier. However, the invitee is a referencable entity that also includes the status of the invitee, as shown in the following snippet:

```
Invitee:
  description: |
    Invitee to a discussion group. The invitee is given by their member
identifier.
    For convenience, the member name is stored as well.
  type: object
  allOf:
    - $ref: '#/components/schemas/InviteeToBeCreated'
  required:
    - id
    - status
  properties:
    id:
      description: Identifier of the invitee
      type: string
      format: uuid
      examples:
        - 2c4365b7-d26b-4ecd-b742-b6e7fe8cdfce
    inviteeName:
      description: For convenience, the name of the invitee is stored
      type: string
      minLength: 2
      maxLength: 65
      examples:
        - Smith
    status:
      - PENDING
      - ACCEPTED
    # when an invitee rejects their invitation, they do not appear in the
group anymore
    # therefore, a status "DECLINED" does not appear
```

Entities in an aggregate can be elegantly handled as sub-resources in an API. The API becomes expressive, and the aggregate secures the domain model.

Let us look at last to the API of the bounded context Notes.

Notes API

We can compare creating and editing notes to Task Management, which almost gives us the same structure as *Notes*. However, notes belong to a particular member and can only be edited by them. They also belong to a specific book, so they must be accessible by both the member and the book.

We could use the book as a path parameter, but that would mean it is directly accessible via notes, even though it does not belong to the bounded context. To avoid the API spoiling the domain model, the book parameter is formulated as a mandatory query parameter:

```
paths:
  /notes:
    get:
      description: The notes are searched by the book identifier and further text
      operationId: getNotesByContent

      tags:
        - Notes

      security:
        - openIdConnect:

            - notes:read

      parameters:
        - $ref: '#/components/parameters/ContentSearchParameter'

        - $ref: '#/components/parameters/BookIdentifierParameter'

        - $ref: '#/components/parameters/VersionParameter'

      responses:
        '200':
          $ref: '#/components/responses/NotesResponse'

    post:

       description: Creates a note

…

/notes/{noteId}:
  get:
    description: Gets a specific note by its identifier
    operationId: getNoteById
    tags:
      - Notes
    security:
      - openIdConnect:
          - notes:read
    parameters:
      - $ref: '#/components/parameters/NoteIdentifierParameter'
      - $ref: '#/components/parameters/BookIdentifierParameter'
      - $ref: '#/components/parameters/VersionParameter'
    responses:
      '200':
        $ref: '#/components/responses/NoteResponse'
…
  put:
    description: Changes a particular note
    operationId: updateNoteById
    tags:
      - Notes
    security:
      - openIdConnect:
          - notes:write
    parameters:
      - $ref: '#/components/parameters/NoteIdentifierParameter'
      - $ref: '#/components/parameters/BookIdentifierParameter'
      - $ref: '#/components/parameters/VersionParameter'
    requestBody:
      $ref: '#/components/requestBodies/NoteToBeChangedRequest'
    responses:
      '200':
        $ref: '#/components/responses/LinkToNoteResponse'
…
  delete:
    description: Description # describe your function
    operationId: deleteResourceById
    tags:
      - Name Tag
    security:
      - openIdConnect:
          - resource:admin
    parameters:
      - $ref: '#/components/parameters/NoteIdentifierParameter'
      - $ref: '#/components/parameters/BookIdentifierParameter'
      - $ref: '#/components/parameters/VersionParameter'
    responses:
      '200':
        $ref: '#/components/responses/SuccessfulOperationResponse'
…
```

The book identifier is used for all endpoints in the *Notes API*. This is shown in the following code snippet:

```
BookIdentifierParameter:
  name: bookId
  description: Book identifier
  required: true
  in: query
  schema:
    type: string
    format: uuid
    examples:
      - 3a2e5b0e-4edc-463e-af30-afdf7530e4cc
```

A note has an owner—comparable to a task requestor—and a corresponding book. The requestor does not appear in the notes' paths because the current user needs to authenticate themselves, and the token can identify the member. However, the owner and book are formulated as properties to all notes belonging to a book:

```
Notes:
  description: All notes belonging to one book
  type: object
  properties:
    bookIdentifier:
      description: Identifier of the book to which the notes belong
      type: string
      format: uuid
      examples:
        - 5c1bb1a0-ea3e-4ced-b89f-1f88253869e3
    bookTitle:
      description: For convenience, the book title is stored besides the
identifier
      type: string
      minLength: 2
      macLength: 255
      examples:
        - Patterns and Tricks
    owner:
      description: Owner of the notes who is allowed to change them, given
by their member ID
      type: string
      format: uuid
      examples:
        - 9963a0b9-934e-41ec-bf62-af2c1669264e
    ownerName:
      description: For convenience, the owner is stored by their name to be
shown e.g. in discussions
      type: string
      minLength: 2
      maxLength: 65
      examples:
        - Miller
    notes:
      type: array
      minItems: 0
      maxItems: 255
      items:
        $ref: '#/components/schemas/Note'
```

The notes array does not only contain notes; it includes their owner and the book to which they belong. This shows that formulating the resource container as its own object and not just as an array helps to avoid breaking changes in the API in the future. When additional properties for the resource container are necessary in the future, they can be added to the already existing object without creating breaking changes.

DDD helped formulate an expressive and well-understood notes API because it crystallized the boundary between member management, catalog management, and notes.

Conclusion

RESTful APIs as synchronous APIs can be formulated from the strategic and tactical design results. Using the ubiquitous language defined gives the APIs a high expressiveness. Client developers can easily understand the business meaning of the APIs. The APIs are structured. They allow not only access to aggregates inside of a bounded context. They enable access to entities contained in aggregates as well by using sub-resources.

As we saw in the strategic design, we need synchronous APIs on the one hand to design successful and modern applications. However, we need asynchronous APIs too.

In the next chapter, we will discuss how to expose the found domain events using AsyncAPI.

Points to remember

- Use the ubiquitous language of a bounded context to formulate APIs.
- All resource names, properties, etc., must follow the ubiquitous language.

  In such a way, it is possible to understand the API well without spoiling the domain model via the API.

- Use the aggregate of a bounded context as a resource in a REST API.

  An aggregate is the essence of a bounded context. It is represented by the API resource and can be easily accessed by a URI.

- Use sub-resources to access entities referenced in aggregates.

  Sub-resources are entities inside the aggregates. URIs can also access them, but only via the resource containing them. Using sub-resources for entities allows the domain model to be kept clean.

- Use standard methods and error codes. For RESTful APIs, use HTTP verbs and HTTP response codes.

  The standards must be used to be understood well. That means using HTTP verbs as they are meant. To avoid side effects, well-defined HTTP response codes also need to be used.

- Use methods of DDD to model threads in an application.

- Avoid using path variables when they do not belong to the boundary context of the API.

  Addressing entities that do not belong to the bounded context—even not as sub-resources—can spoil the domain model. Therefore, it is advisable to use them as query parameters. The query parameter can be used to search for the appropriate aggregate.

Multiple choice questions

1. **When do you use sub-resources in a RESTful API?**
   a. When you want to access aggregates of another bounded context
   b. When you want to access entities referenced in an aggregate
   c. When you do not know the aggregate
   d. When you reference a value object, you do not know

2. **When do you use RESTful APIs?**
   a. To express synchronous behavior
   b. To express asynchronous behavior
   c. To design HTML pages
   d. Only for internal APIs, no other need to understand

3. **Why do you use the ubiquitous language in an API?**
    a. Because my boss ordered me to do so
    b. Because I do not find other words
    c. Because I do not want others to understand my API
    d. Using the ubiquitous language makes the API expressive
4. **When do you use a path parameter in a REST API?**
    a. When a resource should be addressed directly
    b. When a sub-resource cannot be addressed directly
    c. When a value object should be deleted
    d. When difficulties appear in the bounded context
5. **When do you use a query parameter instead of a path variable?**
    a. When the addressed resource belongs to the same bounded context
    b. When the addressed resource does not belong to the same bounded context
    c. When the addressed resource does not exist and needs to be created
    d. When the addressed resource has not been initiated yet

Answers

1. **b**
2. **a**
3. **d**
4. **a**
5. **b**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 12

Exposing Domain Events

Introduction

Modern architectures rely heavily on transferring events in an event-driven architectural style. Events can be defined based on the results of the event storming workshops using the context map as documentation. Using a standardized definition language allows events to be defined independently of implementation details and broker technologies.

Structure

In this chapter, we will cover the following topics:

- Introduction to event-driven architecture
- Broker architecture
- Defining events using AsyncAPI
- Message definitions of the online library
- Definition of the services

Objectives

In this chapter, you will learn how event-driven architectures rely on events. Furthermore, results of the event storming workshops. After this chapter, you can define messages and services in an event-driven architecture using AsyncAPI.

> **Strong minds discuss ideas, average minds discuss events, weak minds discuss people.**
>
> *-Socrates*

Introduction to event-driven architecture

This chapter will discuss the events to be exposed in the online library application. To do so, we first discuss the general approach of an event-driven and broker architecture.

Afterward, we investigate AsyncAPI, the standard for defining events.

In the third part of this chapter, we discuss the events in detail.

Let us explore event-driven architecture. Nowadays, it is the de facto standard for modern and successful architecture. It is highly decoupled and flexible, which allows for good maintainability and a fast time-to-market.

An event-driven architecture uses events to trigger and communicate between decoupled services. An event is a change in state or an update of an aggregate[1].

An event-driven architecture can be best described as a West-East communication style[2], as shown in *Figure 12.1*:

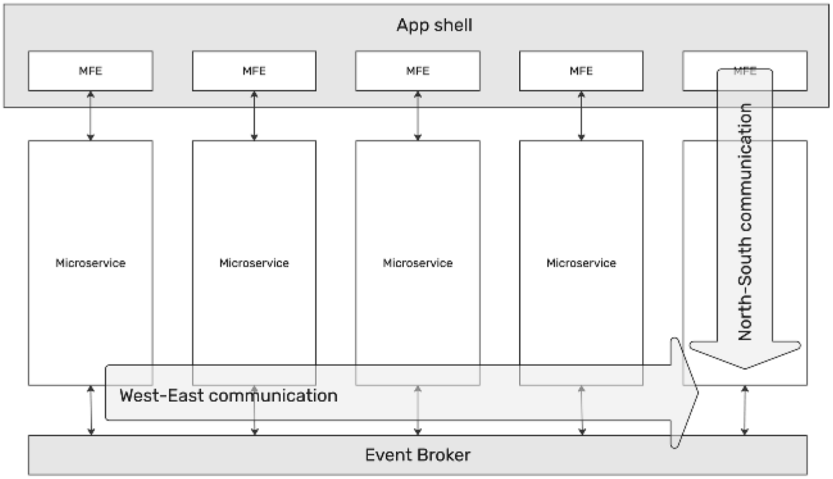


Figure 12.1: *Event-driven architecture with West-East and North-South communication*

Source: *EventDrivenArchitecture.jpg*

---

[1] What is an event-driven architecture? (2024). Retrieved August 2024, from AWS: **https://aws.amazon.com/event-driven-architecture/#:~:text=An%20event%2Ddriven%20architecture%20uses,on%20an%20e%2Dcommerce%20website**.

[2] Junker, A. (2023, May 16). Eventgetriebene Integrationsarchitekturen. Retrieved June 2024, from Informatik aktuell: **https://www.informatik-aktuell.de/entwicklung/methoden/eventgetriebene-integrationsarchitekturen.html**

As discussed in *Chapter 9, Introduction to Tactical Design, a bounded context can be designed using hexagonal architecture*. The microservices expose the functionality as **micro-frontend** (**MFE**). The communication between MFE and microservice is synchronous and designed as RESTful APIs. Usually, the user interface is drawn on the upper part of an architectural sketch and the persistence layer on the lower part; such communication is called **North-South communication**; it reminds us of a map where the North is shown on the upper part.

The microservices communicate via events. The services are drawn from left to right, which is called **West-East communication**. It reminds us of a map where the West is dawn on the left side[3]. The corresponding architecture pattern used here is the messaging pattern[4]. The West-East communication is done via an event broker. The microservices produce and consume events. The events contain information about the status changes of aggregates corresponding to the bounded context they represent.

Event-driven architecture improves responsiveness, flexibility, and extensibility but introduces complexities like operational overhead, event ordering challenges, and the need for effective event modeling and management[5].

In this chapter, we will discuss event modeling. However, let us examine broker architecture in more detail and introduce specific terms.

Broker architecture

Different projects implement event brokers. The most used are Apache Kafka[6] and RabbitMQ[7]. Both are widely used.

A typical broker architecture is shown in *Figure 12. 2*:

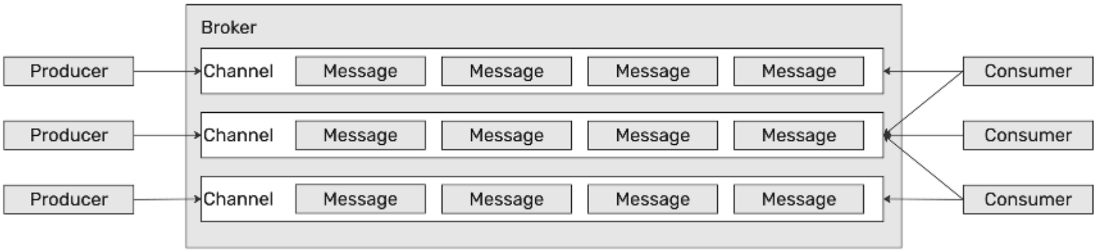


***Figure 12. 2**: Typical broker architecture*

***Source**: BrokerArchitecture.jpg*

---

[3] Junker, A. (2023, May 16). Eventgetriebene Integrationsarchitekturen. Retrieved June 2024, from Informatik aktuell: **https://www.informatik-aktuell.de/entwicklung/methoden/eventgetriebene-integrationsarchitekturen.html**

[4] Richardson, C. (2024). Pattern: Messaging. Retrieved August 2024, from Microservice Architecture: **https://microservices.io/patterns/communication-style/messaging.html**

[5] What is event-driven architecture? (2024). Retrieved August 2024, from Confluent: **https://www.confluent.io/learn/event-driven-architecture/**

[6] Apache Kafka. (2024). Retrieved August 2024, from Kafka: **https://kafka.apache.org/**

[7] RabbitMQ. (2024). Retrieved August 2024, from RabbitMQ:

A broker architecture contains the following:

- **Producers**: As the name suggests, producers produce messages for the broker, who stores them.
- **Broker**: The broker is responsible for receiving and storing messages for consumers to read. It scales its channels and distributes them according to the requirements. In an event-driven architecture, a broker is the most critical infrastructure component. It does not have any business logic.
- **Channel**: A channel is a part of a broker architecture. It stores certain kinds of messages. Please be aware that a channel is called differently in different implementations. We use the term channel to be provider independent. In Kafka, a channel is called **Topic**[8], whereas the channel in RabbitMQ is called **Queue**[9]
- **Message**: A message is an event sent to the consumer via the broker. It contains header and payload information, which can be sent in different formats, such as Avro[10], Protobuf[11], JSON[12], etc.

How to expose domain events via an event-driven architecture should be enough. Be aware that further elements are necessary to set up in both implementations. However, from a logical point of view, the parts, broker, channel, and message are sufficient.

Kafka and RabbitMQ implement two different approaches to event-driven architecture. Kafka follows the event-streaming approach, whereas RabbitMQ implements a message broker following the publish-subscribe approach. Let us discuss the two different approaches briefly.

Event-streaming

Event streaming is an approach where the producer service continuously sends events, which are consumed in a timely sequence. This approach allows the building of real-time systems capable of handling a high data load.

---

[8] Akin, E. (2023, November 18). Kafka Architecture. Retrieved August 2024, from Medium: **https://medium.com/@cobch7/kafka-architecture-43333849e0f4**

[9] Johansson, L. (2019, 09 23). Part 1: RabbitMQ for beginners - What is RabbitMQ. Retrieved August 2024, from CloudAMQP: **https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html**

[10] Apache Avro(TM). (2024). Retrieved August 2024, from Apache Avro(TM): **https://avro.apache.org/**

[11] Protocol Buffers. (2024). Retrieved August 2024, from Protocol Buffers Documentation: **https://protobuf.dev/**

[12] ECMA-404 The JSON interchange syntax. (2017, December). Retrieved August 2024, from ECMA International:

The events are stored on the broker and read by the consumer. The consumer does not need to be up and ready when the broker manages an event; the consumer can read all events when ready. Using a marker, the last read event is marked so that the consumer can detect not-yet-read events.

Kafka implements the event-streaming principle[13].

Publish and subscribe

In the publish and subscribe approach, a publisher produces a message and sends it to the broker. The broker receives the message and queues it to the subscriber. The message is deleted the moment the subscriber has read the message.

RabbitMQ implements this approach[14].

Both approaches are appropriate to implement an event-driven architecture.

Note: Please be aware that the event-sourcing approach means storing small changes, which allows the creation of a full state by replaying the single changes. Those approaches are used in analytics systems, which we do not discuss here.

The following section will discuss how to specify events or messages using AsyncAPI.

Defining events using AsyncAPI

AsyncAPI[15] is an excellent possibility for specifying domain events for an event-driven architecture. The standard is oriented on OpenAPI, so you will feel quite familiar with it.

---

[13] Event Streaming: How it Works, Benefits, and Use Cases. (2024). Retrieved August 2024, from Confluent: **https://www.confluent.io/learn/event-streaming/**

[14] Johansson, L. (2019, 09 23). Part 1: RabbitMQ for beginners - What is RabbitMQ. Retrieved August 2024, from CloudAMQP: **https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html**

[15] 3.0.0. (2023, December 5). Retrieved August 2024, from AsyncAPI: **https://www.asyncapi.com/docs/reference/specification/v3.0.0**

Structure of an AsyncAPI

The structure of an AsyncAPI document is shown in *Figure 12.3*:

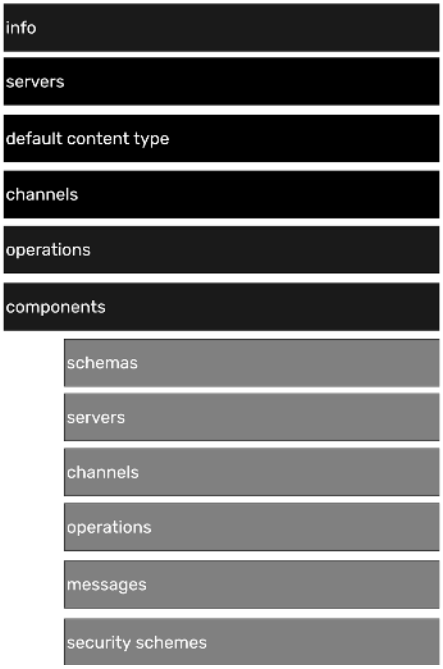


Figure 12.3: *Structure of an AsyncAPI document*

Source: *AsyncApiStructure.jpg*

The *info* block contains information about the title and the version of the specification. Compared to OpenAPI, a description can be formulated in Markdown[16]. Additionally, a contact needs to be given.

The *servers* part contains the broker addresses, e.g., different addresses for different stages.

The corresponding content type can be defined with *default content type*, e.g., JSON.

The *channels* part defines, e.g., the topics in Kafka.

The *operations* part defines which operations are possible on the channels from the point of view of the described service. It contains corresponding actions: **`send`** and **`receive`**.

The *components* part works as we saw in *Chapter 11, Exposing Aggregates via RESTful APIs*.

The schema definition mainly works, as shown in *Chapter 11*.

---

[16] Cone, M. (2024). Markdown Guide. Retrieved August 2024, from Markdown Guide: **https://www.markdownguide.org/**

We do not discuss further technical definitions, such as operation and channel bindings; please refer to the documentation of AsyncAPI[17] and your preferred broker.

Skeleton of an AsyncAPI specification

As for synchronous communication, we also give a specification skeleton[18] for asynchronous communication too.

Let us start with a typical info block, which is quite like the info block of an OpenAPI. This is shown in the following snippet:

```
info:
  title: Skeleton
  version: 1.0.0
  description: |
    Skeleton for an AsyncAPI definition
  contact:
    name: Annegret Junker
    email: annegret.junker@gmx.de
```

The servers part contains the different brokers, shown as follows:

```
servers:
  production:
    # other messaging brokers needs to be defined - look in the
documentation
    host: "events.myonline-library.com:9092"
    protocol: "kafka"
    description: "Production broker"
  testing:
    # other messaging brokers needs to be defined - look in the
documentation
    host: "events.testing.myonline-library.com:9092"
    protocol: "kafka"
    description: "Testing broker"
```

As said, the *servers* part contains the brokers used for the asynchronous events, which are given here as Kafka brokers. The other parts of the definition can be discussed using the definition of the domain events of the online library. We will discuss those in the next section.

---

[17] 3.0.0. (2023, December 5). Retrieved August 2024, from AsyncAPI: **https://www.asyncapi.com/docs/reference/specification/v3.0.0**

[18] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter12-ExposingDomainEvents/AsyncApis/AsyncSkeleton.yaml**

Message definitions of the online library

The domain events of the online library were defined using event storming and a context map. Let us review the context map (see *Chapter 7, Context Map*). For convenience, the context map is repeated in *Figure 12.4*:

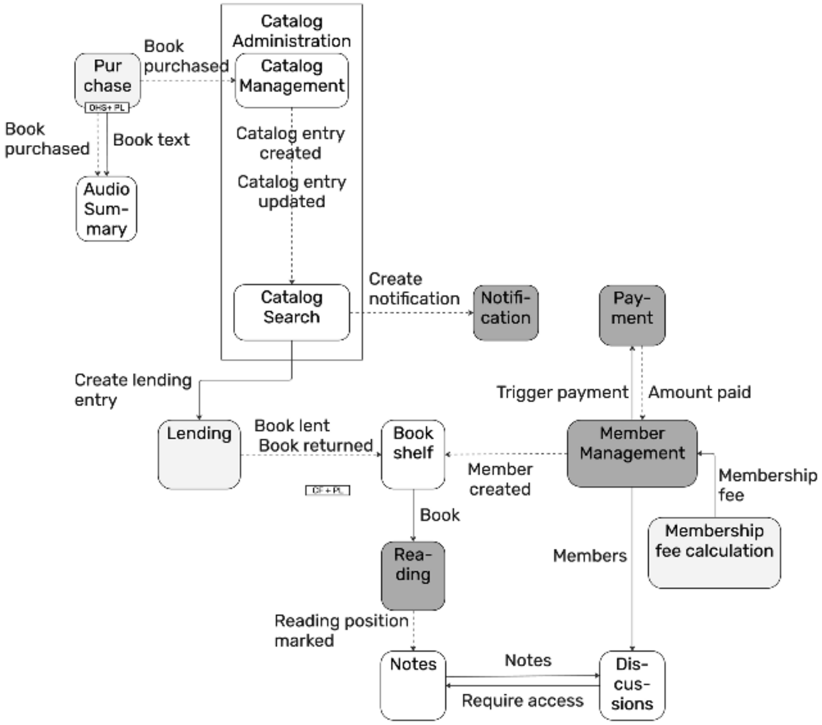


***Figure 12.4**: Context map of the online library*

***Source**: ContextMapOnlineLibrary.jpg*

The context map shows the synchronous data exchange (straight arrows) and the asynchronous data exchange (dashed arrows).

The following events needs to be defined:

- **Book purchased**: The **`bookPurchased`** event is produced by *Purchase* and consumed by *Audio Summary* and *Catalog Management*.
- **Catalog entry created, catalog entry updated**: The events **`catalogEntryCreated`** and **`catalogEntryUpdated`** are necessary to implement the CQRS pattern in the catalog management.
- **Create notification**: **`createNotification`** is a command that creates a notification in the *Notification* context. The command is sent from the *Catalog Search* to *Notification*.

- **Book lent, book returned**: The events `bookLent` and `bookReturned` are produced by Lending and consumed by Bookshelf.
- **Member created**: The `memberCreated` event is produced by *Member Management* and consumed by *Bookshelf*.
- **Amount paid**: The `amountPaid` event is thrown by the payment provider and will be defined. The anti-corruption layer of the *Payment* context will create a generic payment event, so it would be possible to integrate multiple payment providers.
- **Reading position marked**: The event `readingPositionMarked` is produced by *Reading* and consumed by *Notes*.

The events are named according to the ubiquitous languages of the corresponding bounding contexts found during the event storming workshops. Therefore, they can be easily recognized by developers and business experts. Their function can be traced back to the event storming documentation; please check the results in *Chapter 6, Event Storming*.

First, we create an AsyncAPI only containing the message definitions. These can be used in the service definitions.

Pure message definitions of the library

In the following section, we will only define the headers and payloads of the necessary messages.

The header for all messages is the same.

Header of the messages

The header contains two identifiers to track messages. One identifier is a correlation identifier that correlates different events in one business process. The second identifier is the message's identifier. Usually, message identifiers are created by the brokers, but here, we added a provider-independent message identifier.

Additionally, the name of the producer of the message as a source with its bounded context is given as follows:

```
MessageHeader:
  description: Header Information for each message
  type: object
  required:
    - correlationId
    - messageId
  properties:
    correlationId:
      description: Identifier to be set by application to trace end-to-end-
```

```
api-journey
      type: string
      format: uuid
      examples:
        - b3416449-5f69-4cd2-a6c4-1ff82da96bce
    messageId:
      description: Identifier for the particular message, is needed in case
of cascading infrastructure
      type: string
      format: uuid
      examples:
        - 5a1d305b-19f3-46eb-ada1-9a9770e56fd1
    source:
      description: Bounded context name creating the message
      type: string
      minLength: 2
      maxLength: 20
      examples:
        - Catalog Management
```

The message header is already given in the skeleton.

Book purchased

The **`bookPurchaseEvent`** contains all the properties necessary for a new catalog entry. Additionally, the number of text positions is given to give the *Audio Summary* context an idea of the book's size. It may be interesting for which audience the book is intended. An according property is given as follows:

```
BookPurchased:
  description: Message which is sent when a new book was successfully
purchased
  type: object
  required:
    - authors
    - title
    - isbn-10
  properties:
    authors: …
    title: …
    subtitle: …
    publisher: …
    blurb: …
    isbn-10: …
    isbn-13: …
    numberOfTextPositions:
      description: Number of text positions overall in the book
      type: integer
      format: int64
      examples:
        - 327680
    audience:
      description: Audience for which the book is thought
      type: string
      enum:
        - STUDENT
        - TEACHER
        - ADVANCED STUDENT
        - BEGINNER
```

The payload is given as a full-state event to allow the consumers to create the aggregate for which they are responsible.

The according message is defined as **`bookPurchasedEvent`**.

For the message, a separate channel is defined, shown as follows:

```
BookPurchasedEventChannel:
  description: Channel where messages are stored when a book was purchased
  address: book-purchased
  messages:
    BookPurchasedEvent:
      $ref: '#/components/messages/BookPurchasedEvent'
```

The channels for the other messages are defined similarly. It is advisable to define a separate channel for each message to support monitoring and maintaining the channels and messages.

Catalog entry created, catalog entry updated

The creation or update of a catalog entry needs to be transferred to the catalog search. The message payload is the complete catalog entry. It is also a full-state event, so that the search can update and index the entries accordingly. A change event, which only contains the changed data, is insufficient. After all, it would require expensive change handling on the catalog search site, where single changes must be handled to the atomic aggregate. The corresponding events are defined as created and updated event, shown as follows:

```
CatalogEntryCreatedEvent:
  headers:
    $ref: '#/components/schemas/MessageHeader'
  payload:
    $ref: '#/components/schemas/CatalogEntryChanged'

CatalogEntryUpdatedEvent:
  headers:
    $ref: '#/components/schemas/MessageHeader'
  payload:
    $ref: '#/components/schemas/CatalogEntryChanged'
```

The update and creation events are formulated separately, even though the payload is identical. This allows later changes when both are different. Then, only one consumer needs to be adapted, not both. In such a way, greater flexibility in future changes can be achieved.

Book lent, book returned

The events in the *Lending* context can be lean. They only contain the book identifier and the member identifier who lends or returns the book. Further information for convenience reasons is not necessary because the events are used in server-to-server communication. The event only contains the identifier of the book and the member who want to borrow it, as shown in the following code:

```
BookLentReturned:
  description: Book lent or returned by a member
  type: object
  required:
    - catalogId
    - memberId
  properties:
    catalogId:
      description: Catalog identifier of the book lent or returned
      type: string
      format: uuid
      examples:
        - b7aa9669-6995-4793-ad2d-c9fd958bc082
    memberId:
      description: Member identifier of the member who lent or returned the
book
      type: string
      format: uuid
      examples:
        - 30f5421b-9e0c-4c6a-9723-3d4da90c2c54
```

Comparable to the catalog entry created and the update events, the book lent and returned events are formulated separately even though they contain identical payloads. Possible future changes can be made more flexibly and do not cause breaking changes. The events looks as follows:

```
BookLentEvent:
  headers:
    $ref: '#/components/schemas/MessageHeader'
  payload:
    $ref: '#/components/schemas/BookLentReturned'

BookReturnedEvent:
  headers:
    $ref: '#/components/schemas/MessageHeader'
  payload:
    $ref: '#/components/schemas/BookLentReturned'
```

Member created

When a new member is created, a new bookshelf needs to be created. It requires the member identifier and, for convenience reasons, family name and email address. The event contains typical member information as shown in the following code:

```
MemberCreated:
  MemberToBeCreated:
    description: Member to who was created, the member was created after
the payment fee has been paid
    type: object
    required:
      - id
      - emailAddress
    properties:
      id: …
      familyName: …
      emailAddress: …
```

The message is defined as **`MemberCreatedEvent`** with **`MemberCreated`** payload and the header.

Amount paid

When an amount is paid, the payment context can send a message via the necessary anti-corruption layer. The message needs to contain the payment amount and the currency in which the payment was made. Additionally, a correlation identifier is needed. The identifier marks the initial payment request. It could be a one-time request or a request for scheduled payments. The **`Amount`** definition and the **`AmountPaid`** event is shown in the following code:

```
Amount:
  description: Amount of a fee
  type: object
  required:
    - number
    - currency
  properties:
    number:
      description: Number of the amount transferred
      type: number
      format: float
      examples:
        - 12.34
    currency:
      description: Three character acronym of the used currency according
to ISO 4217
      type: string
      minLength: 3
      maxLength: 3
      examples:
        - EUR
        - INR

AmountPaid:
  description: Event when an amount as an member fee has been paid
  type: object
  required:
    - amount
    - requestId
  properties:
    amount:
      $ref: '#/components/schema/Amount'
    requestId:
      description: |
        Identifier of the request for the payment
        The identifier is unique for one time payments and remains for
scheduled payments
      type: string
      format: uuid
      examples:
        - 68818400-bdc3-409f-92c9-51398311fb8d
```

The message is defined as **`AmountPaidEvent`**.

Reading position marked

When a user marks a reading position, the according position in the specific book needs to be sent to the *Notes* context. Therefore, the event contains the text positions and the book as catalog entry identifiers. The definition of the marking is shown in the following code:

```
ReadingPositionMarking:
  description: Marking of a reading position, e.g., to create notes
  type: object
  required:
    - textPositionStart
    - textPositionEnd
  properties:
    bookIdentifier:
      description: Identifier of the catalog entry to which the book read
referenced
      type: string
      format: uuid
      examples:
        - e4667e38-0dd2-481a-b755-69d0ad136fa0
    textPositionStart:
      description: Start of the marking
      type: integer
      format: int64
      examples:
        - 1024
    textPositionEnd:
      description: End of the marking - needs to be higher or equal to the
start position
      type: integer
      format: int64
      examples:
        - 2048
```

The respective event **`ReadingPositionMarked`** contains the payload and the header.

Definitions of the services

The asynchronous APIs for each service in the online library can be defined using the broker channels defined in the *Messages* specification[19].

---

[19]

The according definition is shown as the sample for Catalog Management. Catalog Management consumes the bookPurchasedEvent from Purchase and produces the events **`catalogEntryCreated`** and **`catalogEntryUpdated`**. The following code shows the operations definitions for the *Catalog Management* component:

```
operations:
  BookPurchasedEventReceive:
    action: "receive"
    channel:
      $ref: "../Messages.yaml#/channels/BookPurchaseEventChannel"

  CatalogEntryCreatedEventSend:
    action: "send"
    channel:
      $ref: '../Messages.yaml#/channels/CatalogEntryCreatedEventChannel'

  CatalogEntryUpdatedEventSend:
    action: "send"
    channel:
      $ref: '../Messages.yaml#/channels/CatalogEntryCreatedEventChannel'
```

Only the corresponding operations need to be defined for the services. It must only be determined if a service consumes or produces messages according to the defined channels.

The separated file **`Messages.yaml`** can be referenced, and the **`#`** sign can be used to reference keys in the corresponding AsyncAPI file.

Conclusion

Events and commands can be elegantly defined using the ubiquitous language specified in the event storming workshops. The events and commands are expressive and can be received easily by developers and business experts.

Using a Domain-Driven Design, it seems easy to define synchronous and asynchronous APIs in OpenAPI[20] and AsyncAPI[21]. However, a couple of rules can help avoid a couple of pitfalls. The next chapter will discuss those pitfalls and the corresponding rules.

Points to remember

- Define events using the found domain events in the context map.
- Define messages with header and payload information.

---

[20] OpenAPI. (2022). The world's most widely used API description standard. Retrieved July 2024, from OpenAPIs: **https://www.openapis.org/**

[21] 3.0.0. (2023, December 5). Retrieved August 2024, from AsyncAPI: **https://www.asyncapi.com/docs/reference/specification/v3.0.0**

- Consider whether a consumer needs the full-state information of an aggregate or if only the references are sufficient.
- When references are sufficient, consider whether some additional information is convenient.
- Define channels with the messages in mind. Define one channel per message.
- Define the services that use the defined channels.

Multiple choice questions

1. **When services communicate with each other, which communication form should they use?**
    a. Events
    b. Synchronous events
    c. Asynchronous REST
    d. Proprietary notifications
2. **When defining messages, should you use just the definition of the REST APIs?**
    a. No, I need to think about if a full state is necessary
    b. Yes, I can spare time
    c. No, I only should use plain references
    d. Yes, the full state of an aggregate is always necessary
3. **When defining a channel, do you need to know the underlying technology?**
    a. Yes, otherwise, I would define too many channels for one message
    b. Yes, I cannot define a channel without the broker technology because of the naming conventions
    c. Yes, I need to know the technology to know the maximum number of channel
    d. No, I can define a channel technology-abstract
4. **Is it advisable to define one channel per message?**
    a. No, the overhead of broker management is too high
    b. No, the usage of channels needs to be paid in a cloud infrastructure
    c. Yes, it makes the debugging and monitoring easier
    d. No, I do not know why

Answers

1. **a**
2. **a**
3. **d**
4. **d**

Bibliography

1. 3.0.0. (2023, December 5). Retrieved August 2024, from AsyncAPI: **https://www.asyncapi.com/docs/reference/specification/v3.0.0**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 13

Pitfalls in Tactical Design

Introduction

Tactical design can be tiresome and sometimes seems over-engineered. However, better maintenance and faster onboarding of new team members dramatically increase the rewards. In *Chapters 9* to *12*, we saw the elements of tactical design. We discussed the classical pattern of an entity and repository. We saw how we can expose aggregates via RESTful APIs and how to expose domain events. In this chapter, we will discuss which pitfalls lie on the way to a tactical design of a software application.

Structure

In this chapter, we will discuss the following topics:

- Tactical design and team responsibility
- Pitfalls in tactical design and how to avoid them

Objectives

In this chapter, you will learn how to perform tactical design on a daily basis, how to avoid pitfalls such as solving someone else's problems, and why tactical design is rewarding and worthwhile.

> **We are not fit to lead an army on the march unless we are familiar with the face of the country—its mountains and forests, its pitfalls and precipices, its marshes and swamps.**
>
> *-Sun Tzu*

Team responsibility and tactical design

The tactical design needs to be discussed with the team responsible for the corresponding bounded context.

Starting from the aggregate, the necessary adapters to the user interface via REST interfaces, to other services via domain events, and the persistence layer via a repository can be defined. The team can discuss the necessary structures and interfaces.

The team needs constant support from a business representative, e.g., a product owner. The representative must ensure that the ubiquitous language is representative in all layers—the persistence layer, domain model, business logic, and interfaces.

The team must ensure data consistency, even though some mappings from a reach data model to a lean data model for the interfaces are necessary. The ubiquitous language and the documented models from the domain storytelling and event storming help.

The team should test the interfaces and the language used to see if they are still understandable and if they express the necessary data. That can be done in reverse domain storytelling. A product owner reads the REST interface and tries to tell the domain story out of it. If the story is like the original one, the implementation is still consistent with the business requirements. In the next section, we will discuss how the teams can test the consistency using **large language models** (**LLMs**).

Check implementation consistency by LLMs

The consistency of an API is essential. A client developer needs to understand the API well. Inconsistencies developed over time in the lifecycle of the API cause high maintenance and support effort. Therefore, it is necessary to check the consistency.

LLMs can also do those consistency tests.

To create such a test the following prompt could be used:

Create a domain story after Hofner, using the following interface definition[1].

The answer by ChatGPT looks then as the following:

- **Title**: Managing tasks efficiently:
    - **Actors**:

---

[1] OpenAPI definition: **https://github.com/Grinseteddy/MasteringDdd/blob/main/RestApis/TaskManagement/TaskManagement.yaml**

    - **User (Requester)**: A person who defines a task, assigns it to someone, and monitors its progress.
    - **Assignee**: A person responsible for completing a task assigned by the requester.
    - **System**: Task Management Service that facilitates task creation, updates, deletion, and retrieval.

- **Events and interactions**:
  - **Task creation**:
    - The User wants to create a new task. They log into the Task Management Service via an OpenID Connect authentication flow to ensure they have the necessary permissions (`resource:write`).
    - They navigate to the task creation page and fill out the necessary fields: `title`, `description`, `dueDate`, and choose an assignee from the list of available users.
    - The User sends a POST request to the/tasks endpoint, including the task details in the request body.
    - The System validates the request, checks the API version in the header (x-version), and creates the task, assigning it a unique id and setting the initial status to `"NEW"`.
    - The System returns a response with status 201 (Created) and includes a link to the newly created task (`LinkToTaskResponse`).
  - **Task assignment**:
    - The System sends a notification to the Assignee, informing them about the new task assignment. The Assignee can view the task details by making a GET request to `/tasks/{taskId}`.
    - The Assignee logs in and accesses the task using the provided link, ensuring they have `resource:read` permission.
  - **Viewing tasks**:
    - The User wants to see all tasks matching specific criteria, such as tasks assigned to a particular user or tasks requested by them.
    - They make a GET request to the /tasks endpoint with various search parameters (`title`, `assignee`, `requester`, etc.) provided in the query.
    - The System retrieves and returns a list of tasks (`TaskListResponse`) that match the search criteria, or returns an error (`BadRequestResponse`, `ForbiddenResponse`, or `ServiceNotAvailableResponse`) if the request is invalid or the service is unavailable.

  - **Task status update**:
    - The Assignee begins working on a task. They change the task status to `"IN_PROGRESS"` by sending a PUT request to `/tasks/{taskId}`, including the updated task information.
    - The System checks that the Assignee has the necessary permissions (`resource:write`) and updates the task's status accordingly.
    - Upon successful update, the System responds with status 200 and includes a link to the updated task (`LinkToTaskResponse`).
  - **Task completion or rejection**:
    - When the task is completed, the Assignee marks it as "DONE". If they are unable to complete it, they mark it as "REJECTED".
    - A PUT request is sent to `/tasks/{taskId}` with the new status and any relevant comments.
    - The System processes the update and sends notifications to the User about the task's new status.
  - **Task deletion**:
    - The User decides to delete a task that is no longer needed.
    - They send a DELETE request to `/tasks/{taskId}`. The System verifies that the User has the necessary permissions (`resource:admin`).
    - If authorized, the System deletes the task and responds with status 200 (`SuccessfulOperationResponse`). If not authorized, a 403 (`ForbiddenResponse`) is returned.
  - **Error handling**:
    - If the User or Assignee makes an invalid request, such as providing an incorrect task ID or missing required fields, the System returns an appropriate error response (400 for Bad Request, 404 for Not Found, 500 for Service Not Available, etc.).
  - **Task retrieval by ID**:
    - The User or Assignee wants to view detailed information about a specific task.
    - They send a GET request to `/tasks/{taskId}`. The System retrieves the task details if the User has the necessary `resource:read` permissions and returns the task information (`SingleTaskResponse`).

- **Key scenarios**:
  - **Scenario 1**: A User creates a new task and assigns it to an Assignee.

  - **Scenario 2**: An Assignee retrieves all tasks assigned to them and updates the status of a task to "IN_PROGRESS".
  - **Scenario 3**: A User deletes a task that is no longer relevant.
  - **Scenario 4**: The System handles errors gracefully, ensuring that only valid and authorized operations are performed.

- **Security and permissions**:
  - **Authentication**: Managed via OpenID Connect with different permissions (task:read, task:write, task:admin).
  - **Authorization**: Enforced based on the roles of User and Assignee.[2]

Using those domain stories is helpful to check if the original domain story is still consistent with the implemented interface.

Ubiquitous language

Similarly, the ubiquitous language can be checked. For example, the API definition given in the wrong language[3] can be compared to the visual glossary; see *Figure 13.1*:

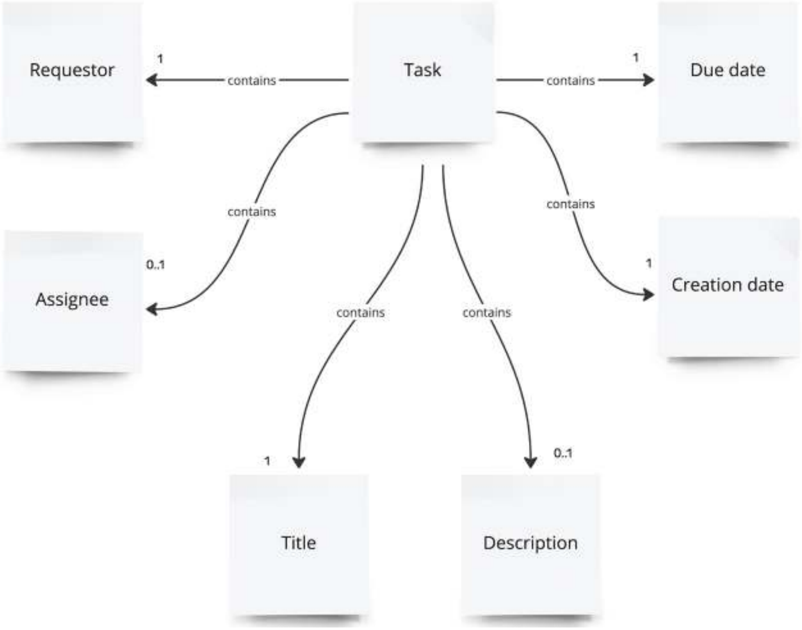


Figure 13.1*: Visual glossary of task management*

Source*: TaskManagementVisualGlossary.jpg*

---

[2] Conversation ChatGPT, 30.8.2024, **https://chatgpt.com/share/c7c75e0e-6945-41a8-9728-87e2e8884016**

[3] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter11-ExposingAggregatesViaA-**

Comparing both, in the API specification, Task is called Nothing, and Assignee is called Layabout.

The sample may be artificial. However, it shows that even though the visual representation is rough, it can be compared to the code. This is especially true for interfaces that can be reviewed by business representatives who recognize the business in the ubiquitous language.

The consistency check helps to avoid the pitfalls discussed in the following section.

Pitfalls and how to avoid them

Pitfalls in tactical design appear when the Domain-Driven Design cannot be applied as theoretically required. Often, only parts of it are implemented, especially in tactical design. Therefore, we want to discuss some of those pitfalls and how they can be avoided.

Driven by database design

In most applications, database design is technically demanding. A lot of data needs to be written quickly, and indices must be created to facilitate fast reading, etc. To manage such a challenging task takes a lot of mental capacity. Usually, the business-driven design stays behind the technical design. The structures are driven by technical decisions and less by business.

Those situations cannot be avoided. However, when the database design defines the domain model, the domain model becomes weak and less expressive. A library member is not called a member anymore; they are called a **user**. From a technical point of view, it is correct because even a library member is just a user with a specific role. From a domain model perspective, it is dramatic because, in an interface, each library user—librarians, HR clerks, and sales representatives could be invited to a discussion. Of course, the discussion service must check the provided role in the **JavaScript Object Notation** (**JSON**) **Web Token** (**JWT**) token, but a member should not be able to invite employees instead of members in the first place. Those mistakes can easily be avoided using ubiquitous language.

The solution is also easy. Applying the repository pattern can avoid the database model's weakening of the domain model.

Purely out of the provider’s point of view

Representatives of the function provider and consumer side should attend. When a tactical design is done inside the provider team, the consumer side is usually no longer available for discussions. Therefore, it happens pretty often that the interfaces are only formulated from the provider’s point of view.

The catalog management provider might provide the same interface for books, audiobooks, and videos. On the client side, those should be handled differently. Therefore, a client would wish for different endpoints, even though the differences for the provider are not so strong.

This pitfall is not as easy to avoid as the previous one. The provider side must think like the client. Usually, it helps if the user interface implementation is part of the team because the user interface developers have the user journey in their heads and know which differences are critical and which are not. Anyhow, when you write an interface, you must ask yourself if you would like to use it or not. When you do not like it, change it as long as you like it.

Purely technical solutions

Sometimes, Domain-Driven Design is established only by the implementation teams without the strategic design of a leading architecture team (see *Chapter 14, Usage of Domain-Driven Design in a Green Field Project*). In those situations, the interfaces and domain models are more technical than business driven. We learned that the ubiquitous language in a bounded context is essential. Still, without strategic design, developing a ubiquitous language with a team of business experts and IT specialists is almost impossible.

The language will use short variable names like **`lm`** for a **`libraryMember`** or `entry` for a `catalogEntry`. Such more technical terms make the code difficult to grasp and delay the onboarding of new team members.

Using strategic design methodologies can avoid those obstacles. Use domain storytelling and event storming together with a visual glossary to develop your design and language, even when you are only in one team responsible for one bounded context. The onboarding process will be much faster, and business experts will understand your code.

The team's product owner or business analyst can play the role of the business expert. They can attend the workshop. A senior developer of the team can take over the role of the moderator. So, the team can develop their ubiquitous language when no strategic design is available.

Anyhow, it is always better to establish a strategic design. It can be done by the architecture guild across multiple teams.

Even if there is only one team for a small application, it makes sense to perform a strategic design to ensure the application's modularization, shared understanding, and ubiquitous language.

Trying to solve someone else’s problem

When you work in a larger company, you face the following problem, most probably: A team implementing a bounded context is part of a more extensive user journey. Another team part of the same user journey cannot deliver on time. The first team wants the user journey to work and the user acceptance test to be delivered.

The solution might be to take over the responsibility and provide the function of the second bounded context in the first team. The first team accesses the second team's database, and the user journey can be completed. The ubiquitous language is used over the two bounded contexts and becomes more unspecific.

That is precisely how **Big Ball of Mud** (**BBOM**) arises. Afterward, it is difficult to separate the code again, and the functionalities are tied together for eternity.

To avoid such a situation, the first team needs to develop on behalf of the second team. They need to create a module that is strongly separated from their code so the second team can quickly take over later. They need to use the ubiquitous language of the second bounded context rather than their own, which makes the whole thing more difficult. A delegate from the second team might help to use the correct language. Anyhow, it is not ideal, but it avoids implementing a BBOM.

The described pitfalls show that DDD needs to play an essential role in daily work. DDD does not mean performing some workshops initially and inviting expensive consultants for a limited time. DDD means developing a structure of bounded contexts with their ubiquitous languages and using the structure and the languages on a daily basis.

Conclusion

We described a process for developing and implementing a software application using strategic and tactical design. DDD is an excellent tool for delivering better software in a shorter time. It enables businesses and IT to work together and meet customers' expectations.

Starting from almost nothing, using DDD on a daily basis is a long journey with several backlashes. However, it is worth starting. One can start even with small applications of tactical design and ubiquitous language in one team.

In the next chapter, we will see how DDD can be applied in a green field project in a start-up or a large enterprise.

Points to remember

- A team is responsible for a bounded context. They need to maintain the domain model and the corresponding ubiquitous language.
- An implementing team needs constant support from business representatives. The product owner or a business analyst can take over this role.
- The consistency of the developed models needs to be tested at regular intervals. The tests can be done manually or with the support of AI tools.
- The domain model needs to be protected from being weakened by the database. The pattern of a repository should be used.

- Interfaces and events should be formulated from the providers' and consumers' perspectives. They must be easy to understand and use.
- Even under pressure, context boundaries must be respected. Interfering in someone else's code is forbidden. The bounded contexts must be respected by using corresponding modules or repositories when necessary. The ubiquitous language must stay clean.

Multiple choice questions

1. **Should one team responsible for one bounded context change the language of another team responsible for another bounded context?**
    a. No, never.
    b. Yes, in times of project pressure, it should be possible.
    c. Yes, but only when the board members agree.
    d. Yes, it should be possible in both ways.
2. **Who should represent businesses in a development team?**
    a. Most senior developer
    b. Junior developer
    c. Product owner or business analyst
    d. External consultant
3. **Is it necessary to test the consistency of the code against the models after a longer development time?**
    a. No, the consistency is guaranteed by unit tests
    b. No, the consistency plays a more minor role after a longer development time
    c. No, the consistency is always given and does not need to be tested
    d. Yes, the consistency must be tested to avoid a weakened domain model

Answers

1. **a**
2. **c**
3. **d**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 14

Usage of Domain-driven Design in a Greenfield

Introduction

This chapter discusses how Domain-Driven Design can be applied to a green field project and methods for achieving a resilient and sustainable architecture when implementing a new product. It also discusses the documentation and implementation of a Domain-Driven Design process.

Structure

In this chapter, we will discuss the following points:

- Overview
- Setting up a workshop
- Finding target audience for the workshops
- Documenting workshop results
- Documentation of bounded contexts
- Documenting architectural decisions
- Importance of documenting software
- Evolving the DDD process

Objectives

You will learn how to start a greenfield project. You will learn how to develop a DDD process. You will also learn how to document the results of the workshops efficiently.

> **Life is like riding a bicycle. To keep your balance, you must keep moving.**
>
> *-Albert Einstein*

Overview

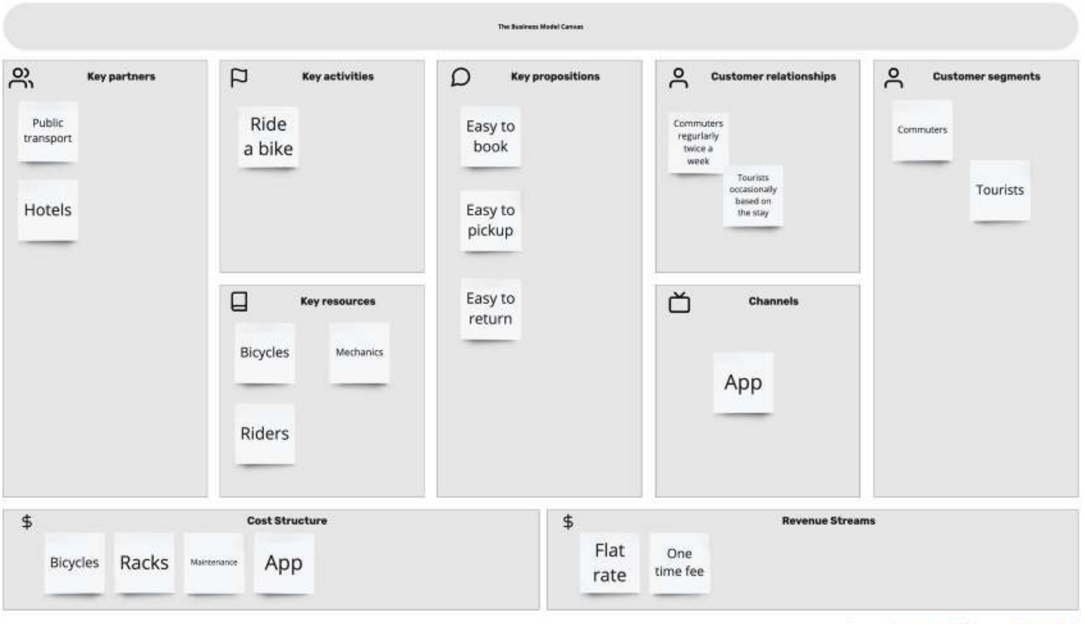
In this chapter, we will explore the operations of a sample bicycle rental shop designed to effectively serve commuters and tourists. This dual-service model not only enhances accessibility for local residents but also provides a convenient option for visitors looking to explore the area.

For commuters, the rental process offers seamless convenience. Bicycles can be rented directly at the train station, allowing commuters to ride home after a long day at work. This system is particularly beneficial for those who prefer to avoid traffic or crowded public transportation. Commuters have flexible options for returning bicycles when they finish their ride. They can either drop them off at a designated rack near their home or return them the following day when they return to the office. To streamline the rental experience, commuters pay a monthly fee that grants them access to the bikes, making it an economical choice for daily transportation.

On the other hand, tourists enjoy a different rental experience tailored to their needs. They can easily rent bikes directly from their hotels and simplify transportation while exploring the local area. Once the tourists request the bikes, the rental company delivers them to the hotel, ensuring a hassle-free experience. After their adventures, tourists simply return the bikes to the hotel. In this arrangement, the hotel charges the tourists for their rentals, while the *Rent a Bike* company invoices the hotel for the service. This partnership enhances the guests' experience and strengthens the hotel's offerings, making it more appealing to potential visitors.

The bicycle rental shop caters to commuters and tourists, creating a versatile and synergetic approach to bike-sharing services. This business model exemplifies the potential for blending convenience with practicality, ensuring that a broader audience can enjoy the benefits of biking while navigating the city. Through this exploration, we will understand the operational dynamics and business implications surrounding this rental shop, illustrating how effective design and organization can lead to a thriving bicycle rental service.

The belonging business plan canvas is shown in *Figure 14.1*:



Source: Strategyzer AG | License: CC By-SA 3.0

***Figure 14.1**: Business plan canvas rent a bike*

The business plan canvas[1], as explained in *Chapter 2, Introduction to the Example Online Library*, is shown for the example *Rent a Bike*. The content is presented in *Table 14.1*:

Part	Topic	Description
Key partners	Public transport	A partnership with the public transport organizations allows to build up bike racks near the stations of the public transport, especially the end points where bikes are necessary to reach homes.
	Hotels	Hotels order bikes at the shop directly and offer them to their guest. Specific racks are not necessary. The tourists will be charged by the hotel and the bike rent service charges the hotel as business partner.
Key activities	Ride a bike	A commuter or a tourist want to ride a bike. A commuter wants to get home or to the office whereas a tourist rides for pleasure.

---

[1] Strategyzer. (2023). The Business Model Canvas. Retrieved March 29, 2024, from **https://www.strategyzer.com/library/the-business-model-canvas**

Part	Topic	Description
Key resources	Bicycles	The bicycles which can be rent by commuters or tourists.
	Mechanics	Mechanics maintain the bikes and take care for distribution to racks and hotels.
	Riders	The riders ride the bikes for pleasure or to come home or to the office.
Key propositions	Easy to book	A ride needs to be easy to book. For commuters it must be easy to find a free bike at their end station. They want to use the service regularly. Therefore, they want to pay per period e.g. per month and not per ride. Tourists want to ride for pleasure. They use the service once or twice during their vacation. They do not want to pay for an abo. It should be as easy and transparent as possible. Therefore, they can fetch the bike at the hotel they are staying in and pay with their hotel bill.
	Easy to pick up	Commuters want to find the bike as near to their end station as possible. They want to unlock it in an easy manner without much putting of codes in locks. Tourists do not care of stations, because they stay in a hotel. The easiest way to pick up a bike is to get it directly at the hotel.
	Easy to return	Commuters do not want to transport the bike over several kilometers. Therefore, it needs to be possible to return a bike at a different rack than it was picked up. Moreover, it needs to be possible to return the bike only the next morning. For tourists it is difficult to find racks because they do not know the area in a detailed manner. Therefore, it is the easiest way to return the bike at the hotel they are staying in.
Customer relationships	Commuters regularly once or twice a week	Commuters want to use the service on a regular base. They travel to the office once or twice a week if we assume that they are allowed to work from home the rest of the time.
	Tourists occasionally	Tourists use the service occasionally when they in a hotel. The service is offered via the hotel. Tourists can download the app but are not forced to.

Part	Topic	Description
Channels	App	The commuters can reach the service via app. The tourists can reach the service via the hotel, but can use the specialized map service via the app.
Customer segments	Commuters	Commuters who want to travel from home to the office and vice versa.
	Tourists	Tourists visiting the city.
Cost structure	Bicycles	Invest into bicycles.
	Racks	Racks for bicycles to pick up and return.
	Maintenance	Maintenance of the bikes and racks.
	App	Costs of the app including development, maintenance, and cloud operations.
Revenue	Flat rate	Commuters can select to pay a flat rate e.g. an abo each month.
	One-time fee	Tourists are usually charged one time via their hotel. But even commuters can select a one-time fee when they use the service only occasionally.

***Table 14.1**: Business model for a rent-a-bike service*

The business model shows the main capabilities of such a service. A capability workshop should detail and prioritize these capabilities. In the following sections, we will discuss those workshops.

Setting up a workshop

All the workshops for architectural design need close cooperation between business experts and IT specialists. They need to be set up with respect and trust. We explained the principles of the brainstorming workshop in *Chapter 6, Event Storming*. All workshops—not only the event storming need to be set up following the given brainstorming principles[2]:

- Focus on quantity, not quality (quantity will lead to quality later)
- Withhold evaluation in the first step
- Encourage wild, outlandish ideas
- Combine or build ideas from others

---

[2] Osborn, A. F. (1953). Applied imagination: principles and procedures of creative thinking. New York: Charles Scribner's Sons.

The workshop area should also follow these principles so that no person is privileged just by their seats in the room. *Hofer and Schwentner*[3] propose a possible solution, as shown in *Figure 14.2*:

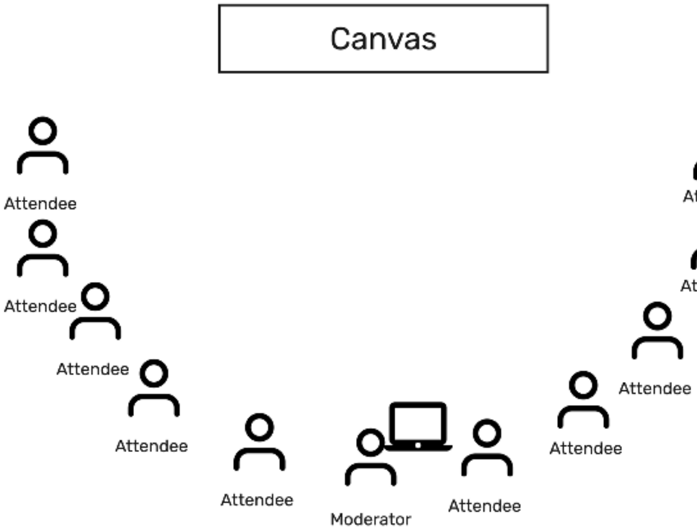


***Figure 14.2**: Possible workshop setup*

The attendees sit around a canvas where ideas can be collected—they form a half circle so that all can see the canvas undisturbed.

The absence of traditional tables in the setting ensures that interactions among attendees remain fluid and unobstructed. However, small tables are provided for holding laptops when utilizing a digital whiteboard, allowing participants to gather and record ideas without any hindrance. These tables facilitate movement around the space, promoting engagement and interaction among attendees. By creating an open environment, participants can easily circulate, collaborate, and share insights, enhancing the overall dynamics of the session.

Now let us look more deeply into the necessary workshops.

Capability map and Wardley Map

We can use the previous workshop setup to find and prioritize the necessary capabilities.

The agenda of such a workshop could look like:

- Presentation of rent-a-bike service by a business expert (20′)
- Collecting of business capabilities by all workshop attendees (30′)

---

[3] Hofer, S., & Schwentner, Henning. (2021). Domain Storytelling: A Collaborative, Visual, and Agile Way to Build Domain-Driven Software. Pearson International.

- Discussing and consolidating the found capabilities (1.5h)
- Creation of the Wardley Map by all workshop attendees (3h)
    - Purpose
    - Users
    - User needs
    - Value chain
    - Map

In the first step, the attendees collect their ideas. They can write their ideas on sticky notes posted on an empty canvas or a meta plan.

Afterward, each attendee presents their ideas to the workshop. During the presentation, they can cluster the ideas and delete duplicates. When a doubling appears, the attendee should present the idea anyway, and the workshop decides whether it is a double. Sometimes, a slight difference that is insignificant to the author might become critical.

The workshop attendees discuss the ideas, cluster them, and consolidate different wording.

The workshop attendees can start creating the Wardley Map using this input.

Let us examine the results of such a workshop. *Figure 14.3* shows the collected capabilities, already consolidated and grouped:

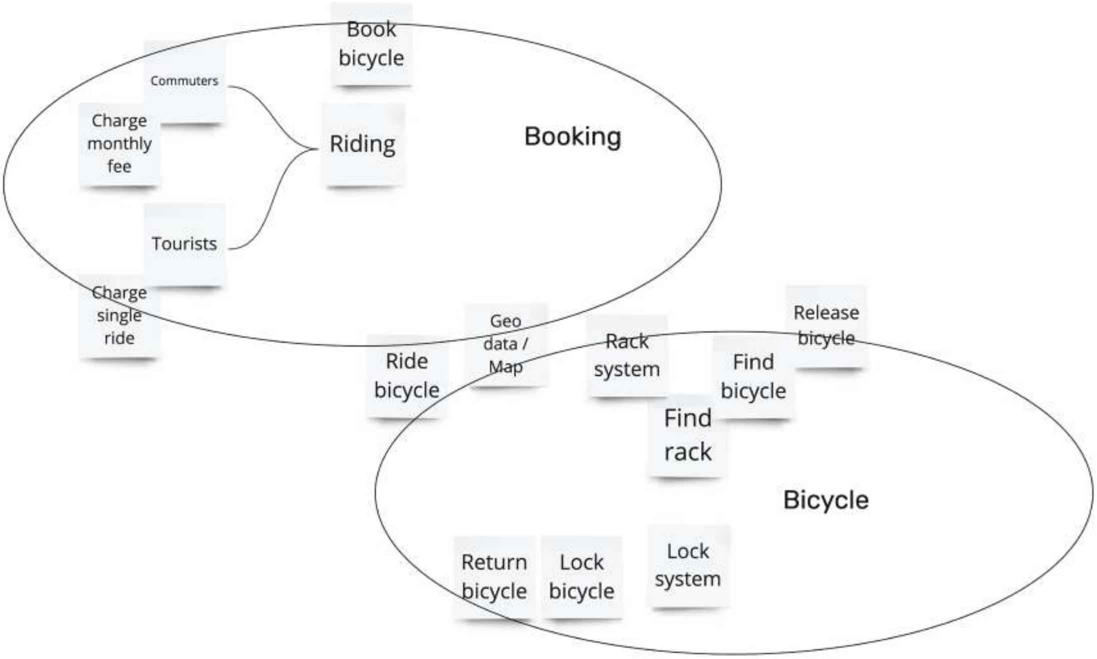


***Figure 14.3**: Capabilities grouped*

***Source**: CapabilitiesGrouped.jpg*

The workshop attendees can use the capabilities to create the Wardley Map, as explained in *Chapter 4, Bounded Context and Domain*. The result is shown in *Figure14.4*:

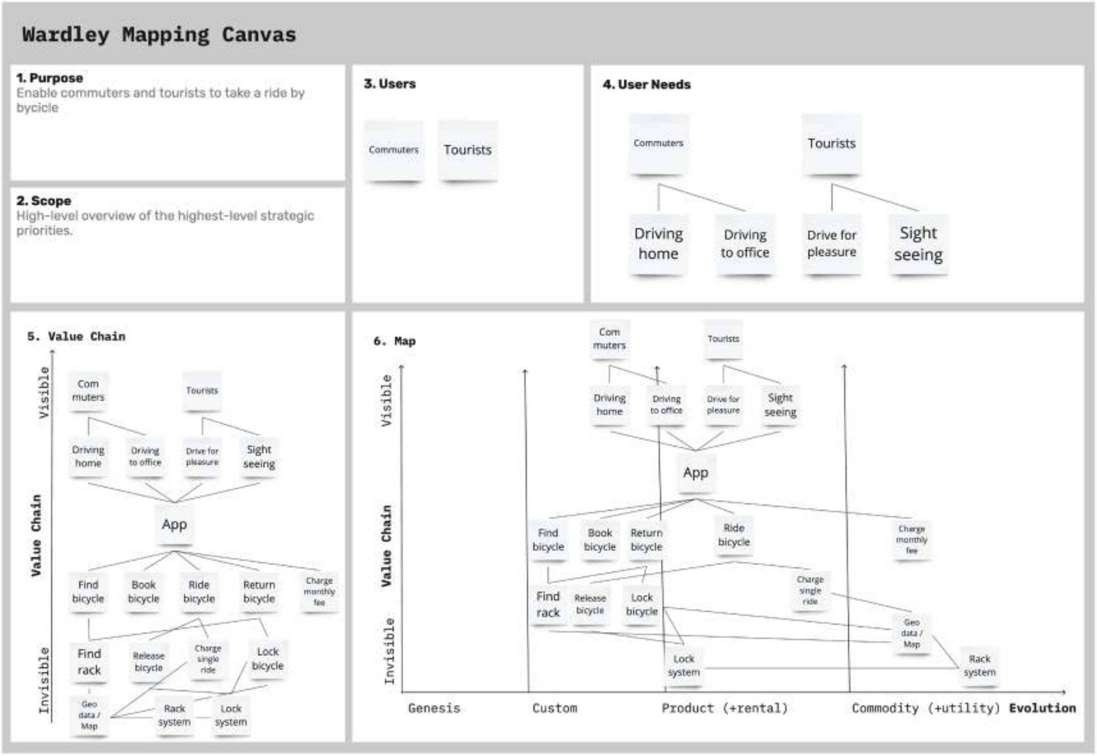


***Figure 14.4**: Wardley Map*

***Source**: WardleyMap.jpg*

The workshop attendees can perform a domain storytelling workshop based on the Wardley Map.

Domain storytelling and event storming

The domain storytelling workshop can be done in the same environment as the capability finding before; see *Chapter 5, Domain Storytelling*.

The business experts tell their stories, which are then consolidated collaboratively. The moderator must encourage a cooperative atmosphere to achieve reliable results and a shared understanding of the problems and solution approaches.

In the case of the rent-a-bicycle shop, at least two domain stories are necessary: one of a tourist and one of a commuter. The Wardley Map defines those major user groups.

The domain story of a tourist is shown in *Figure 14.5*:

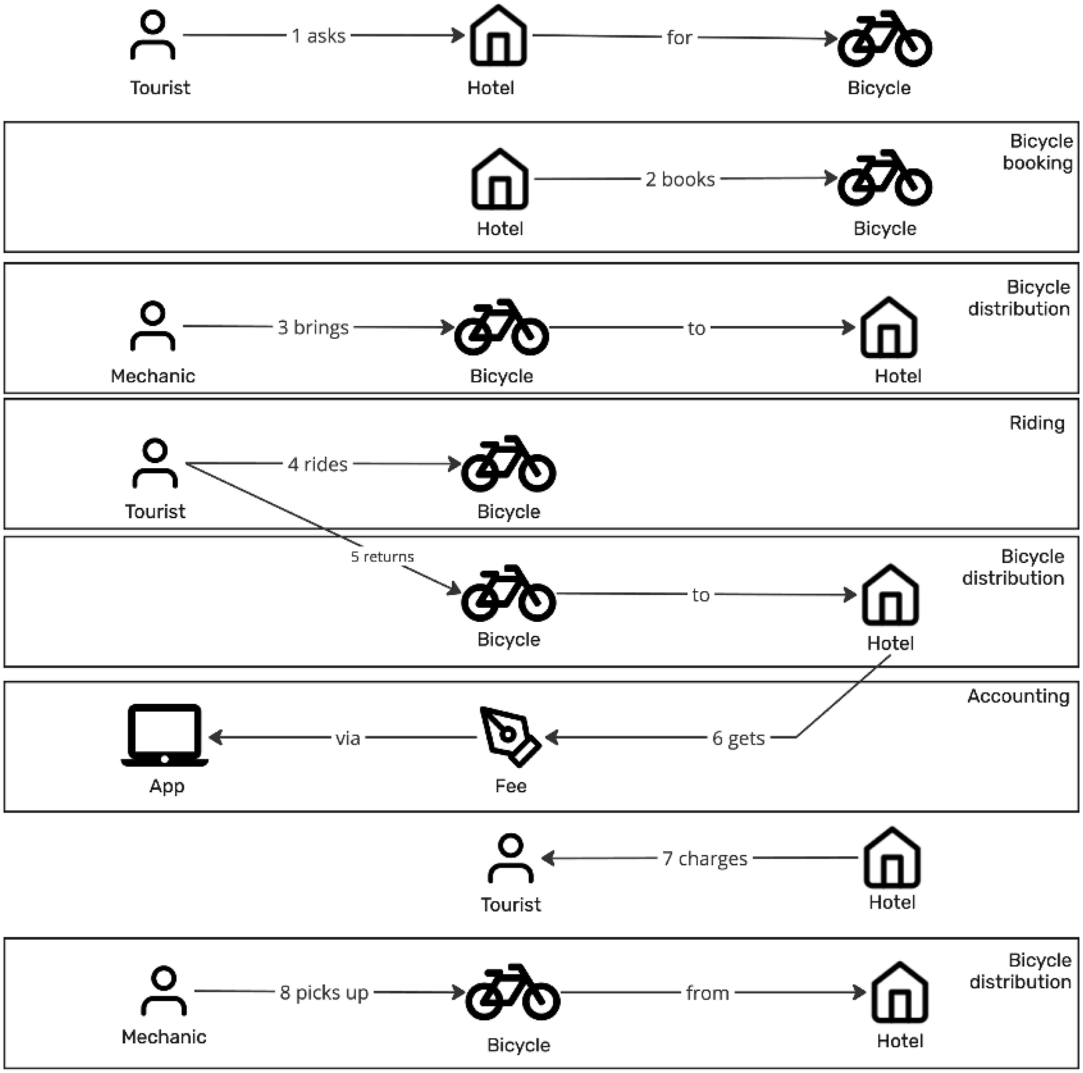


***Figure 14.5**: Domain story from the point of view of a tourist*

The domain story already contains the first bounded contexts, marked as boxes around the sentences belonging to the corresponding bounded context. A domain story is a process-driven view of the problem in which a bounded context can appear multiple times.

The second domain story is told from the point of view of a commuter; see *Figure 14.6*:

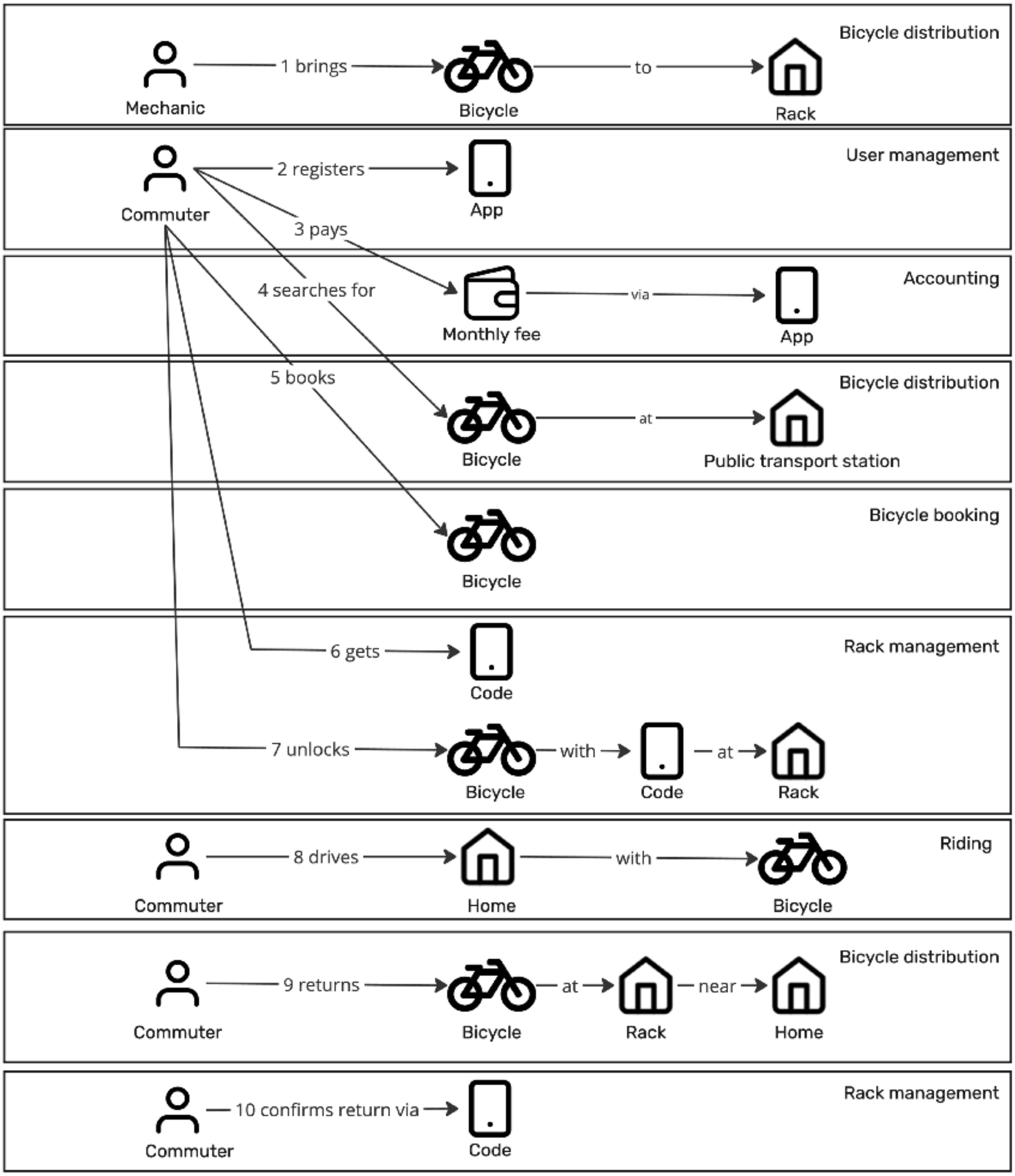


Figure 14.6*: Domain story from the point of view of a commuter*

Source*: DomainStoryCommuter.jpg*

The workshop attendees can use the information from the domain stories to create a visual glossary of the shop rent-a-bike. To do so, they first collect all the terms in the stories and consolidate them together. Afterward, they put them in relations led by the moderator.

The result of this discussion is shown in *Figure 14.7*:

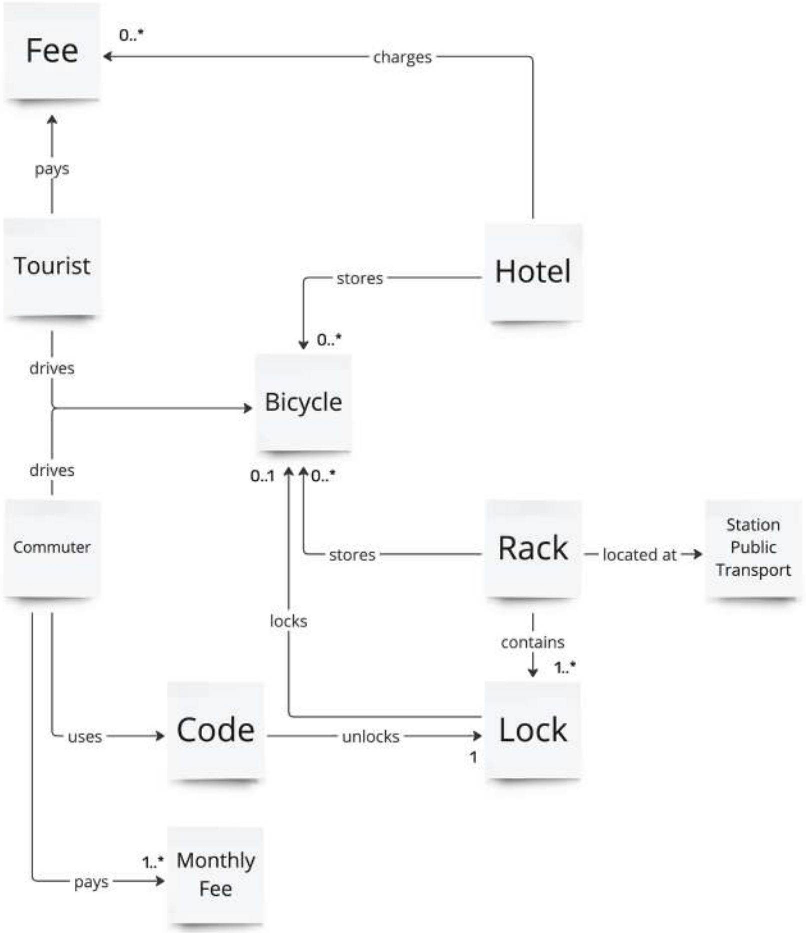


Figure 14.7*: Visual Glossary of the rent-a-bike shop*

Source*: VisualGlossaryBicycle.jpg*

Based on the domain storytelling and the visual glossary, the workshop attendees can perform the corresponding event storming.

The result is documented in *Figure 14.8*:

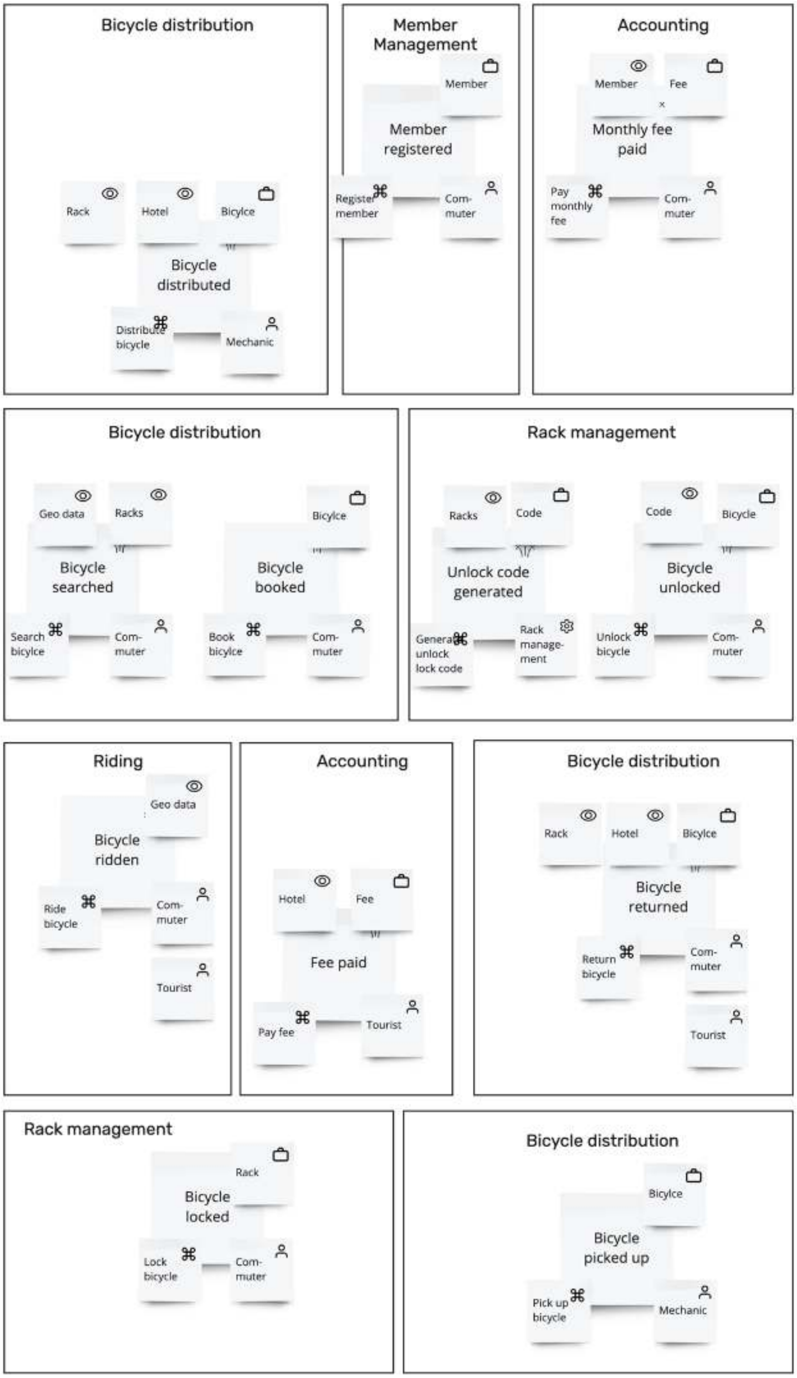


Figure 14.8*: Event storming, including bounded contexts of the rent-a-bike shop*

Source

As described in *Chapter 7, Context Map*, the workshop attendees can create a context map from the event storming results. The context map will be discussed in the following section.

Context map

The context map is built out of the found domain events and the found bounded contexts. The context map is shown in *Figure 14.9*:

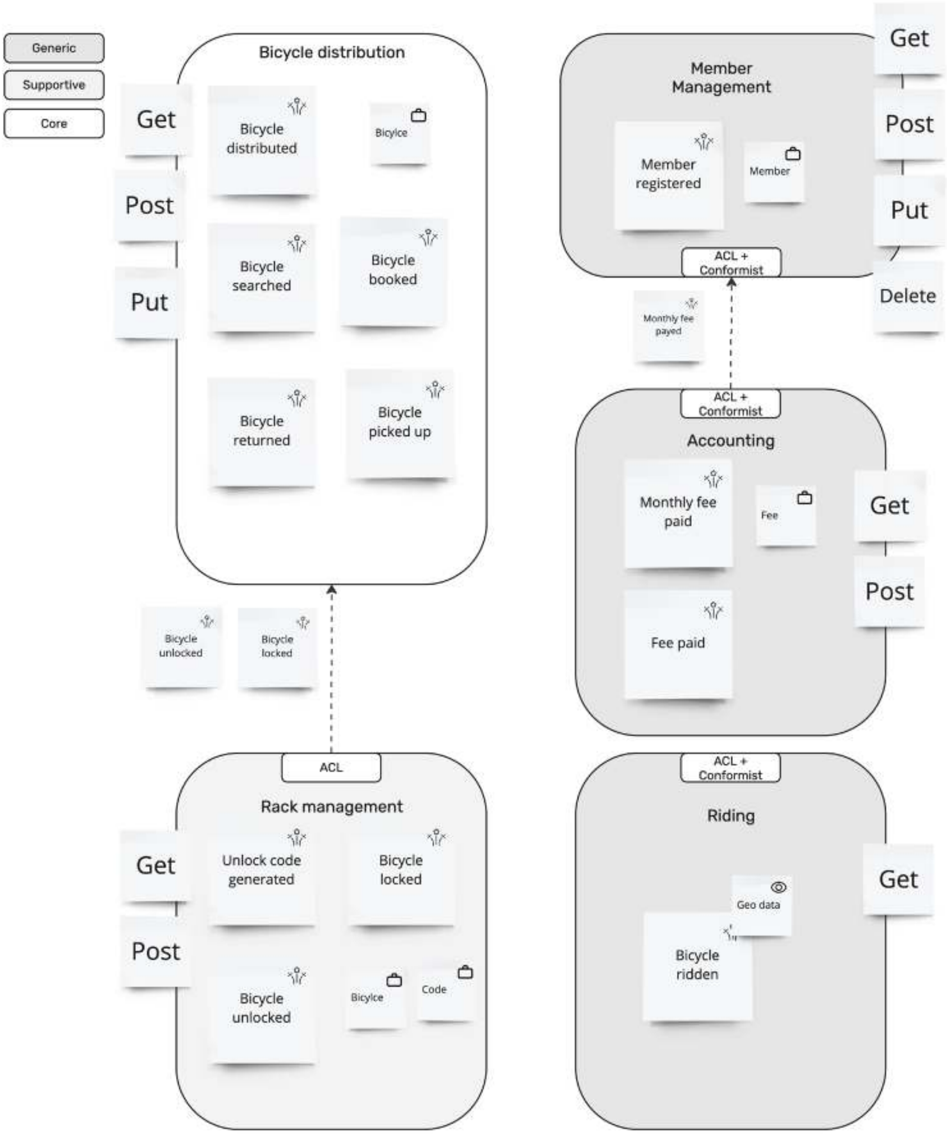


***Figure 14.9**: Context map of the rent-a-bike shop*

***Source**: ContextMapRentABike.jpg*

The context map shown in *Figure 14.9* contains five bounded contexts: bicycle distribution, member management, accounting, riding, and rack management.

According to the analysis presented in the Wardley Map, bicycle distribution is a core capability of the rent-a-bike shop. In contrast, member management, accounting, and rack management are classified as generic capabilities, as they can be readily sourced as commodities in the marketplace. Riding a bike is considered a supportive capability, as it facilitates route discovery for cyclists and includes tailored route suggestions for tourists and commuters specific to the offerings of the rent-a-bike shop.

The supportive and generic bounded contexts rely on bought products. Using those contexts will be implemented as conformists; see *Chapter 4, Bounded Context and Domain*. All those bounded contexts are hidden behind an **anti-corruption layer** (**ACL**), as discussed in *Chapter 4*, to allow the change of the purchased products later.

Furthermore, the APIs were discussed and marked in the context map for each bounded context. We will discuss those APIs in the following section.

API design

Only three events are essential for data exchanges, enabling a highly decoupled architecture.

The rack management system generates two key events: bicycle unlocked and bicycle locked. The bicycle distribution service consumes both of these events, facilitating seamless interactions between the two components.

Additionally, the accounting system produces a third event: the monthly fee paid, which the member management service consumes to ensure accurate billing and account status.

Furthermore, the context map outlines the necessary synchronous APIs that enhance internal communication within the bounded contexts. These APIs are utilized to access services through the user interface, ensuring efficient data flow and user interactions. Detailed explanations of these APIs can be found in *Table 14.12*:

Bounded context	Verb[4]	Payload	Description
Bicycle distribution	GET	Bicycle	Delivers bicycle description with identifier, size, and special features Filter: Rack number, GPS data
	GET	Bicycle location	Delivers back rack lock where the bicycle is stored by bicycle identifier
	POST	Booked status	Books a bicycle for usage by a commuter, response contains rack lock and bicycle identifier
	PUT	Unlock	Unlocks the bicycle with the bicycle and the rack identifier

---

[4] http verb—Hyper Text Transport Protocol

Bounded context	Verb[4]	Payload	Description
Rack management	GET	Code	Delivers code to unlock a code by bicycle identifier
	POST	Code	Locks a bicycle in the rack
Member management	GET	Person data identifier	Delivers back a member with e-mail address and person data
	POST	Person data	Creates a member with e-mail address and person data
	PUT	Person data, identifier	Changes the person data of a member
	DELETE	Identifier	
Accounting	GET	Fee	Delivers the fee to be paid
	POST	Fee	Set the fee to paid
Riding	GET	Geo data by identifier	Delivers the geo data of a certain bicycle

***Table 14.2**: Synchronous APIs of the rent-a-bike-shop*

Using this information and detailed visual glossaries discussed in the workshop, the OpenAPI specification can be written. It is even recommended that the specification work be started in the workshops so business experts can see the ubiquitous language in the specification's code.

Finding target audience for the workshops

For optimal effectiveness, a workshop should have no more than twenty attendees and no fewer than five. When the number of participants exceeds twenty, discussions can become unwieldy, leading to inefficiencies and a lack of focus. With so many voices in the room, reaching a consensus or maintaining a clear direction becomes challenging, ultimately diminishing the workshop's intended outcomes.

On the other hand, workshops with fewer than five participants can also struggle to achieve their objectives. In such settings, the richness of diverse perspectives is often missing. Various opinions and backgrounds are crucial for fostering creativity and stimulating innovative thinking. When only a handful of individuals are present, discussions can become repetitive or lack the dynamism required for a productive session.

An ideal workshop size would be around twelve participants. This number strikes a balance, allowing for a wide array of viewpoints while still being manageable enough to facilitate meaningful conversations. With twelve attendees, there is enough diversity to ensure a range of ideas and insights. Yet, the group remains small enough to allow each participant to contribute actively and engage in discussions effectively. This size encourages collaboration and collective problem-solving, enriching the workshop experience for everyone involved.

Ultimately, careful consideration of attendee numbers can significantly influence the success of a workshop, making it vital to aim for that ideal balance for the best results.

As emphasized in *Chapters 5* and *6, Domain Storytelling and Event Storming*, the workshops must be cross-functional, with business experts and IT specialists.

Business experts can be salespeople who represent customers or even customers themselves. In the sample of an online library, customers can be found easily. In a rent-a-bike shop, it might be more difficult. Product owners of the corresponding bounded contexts can even represent customers.

Additionally, business experts are the ones behind the scenes. That means the liberians in the case of the online library and the mechanics in the case of the rent-a-bike shop. The workshops benefit from their insights and experience.

IT specialists are necessary, and software architects and developers can represent them. However, it might not be easy to find solution architects and developers for those workshops, especially in greenfield projects, because the teams are not yet assigned when they occur. Seasoned developers or solution architects from the implementation company can play that role.

As discussed in *Chapter 5, Domain Storytelling*, the moderator must be a seasoned architect who has already moderated comparable workshops. To be considered for such a role, the person must participate in workshops and thoroughly learn the methodology. First, they should moderate with a seasoned moderator as a backup who can give them some insights for better moderation after the workshop is performed in a lesson-learned session. In critical situations, the backup can give the new moderator more confidence.

The workshops can be documented on an online whiteboard. Because those whiteboards will change when used, it is recommended that you create a **portable document format** (**PDF**) of the direct workshop results. This documentation can be stored in a team working area. When working with physical whiteboards, a photo protocol is essential. The following section discusses how to document the discussion results to amplify the direct workshop results.

Documenting workshop results

To effectively document the workshop results, it is highly recommended that a blueprint be created that presents the information in a more structured and digestible format than merely relying on the documentation on the whiteboards. This blueprint serves as a comprehensive resource, encapsulating critical elements such as the project's scope, context, and initial solution architecture.

The blueprint is designed to align with the first chapters of *arc42*[5] documentation, a reputable standard commonly used in German-speaking regions. This alignment ensures consistency in documentation practices and facilitates better communication among stakeholders who are familiar with this framework.

The first chapter of the blueprint details the scope of the project.

Scope

This section outlines the objectives, goals, and boundaries of what the project intends to achieve. It provides clarity on the key deliverables and sets expectations for all stakeholders involved. In doing so, it helps to ensure that everyone is on the same page and that efforts are directed toward a unified vision.

The section *Overview* contains the project's scope or, better yet, the product. It can contain first ideas in mockups, but it should also include the business model canvas to explain the context and key propositions of the product to be developed.

A template of the chapter is given as a markdown in GitHub[6].

The business model of the rent-a-car shop is shown in *Figure 14.1* and *Table 14.1*.

Additionally, the scope contains the qualitative requirements and the shareholders. Both can be documented as tables, as shown in *Table 14.2* and *Table 14 3* as samples.

Qualitative requirements

The qualitative requirements describe the target quality of the product. Usually, it is not sufficient to require just high quality. They are medium-term targets that thoroughly influence the overall system's design and technology decisions.

They can be collected in a table, as shown in *Table 14.3*. The table structure is given by arc42[7]:

---

[5] Starke, G., & Mueller, R. (2019). arc42 by Example: Software architecture documentation in practice. Packt Publishing.

[6] **https://github.com/Grinseteddy/MasteringDdd/tree/main/Documentation/Blueprint/01%20_Introduction**

[7] Starke, G., & Mueller, R. (2019). arc42 by Example: Software architecture documentation in practice. Packt Publishing.

Priority	Quality target	Note
1	Security	Personal member data must not be revealed to unauthorized users.
2	Security	Moving profiles of members and tourists are not allowed to be stored with the moving profiles of bicycles.
3	Performance	Response times of booking requests must not be longer than 5s in 98% of cases.
4	Flexibility	The architecture must be flexible so that bought modules can be exchanged later.

Table 14 3: *Quality requirements of a rent-a-bike shop*

In the following section of the blueprint, the stakeholders are listed.

Stakeholders

The stakeholders contain the key partners and customers of the product, as shown in the business model canvas. Additionally, project sponsors can appear. The stakeholder table structure is also defined in arc42[8]. A stakeholder example is given in *Table 14.4*:

Role	Target	Expectation
Tourist	Relaxing	Bicycle is available during the vacation directly at the hotel
Commuter	Coming home	The bicycle is available directly at the city train station.
Public transport	More revenue	Commuters decide for public transport instead of car because of available bicycles at the stations
Hotels	More revenue	Tourists decide for the hotel because bicycles are available
Sponsor: community	Recognition of the community	The community wants to attract commuters and investors.

Table 14.4: *Stakeholders of the rent-a-bike shop*

The following chapter of the blueprint discusses the constraints of the rent-a-bike shop.

---

[8] Starke, G., & Mueller, R. (2019). arc42 by Example: Software architecture documentation in practice. Packt Publishing.

Constraints

A system cannot be designed entirely freely. Certain constraints limit the solution area, and an architect needs to be informed about them[9]. Those constraints might be defined by the organization or even by public authorities. There might be organizational as well as technical constraints.

A template of the chapter is given in GitHub[10].

The constraints of the rent-a-bike shop are listed in *Table 14.5*:

Short	**Note**
Kotlin	The rent-a-bike shop server modules are developed in Kotlin current version.
PostgreSQL	The server modules are using PostgreSQL as database.
Kafka	As event bus the open-source version of Kafka is used. The cloud representation needs to be defined.
Documentation	The documentation is published on GitHub.

Table 14.5*: Examples of constraints of the rent-a-bike shop*

The following sections of the blueprint define the business and technical context.

Moreover, the decision to make or buy the modules for the product is constrained. As described in *Chapter 4, Domain and Bounded Contex*t, a Wardley Map can help with those decisions.

In addition to the figure—already shown in *Figure 14.4*—the map needs to be described in a table; see *Table 14.6*:

Evolution stage	**Capability**	**Relation to**	**Note**
Custom	Find bicycle	Find rack	Find a bicycle in a rack on a map.
	Book bicycle		Book a bicycle for riding in a certain rack.
	Return bicycle	Find rack	A commuter brings back a bicycle and locks it in a rack.
		Lock bicycle	
	Find rack	Geo data/Map	A rack can be found by bicycle or in a map.
	Release bicycle	Lock system	A commuter can release a booked bicycle in a rack.
	Lock bicycle	Lock system	A commuter or a mechanic can lock a bicycle in a rack.
		Geo data/ Map	

---

[9] Starke, G., & Mueller, R. (2019). arc42 by Example: Software architecture documentation in practice. Packt Publishing.

[10]

Evolution stage	Capability	Relation to	Note
Product	Ride bicycle	Release bicycle	A commuter or a tourist rides the bicycle.
		Charge single ride	
	Charge single ride	Geo data/Map	A tourist is charged for a single ride e.g. by distance.
	Lock system	Rack system	The system lock and unlock the bicycles in the rack.
Commodity	Charge monthly fee		Commuters are charged with a monthly fee to use the bicycles.
	Geo data/Map		Geo data service to find and track bicycles
	Rack system		System to store bicycles for riding

Table 14.6*: Wardley Map of the rent-a-bike shop*

The following section describes the contexts of the system.

Context

The context allows describing the business and technical scope of system[11]. A capability map defines the context.

Additionally, domain storytelling and event storming results are used for documentation.

The respective template for the chapter is given in GitHub[12].

The following will show how to apply the documentation using the rent-a-bike shop example.

Capability map

The method of a capability map was discussed in *Chapter 4, Bounded Context and Domain*. The found capabilities, as shown in *Figure 14.3*, are sorted as generic, supportive, and core capabilities. Usually, the capabilities are regrouped after being applied in the Wardley Map. In this chapter, the result is shown after regrouping.

The capability map of the rent-a-bike shop is shown in *Figure 14.10*:

---

[11] Starke, G., & Mueller, R. (2019). arc42 by Example: Software architecture documentation in practice. Packt Publishing.

[12]

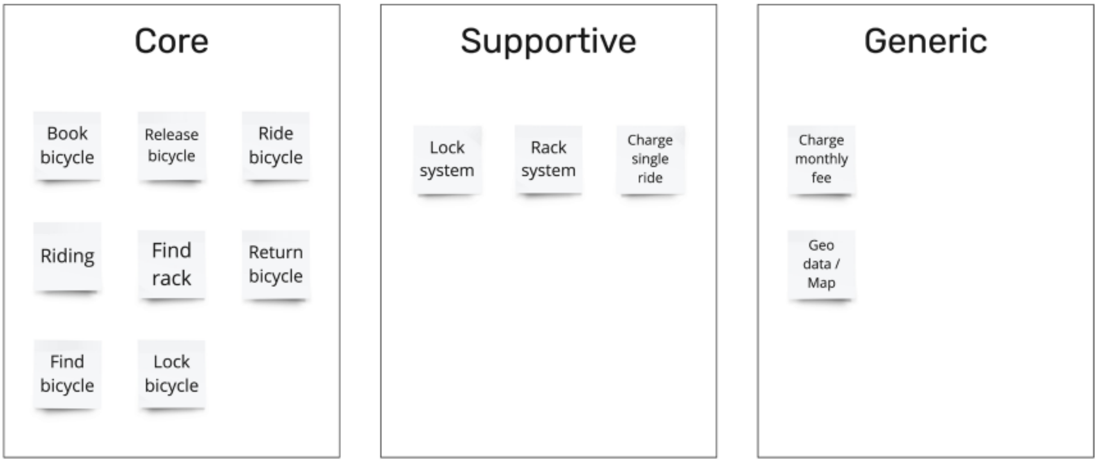


***Figure 14.10**: Capability map of the rent-a-bike shop*

***Source**: CapabilityMapRentABike.jpg*

As the other results, the figure, see *Figure 14.10*, must be described in a table:

Subdomain	Capability	Note
Core	Book bicycle	A commuter can book a bicycle to take it later the day.
	Release bicycle	A commuter can release a bicycle in rack using a code.
	Ride bicycle	A commuter or a tourist can ride a bicycle.
	Find rack	A commuter can find a rack containing a suitable bicycle on a map.
	Return bicycle	A commuter or a tourist can return a bicycle to a rack or the hotel.
	Find bicycle	A commuter can find a suitable bicycle by its properties, e.g., size and motor support.
	Lock bicycle	A commuter or a mechanic can lock a bicycle in a rack.
Supportive	Lock system	A system to lock bicycles in a rack.
	Rack system	System to store and lock bicycles in public areas.
	Charge single ride	A single ride of a tourist must be chargeable via the hotel.
Generic	Charge monthly fee	A commuter pays a monthly fee as a member.
	Geo data map	A map where bicycles, racks, and rides can be shown.

***Table 14.7**: Capability map of the rent-a-bike shop*

The context contains a section on the business context that will be discussed in the following.

Business context

The results of domain storytelling can excellently describe the business context. In addition to the domain story as a figure (see *Figure 14.5* and *Figure 14.6*), the domain stories need to be defined in a table. The domain story in the point of view of the tourist is defined in *Table 14.8*:

No.	Bounded context	Sentence	Note
1		A tourists ask the hotel for a bicycle.	Outside of the rent-a-bike shop application.
2	Bicycle booking	The hotel books the bicycle.	
3	Bicycle distribution	The mechanic brings the bicycle to the hotel.	The hotel stores the bicycles in their rooms so that an automatic rack is not necessary.
4	Riding	The tourist rides the bicycle.	The tour can be followed on a map in the app.
5	Bicycle distribution	The tourist returns the bicycle to the hotel.	
6	Accounting	The hotel gets the fee invoice via the app.	Alternatively, the invoice can be sent via email.
7		The hotel charges the tourists.	Outside of the rent-a-bike shop application.
8	Bicycle distribution	The mechanic picks up the bicycle from the hotel.	The mechanic can book the return to the garage via the app.

***Table 14.8**: Domain story of the rent-a-bike shop from the point of view of a tourist*

The corresponding domain story from the point of view of the commuters is described in *Table 14.9*:

No.	Bounded context	Sentence	Note
1	Bicycle distribution	A mechanic brings the bicycle to the rack.	The bicycle is locked in the rack.
2	User management	Commuters registers themselves at the app.	
3	Accounting	The commuter pays the monthly fee via app.	The integration of a payment system needs to be discussed.

No.	Bounded context	Sentence	Note
4	Bicycle distribution	The commuter searches for a bicycle at a public transport station.	The search can be done by lists of stations or in a map.
5	Bicycle booking	The commuter books the bicycle.	
6	Rack management	The commuter gets the code.	The code can be used to unlock the bicycle in the rack.
7		The commuter unlocks the bicycle with the code at the rack.	
8	Riding	The commuter drives home with the bicycle.	
9	Bicycle distribution	The commuter returns the bicycle at the rack near home.	Suitable racks can be found on the map.
10	Rack management	The commuter confirms the return via code.	Using the code the return is confirmed and the bicycle is locked in the rack.

***Table 14.9**: Domain story of the rent-a-bike shop from the point of view of a commuter*

The terms used in the domain stories need to be explained in addition to the domain stories. A visual glossary, such as a figure or a table, can be used. The visual glossary of the rent-a-bike shop is shown in *Figure 14.7*. The corresponding table is given in *Table 14.10*:

Term	Description	Relation to
Fee	A fee a tourist or a commuter pays.	
Tourist	A tourist who wants to ride a bike.	Pays fee
		Drives bicycle
Hotel	A hotel providing bicycles to tourists.	Charges 0..* fees
		Stores 0..* bicycles
Bicycle	A bicycle ridden by tourists or commuters.	
Commuter	A commuter who wants to drive home.	Drives the bicycle
		Uses code
		Pays 1..* monthly fee
Rack	A rack where bicycles can be provided to commuters.	Located at station public transport
		Contains 1..* locks
		Stores 0..* bicycles

Term	Description	Relation to
Station public transport	A station of a public transport system, e.g., a city train station.	
Code	A code to lock or unlock a bicycle.	Unlocks 1 lock
Lock	A lock which secures a bicycle in a rack.	Locks 0..1 bicylce
Monthly fee	A monthly fee which is paid by commuter to uses bicycles.	

Table 14.10: *Description of the visual glossary of a rent-a-bike shop*

The event storming result is the basis for describing the deployment context. In the following section, we will discuss the corresponding documentation.

Technical context

Event storming is a pretty good base for describing the technical context, which should explain the system's technical context. The events can be characterized by bounded context, role, command, read model, event, and aggregate. The event storming result is shown in *Figure 14.8*.

The result of the event storming is shown in *Table 14.10*:

Bounded context	Role	Command	Read model	Event	Aggregate
Bicycle distribution	Mechanic	Distribute bicycle	Rack, hotel	Bicycle distributed	Bicycle
Member management	Commuter	Register member		Member registered	Member
Accounting	Commuter	Pay monthly fee	Member	Monthly fee paid	Fee
Bicycle distribution	Commuter	Search bicycle	Geo data Racks	Bicycle searched	
	Commuter	Book bicycle		Bicycle booked	Bicycle
Rack management	Rack management	Generate unlock lock code	Racks	Unlock code generated	Code
	Commuter	Unlock bicycle	Code	Bicycle unlocked	Bicycle
Riding	Commuter	Ride bicycle	Geo data	Bicycle ridden	
	Tourist				

Bounded context	Role	Command	Read model	Event	Aggregate
Accounting	Tourist	Pay fee	Hotel	Fee paid	Fee
Bicycle distribution	Commuter	Return bicycle	Rack	Bicycle returned	Bicycle
	Tourist		Hotel		

***Table 14.11**: Event storming result of the rent-a-bike shop*

The product's solution approach is described in the next section.

Solution approach

The solution approach can be nicely documented using the context map. It contains the bounded contexts, the belonging events, the aggregates belonging to the bounded context, and the asynchronous and synchronous **application programming interfaces** (**APIs**). The template for the chapter is available in GitHub[13].

Figure 14.9 shows the context map of the rent-a-bike shop. The figure created in the workshops needs to be documented in a table, which is given in *Table 14.12*:

Bounded context	Pattern	Event	Aggregate	Asynchronous events	Synchronous APIs
Bicycle distribution Core		Bicycle distributed	Bicycle	Consume: Bicycle locked	Get
		Bicycle searched		Consume: Bicycle unlocked	Post
		Bicycle returned			Put
		Bicycle booked			
		Bicycle picked up			
Member management Generic	Anti-corruption layer, Conformist	Member registered	Member	Consume: Monthly fee paid	Get
					Post
					Put
					Delete

---

[13] **https://github.com/Grinseteddy/MasteringDdd/tree/main/Documentation/Blueprint/04_Solution-Approach**

Bounded context	Pattern	Event	Aggregate	Asynchronous events	Synchronous APIs
Accounting Generic	Anti-corruption layer, Conformist	Monthly fee paid	Fee	Produce: Monthly fee paid	Get
		Fee paid			Post
Rack management Supportive	Anti-corruption layer	Unlock code generated	Bicycle	Produce: Bicycle unlocked	Get
		Bicycle locked	Code	Produce: Bicycle locked	Post
		Bicycle unlocked			
Riding Generic	Anti-corruption layer, Conformist	Bicycle ridden			Get

Table 14.12: *Context map description of the rent-a-bike shop*

Overall, utilizing a blueprint to document workshop outcomes enhances clarity and accessibility of information. It allows stakeholders to revisit the topics discussed and understand the rationale behind decisions made during the workshop. By transforming discussions into a well-organized framework, the blueprint becomes an invaluable tool for guiding subsequent phases of the project and ensuring alignment throughout the development lifecycle.

The introduction, constraints, context, and solution approach make the blueprint highly expressive and can serve as a base for developing single-bounded contexts in corresponding development teams. However, the bounded contexts themselves need to be documented, and the necessary documentation will be discussed in the following section.

Documentation of bounded contexts

As discussed in the previous section, the blueprint documentation describes the product's logical architecture. A blueprint for the bounded context can address its architecture.

It uses the bounded context canvas as discussed in *Chapter 6, Event Storming*, an arc42 canvas[14], an OpenAPI specification as discussed in *Chapter 11, Exposing Aggregates via APIs*, and an Async API as discussed in *Chapter 12, Exposing Domain Events*, belonging to the according to bounded context.

We will discuss them as documentation artifacts in the following section.

Bounded context canvas

The first part of a bounded context documentation is the bounded context canvas.

The bounded context canvas contains the following points[15] (for detailed description, please refer to *Chapter 6, Event Storming*):

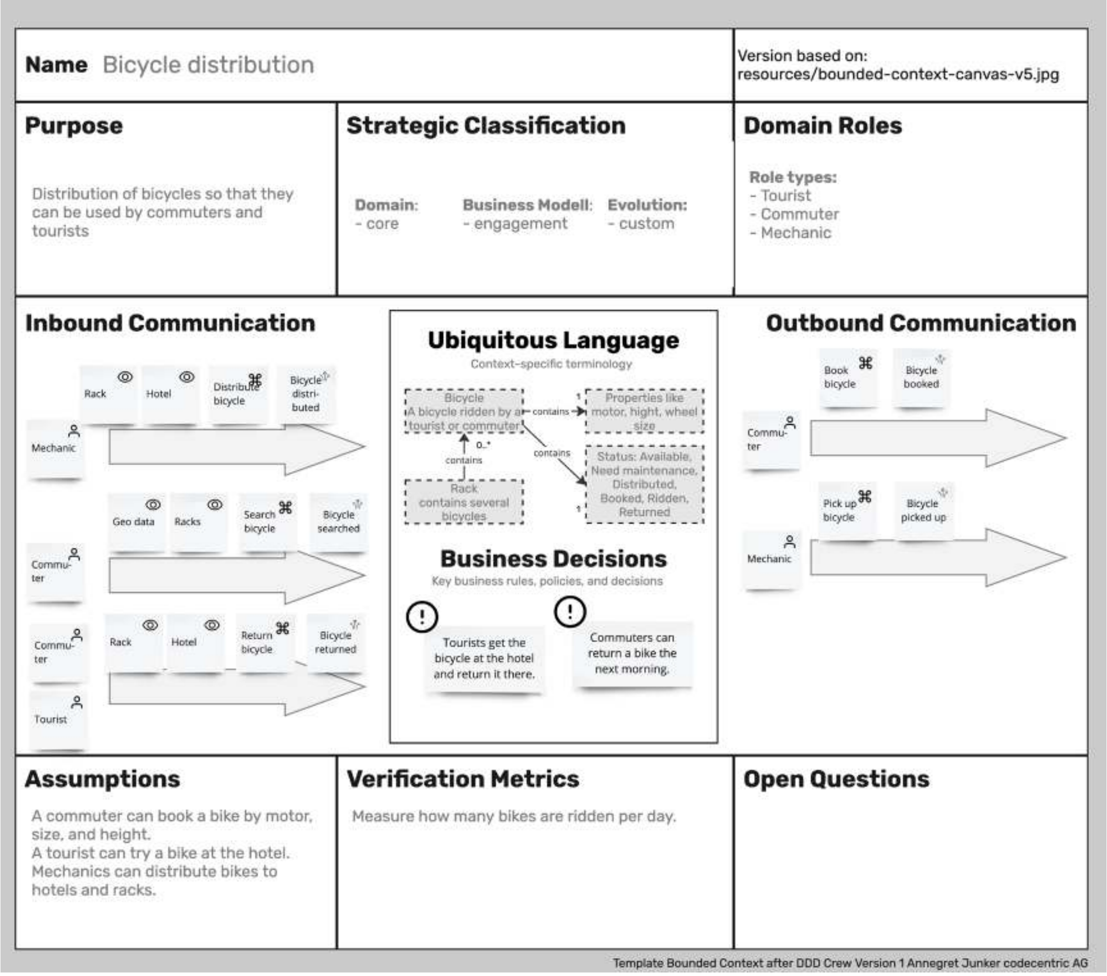
- Name
- Purpose
- Strategic classification: Domain, business model, evolution
- Domain roles
- Inbound communication
- Ubiquitous language
- Business decisions
- Outbound communication
- Assumptions
- Verification metrics
- Open questions

---

[14] Architecture Communication Canvas. (2024). Retrieved October 2024, from arc42: **https://canvas.arc42.org/architecture-communication-canvas**

[15] Baas, K., Lindhard, K., Plöd, M., & Sanglan-Charlier, M. (2024, May 20). The Bounded Context Canvas. Retrieved May 2024, from GitHub: **https://github.com/ddd-crew/bounded-context-canvas/tree/master**

The bounded context canvas of the bicycle distribution is shown in *Figure 14.11*:



Name Bicycle distribution

Version based on: resources/bounded-context-canvas-v5.jpg

Purpose

Distribution of bicycles so that they can be used by commuters and tourists

Strategic Classification

Domain: - core

Business Modell: - engagement

Evolution: - custom

Domain Roles

Role types:
- Tourist
- Commuter
- Mechanic

Inbound Communication

Ubiquitous Language

Context-specific terminology

Business Decisions

Key business rules, policies, and decisions

Tourists get the bicycle at the hotel and return it there.

Commuters can return a bike the next morning.

Outbound Communication

Assumptions

A commuter can book a bike by motor, size, and height.
A tourist can try a bike at the hotel.
Mechanics can distribute bikes to hotels and racks.

Verification Metrics

Measure how many bikes are ridden per day.

Open Questions

Template Bounded Context after DDD Crew Version 1 Annegret Junker codecentric AG

Figure 14.11: *Bounded context canvas of bicycle distribution*

Source: *BoundedContextCanvasBicycleDistribution.jpg*

The bounded context canvas gives an overview of the significant business decisions of the bounded context.

Second, the technical design of a bounded context needs to be described. An architectural communication canvas can be used to do so.

Architectural communication canvas

The arc42 group developed the architecture communication canvas[16] as the comprehensive architecture documentation template arc42[17].

The canvas contains the following areas:

- **Value proposition**: The value proposition contains the business case or the economic driver of the corresponding bounded context. It can be taken over from the *Purpose* section of the bounded context canvas.
- **Key stakeholder**: The roles listed in the bounded context canvas are also the key stakeholders of the bounded context. Sponsors who are especially interested in the particular bounded context can be listed additionally.
- **Core functions**: The core functions can be listed as the commands handled in the bounded context or using the events.
- **Quality requirements**: The quality requirements are derived from the general quality requirements listed in the blueprint of the overall product. They must be measurable.
- **Core decisions**: The core decisions of architecture need to be listed. The team made these decisions to implement the bounded context. Overall constraints are listed in the blueprint.
- **Technologies**: This part listed the significant technologies of the bounded context.
- **Business context**: The part contains the necessary external interfaces of the neighboring systems. To do so, a C4-model diagram[18] should be used. It can be easily derived from the context map in the blueprint section *Solution approach*.
- **Components/modules**: The part contains the bounded context's internal point of view of the bounded context. This can be done at the container level of the C4 model.
- **Risks and missing information**: This part contains the risks and probably needs more information when developing and designing the bounded context.

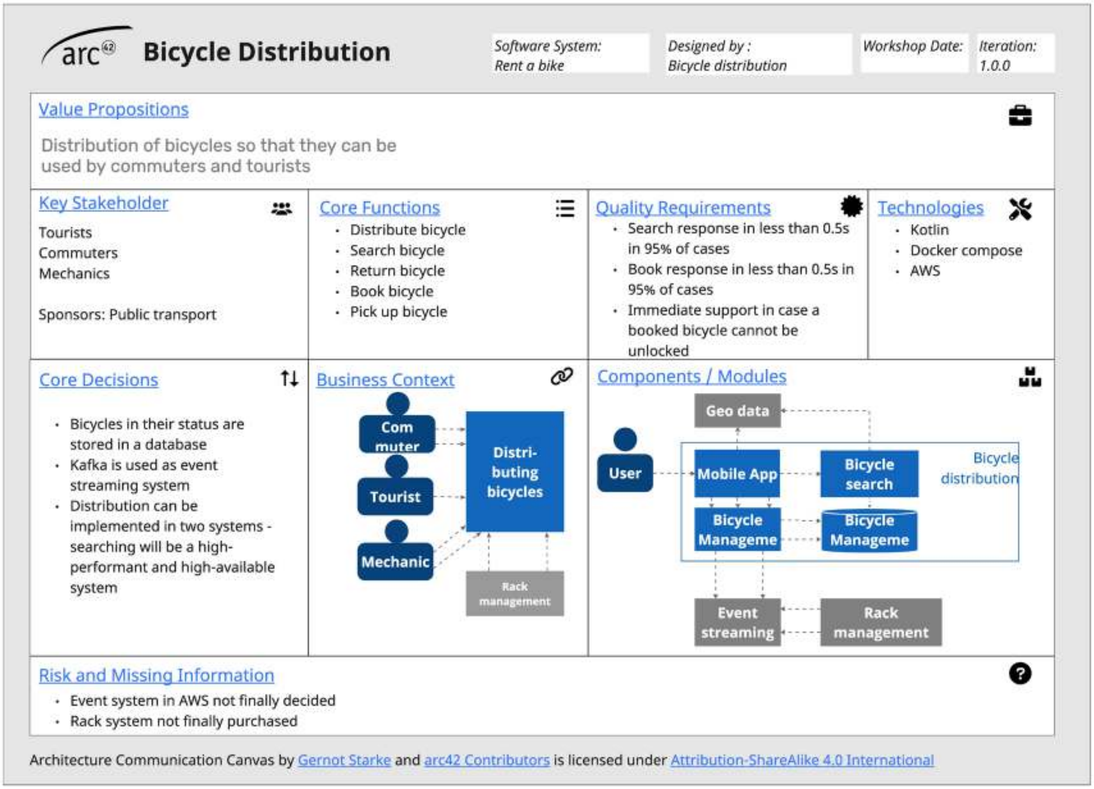
---

[16] Architecture Communication Canvas. (2024). Retrieved October 2024, from arc42: **https://canvas.arc42.org/architecture-communication-canvas**

[17] Starke, G., & Mueller, R. (2019). arc42 by Example: Software architecture documentation in practice. Packt Publishing.

[18] Brown, S. (2023). C4 model. Self-published.

When we apply the areas to a canvas and fill it with the bicycle distribution bounded context information, we get the canvas, as shown in *Figure 14.12*:



Bicycle Distribution

Software System: Rent a bike

Designed by : Bicycle distribution

Workshop Date:

Iteration: 1.0.0

Value Propositions

Distribution of bicycles so that they can be used by commuters and tourists

Key Stakeholder

Tourists
Commuters
Mechanics

Sponsors: Public transport

Core Functions

- Distribute bicycle
- Search bicycle
- Return bicycle
- Book bicycle
- Pick up bicycle

Quality Requirements

- Search response in less than 0.5s in 95% of cases
- Book response in less than 0.5s in 95% of cases
- Immediate support in case a booked bicycle cannot be unlocked

Technologies

- Kotlin
- Docker compose
- AWS

Core Decisions

- Bicycles in their status are stored in a database
- Kafka is used as event streaming system
- Distribution can be implemented in two systems - searching will be a high-performant and high-available system

Business Context

Components / Modules

Risk and Missing Information

- Event system in AWS not finally decided
- Rack system not finally purchased

Architecture Communication Canvas by Gernot Starke and arc42 Contributors is licensed under Attribution-ShareAlike 4.0 International

***Figure 14.12**: Architecture communication canvas of the bounded context bicycle distribution*

***Source**: ArchitectureCommunicationCanvas.jpg*

The canvas was created in *Miro*[19] using the template provided by *Gernot Starke*[20].

The business context and component areas contain only limited labels to support readability in print. They can be provided separately. Such separate provisioning of parts of the canvas—especially diagrams—is good practice. The canvas should be readable and easy to understand in an overview. If the diagrams are too detailed, they cannot be understood without zooming, which contradicts the overview idea.

---

[19] **https://miro.com/app/dashboard/**

[20] **https://miro.com/miroverse/architecture-communication-canvas/**

The business context diagram with details is given in *Figure 14.13*:

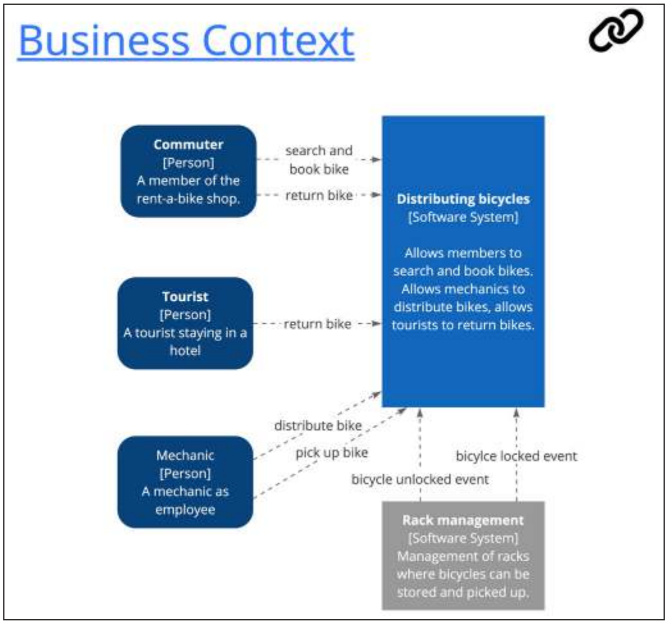


***Figure 14.13**: Business context of the bicycle distribution context*

***Source**: BusinessContext.jpg*

The same is valid for the internal design of the bounded context, which is given in the *Components/modules* section. The component diagram is shown in *Figure 14.14*:

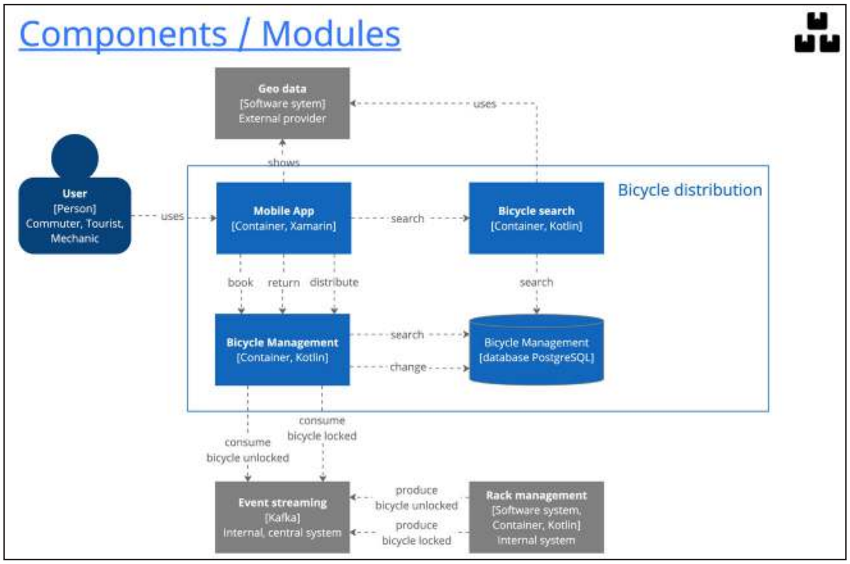


***Figure 14.14**: Components and modules of the bicycle distribution bounded context*

***Source:** Components.jpg*

Synchronous API specification

The synchronous API specification can be given as OpenAPI specification. The specification details were discussed in *Chapter 11, Exposing Aggregates via APIs*.

For the bounded context, the endpoints listed in the context map need to be specified; see *Figure 14.9* and *Table 14.12*.

A bicycle is defined by its status and properties as listed in the ubiquitous language of the bounded context canvas, see *Figure 14.11*.

The endpoints should be defined as follows:

- **Get**: Gets a list of available bicycles in racks by a current point on a map and a defined distance.
- **Post**: Books a bicycle so that it is not available anymore for other commuters.
- **Put**: Cancels a booking.

A bicycle should contain the following properties:

- Identifier as a **universal unique identifier** (**UUID**)
- **Physical identifier**: Number printed onto the bicycle frame to identify the bike physically, too
- **Properties**[21]:
  - Height in inches as a number
  - Wheel size in inches as a number
  - Motor as Boolean
- **Status as enum**: AVAILABLE, NEED_MAINTENANCE, DISTRIBUTED, BOOKED, RIDDEN, RETURNED
- **Rack number**: The rack number is stored if the bicycle is distributed in a rack

With such a specification, developers can implement clients using the synchronous interface.

---

[21] There might be many more properties to describe a bicycle, but for the example here, it might be sufficient to know that describing properties is essential.

Asynchronous specification

The asynchronous specification contains the produced and consumed events of the corresponding service. *Chapter 12, Exposing Domain Events*, describes the parts of such a specification and how to define each part.

For the bounded context of bicycle distribution, this means defining the consumption of the events: bicycles locked and unlocked.

The event must contain the bicycle's identifier, the rack's identifier, and the lock's identifier where the bike is taken or returned. Based on that, the service can change the status of the corresponding bike.

As we can see, several architectural decisions need to be made, from the first ideas to the implementation of single-bounded contexts. They also need to be documented. The following section discusses how architectural decisions can be documented.

Documenting architectural decisions

Architectural decisions need to be documented. Several decisions need to be made during the design process. They need to be made consciously, and they need to be documented so that the reasoning behind a decision can be followed.

This is valid for a product's overarching architecture and technology decisions and decisions made by an implementing team. The decisions can be documented using an **architecture decision record** (**ADR**)[22].

ADRs should be stored along with the blueprint for overarching decisions and the bounded context bundle for decisions inside a bounded context.

The ADR contains the decision, its reasoning, the deciders who made it, and the presented alternatives.

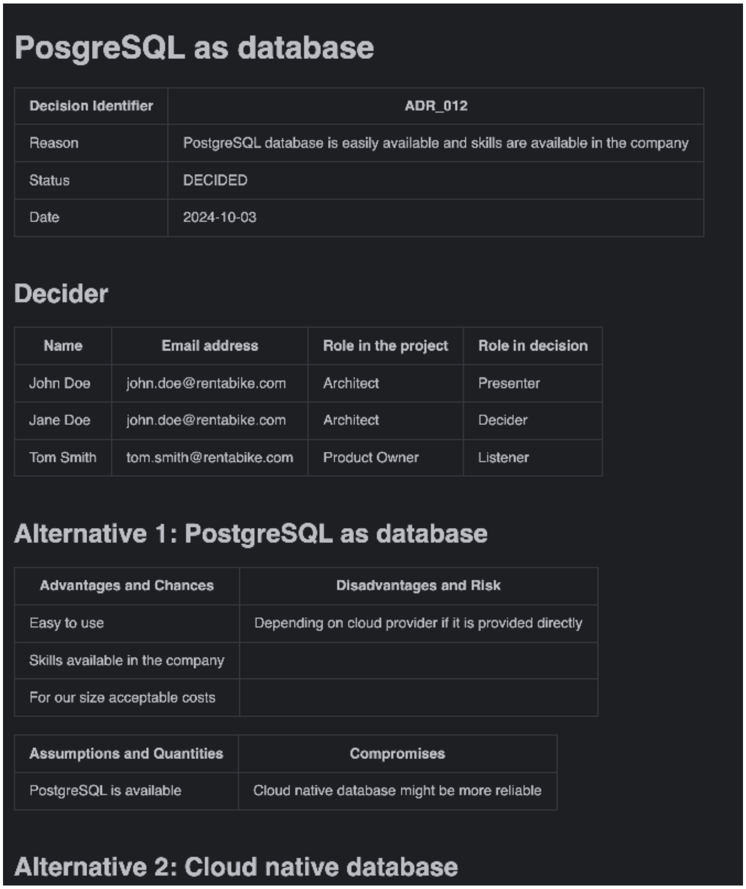
A template for an ADR is given as a markdown in GitHub[23].

---

[22] Architectural Decision Records (ADRs). (2024). Retrieved October 2024, from Homepage of the ADR GitHub organization: **https://adr.github.io/**

[23] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter14-UsageOfDomainDrivenDesignInAGreenfield/Documentation/ADR/README.md**

An ADR for using PostgreSQL as a database in the rent-a-bike shop could be like the one documented in *Figure 14.15*:



PosgreSQL as database

Decision Identifier	ADR_012
Reason	PostgreSQL database is easily available and skills are available in the company
Status	DECIDED
Date	2024-10-03

Decider

Name	Email address	Role in the project	Role in decision
John Doe	john.doe@rentabike.com	Architect	Presenter
Jane Doe	john.doe@rentabike.com	Architect	Decider
Tom Smith	tom.smith@rentabike.com	Product Owner	Listener

Alternative 1: PostgreSQL as database

Advantages and Chances	Disadvantages and Risk
Easy to use	Depending on cloud provider if it is provided directly
Skills available in the company	
For our size acceptable costs	

Assumptions and Quantities	Compromises
PostgreSQL is available	Cloud native database might be more reliable

Alternative 2: Cloud native database

Figure 14.15*: Architecture decision record filled with database decision*

Source*: AdrFilled.png*

The corresponding example is also given in GitHub[24]. It is recommended that architecture decisions be documented using documentation as code, such as Markdown[25] or AsciiDoc[26]. The decisions can then be stored near the code in a suitable version control system.

However, readers might wonder why software and architecture need to be documented at all. We will discuss this shortly in the following section.

---

[24] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter14-UsageOfDomainDrivenDesignInAGreenfield/Documentation/ADR/postgre.md**

[25] Markdown Guide. (2024). Retrieved October 2024, from Markdown Guide: **https://www.markdownguide.org/**

[26] Publish presentation-rich content from a concise and comprehensive authoring format. (2024). Retrieved October 2024, from AsciiDoc: **https://asciidoc.org/**

Importance of documenting software

The comprehensive documentation of the product, complemented by the specific documentation for each bounded context, illuminates the product's complexities from both business and IT perspectives. This robust documentation framework empowers independent teams to implement bounded contexts efficiently while also facilitating a swift onboarding process for new team members and entire teams.

The documents are designed to be concise and interconnected, offering clear guidance on where newcomers should begin their journey and where they can find more detailed information. This streamlined approach fosters understanding and supports effective collaboration among team members.

Based on the author's experience, documenting a workshop through domain storytelling and event storming generally requires approximately two days. Additionally, the relevant team can create the bounded context canvas and architecture communication canvas within the same time frame. This efficient documentation process ensures teams have the necessary insights and visualizations to move forward confidently.

API specifications should be developed using an API-first approach, prioritizing clear and precise definitions. The effort involved in creating and maintaining this documentation is relatively minimal compared to the extensive resources required for onboarding new personnel.

Moreover, the proposed documentation methodology—featuring the blueprint and bounded context documentation bundle—is widely regarded in its parts by authorities during software product reviews, providing an added layer of assurance regarding the quality and clarity of the documentation.

We explored the application of DDD in a greenfield project. However, navigating the initial steps of such an endeavor can be challenging, leaving teams needing guidance on where to start and how to evolve their approach effectively. This highlights the importance of having a structured framework and clear documentation to guide teams through the complexities of DDD and ensure they can adapt and grow as their projects develop. The corresponding implementation process of DDD will be discussed in the next section.

Evolving the DDD process

It is essential to evolve to master a Domain-Driven Design process. In this section, we will discuss how to implement and evolve the process. The following describes how a DDD process can be implemented step by step:

1. **Start with domain storytelling**: The foundation of an effective DDD process begins with domain storytelling. This technique involves gathering stakeholders—business leaders, developers, and end-users—to narrate the domain's key

real-world scenarios through storytelling, capturing the essential contexts, roles, and business objectives. This initial phase fosters a shared understanding of the domain, highlighting the complexities and nuances that will guide subsequent design efforts. It also promotes engagement among team members, encouraging them to contribute their unique perspectives and knowledge.

The workshop format can be studied simply. It can be learned by attending workshops moderated by seasoned architects. Afterward, coached self-moderating is possible.

2. **Conduct event storming in single teams**: Once a foundational understanding is established through domain storytelling, the next step is to perform event storming sessions within individual teams. Modifying overview event stormings, such as those presented in *Chapter 6, Event Storming*, and the current chapter, is challenging. However, in a smaller size, it can be moderated in a group and steer the necessary discussion in a team to find the best solution.

   Event storming is a collaborative workshop technique that allows teams to explore domain events—significant occurrences that affect the system. During this phase, team members map out events on a timeline, identifying commands and aggregates and reading models associated with each event. This process uncovers required functionalities and reveals how different components interact within the system. By focusing on a single team's bounded context, participants can delve deeper into specific challenges and design solutions tailored to their unique bounded context, ensuring that each bounded context is well understood and delineated.

   When the event storming is well established in the teams, overview workshops for comprehensive products and projects can be conducted.

3. **Formulate the business using a Wardley Map**: With a clearer understanding of the domain established through storytelling and event storming, teams can begin to visualize the business strategy using a Wardley Map. This strategic tool maps out the business components, identifying their maturity levels and relationships.

   The teams are used to discuss problems not only from a technical perspective. Since they already use domain storytelling and event storming, they are used to examine the business perspective similarly.

   In an early phase, teams using a Wardley Map categorize activities, services, and technologies, plotting them across the map based on their evolution—from genesis to commodity. By leveraging a Wardley Map, teams gain insights into competitive positioning, resource allocation, and areas requiring innovation or attention. This visualization aids in aligning technical decisions with business goals, ensuring that every team member understands the broader implications of their work.

4. **Introduce formalized documentation**: As the design process evolves, formalized documentation becomes essential. Establishing comprehensive documentation

involves creating clear and concise records of capability maps, Wardley Maps, domain stories, visual glossaries, and event storming outcomes. This documentation is a reference point for existing and new team members, promoting continuity and reducing onboarding time. By maintaining updated documentation, teams can ensure transparency and facilitate better communication within and across teams. Formal documentation should be structured, linked, and easily accessible, allowing team members to quickly find relevant information and understand how their work fits into the larger picture.

5. **Require the blueprint process**: The final phase in evolving the DDD process requires using the blueprint process. The blueprint process is a guiding framework that outlines best practices for implementing and managing the various bounded contexts identified throughout the earlier stages. This entails creating blueprints of the overarching architecture and detailed system architecture documented in canvases and API specifications. By formalizing this process, teams can ensure that their designs adhere to architectural principles and maintain coherence across multiple development efforts. The blueprint serves as a living document, adapting alongside ongoing development while ensuring that teams remain aligned with the overall vision and objectives of the project.

Additionally, it is essential to provide training for all team members. Training programs should include foundational sessions covering domain storytelling and event storming and advanced training formats for business professionals focused on capability mapping and Wardley Mapping. Furthermore, technical training formats should address aggregate design and API specifications to ensure all team members are well-equipped with the necessary skills.

Well-trained team members and an adequately executed DDD process significantly enhance an organization's ability to implement business requirements efficiently and with superior quality. When team members understand DDD principles, including domain storytelling, event storming, and architecture documentation, they are better equipped to comprehend complex business needs and translate them into effective technical solutions.

A well-implemented DDD process fosters collaboration across diverse expertise, encouraging clear communication among business stakeholders and development teams. This alignment helps minimize misunderstandings and ensures everyone has a shared vision of the objectives. Consequently, resources can be allocated more effectively, as team members are adept at identifying critical areas that require attention and improvement.

Moreover, a structured DDD process promotes best practices in software development, allowing for improved code quality and maintainability. By emphasizing the importance of well-defined bounded contexts, the teams can reduce dependencies and streamline their workflows, which leads to faster delivery times. When all team members understand the bounded context and the related technical design, they can work more autonomously and collaboratively. This boosts productivity and empowers individuals to take ownership of their work, leading to more innovative solutions.

Conclusion

In conclusion, investing in training and establishing a robust DDD framework lays the groundwork for achieving business objectives more swiftly and efficiently. The combination of skilled team members and a cohesive design approach ensures that the solutions delivered are timely and meet high-quality standards, ultimately driving success for the organization.

We discussed the DDD process in two examples. The online library and the rent-a-bike shop were discussed as greenfield projects, starting from the idea to implementing bounded contexts. The DDD process supports a deeper understanding of the business domain and, insofar as better and more efficient software.

Today, the majority of development projects fall within the category of brownfield projects. A brownfield project refers to an existing codebase developed and modified over many years. While these systems require ongoing maintenance to ensure stability and functionality, they must also be modernized to remain relevant in a fast-evolving technological landscape.

The challenge of brownfield development lies in balancing the preservation of valuable legacy code with the necessity for updates and advancements. Modernizing these long-standing systems can be daunting as organizations strive to adapt to new business requirements and customer expectations.

The following chapter will explore how DDD can serve as a robust methodology for revitalizing aging systems.

Points to remember

- Start a design process with a business model documented in a business model canvas.
- Set up workshops with at most twenty people and at least five.
- Engage as diverse people as possible in the workshops, including customers and sponsors.
- Ensure that each attendee can see the canvas clearly and be involved in the discussion similarly.
- Create a capability map and a Wardley Map.
- Conduct domain storytelling and event storming.
- Create a visual glossary.
- Create a context map and define asynchronous and synchronous data flows.

- Define synchronous APIs in the context map for user interactions.
- Document the workshops in a blueprint with the following:
    - Project scope documented as business model canvas, including quality requirements and stakeholders,
    - Constraints such as business constraints and technical constraints,
    - Business context is documented in domain stories and visual glossaries, technical context is reported in event storming outcomes, and
    - The solution approach is documented as a context map and documented data flow.
- Document the workshop results using their diagrams and explaining tables.
- Document a bounded context with:
    - Bounded context canvas
    - Architecture communication canvas
    - OpenAPI
    - AsyncAPI

Multiple choice questions

1. **Which canvas can be used to define a business model?**
    a. Wardley Map
    b. Bounded context canvas
    c. Business model canvas
    d. Architecture communication canvas

2. **How many attendees should be part of a DDD workshop?**
    a. 2..5
    b. 5..20
    c. 32
    d. More than 25

3. **What map organizes capabilities as core, supportive, and generic?**
    a. Wardley Map
    b. Bounded context canvas
    c. Business model canvas
    d. Capability map

4. **Which map can be used to define the data flow be defined?**
    a. Context map
    b. Capability map
    c. Wardley Map
    d. Bounded context canvas

5. **Who should attend the workshops?**
    a. Only developers
    b. As diverse people as possible, including customers and sponsors
    c. Developers and architects
    d. Only businesspeople

6. **What part of the design process is a blueprint documenting?**
    a. The overarching architecture and business goal
    b. The architecture of a bounded context
    c. Only the technical solution architecture
    d. The architecture of the infrastructure

7. **What is needed to document a bounded context sufficiently**
    a. Only bounded context canvas
    b. Only architecture communication canvas
    c. Only API specifications
    d. Bounded context canvas, architecture communication canvas, API specification

Answers

1. **c**
2. **b**
3. **d**
4. **a**
5. **b**
6. **a**
7. **d**

CHAPTER 15

Domain-driven Design in a Brownfield Project

Introduction

The chapter discusses how Domain-Driven Design can be applied to a brownfield project and methods for achieving a resilient and sustainable architecture when refactoring a legacy system.

Structure

In this chapter, we will cover the following topics:

- Overview
- Physical library
- Renaissance of the domain
- Comparison of the legacy and target architecture
- Decoupling layer
- Modernization layer
- Step-by-step approach for target architecture
- Remaining functionality
- Organizational changes

Objectives

In this chapter, you will learn how to analyze and model a system to be modernized. You will see how to apply the DDD methodology to formulate the modernized architecture and plan the process.

> **A room without books is like a body without a soul.**
>
> *-Marcus Tullius Cicero*

Overview

In *Chapter 5, Domain Storytelling, Chapter 6, Event Storming, and Chapter 7, Context Map*, we discussed the development of an online library from the first idea to the strategic design.

In most cases today, an existing system has served the requirements. Implementing new features is difficult and takes a long time because many dependencies must be considered.

By applying DDD principles, teams can better understand the existing domain and identify critical areas for improvement. This structured approach will facilitate the gradual modernization of the system, allowing for the introduction of new technologies and architectural patterns without compromising the integrity of the original codebase. Ultimately, DDD can provide a blueprint for navigating the complexities of brownfield projects, enabling organizations to evolve their systems while continuing to deliver value.

Now, we want to discuss how teams can benefit from DDD in a brownfield project using an existing library system as an example.

Let us assume a seasoned library system, which should be modernized so members can access the information online, make reservations for physical books, and read digital books online.

Physical library

A community's library should be modernized.

It contains physical books that library members can borrow. *Figure 15.1* shows the current architecture of the library:

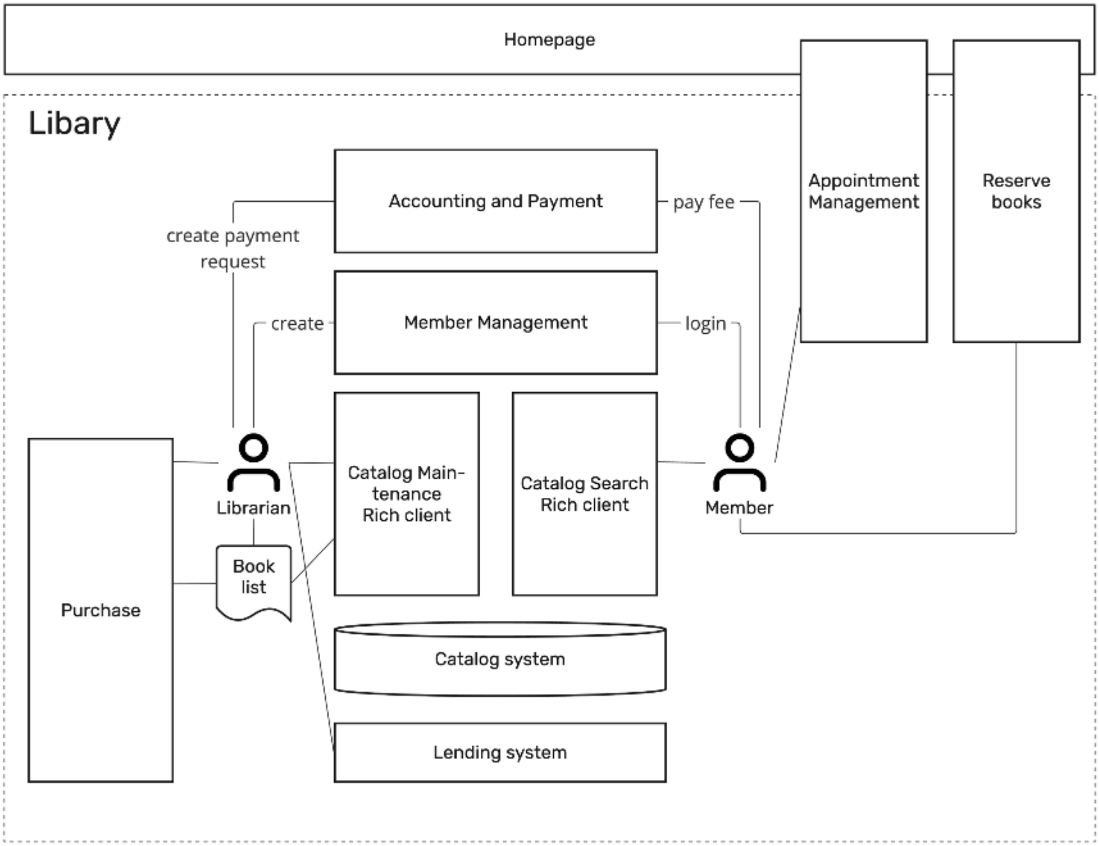


Figure 15.1*: Architecture overview of the community library*

Source*: PhysicalLibrary.jpg*

The library's homepage features a public section that provides essential information, such as opening hours and book reviews. Additionally, it includes restricted areas for appointment management and a module for reserving books.

Through the appointment management system, members can schedule discussions about books, which typically take place in the library's coffee corner or at members' homes.

Members can provide the book's library code to reserve it, allowing librarians to set it aside at the library desk. However, reservations cannot be made solely by author or title, and the reservation system does not support any search functionality for locating books.

When searching for books, a member can use library stations, allowing a fat client to search for books by title, author, keywords, etc. The search provides the book's metadata as a response enhanced by the place in the library of the physical book.

To utilize the search function, members must log into the station using their username and password, which are the same credentials for accessing restricted homepage areas.

Librarians manage member accounts, including creating, updating, and deleting memberships.

To become a member, individuals must visit the library and present a valid identity card for verification. Librarians handle member management tasks and generate payment requests, allowing members to pay their membership fees. Payments and payment requests are handled in the accounting and payment module, which librarians need to log into. A **single-sign-on** (**SSO**) is not available.

Members should present a book at the librarian's desk when they want to borrow it. The librarian will input the book's library code and the member's identification into the lending system, which will then record the due date for returning the book. Additionally, the librarian will annotate the expected return date within the book itself.

When a member exceeds the expected return date, the lending system will create a payment request in the accounting system. The member needs to pay a punishment fee for exceeding the return date.

Librarians are responsible for maintaining the library collection. They identify items that need to be replaced or new books that should be added to the collection. To procure books, they utilize the purchase module, which allows them to order from various suppliers and manage the budget for replacing existing items and acquiring new materials. The purchase module requires a separate login.

At the end of each day, the responsible librarian exports a list of all newly arrived books into an Excel spreadsheet. This spreadsheet is then imported into the catalog system, ensuring that all new acquisitions are correctly recorded. Within the catalog system, the librarian then updates the location of each book, organizing them according to their designated shelves.

Library members developed the catalog system to support their community. It contains the library collection, which the librarians maintain, and it is searched for books by members. The maintenance and the search use the same database.

The library contains exclusively physical books and does not have any digital offerings. Members wish to allow the procurement of digital exemplars, which are accessible online. Moreover, young members especially want to enable remote appointments to discuss books without meeting in person.

The library system has become outdated and requires significant modernization to meet its members' evolving needs. Today's users increasingly seek more extraordinary digital offerings and flexible access to the library collection. Beyond the availability of online digital books, members express a strong desire for the capability to search for titles online and reserve books for specific dates.

Additionally, they want the option to schedule appointments both in person and remotely, allowing for greater convenience and accessibility. To address these demands, the library system must undergo a comprehensive reengineering process to address these demands.

This modernization effort should focus not only on enhancing digital access but also on improving user experience through intuitive interfaces and streamlined processes. The library can better serve its community by integrating advanced technology and user-friendly features, ensuring that resources are accessible to all members, and fostering a more engaging and interactive library environment.

In the next section, we want to define the target of such a modernization before we discuss the step-by-step approach to reaching it.

Renaissance of the domain

When defining a target system, more than merely modernizing the technical components is required. Over time, the usage and significance of these components have evolved. Therefore, it is essential to reevaluate the domain's essence within the new context of how the system is intended to function in the future[1]. This involves understanding not only the current technological landscape but also the needs and expectations of users to ensure that the system is effectively aligned with future requirements.

A starting point in such a situation can be the definition of the vision statement.

Vision statement

A vision statement contains the most essential and product outcomes. It might help to define the target of the modernization.

A **North Star Metric** (**NSM**)[2][3] can help formulate a vision statement. Specific frameworks and boards, such as the board provided by *John Cutler*[4], can be used to find the right NSM. Formulating a vision statement takes work and requires many discussions. Anyhow, using the following checklist helps to find the right balance between endless discussions and a valuable vision statement, which allows teams to align their efforts[5]:

- Does the statement express the customer value?
- Does it represent your vision and strategy?
- Is it a leading indicator of success?
- Is it actionable?

---

[1] Lilienthal, C., & Schwentner, H. (2023). *Domain-Driven Transformation.* dpunkt.verlag.

[2] North Star is meant to help navigate the ocean. It works on the northern half of the earth. On the southern part of the earth, it probably should be called the Southern Cross, meaning the star constellation used to navigate the sea.

[3] Ellis, S. (2017, June 5). *What is a North Star Metric*. Retrieved 2024 October, from blog.growthhackers.com: **https://blog.growthhackers.com/what-is-a-north-star-metric-b31a8512923f**

[4] Cutler, J. (2022, August 13). *North Star Framework Template & Activity Library*. Retrieved October 2024, from The Beautiful Mess: **https://cutlefish.substack.com/p/north-star-framework-template-and**

[5] Kouzmanoff, A., Bashir, I., Cutler, J., & Clark, T. (2024).

- Is it understandable to non-technical partners?
- Is it measurable?
- Is it not a vanity metric?

When we define the targets of library modernization, we need to check our targets against these checkpoints against the statement.

The statement is developed with a mind-storming workshop, following the principles of *Osborn*[6] and *Miller*[7], as already discussed in *Chapter 6, Event Storming*:

- Focus on quantity, not quality (quantity will lead to quality later)
- Withhold evaluation in a first step
- Encourage wild, outlandish ideas
- Combine or build ideas from others

The first step in creating a statement is collecting, consolidating, and combining ideas. *Figure 15.2* shows a collection of ideas:

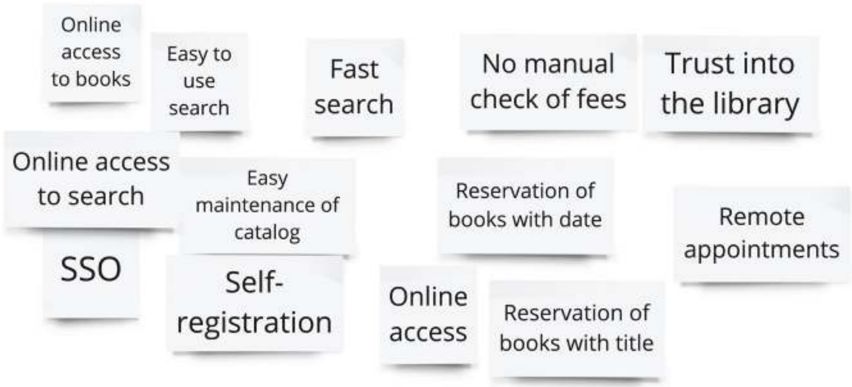


Figure 15.2: *Ideas collected for the modernization of the library*

Source: *Ideas.jpg*

To formulate a vision statement out of those would mean a list of features like the following:

The community library offers digital books online. All books can be searched and reserved online. Appointments can be made in person or remotely. Liberians spend no time in catalog maintenance. New arrivals are entered 100% automatically.

---

[6] Osborn, A. F. (1953). *Applied imagination: principles and procedures of creative thinking.* New York: Charles Scribner's Sons.

[7] Miller, B. C. (2012). *Quick brainstorming activities for busy managers: 50 exercises to spark your team's creativity and get results fast.*

Anyhow, it helps to formulate the essence of it and the target of the community:

Note: Our library is the center of our community. People can meet here and follow their interests. All member of the community can use the library services independent of their age, online affinity, or mobility. People with limited mobility or time can use the library's services online. At least 95% of the citizens of our community are members of the library and trust it as a public service. Librarians consult the members, propose new books, and organize reading circles. Tedious work like manually importing new arrivals to the catalog or creating payment statements is 100% automated so that the librarians can serve the community 100% with their knowledge and passion. The modernization of library software allows us to reach those goals.

Let us check if the vision statement fulfills the requirements of the NSM:

- **Customer value**: The vision expresses customer value through a *center of community*.
- **Vision and strategy**: The statement expresses a community center's vision by engaging all community members through different channels.
- **Leading indicator of success**: Indicators of success are the ratio of library members to the number of citizens and the ratio of automated processes for librarians.
- **Actionable**: The statement is actionable. The modernization of the software is the tool to reach the goals.
- **Measurable**: The statement gives metrics to measure the success of the action.
- **No vanity metrics**: The metrics are not vanity metrics because they state a clear target that modernization can reach in the future. The metrics are not metrics directed to the past.

Before implementing modernization, we must define the necessary business changes based on the vision statement. We use domain storytelling to detail the business requirements.

Domain storytelling

In modernization, we usually do not need a business model and capability map, as discussed in *Chapter 5, Domain Storytelling,* because most are already in place. However, we need domain storytelling workshops and a process, as *Lilienthal* and *Schwenter*[8] proposed.

In the corresponding workshops, the business experts tell a story about the current situation, and afterward, the intended situation is discussed and documented as a domain story. In the following, we will discuss both the current and intended situations. However, to make the reading easier, bounded contexts are already marked in the stories, even though they are discussed after the domain stories are finished.

We start with the purchasing of books.

---

[8] Lilienthal, C., & Schwentner, H. (2023).

Purchasing books

To describe the purchasing of books, we first model the librarians' current situation. In the second step, we model the intended situation. *Figure 15.4* shows the current situation:

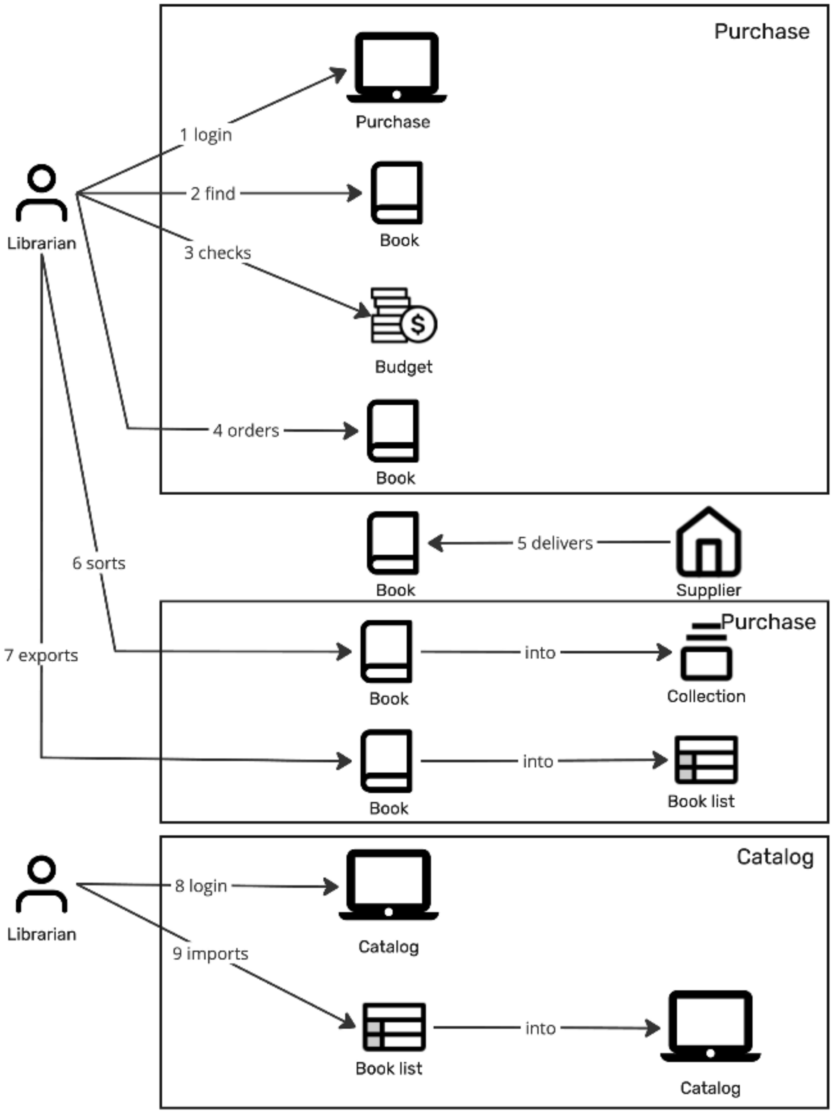


Figure 15.3: *Purchasing and cataloging of physical books—current situation*

Source: *PurchaseBooksCurrentSituation.jpg*

The current situation can be documented as intended situations using corresponding tables, as discussed in *Chapter 14, Usage of Domain-Driven Design in a Greenfield*. The explanation is documented in *Table 15.1*:

No.	Bounded context	Sentence	Note
1	Purchase	The librarian logs into the purchase system.	
2		The librarian finds a book, they want to purchase.	The purchase system allows access to different supplier catalogs.
3		The librarian checks the budget.	The budget is sufficient for the purchase.
4		The librarian orders the book.	
5		The supplier delivers the book.	
6	Purchase	The librarian takes the book in the collection.	To sort a book into collection means to give it a tag with the library code. To stamp it with an ex libris and to put it into the belonging bookshelf in the library.
7		The librarian exports all new arrived books into a list.	The list contains all books sorted into the collection of the day.
8	Catalog	The librarian logs into the catalog.	
9		The librarian imports the book list into the catalog.	

***Table 15.1**: Current situation of purchasing and cataloging books*

The intended situation should avoid tedious and formal work for librarians. Book purchases should stay the same; suppliers' catalogs already contain digital books. The intended situation is shown in *Figure 15.5*. The unchanged parts of the system are shown in grey:

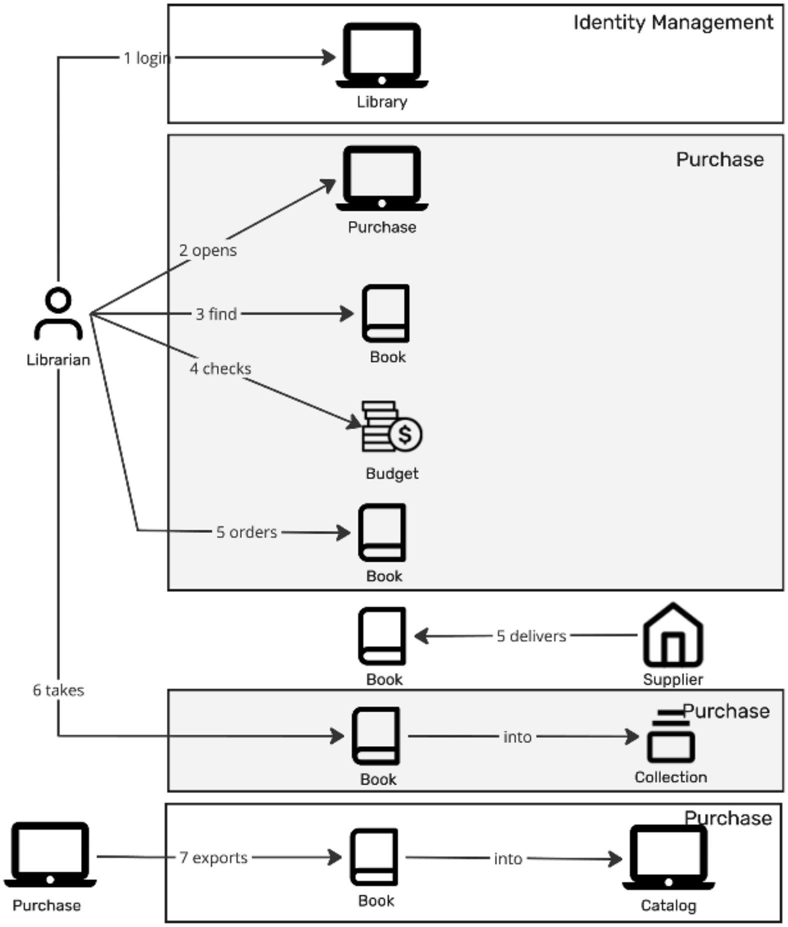


***Figure 15.4**: Intended situation of purchasing books*

***Source**: PurchaseBookIntendedSituation.jpg*

In the intended situation, automated processes take over formalized work from the librarian. The original librarian's work stays unchanged. The unchanged parts are marked in grey in *Figure 15.5*. The changed parts are described in *Table 15.2*:

No.	Bounded context	Sentence	Note
1	Identity management	The librarian logs into the library system.	A single login into the different systems is no longer necessary. The systems access an SSO.
7	Purchase	The purchase system exports the newly arrived books to the catalog.	

Table 15.2

Library members need to be onboarded to lend books so that the library can become a community center.

Onboarding of new members

The onboarding of new members is tiresome and error-prone. It requires a lot of manual work from the librarians. The current situation is displayed in *Figure 15.6*:

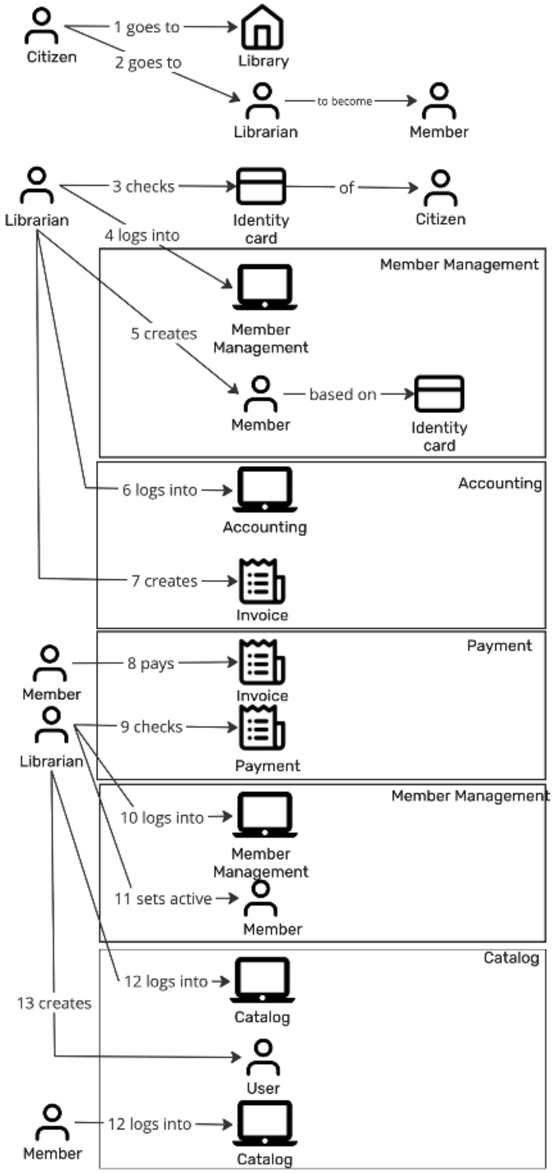


Figure 15.5*: Current situation to onboard a library member*

Source

The domain story is explained in *Table 15.3*:

No.	Bounded context	Sentence	Note
1		A citizen goes to the library.	
2		The citizen goes to the librarian to become a member.	
3		The librarian checks the identity card of the citizen.	
4	Member management	The librarian logs into the member management system.	
5		The librarian creates the member based on the information on the identity card.	
6	Accounting	The librarian logs into the accounting.	
7		The librarian creates an invoice for the citizen.	
8	Payment	The member pays the invoice.	
9		The librarian checks the payment.	
10	Member management	The librarian logs into the member management.	
11		The librarian sets the member to active.	
12	Catalog	The librarian logs into the catalog.	
13		The librarian creates a user in the catalog system.	
14		The member logs into the catalog.	

***Table 15.3**: Onboarding process of new members, current situation*

The situation is not acceptable. Members should be onboarded online and with a low threshold. The community only wants to check that library members are citizens to provide a discount. The discount is financed out of taxes paid by community citizens. Other users can become members but must pay a higher membership fee.

The intended situation is similar to the member management of the online library discussed in *Chapter 5, Domain Storytelling*. It is enhanced by invoking a citizen register and a fee calculation. The intended situation of onboarding members is shown in *Figure 15.7*:

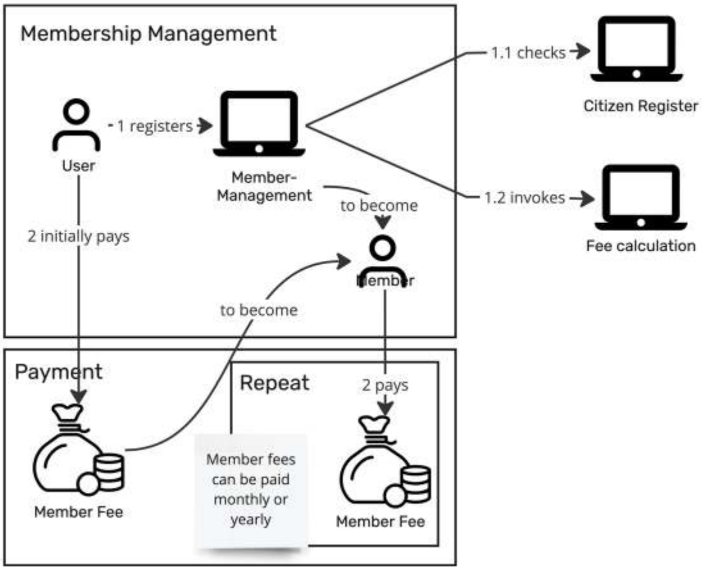


Figure 15.6: *Onboarding process, intended situation*
Source: *OnboardingProcessIntendedSituation.jpg*

The next part will discuss how library members can reserve physical books in the future.

Reservation of books

Currently, members enter the library code of the book into a specific field on the homepage. The librarian receives an email notification and can reserve the book. The current situation is shown in *Figure 15.8*:

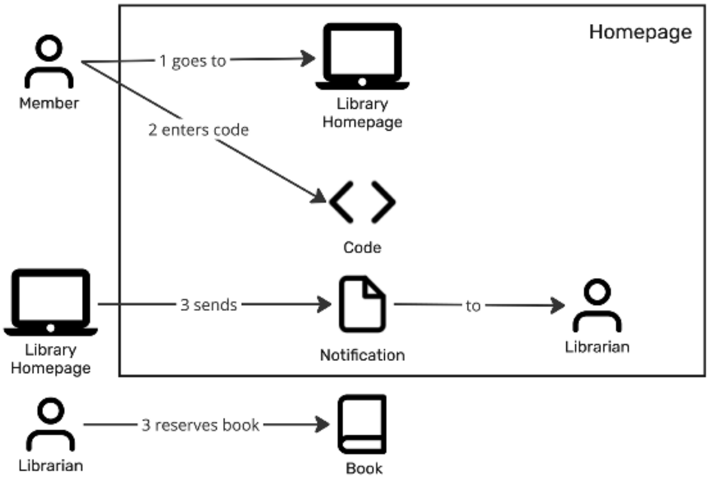


Figure 15.7: *Reservation of books current situation*
Source

The description of the situation is given in *Table 15.3*:

No.	Bounded Context	Sentence	Note
1	Homepage	The member goes to the library homepage.	The member must be logged in via username and password.
2		The member enters the library code for the book.	The member needs to know the code. A search for a book is not possible.
3		The library homepage sends a notification to the librarian.	The notification takes the form of an email and contains the library code of the book and the name of the library member.
4		The librarian reserves the book for the member.	

Table 15.4: *Current situation of reservation of books*

The current situation is unsatisfying because the member needs to go to the library to search for the book and note down the library code. The homepage functionality is well-meaning but has yet to be used by the members. The intended situation is entirely different. It is shown in *Figure 15.7*:

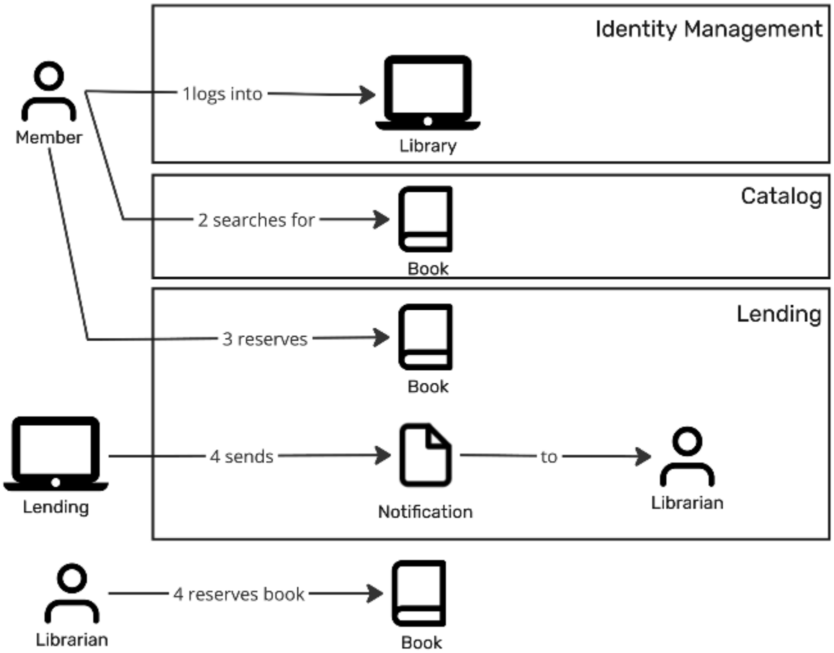


Figure 15.8: *Intended situation for reservation of books*

Source: *ReservationOfBooksIntendedSituation.jpg*

The situation is entirely different because all library functions are available at the library stations and online. Anyhow, the librarian has to reserve the physical book in both cases. The new reservation of books is described in *Table 15.4*:

No.	Bounded context	Sentence	Note
1	Identity management	The member logs into the library system.	
2	Catalog	The member search for a book.	The search for physical books and digital books is not different.
3	Lending	The member reserves the book.	
4		The lending system sends a notification via email to the librarian.	The functionality is no longer function of the homepage. It is integrated into the lending system.

***Table 15.5**: Reservation of books intended situation*

When members have reserved a book, they want to borrow it. The corresponding lending process is described in the next section.

Lending of books

To lend and read a book, a library member must go to the library at the moment, as shown in *Figure 15.9*:

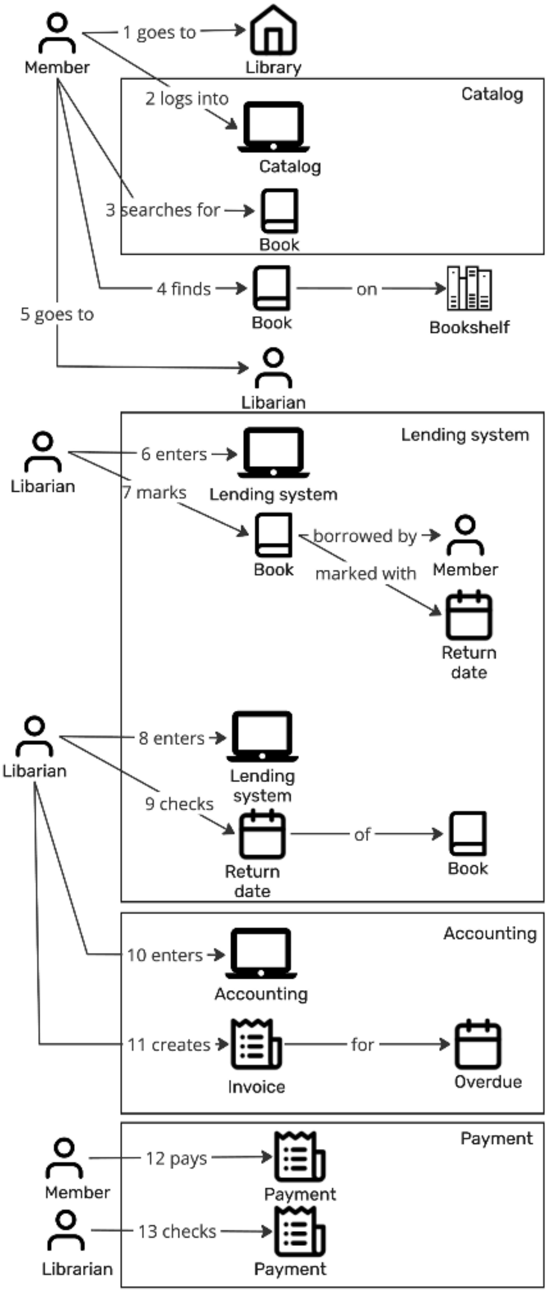


***Figure 15.9**: Current situation to lend a book*

***Source**: LendingCurrentSituation.jpg*

The current situation relies on a physical library and physical books. *Table 15.6* explains the domain story:

No.	Bounded context	Sentence	Note
1		A member goes to the library.	
2	Catalog	The member logs into the catalog system.	
3		The member searches for a book.	The search is not comfortable and needs to be reworked.
4		The member finds a book on the bookshelf in the library.	
5		The member goes to the librarian at the desk.	
6	Lending	The librarian enters the lending system.	We assume that the librarians are logged in the entire day.
7		The librarian marks the books by the member and with a return date.	
8		The librarian enters the lending system.	The checks are done at the end of the day. The librarian is still logged in.
9		The librarian checks the return dates of the lent books.	
10	Accounting	The librarian enters accounting.	We assume that the librarian has been logged into accounting before.
11		The librarian creates an invoice for overdue books.	Punishment fees are configured in accounting.
12	Payment	The member pays the invoice.	
13		The librarian checks the payment.	

***Table 15.6**: Lending of books; current situation*

In the future, members should be able to lend books digitally and physically. Punishment fees for overdue books should be created automatically to release librarians from tedious work. Additionally, it would be possible to borrow digital books through a standard catalog search as physical books. The intended situation is shown in *Figure 15.10*:

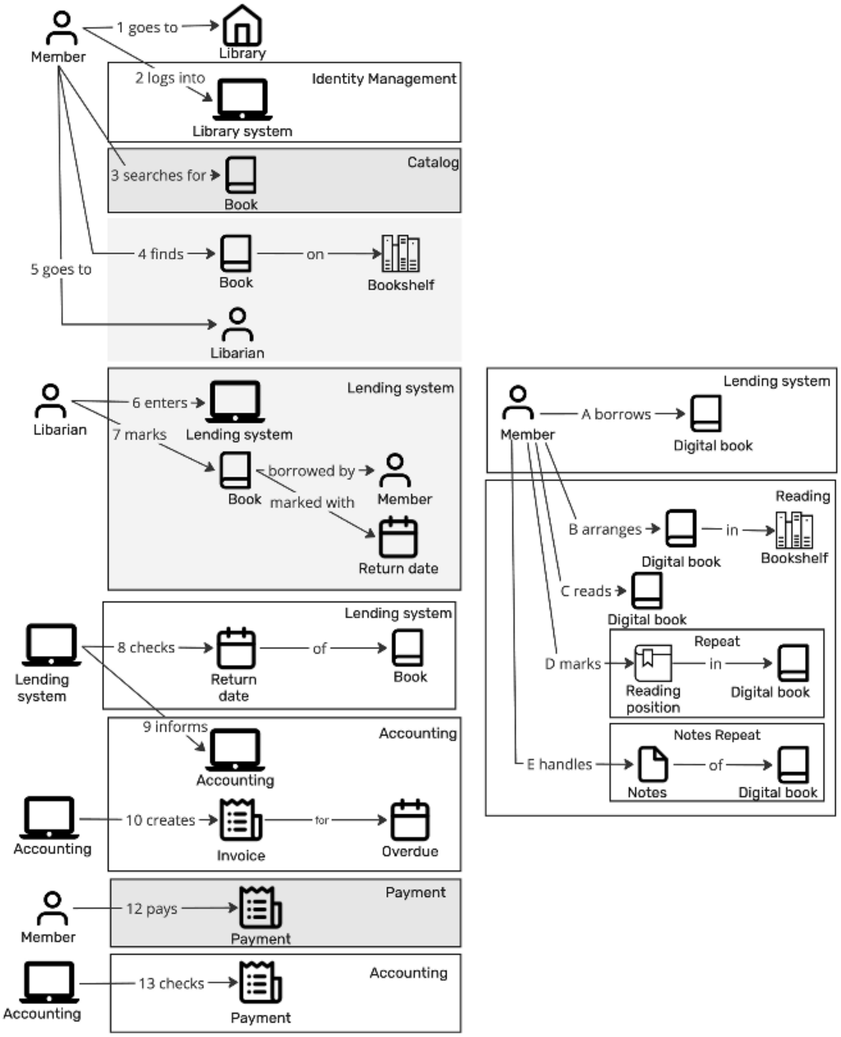


Figure 15.10*: Intended situation for lending books*

Source*: LendingBooksIntendedSituation.jpg*

The domain story contains digital reading, similar to the lending and reading part in *Chapter 5, Domain Storytelling*. The unchanged parts are marked in grey compared to the current system.

Anyhow, all new parts of the domain story are explained in *Table 15.7*:

No.	Bounded Context	Sentence	Note
1		A member goes to the library.	
2	Identity Management	The member logs into the library system.	
3	Lending system	The member borrows the digital book.	Digital reading
4	Reading	The member arranges the digital book on the bookshelf.	Digital reading
5		The member reads the digital book.	Digital reading
6		The member marks the reading position in the digital book.	Digital reading
7		The member handles notes of digital books.	Digital reading Handles mean creation, updating, and deletion of notes.
8	Lending system	The lending system checks the return dates of books.	
9	Accounting	The lending system informs accounting about overdue books.	
10		Accounting creates an invoice for overdue books.	Punishment fees are configured in accounting.
11		Accounting checks the payment.	

***Table 15.7**: Intended situation of lending*

The context map can be created directly based on the analyzed domain stories. The event storming step can be skipped because the domain is well-known to business experts and IT specialists. Event storming can be applied in the tactical design of a bounded context when the internal processes are complex and need to be visualized.

The following section will discuss the context map of the target behavior.

Context map

The context map shows the bounded context determined during the domain storytelling; see *Figure 15.11*:

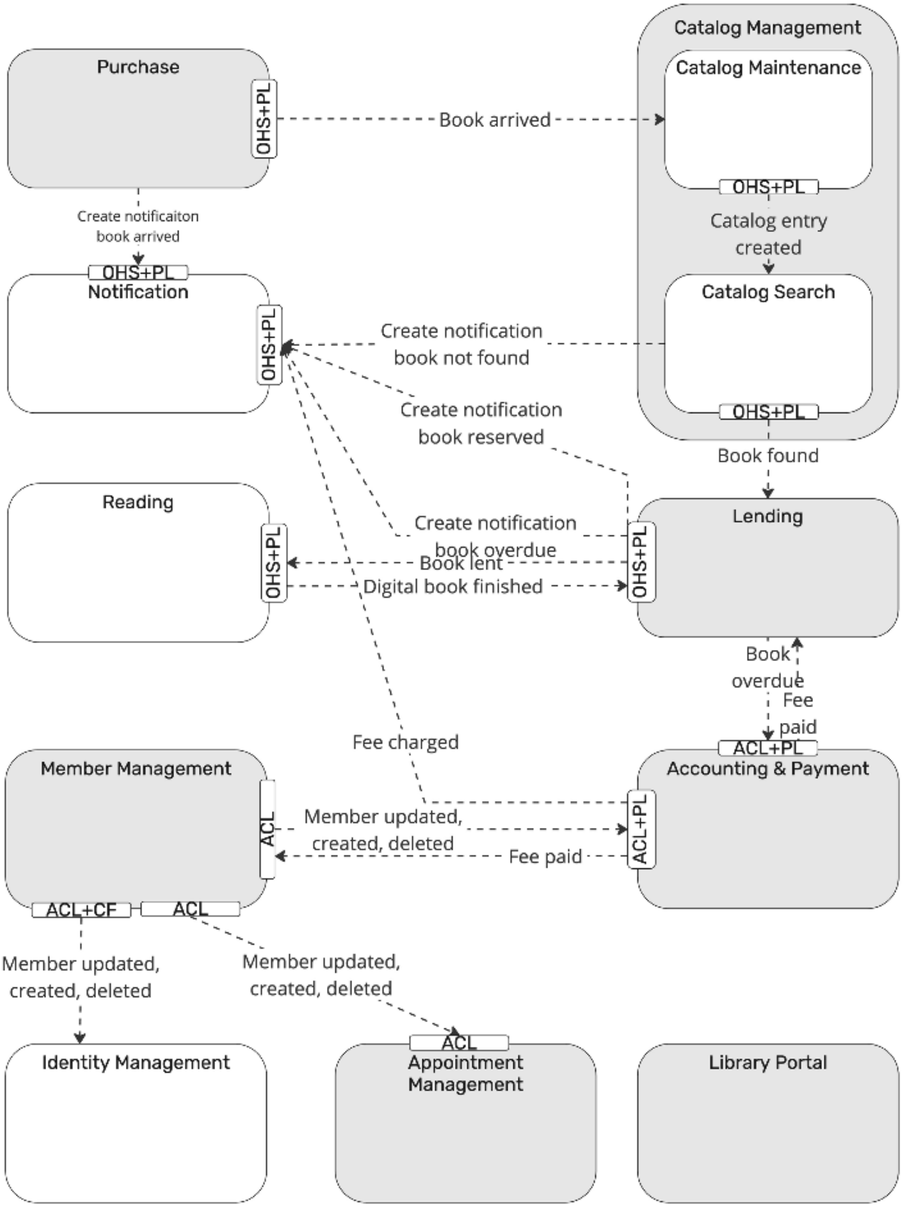


***Figure 15.11**: Context map of the library's target*

***Source**: ContextMapTarget.jpg*

The context map contains parts that need to be developed and parts that need to be adopted. In *Figure 15.11*, the new parts are white, and the others are grey.

Bounded context Purchase allows librarians to purchase and integrate books into the library's collection. This is valid for physical as well as digital books. After a book arrived and was integrated, Purchase produces an appropriate event. Additionally, a command for Notification is produced to inform the librarians about the new book.

Catalog Maintenance consumes the Book Arrived event. Based on this event, the corresponding catalog entry is created. When the catalog entry was created, an event Catalog entry created is produced.

Catalog search consumes the event.

As discussed in *Chapter 7*, *Context Map*, a CQRS pattern is beneficial for catalog management. Therefore, two services represent the Catalog Management bounded context.

When a book cannot be found, a command is produced to send the information to the librarian via Notification context.

The Book found event is consumed by Lending. Lending produces a corresponding event consumed by Reading when a digital book is lent, and Reading itself produces an event when the reader finishes a book. Additionally, Lending produces two Notification commands, one to inform the librarian and member that a book is overdue and the other to notify the librarian that a member reserved a book.

Lending produces an event consumed by Accounting & Payment when a physical book is overdue. Accounting & Payment produces the corresponding event when the corresponding fee is paid. Additionally, both services produce commands to Notification to inform librarians and members about overdue books and fees to be paid.

Member Management produces an event when a new member is created, or members are updated or deleted. These events are consumed by Accounting & Payment to charge the belonging fees and by Identity Management for access control.

The Library Portal manages the micro-frontends of the bounded contexts and the single-SSO.

Appointment Management can be used to arrange appointments in the same way as today. To arrange appointments comfortably, members can see other members. Therefore, the event from Member Management is consumed.

Wardley Map

Using the described contexts, a Wardley Map[9] can be created. The Wardley Map for the target library is shown in *Figure 15.12*:

---

[9] Wardley, S. (2022). *Wardley Maps*. Simon Wardley.

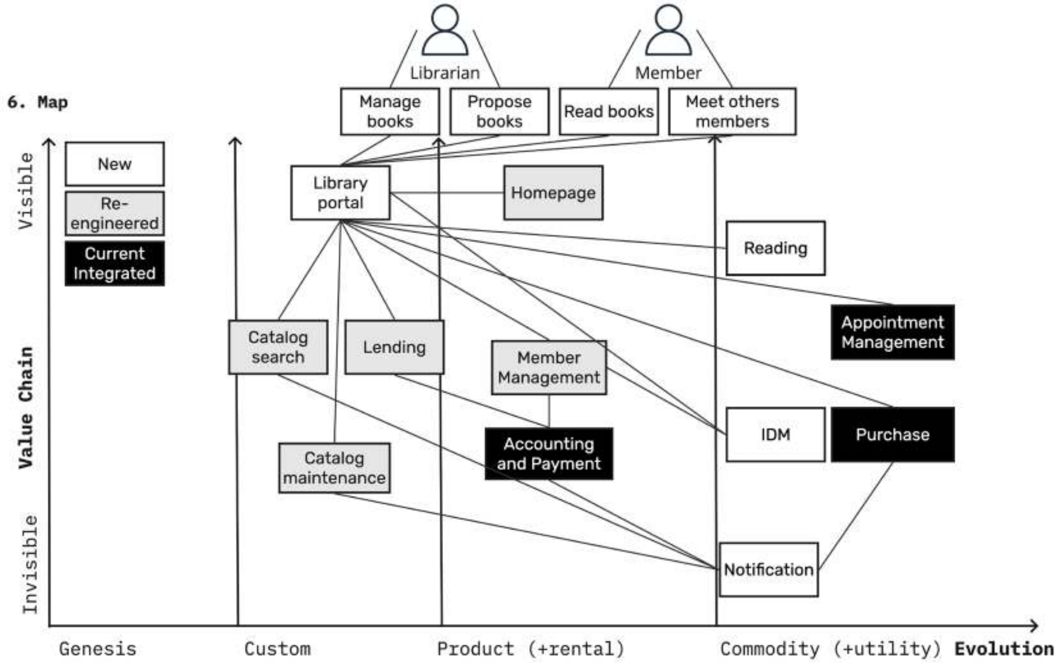


***Figure 15.12**: Wardley Map of the library to be modernized*

***Source**: WardleyMap.jpg*

The Wardley map shows the bounded contexts in white when they need to be implemented or purchased new. Grey-bounded contexts are available but need to be re-engineered. Black-bounded contexts are available and must be adopted to integrate them into the new portal and SSO.

It suggests that the portal needs to be developed as a custom project. Custom development teams must re-engineer Catalog Search, Lending, and Catalog Maintenance. This aligns with the North Star statement to automate library processes as much as possible, freeing librarians to consult library members with their knowledge and passion for books.

Homepage and Member Management need to be re-engineered. Accounting and Payment can stay but must be integrated into the new library portal environment. Reading and Notification need to be purchased as commodities.

Reading, **Identity Management** (**IDM**), and Notification are new services that must be purchased as commodities. Appointment Management and Purchase can stay as they are but must be integrated into the new environment.

A target architecture can be designed based on the context and Wardley Maps. We will discuss the target architecture in the following section.

Target architecture

The target architecture should follow the principle of event-driven architecture using events for server-to-server communication, as discussed in *Chapter 12, Exposing Domain Events*.

The library portal takes over the task of the app shell and choreographer of the Micro-frontends of the bounded context. The bounded contexts communicate via events described in the context map; see *Figure 15.11*.

Use cases derived from the domain stories enhance the bounded contexts as services in the target architecture. The resulting architecture is shown in *Figure 15.13*:

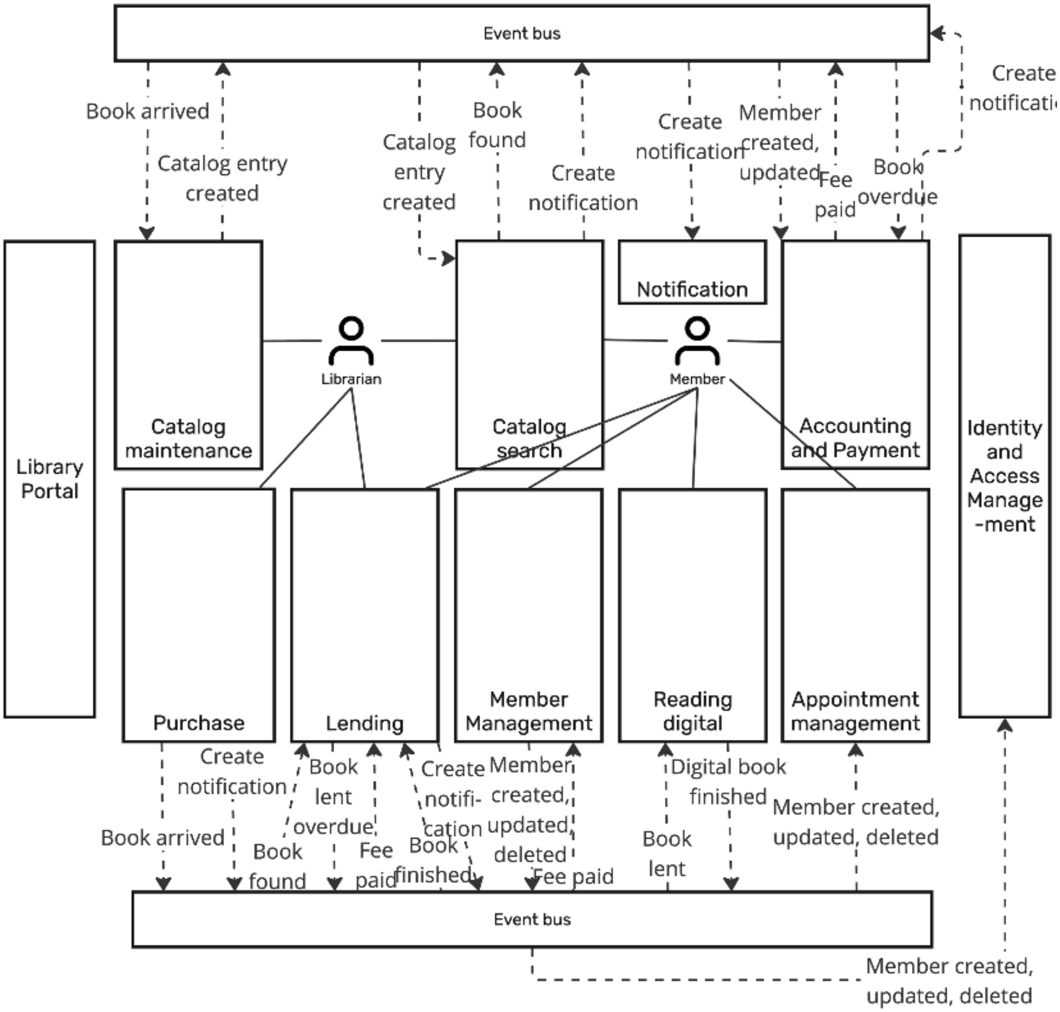


Figure 15.13: *Target architecture of the modernized library*

Source: *TargetArchitecture.jpg*

A new library portal takes over the homepage functionality. An SSO needs to be implemented new.

Using the part of the Catalog Maintenance a librarian can update catalog entries. The initial creation of catalog entries based on an event Book arrived produced by Purchase.

Using the Purchase service, a librarian can search for books to be purchased, check the available budget, order books, and create library codes for physical books. When an ordered book arrives, the belonging event is produced.

Using the Lending service, a librarian can lend physical books. The member can reserve books, return books, and borrow digital books. Additionally, the service checks if physical books are overdue and sends corresponding events.

Librarians and members use the Catalog Search to search for books.

The Notification service sends notifications to librarians and members based on commands produced by other services.

Using Member Management, new members can register as a member, update their membership, or cancel it.

A member can read digital books via the library portal using the Reading service.

Members can arrange appointments with each other using Appointment Management, which consumes membership data from Member Management.

Accounting and Payment creates invoices for membership and punishment fees for overdue books. The fee calculation is part of Accounting and Payment.

The target architecture shows an idealized picture of the future status. In the next step, a step-by-step transition architecture must be described. We must compare the target and legacy architecture to determine the steps needed to modernize the library application. The comparison will be discussed in the next section.

Comparison legacy and target architecture

To determine the transition architecture, the target we overlay the target architecture, as shown in *Figure 15.1*, and the target architecture, shown in *Figure 15.13*. The respective results are shown in *Figure 15.14*:

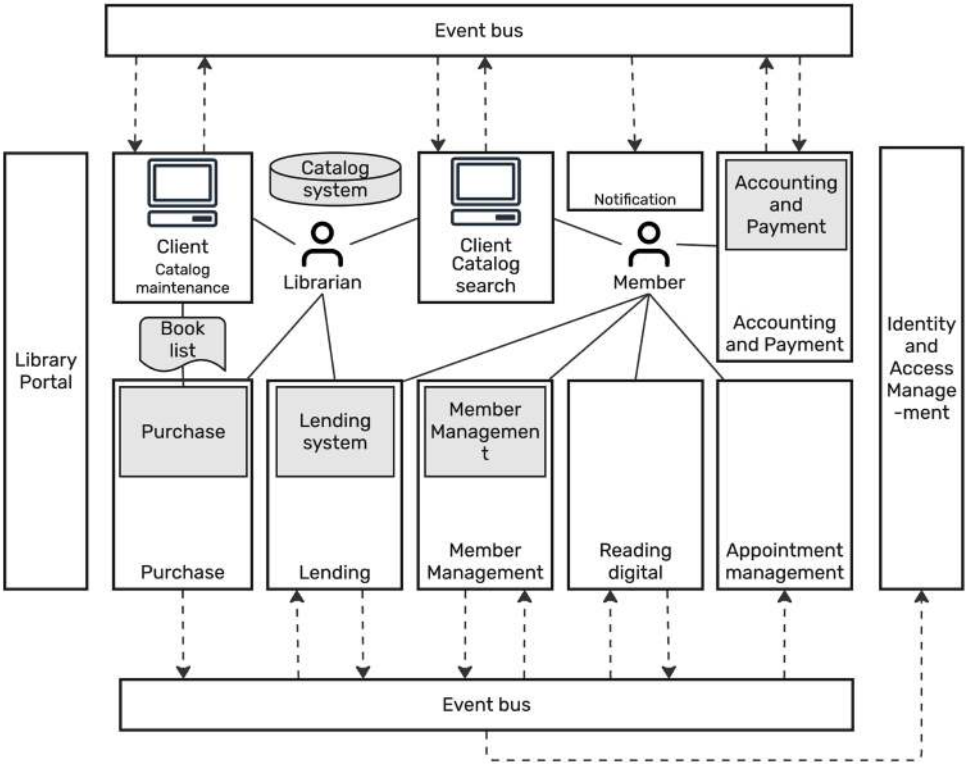


Figure 15.14: Layer of target architecture and legacy

Source: *ArchitectureLayered.jpg*

The layered architecture shows the large areas which need to be developed, such as:

- **Library portal**: The Library Portal is a new user interface module that needs to be developed. It is the portal to all other library services. Therefore, it needs to be designed carefully. Even though it is a custom part of the overall application, specific frameworks for portals and application shells can be used.
- **Catalog maintenance**: Catalog Maintenance provides a rich client. This rich client needs to be substituted by a modern web application integrated into the Library Portal. The catalog system's database is used for Catalog Maintenance, Catalog Search, and Lending. Each service needs its database in the target architecture, and communication is done via events. It is the most significant change in the library system's core. Therefore, it needs to be done with the highest priority.
- **Purchase**: The Purchase area must be automated so librarians can avoid manually exporting the booklist. The solution is events consumed by the catalog system. Because the legacy purchase module covers the current requirements, it can still be used in the future. However, the integration between the Library Portal and the IDM (Identity Management) for login must be implemented.

- **Catalog search**: Catalog Search provides its rich client today based on a central database. The database must be separated according to the target architecture and follow the targeted CQRS pattern for Catalog Maintenance and Catalog Search.
- **Member management**: Member Management is currently a management application exclusive to librarians. In the future, it should provide a self-registration service. The current membership management can stay because it provides a web application that needs to be integrated into the SSO and Library Portal. The self-registration service can be developed parallel to the librarian service, but both need to produce and consume the necessary events.
- **Notification**: The Notification service is new and needs to be implemented to allow the sending of messages and emails to librarians and members.
- **Reading digital books**: The Reading service is a commodity that can be purchased. It follows standards and can easily integrate into the Library Portal and SSO.
- **Accounting and payment**: The Accounting and Payment service can stay as it is. However, it needs to be integrated into SSO and Library Portal. Furthermore, the service must be adapted to produce and consume the corresponding events.
- **Identity and access management**: The IDM system is a commodity. It can usually be purchased as **Software as a Service** (**SaaS**) provided by the cloud provider.

Using this overview, we can discuss the steps of re-engineering the library system. We will concentrate on Catalog Maintenance, Catalog Search, and Lending. First, we need a decoupling layer between the legacy system and the new generation system. The decoupling layer will be discussed in the following section.

Decoupling layer

Using a decoupling layer allows the legacy system to be decoupled from the modern system. It even allows the new system's domain model to be secured from the legacy system's model.

Benefits of decoupling

To create resilient systems that can be developed independently in different teams, the services and business functionalities must be tailored to minimize runtime dependencies. Because of the systems' resilience and the reduced dependencies, the availability of the entire application increases[10].

---

[10] Richardson, C. (2024). *Essential characteristics of the microservice architecture: loosely coupled*. Retrieved October 2024, from microservices.io: **https://microservices.io/post/architecture/2023/03/28/microservice-architecture-essentials-loose-coupling.html**

The *Strangler Fig* pattern after *Fowler* can be applied to reach the transition from the monolith to the modernized system[11]. This means that a decoupling layer hides the legacy system insofar as the legacy model is concerned. The modernized system substitutes the hidden legacy system step by step.

Cartwright, Horn, and Lewis described those processes as challenging and containing many pitfalls. The organization does not allow change as it needs to change, like the technical system. The change is organized with an extensive program, which is constituted of several projects and teams. Those programs often exist parallel to the usual business or **business as usual** (**BAU**). So, the delivering teams of the normal business deliver against the requirements of the modernization teams when they are not closely synchronized[12].

To break this circle, the following categories need to be answered[13]:

- **Understand the outcomes you want to achieve**: We did this by formulating the North Star at the beginning of this chapter.
- **Decide how to break the problem up into smaller parts**: We did this by analyzing the current and the intended business by domain storytelling.
- **Successfully deliver the parts**: We will discuss the *step-by-step approach* later in this section.
- **Change the organization to allow this to happen on an ongoing basis**: We will discuss the organizational challenges later in the section on *Organizational changes*.

Decoupling via capabilities

As we learned before, the monolith needs to be decoupled by capabilities[14]. The capabilities were detected by domain storytelling of the current system and the system-to-be.

We need to model a modern system and hide the anemic model of the legacy system[15]. The anemic model evolved over the years and overarched several business contexts. Another mistake is to reuse unsuitable models because they look like they fit into the business. That leads to large, unfit, and tight coupled models[16]. It is essential to secure the modern model from stains coming from the legacy system. As discussed in *Chapter 4, Bounded Context and Domain*, an anti-corruption layer suits the task.

---

[11] Fowler, M. (2024, August 22). *Strangler Fig*. Retrieved October 2024, from Martin Fowler: **https://martinfowler.com/bliki/StranglerFigApplication.html**

[12] Cartwright, I., Horn, R., & Lewis, J. (2024, March). *Patterns of Legacy Displacement*. Retrieved October 2024, from Martin Fowler: **https://martinfowler.com/articles/patterns-legacy-displacement/**

[13] Cartwright, I., Horn, R., & Lewis, J. (2024, March). *Patterns of Legacy Displacement*. Retrieved October 2024, from Martin Fowler: **https://martinfowler.com/articles/patterns-legacy-displacement/**

[14] Dehghani, Z. (2018, April 24). *How to break a Monolith into Microservices*. Retrieved October 2024, from Martin Fowler: **https://martinfowler.com/articles/break-monolith-into-microservices.html#Decouple-CapabilityAndNotCode**

[15] Lilienthal, C., & Schwentner, H. (2023). *Domain-Driven Transformation.* dpunkt.verlag.

[16] Lilienthal, C., & Schwentner, H. (2023).

Let us discuss how the principles can be applied to the anti-corruption layer in the catalog and lending and how the library system can be modernized.

Decoupling catalog and lending

A decoupling layer decouples a legacy system from a modern system. It targets to hide the legacy application completely as an anti-corruption layer.

The decoupling layer adds adapters to legacy applications that provide the necessary events for service-to-service communication. Additionally, adapters are supplied with synchronous APIs to use modern web clients even onto the legacy system. The adapters work as a modern facade for the legacy systems.

An overview of this transition architecture step is shown in *Figure 15.15*:

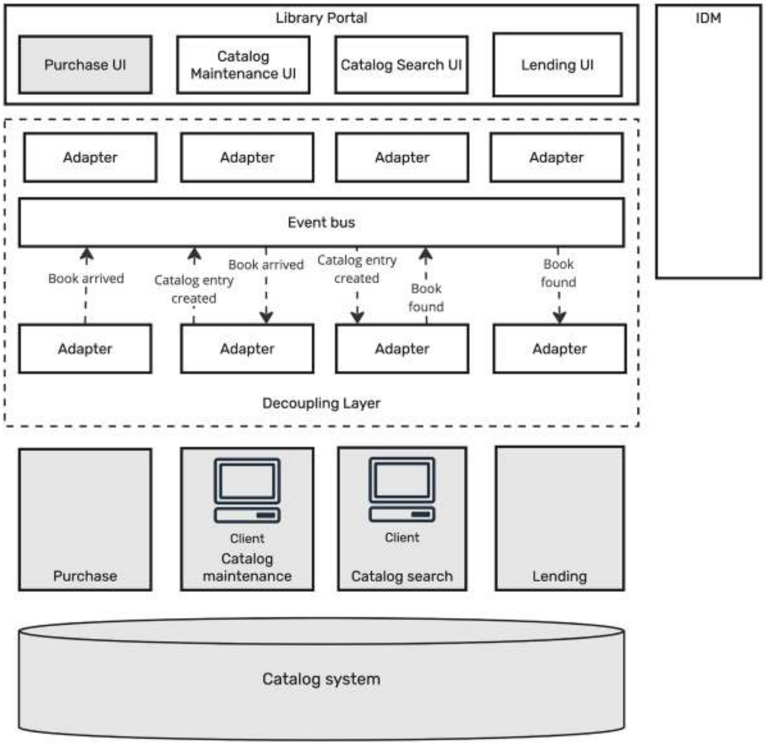


Figure 15.15*: Decoupling layer of a legacy system and a modern system*

Source*: DecouplingLayer.jpg*

On top of the anti-corruption layer, modern **user interfaces** (**UI**) can be added, which already use the Library Portal as a new module. The user interfaces can be implemented as mobile and web applications using the synchronous REST APIs provided by the adapters of the anti-corruption layer. The Purchase UI must only be adapted using the REST interfaces and integrated into the Library Portal.

The rich clients of Catalog Maintenance and Catalog Search are still available for librarians and members. The new clients substitute their features step-by-step.

The events belong to the current features delivered by the current system. As discussed in the next section, additional events belonging to the new modules can be implemented when the modernization layer is added.

Modernization layer

The modernization layer adds services to the architecture, corresponding to bounded contexts determined during the domain storytelling workshops, see *Figure 15.11*. They take over the data ownership from the legacy system, which might still be necessary in exceptional cases.

When introducing a new service, e.g., Catalog Search, the legacy service needs to be set to feature-closed because all new features need to be developed in the new service. The legacy services can be removed when the new services take over all necessary features and data.

The architecture of the system containing Catalog Maintenance, Catalog Search, Purchase, and Lending, the modernization and anti-corruption layer is shown in *Figure 15.16:*

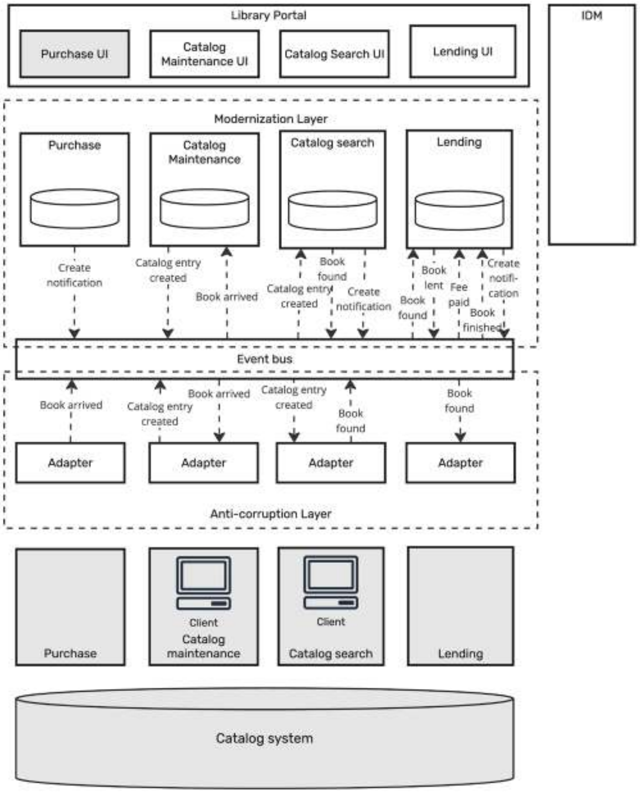


Figure 15.16

The new services introduced produce and consumed the events discussed in the context map (see *Figure 15.11*). The adapters for the REST services are no longer necessary because the services take over this task. However, the REST interface specifications are not changed in this step so that the user interfaces can use the new services without any changes.

Such a system can be implemented step-by-step. The following section will discuss how to do it.

Step-by-step approach for target architecture

According to the Wardley Map in *Figure 15.12*, the catalog, lending functionality, and portal must be implemented first. All other functionalities can follow. The dependencies and rough forecasts are necessary to create a project overview.

Those forecasts are challenging to create because, in an early phase of a project, standard team estimations, as usual in *Scrum*[17] or *Kanban*[18], are not possible because the teams are not built yet. However, business experts and IT specialists working together on domain storytelling, context maps, and architecture overviews can create rough forecasts.

First, they collect the necessary work items based on their knowledge of the program. They can do it for each bounded context separately.

Such a collection is shown for Catalog Maintenance, Catalog Search, and Lending in *Figure 15.17*:

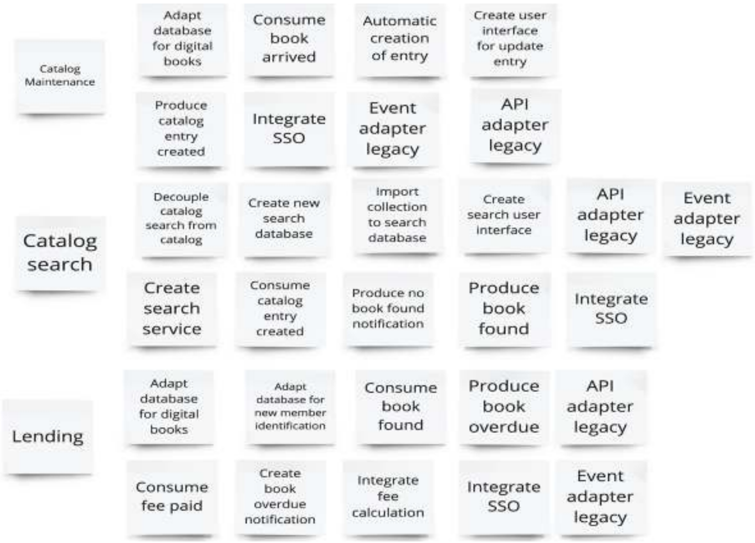

Catalog Maintenance
Adapt database for digital books
Consume book arrived
Automatic creation of entry
Create user interface for update entry
Produce catalog entry created
Integrate SSO
Event adapter legacy
API adapter legacy
Catalog search
Decouple catalog search from catalog
Create new search database
Import collection to search database
Create search user interface
API adapter legacy
Event adapter legacy
Create search service
Consume catalog entry created
Produce no book found notification
Produce book found
Integrate SSO
Lending
Adapt database for digital books
Adapt database for new member identification
Consume book found
Produce book overdue
API adapter legacy
Consume fee paid
Create book overdue notification
Integrate fee calculation
Integrate SSO
Event adapter legacy

***Figure 15.17**: Collection of work items to implement services and integrate them*[19]

***Source**: Forecast.jpg*

---

[17] Moskalenko, S. (2017, October 11). *What Scrum Says About Estimates*. Retrieved October 2024, from Scrum.org: **https://www.scrum.org/resources/blog/what-scrum-says-about-estimates**

[18] Krasteva, I. (2022, July 20). *Kanban Estimation: Is There Such a Thing?* Retrieved October 2024, from businessmap: **https://www.scrum.org/resources/blog/what-scrum-says-about-estimates**

The single work items can now be sorted in a sequence of more, less, or comparable complexity. A short example is shown for the proposed work items of Catalog Search in *Figure 15.18*:

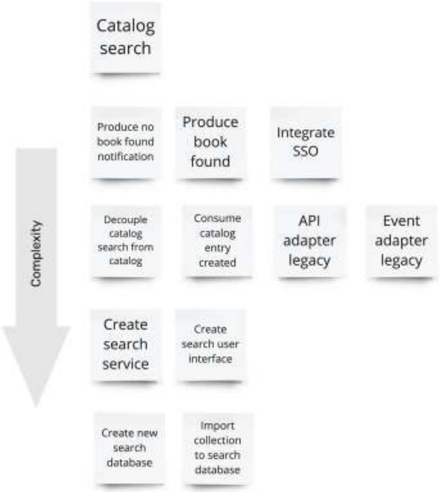

Catalog search
Complexity
Produce no book found notification
Produce book found
Integrate SSO
Decouple catalog search from catalog
Consume catalog entry created
API adapter legacy
Event adapter legacy
Create search service
Create search user interface
Create new search database
Import collection to search database

***Figure 15.18**: Work items sorted by complexity*

***Source**: SortByComplexity.jpg*

Each row can be assigned a number representing complexity. The numbers follow a Fibonacci sequence. Additionally, work items with certain risks can be marked so that the forecast can be adapted accordingly[20].

A project overview can be created using the forecast's complexities and the Wardley Map's dependencies and priorities. The size of the work items can represent the complexity, and arrows can mark the dependencies. The result is shown in *Figure 15.19*:

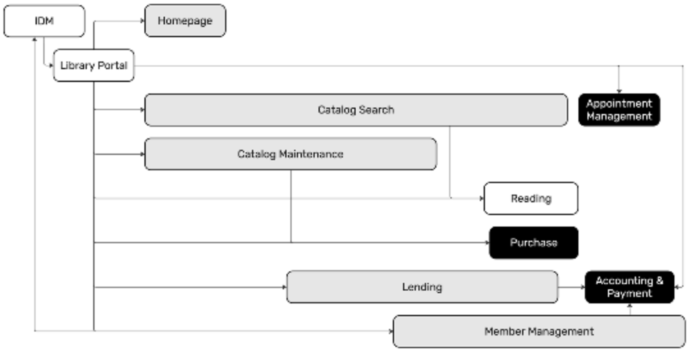


***Figure 15.19**: Project plan of the library to be modernized*[21]

***Source**: ProjectPlan.jpg*

---

[20] Junker, A. (2018, July). *Gib mir eine Zahl—Schätzungen entlang des Entwicklungsprozesses*. Retrieved October 2024, from heise online: **https://www.heise.de/ratgeber/Gib-mir-eine-Zahl-Schaetzungen-entlang-des-Entwicklungsprozesses-4119174.html**

Even though we want to modernize the entire library following the North Star, some functionality might remain. In the next section, we will discuss whether or not the remaining functionality contradicts the approach.

Remaining functionality

When a software system is modernized, not all parts are modernized. This happens sometimes intentionally and sometimes unintentionally. From the business value of collaborative modeling to collaborative project planning, as described in this chapter, unintentional remains do not occur. However, intentional remains happen and can be helpful.

Imagine a library collecting valuable and rare first editions of books. Those books should not be lent as the other books in the library. It makes sense to manage them completely, separated from all other books and media.

So, it might make sense to manage them still in the legacy system as a specialized system for first editions, which can be read only in the presence of a well-schooled librarian.

These exceptions are possible and do not contradict the overall approach. However, the domain storytelling should reveal the business requirements and be documented. In such a case, the modeling team can decide if the according functionality remains after modernization.

Modernization is a long process that requires careful planning and preparation. In addition to technical changes, it requires organizational changes. We will discuss the organizational approaches in the next section.

Organizational changes

An implementation organization must follow the required architecture as *Conway's law*[22] requires, as discussed in *Chapter 3, Why Strategic Design*. When we want to change an architecture, we must first change the organization. Such a move is called **Inverse-Conway-Maneuver**[23].

It would mean for our library a couple of teams:

- A membership and identity management team for Member Management and Identity Management
- A portal team taking over the Library Portal and Homepage
- An accounting and payment team

---

[22] Conway, M. E. (1968, April). How do committees invent? *Datamation magazine*.

[23] Soffner, T., & Cremer, B. (2023, May 26). *Inverse-Conway-Maneuver: How to speed up product development teams successfully*. Retrieved October 2024, from thoughworks: **https://www.thoughtworks.com/**

- A team responsible for catalog management
- A team accountable for lending
- a team responsible for integrating the purchase module
- And a technical integration team handling Appointment Management and Notification

Anyhow, the legacy teams remain. The legacy systems still need to be maintained until they are switched off completely. So legacy teams for catalog management, purchase, and the homepage, including integration, remain. They do not develop new features but fix bugs and integrate new versions of databases or development software. Usually, the most experienced developers stay on those teams because they know the system best.

Meanwhile, the new teams develop existing features already available in the legacy system to substitute them. Anyhow, the new developers in the modernization layer do not know the business as well as the experienced developers. That is a constant problem using Inverse-Conway-Maneuver.

Attempting to formalize those organizational changes as described will be unsuccessful[24]. Organizational changes are about people. Therefore, the people affected should do the reorganization. The modeling team can discuss the changes with the teams and find possible ways.

One way might be for teams responsible for corresponding business contexts to jointly take over the responsibility of maintenance and modernization.

Team members might cycle and be involved in maintenance and modernization teams.

Or entire teams might change responsibility for modernization and maintenance in defined schedules.

However, the solution might depend heavily on the existing organization and needs to be discussed thoroughly, including all affected people.

---

[24] Dupuydauby, C. (2023, February 22). *ICM#1: Say no to the 'Inverse Conway Maneuver'*. Retrieved October 2024, from Medium: **https://medium.com/@Cyrdup/icm-1-say-no-to-the-inverse-conway-maneuver-6672ba2373cb**

Figure 15.20 shows a possible approach:

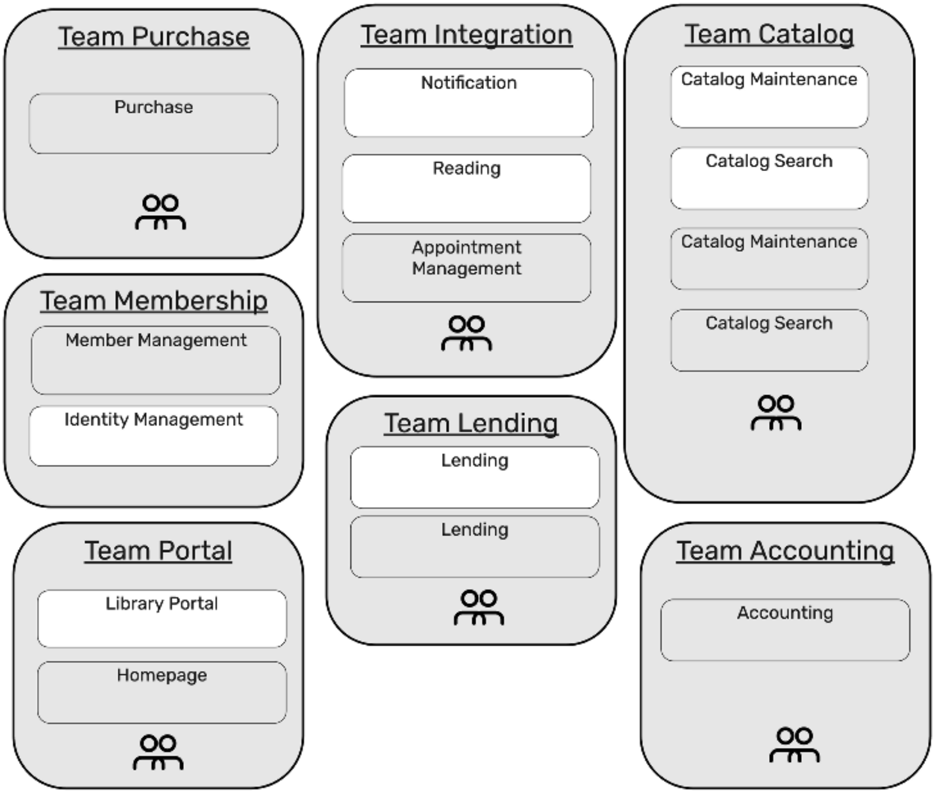


Figure 15.20*: Team organization along the business capabilities*

Source*: TeamApproach.jpg*

The teams are organized according to business capabilities. Some have to integrate the existing modules, such as *Team Accounting* and *Team Purchase*. Others must substitute the existing module with a new one, such as *Team Catalog* and *Team Lending*. The teams are responsible for the entire capability, including modernization and legacy systems. The teams can decide independently from each other which organizational approach between maintenance and new development they want to apply.

What is clear is that the organization needs to change in the same manner as the architecture.

Conclusion

We followed a method for modernizing a system from a North Star, formulating the change's target and transferring it to the team organization for modernization. We saw that DDD can be applied to modernizations as successfully as to greenfield projects.

DDD is an excellent approach to developing successful products. It can be implemented step-by-step in teams, and it supports the necessary collaboration between business experts and IT specialists in workshop formats.

In the last chapter of this book, we will summarize what we learned and what might come next.

Points to remember

- Start with a brownfield project to modernize an existing system and formulate the target. The target can be formulated using a North Star.
- Formulate the domain using domain storytelling and event storming.
- Formulate the current situation to re-event the domain.
- Formulate the target situation using DDD methods.
- Mark parts of the target situation that the existing modules can implement.
- Use the context map to mark new systems and systems to adopt.
- Formulate the target architecture as a greenfield architecture would be formulated.
- Overlay the target architecture with the current architecture to define how the current architecture should be changed to reach the target architecture.
- Introduce an anti-corruption layer to decouple the legacy systems from the new systems. The anti-corruption layer avoids stains from the legacy model to the modern model.
- Introduce a modernization layer using contemporary architecture approaches on top of the anti-corruption layer, hiding the legacy model.
- Plan a step-by-step approach using a rough forecast and prioritization of the Wardley map.
- Allow specific functions to remain in the legacy system.
- Plan organizational changes as carefully as technical changes.

Multiple choice questions

1. **What could be the starting point of a brownfield project?**
   a. The formulation of the target using a North Star
   b. The calculation of the costs
   c. The definition of the delivery date
   d. The definition of remaining functionality

2. **Do I need to formulate the current situation using DDD methods?**
    a. No, it is not necessary
    b. Yes, but only when the software is taken over in a company merger
    c. No, it is not required. Everyone knows the current situation
    d. Yes, the current situation needs to be formulated to re-invent the domain
3. **How can I see if current modules could be used in a modernized system?**
    a. It cannot be seen at all. It needs to be decided later.
    b. It can be marked in the domain stories.
    c. The management needs to define the remaining modules before the start of the project.
    d. There are no remaining modules in a modernized system.
4. **Which tool should be used to prioritize the capabilities?**
    a. North Star
    b. Domain Story
    c. Wardley Map
    d. Context map
5. **Is it possible to formulate a transition architecture without a target?**
    a. No, it is not possible. The target needs to be precise.
    b. Yes, it is possible. Everyone knows the target by the North Star.
    c. Yes, it is possible only when the management has formulated the target.
    d. No, it is not possible. However, the target needs to be formulated after the transition architecture is formulated.
6. **What can be used to decouple a modernized system from a legacy system?**
    a. A conformist
    b. A shared kernel
    c. An anti-corruption layer
    d. An open host service
7. **Should modernization be planned as a one-step approach?**
    a. Yes
    b. Yes, but the one-step should be planned by the management
    c. No, but the plan should contain only three steps in maximum
    d. No, a step-by-step continuous approach is necessary

8. **Are organizational changes essential in a brownfield project in the same manner as in greenfield projects?**
    a. Yes
    b. No
    c. Yes, but only when the project affects more than five teams
    d. Yes, but only when a consulting company proposes the project

Answers

1. **a**
2. **d**
3. **b**
4. **c**
5. **a**
6. **c**
7. **d**
8. **a**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 16

Summary

Introduction

Domain-Driven Design helps master the complexity of software development. Strategic design helps tailor the bounded context along the enterprise's business and find a ubiquitous language. Tactical design helps find suitable application interfaces and events for communication with the outside world and between services.

Overall, using the *Synergetic Blueprint* process, your software will improve, be more successful, and be highly quality.

Structure

This chapter summarizes the process and the chapters of this book with

- Synergetic Blueprint
- Strategic design
- Tactical design
- Applying Domain-Driven Design in projects
- Artifacts of the Synergetic Blueprint process
- Using AI in the process

Objectives

This chapter outlines the steps and artifacts of the Synergetic Blueprint process and the chapters in which they are detailed.

> **To summarize the summary of the summary: people are a problem.**
>
> *-Douglas Adams*

Synergetic Blueprint

Throughout this book, we discussed a process from the first ideas sketched on paper to domain model specifications and code. The process is marked by collaborative and trustful work between quite different disciplines: business experts, IT specialists, UX experts, security specialists, etc. The author calls this process *Synergetic Blueprint*.

The process covers the strategic and tactical Domain-Driven Design. First, let us look at the strategic design.

Strategic design

The strategic design of Domain-Driven Design tailors the problem to smaller, digestible parts. An overview of the presented process is given in *Figure 16.1*:

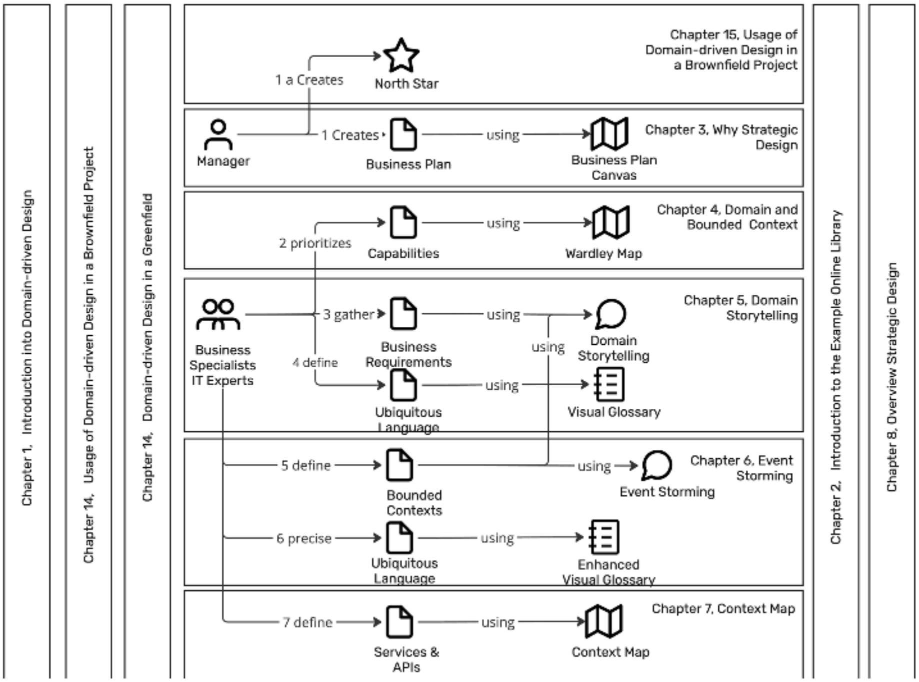

Chapter 1, Introduction into Domain-driven Design
Chapter 14, Usage of Domain-driven Design in a Brownfield Project
Chapter 14, Domain-driven Design in a Greenfield
Chapter 15, Usage of Domain-driven Design in a Brownfield Project
1 a Creates
North Star
Manager
1 Creates
Business Plan
using
Business Plan Canvas
Chapter 3, Why Strategic Design
2 prioritizes
Capabilities
using
Wardley Map
Chapter 4, Domain and Bounded Context
Business Specialists IT Experts
3 gather
Business Requirements
using
Domain Storytelling
Chapter 5, Domain Storytelling
4 define
Ubiquitous Language
using
Visual Glossary
5 define
Bounded Contexts
using
Event Storming
Chapter 6, Event Storming
6 precise
Ubiquitous Language
using
Enhanced Visual Glossary
7 define
Services & APIs
using
Context Map
Chapter 7, Context Map
Chapter 2, Introduction to the Example Online Library
Chapter 8, Overview Strategic Design

***Figure 16.1**: Overview Synergetic Blueprint—Strategic Part*

***Source**: StrategicDesign.jpg*

The Synergetic Blueprint contains the following steps, as shown in *Figure 16.1*:

1. A manager creates a business plan using a business plan canvas or a North Star formulation.
2. Business specialists and IT experts prioritize the application's capabilities using a Wardley map.
3. Business specialists and IT experts gather business requirements using domain storytelling and
4. Define the ubiquitous language using a visual glossary.
5. Business specialists and IT experts define the bounded contexts using event storming or domain storytelling.
6. Business specialists and IT experts precise the ubiquitous language using an enhanced visual glossary.
7. Business specialists and IT experts define services and APIs using a context map.

This process is detailed in the different chapters of this book, as the following overview shows:

- *Chapter 1, Introduction:* This chapter provides a first overview of DDD[1]. It introduces readers to the complexity of software architecture and development and shows how DDD can help overcome such obstacles.

  This text explores the concept of complexity in a rapidly changing and uncertain world. It provides a thorough explanation of software architecture, detailing the responsibilities of a software architect and referencing specific chapters relevant to each task.
- *Chapter 2, Introduction to the Example Online Library: Chapter 2* introduced an online library, which serves as an example throughout the book. First, the library is roughly sketched, and more details are added using a paper prototype. The library's core, supportive, and generic domains are detected based on capabilities.

  Sketching ideas, e.g., using wireframes, can be the first step in creating a business plan.
- *Chapter 3, Why Strategic Design*: Strategic design, a key element of Domain-Driven Design, involves the identification of bounded contexts, establishing a ubiquitous language, and explicitly clarifying inter-context dependencies. Bounded contexts do more than delineate the applicability and coherence of domain models; they also restrict the scope of services within a service architecture. This delineation informs the division of responsibilities among the development teams, facilitating

---

[1] Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Boston: Addison-Wesley.

a structure where teams are aligned with distinct segments of the system they are implementing.

It introduces the significance of business clarity and the collaborative formulation it. The methodology is a business plan canvas.

The chapter outlines the significance of strategic design within the macro architecture of software systems. It emphasizes that strategic design is vital not only for ensuring autonomy among teams but also for exerting a substantial impact on the overall quality of the software. Additionally, the chapter provides a brief overview of DDD methodologies, highlighting their influence on specific quality and security aspects of the software.

- *Chapter 4, Domain and Bounded Context*: The chapter introduces the most essential terms in DDD: domain, subdomain, and bounded context[2]. It differentiates core, supportive, and generic domains related to an enterprise's business. It deep-dives into the relationship between subdomain and bounded context and discusses typical architectural patterns.

  It highlights the critical role of domains within enterprise structures and details the process of distilling subdomains utilizing Wardley maps[3]. It emphasizes the significance of bounded contexts and adopting a ubiquitous language. Additionally, the chapter explores architectural patterns, like *Shared Kernel*, *Conformist*, *Anti-Corruption Layer*, and *Open Host Service*, which are instrumental in implementing bounded contexts.

- *Chapter 5, Domain Storytelling*: *Chapter 5* introduces domain storytelling[4] as a collaborative modeling technique that allows business experts and IT specialists to work together. It crafts a shared understanding and ubiquitous language between people with different backgrounds. The chapter introduces the methodologies of domain storytelling and visual glossary[5] as workshop formats.

  Employing straightforward and accessible grammar, domain stories effectively steer complex discussions, ensuring all participants remain aligned. While the grammar and associated iconography are intentionally simple to facilitate clear communication, there are instances where prioritizing comprehension over strict adherence to predefined rules may yield better collaborative outcomes. Flexibility in this method allows for nuanced interpretation and alignment with the unique context of each discussion.

---

[2] Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Boston: Addison-Wesley.

[3] Wardley, S. (2022). *Wardley Maps.* Simon Wardley.

[4] Hofer, S., & Schwentner, Henning. (2021). *Domain Storytelling: A Collaborative, Visual, and Agile Way to Build Domain-Driven Software.* Pearson International.

[5] Zörner, S. (2015). *Softwarearchitekturen dokumentieren und kommunizieren, Entwürfe, Entscheidungen und Lösungen nachvollziehbar und wirkungsvoll festhalten.*

Integrating domain stories with visual glossaries empowers stakeholders to define bounded contexts and subdomains more precisely. This combination lays a strong foundation for subsequent modeling activities and implementation processes.

- *Chapter 6, Event Storming*: Event storming is another workshop format for Domain-Driven Design[6]. It concentrates on domain events along a business process. These domain events are enhanced by commands triggering them, aggregates and read-only data, and roles and processes performing the triggering command. Event storming is used to define bounded contexts along the discussed business processes.

  Event storming workshops can be held without previous workshops or subsequent to domain storytelling workshops.

  The collaborative nature of these workshops allows people with entirely different experiences to attend, even though a seasoned architect should moderate them.

  Event storming workshops result in the definition of bounded contexts. Those are the bases of context maps, which will be discussed in the next chapter. During the discussions, previous visual glossaries can be enhanced or even created.

- *Chapter 7, Context Map:* The derived bounded contexts from an event storming or domain storytelling Workshop can be summarized in a context map[7].

  A context map overviews the bounded contexts in one or more subdomains. It facilitates the team dependencies between the bounded contexts and the belonging services. Moreover, a context map can deduce the data flows synchronously and asynchronously between the bounded contexts.

  A context map is invaluable for analyzing business requirements and designing modern software architectures. It effectively bridges the tight coupling between business requirements, represented by bounded contexts, and technical design, defined by asynchronous and synchronous APIs. This makes context maps ideal for facilitating collaborative software architecture design between business experts and technical teams.

  A context map also explains how implementing teams can work together. Those dependencies can easily be recognized by business and IT specialists. In such a way, team tailoring can be adapted corresponding to the business needs of the project.

---

[6] Brandolini, A. (2023). *EventStorming*. Retrieved 04 2024, from EventStorming: **https://www.eventstorming.com**

[7] Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Boston: Addison-Wesley.

The bounded contexts within the context map are derived from the event storming analysis. The domain events used for data flows are explicitly taken from this analysis. However, the resources for the planning are deduced from the identified aggregates.

- *Chapter 8, Overview Strategic Design*: Strategic design is an excellent tool for defining the service structure of applications. It helps to understand the business better and establish a ubiquitous language for the implementation teams. However, it needs to be implemented sustainably if we want to reap its benefits. This chapter explains why strategic design is essential and gives tips for introducing it to a company.

  It shows the software application's design process, from defining business capabilities through gathering business requirements using domain storytelling to creating the application and APIs using event storming and context mapping. It explains which step of the overall process and which artifacts need to be produced using workshops and an overall collaborative approach.

  It shows how a strategic design can be implemented in an organization and hints at how to apply it successfully.

Strategic design is the base for any tactical design within a bounded context. It is the groundwork. Without strategic design, a successful tactical design is not possible. The following section will go through the process of tactical design in more detail.

Tactical design

Tactical design is based on strategic design. It uses the artifacts created during strategic design. The collaboration approach is slightly changed. The process is shown in *Figure 16.2*:

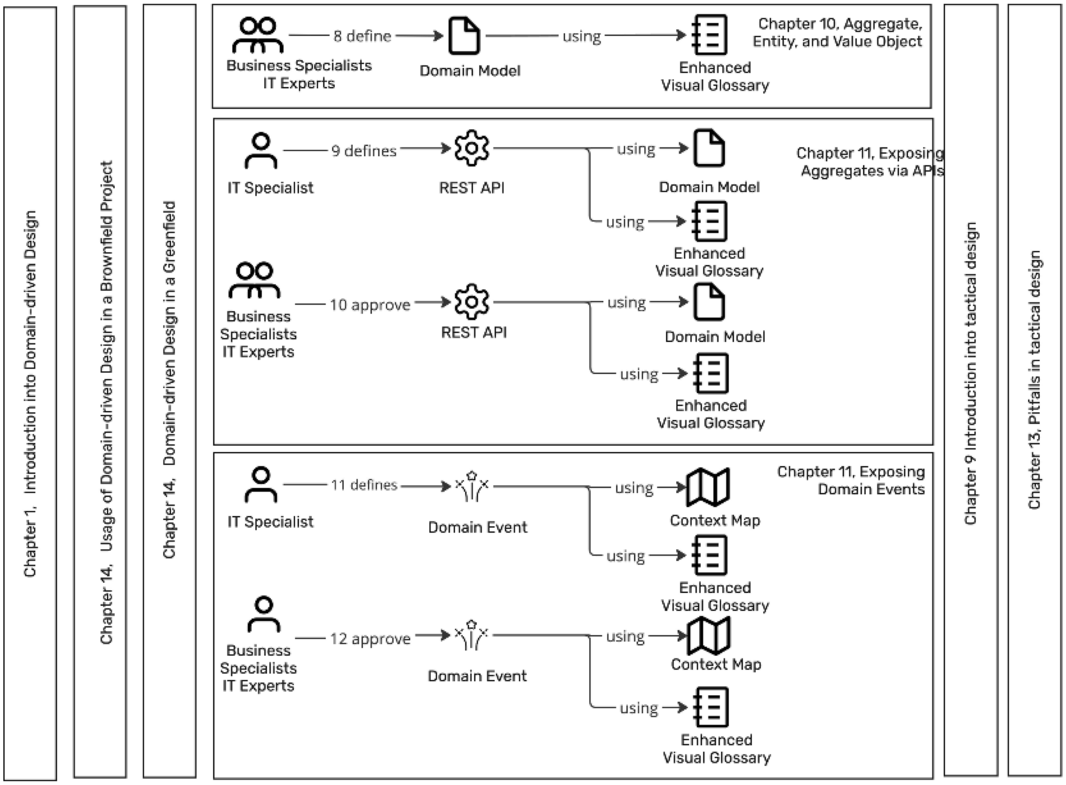


***Figure 16.2**: Tactical design of the Synergetic Blueprint process*

***Source**: TacticalDesign.jpg*

Figure 16.2 shows the blueprint process as the sequel of the strategic design. The single steps are listed in the following points:

1. Business specialists and IT experts define the domain model using the enhanced visual glossary.
2. An IT specialist defines the synchronous REST API using the domain model and the enhanced visual glossary.
3. Business specialists and IT experts approve the REST API using the domain model and the enhanced visual glossary.
4. An IT specialist defines the domain events using the context map and the enhanced visual glossary.
5. Business specialists and IT experts approve the domain events using the context map and the enhanced visual glossary.

The collaboration pattern has been changed insofar as the developers create drafts of APIs, The process is described in the chapters discussing the tactical design:

- *Chapter 9, Introduction to Tactical Design*: This chapter discusses tactical design as the other side of DDD[8]. Developers should apply the principles in their daily professional lives. However, business analysts and product owners should also understand the principles and concepts of tactical design to discuss the designs with developers.

  Tactical design allows a team responsible for a bounded context to model their solution inside the context. They can design corresponding aggregates and decide how to structure modules or services.

  A deep understanding of the business requirements is necessary to successfully model the bounded context using modern design patterns like hexagonal architecture[9] and Domain-Driven Design patterns.

- *Chapter 10, Aggregate, Entity, and Value Object*: This chapter discusses the patterns of Domain-Driven Design. Patterns help us understand successful, well-structured code's underlying structure and design. It discusses the bounded context patterns aggregate, entity, and value object[10]. Furthermore, it discusses the pattern repository for persisting objects and the factory[11] for creating objects[12]. The pattern orchestrator and choreography are also discussed to access aggregates[13].

  Tactical design is as essential to successful software products as strategic design. Developers can only depict a business sense by representing the ubiquitous language in the implemented bounded context. Without an understanding of published language, a bounded context stays meaningless.

  Remember that only a tight collaboration between business experts and IT specialists guarantees success and that the results of the strategic design documented during the domain storytelling and event-storming workshops need to be reflected in the business logic code and the database design.

---

[8] Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Boston: Addison-Wesley.

[9] Cockburn, A., & Garrido de Paz, J. (2024). *Hexagonal Architecture Explained.* Human and Technology Inc.

[10] Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Boston: Addison-Wesley.

[11] Gamma, E., Helm, R., & Johnson, R. (1997). *Design Patterns. Elements of Reusable Object-Oriented Software.* Prentice Hall.

[12] Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Boston: Addison-Wesley.

[13] Richardson, C. (2019). *Microservices Patterns.*

- *Chapter 11, Exposing Aggregates via APIs*: Aggregates could be accessed via a published language. Aggregates, entities, and value objects are part of a domain model. Aggregates maintain the content of the bounded context.

  RESTful APIs as synchronous APIs can be formulated from the strategic and tactical design results[14]. Using ubiquitous language gives the APIs a high expressiveness. Client developers can easily understand the business meaning of the APIs. The APIs are structured. They allow access to aggregates inside a bounded context and enable access to entities in aggregates using sub-resources.

- *Chapter 12, Exposing Domain Events*: Modern architecture relies heavily on transferring events in an event-driven architectural style. Events can be defined based on the results of the event storming workshops using the context map as documentation. Using a standardized definition language allows events to be defined independently of implementation details and broker technologies[15].

  Events and commands can be elegantly defined using the ubiquitous language specified in the event storming and domain storytelling workshops. The events and commands are expressive and can be understood elegantly by developers and business experts.

- *Chapter 13, Pitfalls in Tactical Design*: Tactical design can be tiresome and sometimes seems over-engineered. However, better maintenance and faster onboarding of new team members dramatically increase the rewards. Certain pitfalls need to be avoided.

  One pitfall might be that the created model and the implementation differ over time. **Large language models** (**LLMs**) can help check the consistency of the domain story against the implementation.

  Another one might be to drive the language by database design. That usually leads to technical language that business experts do not understand. Therefore, the ubiquitous language must drive the database design, not vice versa.

  A typical pitfall is designing an API purely from the provider's point of view. The client cannot understand the business meaning behind it. The same is true for pure technical solutions.

  A wrong tailoring of the bounded contexts might lead to overwhelming APIs trying to solve someone else's problem.

---

[14] OpenAPI. (2022). *The world's most widely used API description standard*. Retrieved July 2024, from OpenAPIs: **https://www.openapis.org/**

[15] *3.0.0*. (2023, December 5). Retrieved August 2024, from AsyncAPI: **https://www.asyncapi.com/docs/reference/specification/v3.0.0**

> Starting from almost nothing, using DDD daily is a long journey with several backlashes. But it is worth starting. One can begin to even with small applications of tactical design and ubiquitous language in one team.

The importance of tactical design is evident, and it shows how it can be applied practically. The entire blueprint process is used in a greenfield and a brownfield environment.

Applying Domain-Driven Design in projects

Two examples show the relevance and practicability of the blueprint process.

First, a greenfield project is discussed. *Chapter 14, Usage of Domain-Driven Design in a Greenfield*, discusses the design and implementation of a rent-a-bike shop.

This chapter discusses how Domain-Driven Design can be applied to a greenfield project and methods for achieving a resilient and sustainable architecture when implementing a new product. It also discusses the documentation and implementation of a Domain-Driven Design process.

The strategic design can be documented in a concept using the first chapters of architecture documentation. A bounded context canvas documents bounded contexts, and the architecture of a bounded context is documented using an architecture communication canvas. The canvases can reference architecture decisions that must be documented using an architecture decision record.

Additionally, a way to implement DDD processes in an enterprise is as follows:

1. Start with domain storytelling
2. Conduct event stormings in single teams
3. Formulate the business using a Wardley map
4. Introduce formalized documentation
5. Require the blueprint process

Along with those steps, training needs to be provided to all participants of the blueprint process. Foundation training can be provided to all members, whereas advanced training should focus either on strategic or tactical design.

In conclusion, investing in training and establishing a robust DDD framework lays the groundwork for swiftly and efficiently achieving business objectives. Skilled team members and a cohesive design approach ensure that the solutions delivered are timely and meet high-quality standards, ultimately driving the organization's success.

We discussed the DDD process in two examples. The online library and the rent-a-bike shop were discussed as greenfield projects, starting from the idea to implementing bounded contexts. The DDD process supports a deeper understanding of the business domain and, insofar as better and more efficient software.

Today, most development projects fall within the category of brownfield projects. A brownfield project refers to an existing codebase developed and modified over many years. While these systems require ongoing maintenance to ensure stability and functionality, they must also be modernized to remain relevant in a fast-evolving technological landscape.

The challenge of brownfield development lies in balancing the preservation of valuable legacy code with the necessity for updates and advancements. Modernizing these long-standing systems can be daunting as organizations strive to adapt to new business requirements and customer expectations. The following chapter will explore how DDD can be a robust methodology for revitalizing aging systems.

Chapter 15, Usage of a Domain-Driven Design in a Brownfield Project, shows how the blueprint process can be applied to a brownfield project. Again, a library is an example, but the existing library software should be modernized this time.

The chapter discusses how DDD can be applied to a brownfield project and methods for achieving a resilient and sustainable architecture when refactoring a legacy system.

Modernizing a system starts from a North Star, formulating the change's target and transferring it to the team organization for modernization. We saw that Domain-Driven Design can be applied to modernizations as successfully as to greenfield projects.

DDD is an excellent approach to developing successful products. It can be implemented step by step in teams and supports collaboration between business experts and IT specialists in workshop formats.

In the following section, let us examine the artifacts that must be provided throughout the process.

Artifacts of the Synergetic Blueprint process

Many artifacts are created during the process and need to be documented. *Figure 16.3* shows these artifacts:

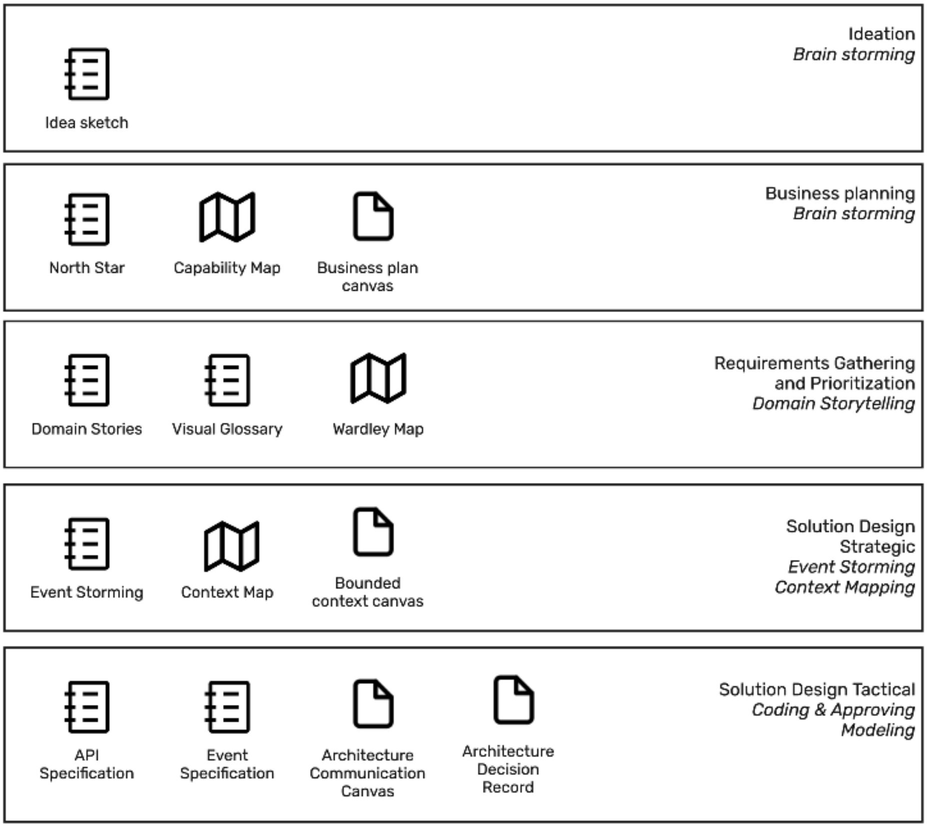


***Figure 16.3**: Artifacts of the Synergetic Blueprint process*

***Source**: Artifacts.jpg*

In the following sections, we want to discuss the artifacts produced during the phases of the blueprint process.

Ideation

During the ideation phase, creating sketches and drafting rough descriptions of the software to be implemented serve as valuable tools for clarifying the business objectives. These visual and conceptual representations facilitate a deeper understanding of the project's goals and requirements.

To foster creativity and innovation, various established brainstorming techniques can be employed. Approaches like brainwriting encourage individual idea generation before sharing with the group, helping to harness diverse perspectives while minimizing the risk of dominant voices overshadowing others[16]. Additionally, brainstorming workshops

---

[16] Osborn, A. F. (1953). *Applied imagination: principles and procedures of creative thinking*. New York: Charles Scribner's Sons.

create a collaborative environment, allowing team members to collectively build on each other's ideas and refine their concepts.

In the next phase, the business planning must detail the ideas and give them a reliable base.

Business planning

In the business planning phase, it is crucial to establish a shared understanding of the business objectives among all stakeholders. One practical approach is to create a North Star metric[17], which serves as a guiding principle for the project. This is particularly beneficial in brownfield projects where existing systems and practices must be integrated or transformed. This metric helps align the team's efforts toward a common goal, ensuring that all decisions and actions contribute to the overarching vision.

Alternatively, developing a business model canvas can provide a comprehensive framework for articulating the business strategy[18]. This visual representation identifies critical elements such as value propositions, customer segments, channels, and revenue streams, fostering holistic discussions and clarifying the business landscape.

In tandem with these strategies, a capability map can be constructed based on the identified competencies within the organization[19]. This map categorizes capabilities into three distinct tiers: core capabilities essential for delivering unique value, supportive capabilities enhancing operational efficiency, and generic capabilities necessary for basic functionality.

By articulating these capabilities, organizations can prioritize their development efforts and allocate resources effectively. The capability map serves as a strategic tool that guides current initiatives and informs future planning and decision-making, helping ensure long-term alignment with business targets. Through these combined methods, teams can build a robust foundation for successful implementation and sustainable growth.

Using a capability map or a business plan canvas, the business plans can be detailed by structured requirements gathering.

---

[17] Ellis, S. (2017, June 5). *What is a North Star Metric*. Retrieved 2024 October, from blog.growthhackers.com: **https://blog.growthhackers.com/what-is-a-north-star-metric-b31a8512923f**

[18] Strategyzer. (2023). *The Business Model Canvas*. Retrieved March 29, 2024, from **https://www.strategyzer.com/library/the-business-model-canvas**

[19] Blair, A., Lail, J. B., & Marshall, S. (2022). *Business Capabilities, Version 2.* Berkshire: The Open Group.

Requirements gathering

Domain storytelling effectively gathers the requirements for newly built or modernized software[20]. This approach facilitates a deeper understanding of user needs and business contexts and enhances stakeholder collaboration. To bridge the gap between IT specialists and business experts, creating a visual glossary can help establish a ubiquitous language, fostering more transparent communication and mutual understanding throughout the development process.

A Wardley Map can be employed to prioritize the identified bounded contexts within the domain of storytelling sessions[21]. This mapping technique is precious in brownfield projects, where documenting the current state of software is essential for guiding future enhancements and modifications. By visually organizing the components of the software landscape, teams can identify critical areas for improvement and establish priorities that align with business goals.

In the context of greenfield projects, the capabilities discovered during the initial phases can also be prioritized using a Wardley Map. This allows teams to categorize the capabilities of the future software according to their maturity and evolution stages. By sorting these capabilities, stakeholders can focus on foundational elements that require immediate attention while planning for more advanced features in the future.

Overall, integrating these techniques not only aids in gathering and prioritizing requirements but also enhances alignment among team members. Through domain storytelling, visual glossaries, and Wardley Mapping, organizations can create a structured framework for developing software that meets current and future needs, ensuring a seamless transition from concept to implementation.

A solution design can outline future software components when business targets and requirements are well documented and agreed upon.

Solution design

The solution design phase outlines the services to be implemented or adopted. In greenfield projects, event storming proves highly effective, providing clearly defined bounded contexts that can be documented in a comprehensive context map[22]. This visual representation clarifies the interactions between different components and is a valuable reference for stakeholders throughout the development process.

---

[20] Hofer, S., & Schwentner, Henning. (2021). *Domain Storytelling: A Collaborative, Visual, and Agile Way to Build Domain-Driven Software.* Pearson International.

[21] Wardley, S. (2022). *Wardley Maps.* Simon Wardley.

[22] Brandolini, A. (2023). *EventStorming*. Retrieved 04 2024, from EventStorming: **https://www.eventstorming.com**

However, event storming may not be necessary for brownfield projects in every instance[23]. Instead, the context map can be effectively created based on the insights derived from domain storytelling sessions. By leveraging the rich narratives captured during these discussions, teams can gain a holistic view of the existing system's functionalities and limitations, allowing for a more targeted approach to modernization.

Regardless of the project type, it is essential to enhance the visual glossary to articulate further the ubiquitous language shared among team members[24]. This living document should be continually updated to reflect new terms, concepts, and nuances that emerge throughout the design process. By maintaining an accessible and detailed glossary, teams can ensure that all participants—whether from IT or the business side—remain aligned in their understanding and communication.

Furthermore, the identified bounded contexts should be documented using bounded context canvases[25]. These canvases provide a structured format for capturing key aspects such as the purpose, interactions, and responsibilities within each context. By systematically organizing this information, teams can promote clarity and coherence in their design efforts.

Ultimately, combining these techniques fosters a collaborative environment where all stakeholders can engage meaningfully in the solution design, driving the project toward successful implementation. The thoughtful integration of event storming, domain storytelling, and visual documentation not only enhances clarity but also aids in aligning technical and business objectives.

With a straightforward solution design, the tactical design and implementation of the bounded contexts can be applied.

Tactical design

The tactical design phase is pivotal in defining the software system's structure, particularly through the exposure of aggregates and domain events. These elements are essential for capturing the core business logic and facilitating communication between different system parts. To ensure these components are well-understood and accessible, they must be documented in corresponding specifications using tools like OpenAPI[26] and AsyncAPI[27]. OpenAPI is beneficial for outlining RESTful APIs, while AsyncAPI is ideal for detailing event-driven architectures, providing clarity for developers and stakeholders alike.

---

[23] Lilienthal, C., & Schwentner, H. (2023). *Domain-Driven Transformation.* dpunkt.verlag.

[24] Zörner, S. (2015). *Softwarearchitekturen dokumentieren und kommunizieren, Entwürfe, Entscheidungen und Lösungen nachvollziehbar und wirkungsvoll festhalten.* München: Carl Hanser Verlag.

[25] Baas, K., Lindhard, K., Plöd, M., & Sanglan-Charlier, M. (2024, May 20). *The Bounded Context Canvas.* Retrieved May 2024, from GitHub: **https://github.com/ddd-crew/bounded-context-canvas/tree/master**

[26] OpenAPI. (2022). *The world's most widely used API description standard.* Retrieved July 2024, from OpenAPIs: **https://www.openapis.org/**

[27] *3.0.0.* (2023, December 5). Retrieved August 2024, from AsyncAPI: **https://www.asyncapi.com/docs/reference/specification/v3.0.0**

The architecture of the bounded context can be further articulated using a lead approach incorporating an Architecture Communication Canvas[28]. This canvas is a comprehensive visual tool that captures essential architectural elements, stakeholder concerns, and communication flows. Using this structured format, teams can successfully convey the relationships and responsibilities within the bounded context, enhancing collaboration and understanding among team members and stakeholders.

Moreover, an **Architecture Decision Record** (**ADR**)[29] must meticulously document each architectural decision. These records provide a historical context for why specific choices were made, capturing each decision's rationale, alternatives considered, and potential impacts. By maintaining a well-organized set of ADRs, teams can create a transparent decision-making process that can be referred to in future design discussions, facilitating learning and continuity within the project.

Overall, exposing aggregates and domain events, using specified frameworks for documentation, and systematically recording architectural decisions help create a robust technical foundation. This thorough documentation promotes clarity, enhances communication, streamlines the development process, and contributes to the project's success.

Artifacts are not the product of a solitary individual working alone; instead, they emerge from a collaborative process called the Synergetic Blueprint process. This process fosters the creation of blueprints and shared understandings among team members. While this collaboration is essential for ensuring diverse perspectives and comprehensive insights, generating these artifacts can often be tedious, monotonous, and susceptible to errors.

Leveraging **artificial intelligence** (**AI**) tools to enhance this process and alleviate some burdens is advantageous. These tools can streamline documentation efforts, reduce the risk of human error, and allow team members to focus on higher-level strategic thinking and creative problem-solving. By integrating AI solutions, teams can improve efficiency and enhance the quality of the artifacts produced, leading to more effective communication and a more apparent alignment on project goals.

Ultimately, embracing technology in this collaborative framework makes the process more enjoyable and engaging and empowers teams to deliver artifacts that are accurate and reflective of the group's collective intelligence.

Using AI in the process

Let us explore where AI tools might benefit the process. For example, creating a REST API based on an appropriate skeleton might be beneficial.

---

[28] *Architecture Communication Canvas*. (2024). Retrieved October 2024, from arc42: **https://canvas.arc42.org/architecture-communication-canvas**

[29] *Architectural Decision Records (ADRs)*. (2024). Retrieved October 2024, from Homepage of the ADR Github organization: **https://adr.github.io/**

Creating a RESTful API based on a visual glossary

RESTful APIs are expressive and benefit from ubiquitous language. We can document them using a visual glossary. On the other hand, a RESTful API specified in an OpenAPI needs a profound structure, which a skeleton can give.

Let us create an OpenAPI specification using the domain model of task management (as discussed in *Chapter 10, Aggregate, Entity, and Value Object*) and the skeleton (as discussed in *Chapter 11, Exposing Aggregates via APIs*).

A prompt could be like after the AI is set to an OpenAPI expert:

`Create an OpenAPI 3.1.0 specification based on the provided domain model and the provided skeleton.`

The answer looks promising[30][31]:

`I've created an OpenAPI 3.1.0 specification for a Task Management API based on the provided domain model and skeleton. Here's a summary of the key components:`

1. **`Basic Information: I've set up the API info, including title, description, and version.`**
2. **`Security: I've implemented OpenID Connect for authentication with different scopes for read, write, and admin operations.`**
3. **`Paths: I've defined endpoints for:`**
    - **`GET /tasks: Retrieve a list of tasks`**
    - **`POST /tasks: Create a new task`**
    - **`GET /tasks/{taskId}: Retrieve a specific task`**
    - **`PUT /tasks/{taskId}: Update an existing task`**
    - **`DELETE /tasks/{taskId}: Delete a task`**
4. **`Components:`**
    - **`Parameters: I've defined parameters for taskId, search criteria, and API version.`**
    - **`Responses: I've created responses for success, error, and various HTTP status codes.`**
    - **`Request Bodies: I've defined request bodies for creating and updating tasks.`**

---

[30] Claude 3.5 Sonnet Creation of an OpenAPI, generated 2024-10-20

[31] A copy of the full chat can be found under: **https://github.com/Grinseteddy/MasteringDdd/blob/main/AI/GenerateRest.md**

    - **Schemas: I've created schemas for Task, TaskToBeCreated, LeanUser, Status, and other necessary components based on the domain model.**

5. **Tags: I've added a "Tasks" tag to group related operations.**

However, even the generated OpenAPI looks well-structured and expressive. The following is an example of a task schema definition.

```
TaskToBeCreated:
  description: Task to be created
  type: object
  required:
    - title
    - dueDate
    - requester
  properties:
    title:
      description: Title of the task
      type: string
      minLength: 1
      maxLength: 255
      example: "Complete project proposal"
    description:
      description: Detailed description of the task
      type: string
      example: "Write a comprehensive project proposal including timeline
and budget"
    dueDate:
      description: Due date of the task
      type: string
      format: date
      example: "2024-12-31"
    creationDate:
      description: Creation date of the task
      type: string
      format: date-time
      example: "2024-07-01T10:30:00Z"
    assignee:
      $ref: '#/components/schemas/LeanUser'
    requester:
      $ref: '#/components/schemas/LeanUser'
    status:
      $ref: '#/components/schemas/Status'
```

The brief example demonstrates that OpenAPIs and AsyncAPIs can be effectively generated using AI technologies, which helps streamline the development process and ensure consistency. To achieve this, it is essential to have a thoroughly discussed visual glossary in place. This glossary serves as a critical reference for defining terms, concepts, and relationships, ensuring that all stakeholders share a common understanding of the language used.

Additionally, a well-structured skeleton of the corresponding specification language is necessary. This structured framework provides a clear outline for the API specifications, guiding AI tools in generating more accurate and relevant documentation. By establishing this foundation, organizations can significantly reduce ambiguities and promote clarity in their API implementations.

Combining a robust visual glossary and a well-defined specification skeleton enhances collaboration among team members and empowers AI technologies to function more effectively. When these elements are clearly articulated, the likelihood of generating high-quality, compliant APIs increases, ultimately contributing to a more efficient and successful development lifecycle.

Moreover, integrating AI in this context allows teams to focus on innovation and problem-solving rather than getting bogged down in the minutiae of documentation. As AI tools continue to evolve, their capability to support and enhance the API design process will only improve, making them an invaluable asset for modern software development.

Check on consistency

Chapter 13, *Pitfalls in Tactical Design*, already discusses how AI tools can detect inconsistencies between implementations.

The prompt can look like this:

```
Check if the OpenAPI specification and the domain model are consistent.
```

For this check, the incorrect task management model was used, and inconsistent language was used[32].

The AI tool detects inconsistencies[33][34]:

```
Inconsistencies and issues:
   1. Naming: The OpenAPI spec uses "Nothing" instead of "Task" for the
      main entity.
   2. LeanUser: The OpenAPI spec includes a "link" attribute not present
      in the domain model.
```

---

[32] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter11-ExposingAggregatesViaApis/RestApis/TaskManagement/TaskManagementWrongLanguage.yaml**

[33] Chat with Claude 3.5 Sonnet 2024-10-20

[34] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter16-Summary/AI/ConsistenceCheck.md**

3. **`Text value object: Not represented in the OpenAPI specification.`**
4. **`Paths and operations: The OpenAPI spec defines CRUD operations for tasks, which align with the domain model, but the naming is inconsistent (using "Nothing" instead of "Task").`**

The AI detects the inconsistencies. However, a visual glossary is not as accurate as code. It is a tool to establish a ubiquitous language, so some of those inconsistencies might be intentional. Anyhow, checking inconsistencies between modeling and implementation is a great help in avoiding anemic and incorrect models in code compared to the collaboratively created rich models.

In the examples before, we created code based on visual models. Sometimes, we need it the other way around to create visual models based on text. In the next section, we will discuss this point using AI.

Using AI to generate models

Let us assume we have text describing a bounded context and want a data flow diagram showing the bounded context and the data flows.

Still, let us use the task management model because of its low complexity.

1. A user creates a task which is stored in an inbox.
2. The team reviews those tasks and assigns them to team members.
3. An assigned task is set to in progress by the assignee.
4. The assignee adds documents and notes to the task.
5. The assignee finishes the task and creates a follow-up task in the inbox.

The prompt could look like this:

`Please create a context flow diagram based on the following description:`

The AI creates a graph that shows the contexts[35]; see *Figure 16.4*:

---

[35] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter16-Summary/AI/ContextFlow-Generation.md**

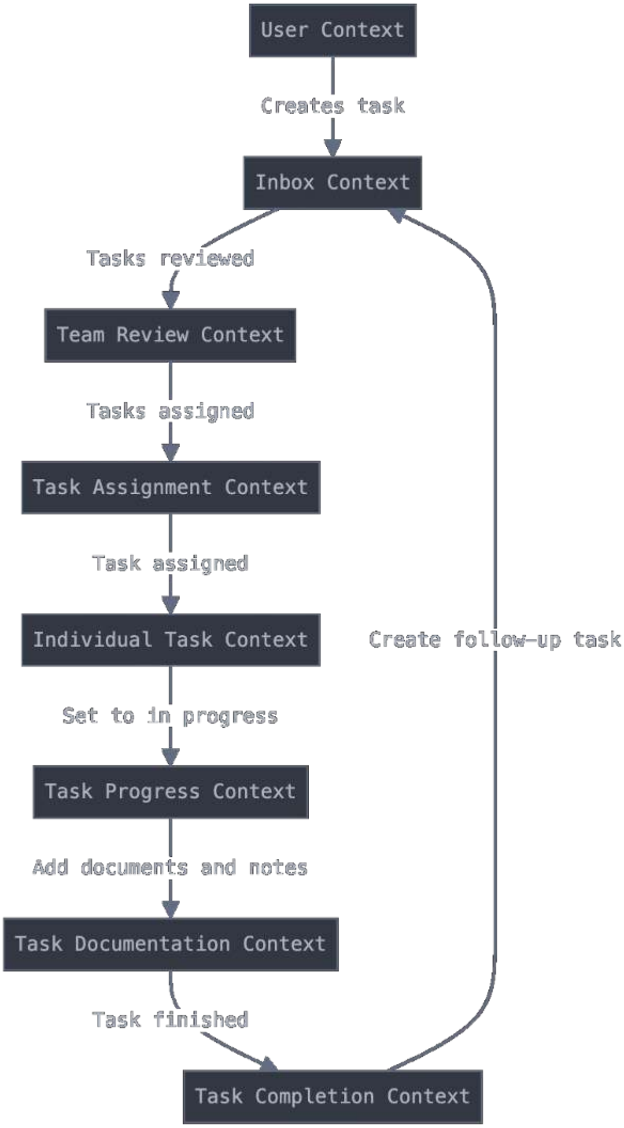


Figure 16.4: *Created context graph by Claude 3.5 Sonnet 2024-10-20* [36]

Source*: ContextFlow.png*

The examples illustrate that even the generation of diagrams can significantly aid in the modeling process. Additionally, AI can be vital in fostering collaborative modeling throughout the Synergetic Blueprint process. By leveraging AI technologies, teams can enhance their ability to visualize complex concepts, facilitate communication, and ensure a more cohesive understanding among participants. This integration streamlines the

---

[36] **https://github.com/Grinseteddy/MasteringDdd/blob/main/Chapter16-Summary/AI/ContextFlow.mermaid**

modeling efforts and allows for greater creativity and innovation, ultimately leading to more effective and comprehensive blueprints.

At the end of this book, let us see what we can conclude.

Conclusion

The Synergetic Blueprint process facilitates software design and implementation through collaborative, cross-functional, and cross-skilled teams. Beginning with vague ideas, this process effectively guides teams toward creating successful, high-quality software solutions.

While AI tools enhance various stages of this process, collaboration among team members is the real key to success. Individuals must work together and develop a shared language that fosters mutual understanding. This collaborative effort strengthens communication and enriches the team's overall problem-solving capability.

Ultimately, the Synergetic Blueprint process nurtures an environment that promotes comprehension and encourages successful collaboration, ensuring that diverse perspectives are integrated into the final product.

Points to remember

- The Synergetic Blueprint process covers the steps of ideation, business planning, requirements gathering, strategic solution design, and tactical design.
- The Synergetic Blueprint process can be applied successfully in greenfield and brownfield projects.
- AI tools can be applied during the process:
  - Generating API and eventing specifications
  - Checking consistencies from modeling to implementation and vice versa
  - Generating diagrams

Multiple choice questions

1. **Which parts of DDD are covered by the Synergetic Blueprint?**
   a. Only strategic design
   b. Only tactical design
   c. Tactical and strategic design
   d. Ideation, business planning, requirements gathering, strategic and tactical design

2. **Can business experts be part of tactical design?**
    a. Yes, they approve implementations.
    b. No
    c. Yes, but only as a watcher without any voice
    d. Yes, as chickens

3. **Can the Synergetic Blueprint process applied to brownfield projects?**
    a. No
    b. Yes, but only in the strategic part
    c. Yes, but only in the tactical part
    d. Yes

4. **For which artifacts can AI be used during the process?**
    a. For generating specifications
    b. For checking consistencies
    c. For generation of diagrams
    d. For all of the points above

Answers

1. **d**
2. **a**
3. **d**
4. **d**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Index

A

ADR, importance 293
Aggregate 63, 64
Aggregate/Views,
    preventing 111, 112
AI, concepts
  consistency,
    checking 355, 356
  data flow, analyzing 356-358
  RESTful API 353-355
Anti-Corruption Layer 67, 68
API Definitions/Context Map,
    analyzing 150-152
API Definitions, factors 151
API Definitions, sections
  discussions 148, 149
  member,
    managing 144-146
  purchase 144
  reading 146-148
API, fundamentals
  catalog, managing 215
  discussions 218-222
  member,
    managing 215-218
  Notes, optimizing 224-227
Application Programming
    Interface (API) 204
arc42, areas
  business, context 287
  core, decisions 287
  core, functions 287
  key stakeholder 287
  modules 287
  quality, requirements 287
  risks, information 287
  technologies 287
  value, proposition 287

Architecture Decision Record (ADR) 291, 292
AsyncAPI 235
AsyncAPI, architecture 237
AsyncAPI, structure 236, 237

B

BASE, contrasts
  basically available 14
  eventually, consistent 15
  soft-state 15
Blueprint 275
Blueprint, key points
  Constraints 277
  Context 278
  Qualitative Requirements 275, 276
  scope 275
  solution approach 283, 284
  Stakeholders 276
Bounded Context 63
Bounded Context, repository 193, 194
Bounded Context, section
  arc42 287
  Asynchronous, events 291
  Canvas 285, 286
  Synchronous API 290
Brainstorming 102, 103
Brainstorming, rules 102
Broker Architecture, aspects
  Event Streaming 234, 235
  Publish/Subscribe 235
Broker Architecture, concepts
  Broker 234
  Channel 234
  Message 234
  producers 234
Business Plan, points
  channels 27
  cost, structure 27
  customer, relationships 27
  customer, segments 27
  key activities 27
  key partners 26
  key propositions 27
  key resources 27
  revenue, streams 27

C

Canvas 128
Canvas, sections
  assumptions 130
  business, decisions 130
  domain, roles 129
  inbound, communicating 129
  Name 129
  open, questions 130
  outbound, communicating 129
  purpose 129
  strategic, classifying 129
  Ubiquitou, language 130
  verification, metrics 130
Capability Map 27-30
Capability Map, approaches
  Bottom-up 30
  Top-down 30

Capability Map, base
  business, model 30
  financial, plans 30
  organizational, structure 30
CAP Theorem 14
CAP Theorem, properties 14
Catalog Management 245, 246
Choreography 197, 198
Choreography, sequences
  reading 199
  searching 198
Command 109, 110
Complex Domain 6
Conformist 66, 67
Context Map 136-139
Context Map, optimizing 143
Context Map, steps 139, 140
Conway Law 41, 42
Conway Law, capabilities 42

D

DDD, architecture 2, 3
DDD, challenges 43, 44
DDD, complexity
  CAP Theorem 14, 15
  Commensurate Complexity 15-17
  Design Thinking 9
  Modern Software,
      analyzing 10-14
  Software Architecture 17
DDD, evolution 3, 4
DDD, history 2
DDD, methods
  API-first, strategy 164
  Governance, process 165-167
  Team Structure 167
DDD, patterns
  Aggregates 184-186
  Entity 186, 187
  Value Object 187
DDD Resource,
      analyzing 346, 347
DDD, security aspects 45, 46
DDD, steps 293-295
DDD, strategic
  accountability 45
  confidentiality 44
  integrity 44
Decoupling 324
Decoupling,
      architecture 326, 327
Decoupling, benefits 324, 325
Decoupling, breakdown 325
Decoupling, capabilities 325
Design Thinking 9
Design Thinking, principles
  ambiguity, rule 10
  Human Rule 9
  redesign 10
  tangibility 10
Domain, core 33
Domain-Driven Design (DDD) 2
Domains 31, 50, 51
Domains, capabilities 31-34
Domains, points
  Context Map 318, 319
  Storytelling 305
  Target Architecture 321, 322
  Vision Statement 303, 304
  Wardley Map 319, 320
Domain Storytelling 74
Domain Storytelling,
      fundamentals
  books, purchasing 306-308
  lending 313, 314

  - reservation 311, 312

Domain Storytelling, insights 96, 97

Domain Storytelling, language 81-83

Domain Storytelling Library, aspects
  - books, purchasing 88, 89
  - discussion 91, 92
  - membership, managing 89, 90
  - reading 90, 91
  - recommendation 92, 93

E

Event, collections 106, 107

Event, consolidation 107-109

Event-Driven Architecture 232, 233

Event Role, processes 110, 111

Event Storming 102

Event Storming Library, section
  - discussions 125-127
  - member, managing 115-117
  - membership, calculating 118, 119
  - purchase 114, 115
  - reading 120-123
  - recommendation 127

Event Storming, points 130-132

Event Storming, process 103-106

Event Storming, sessions
  - Scheduling 104
  - Tooling 104, 105

Eventual Consistency 14

F

Factory Pattern 195, 196

M

Minimum Viable Product (MVP) 11, 12

Model-Driven Design 173

Model-Driven Design, insights
  - Aggregates 177
  - Factories 177
  - Repositories 177

Model-Driven Design, parts 174-176

Modernization Layer 327, 328

Modern Software Systems (MSS) 5

MSS Complexity, handling 5-7

MVP, steps
  - Check 12
  - Do 12
  - Think 12

O

Online Library 22-26

Online Library, configuring 187-193

Online Library, section
  - amount paid 243
  - book identifier, optimizing 242
  - book, purchasing 240, 241
  - catalog entry, updating 241, 242
  - header, messages 239
  - member, creating 243
  - position, reading 245

OpenAPI 206, 207

Open Host Service 68, 69

Orchestrator 196, 197

Orchestrator, points
  reading 197
  searching 197

P

Physical Library,
    optimizing 300-303
Pitfalls 254
Pitfalls, points
  Database Design,
    preventing 254
  function provider,
    optimizing 254, 255
  technical solutions,
    implementing 255
  user acceptance,
    testing 255, 256

R

Representational State Transfer
    (REST) 141
Resource Paths, methods
  Single Task Delete 213, 214
  Single Task Get 213
  Tasks Get 211, 212
  Tasks Post 212, 213
REST 204-206
RESTful APIs, architecture
  Application Programming
    Interface (API) 204
  OpenAPI 206
  RUST 204
REST Interfaces, sections
  Asynchronous,
    communication 141
  Synchronous,
    communication 141
  West-East/North-South,
    communication 142, 143

S

Shared Kernel 64-66
Simple Domain 6
Software Architecture 17, 18
Software Architecture,
    points 18, 19
Software Design 38
Software Design,
    dependencies 39-41
Storytelling 74
Storytelling, approach 75
Storytelling, aspects 75
Storytelling System,
    enhancing 87, 88
Storytelling,
    visualization 75-80
Storytelling, ways 75
Strategic Design 64
Strategic Design Business,
    capabilities 164
Strategic Design, key point
  Canvases 161, 162
  modernization 160, 161
  purchase, prioritizing 161
Strategic Design, patterns
  Anti-Corruption Layer 67
  Conformist 66
  Open Host Service 68
  Shared Kernel 64-66
Strategic Design, sections
  anti-corruption, layer 162, 163
  Big Ball of Mud 163
  conformist 163
  open host, service 162
Strategic Design,
    steps 157-159
Subdomains 51

Synergetic Blueprint 338
Synergetic Blueprint Data, optimizing 339-342
Synergetic Blueprint, process
  business, planning 349
  Ideation 348
  requirements, gathering 350
  solution design, preventing 350, 351
  Tactical Design, optimizing 351, 352
Synergetic Blueprint, steps 339

T
Tactical Design 172
Tactical Design, aspects
  Resource Paths 211
  Schema Design 208-210
Tactical Design, configuring 172, 173
Tactical Design Data, analyzing 344-346
Tactical Design, lifecycle 250-253
Tactical Design, points 343
Tactical Patterns 179, 180
Target Architecture 322
Target Architecture, areas
  account/payment 324
  book, reading 324
  catalog, maintenance 323
  catalog, searching 324
  identity access, managing 324
  library, portal 323
  member, managing 324
  notification 324
  purchase 323
Target Architecture, configuring 330-332
Target Architecture, functionality 330
Target Architecture, process 328-330
Task Management, contexts 112, 113

U
ubiquitous language 177
ubiquitous language, best practices
  business experts 178
  consistency, ensuring 178
  domain, avoiding 178
  language, evolving 178
  onboard, facilitates 178
  visual glossary, developing 178
  weak words, avoiding 178
ubiquitous language, optimizing 253, 254
Universally Unified Identifier (UUID) 210

V
Vision Statement 303
Vision Statement, values 305
Visual Glossary 93, 94
Visual Glossary Subdomains, entities
  catalog 95
  discussions 95
  lending 95
  member, managing 95
  notes 95
  payment 95

  purchase 95
  reading 95
  recommendation 95
VUCA Software, complexity 7-9

W

Wardley Map 52
Wardley Map, methodology
  purpose 52
  scope 52
  user needs 53
  users 53
  value chain 54-56
Wardley Map Process, analyzing 57-59
Wardley Map, stages
  Commodity 56
  Custom 56
  Genesis 56
  Growing 56
Wardley Map/Subdomain, capabilities 61-63
Workshop 83
Workshop Context, template
  Business Context 280-282
  Capability Map 278
  Technical Context 282
Workshop, principles
  API Design 272, 273
  Context Map 271, 272
  Domain Storytelling 266
  Event Storming 266-270
  Wardley/Capability, map 264-266
Workshop, setup
  attendees 83
  length, preventing 86
  model tools, optimizing 84-86
  whiteboards 84
Workshop, visualizing 273, 274