



Effective Software Development for the Enterprise

Beyond Domain Driven Design,
Software Architecture, and
Extreme Programming

—
Tengiz Tatisani

Apress®

Effective Software Development for the Enterprise

Beyond Domain Driven Design,
Software Architecture,
and Extreme Programming

Tengiz Tutisani

apress®

Effective Software Development for the Enterprise: Beyond Domain Driven Design, Software Architecture, and Extreme Programming

Tengiz Tutsani
Charlotte, NC, USA

ISBN-13 (pbk): 978-1-4842-9387-4

<https://doi.org/10.1007/978-1-4842-9385-0>

ISBN-13 (electronic): 978-1-4842-9385-0

Copyright © 2023 by Tengiz Tutsani

This work is subject to copyright. All rights are reserved by the publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spaehr
Acquisitions Editor: Aditee Mirashi
Development Editor: James Markham
Coordinating Editor: Mark Powers
Copy Editor: April Rondeau

Cover designed by eStudioCalamar

Cover image by Luemen Rutkowski on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, email orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (<https://github.com/Apress>). For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

*I dedicate this book to my life's two most precious women:
my mother, Eteri Tetrauli, and my wife, Romana Stasiv.*

Table of Contents

About the Author	xiii
About the Technical Reviewer	xv
Acknowledgments	xvii
Preface	xix
Chapter 1: Introduction.....	1
History of Inefficient Monoliths	1
Why People Avoid Building Effective Software	2
Learning Curve	3
Quality over Quantity	3
Highly Paid Experts.....	5
Naive Hopes to Survive	5
Software Development Perfectionism as a State of Mind.....	6
Is It Crazy?.....	6
Desire Behind Perfectionism	7
Is It Worth It?	7
Six Pillars of Effective Software.....	8
#1. Meet Users' Expectations	8
#2. Allow No Defects	9
#3. Scale Out Horizontally	10
#4. Have No Dedicated Production Support Team	11

TABLE OF CONTENTS

#5. Accelerate Development Pace	11
#6. Double the ROI per Developer, Team, and Software	11
Summary.....	12
Chapter 2: Cross-cutting Concerns.....	13
Execution, Leadership, Management.....	14
Importance of Software Development Transformation.....	14
How to Achieve Software Development Transformation.....	15
Transformation Is for Everybody.....	15
Ad-Hoc Tasks vs. Process.....	17
Hands-on Leaders	18
Overall Support from Management	19
Organizational Structure	19
A Couple of Important Definitions Upfront.....	19
Forming Organizations	20
Forming Subsystems.....	22
Forming Microservices.....	24
Forming Teams.....	25
Forming Programs, Projects, Requirements, and Deliveries	29
Organizational Silos.....	30
Autonomy vs. Reuse	33
Processes, Ongoing Efforts, Teams	34
Agile or Waterfall	34
Transparent Teams	36
Managing Work in Progress.....	37
One for All, All for One	38
Transformation from Inside	39
Culture	40

TABLE OF CONTENTS

Professionalism in Software Development.....	40
Trust.....	42
Delegation of Responsibilities.....	43
Identifying Talent.....	44
Relaxed Team Atmosphere	45
Work–Life Balance.....	46
Candid Feedback.....	46
Change of Mind	47
Social Aspect of Engineering.....	48
Complexity as Job Safety.....	50
Team Spirit	51
Keep It Fun	53
Recruitment	54
Supporting Role of Recruitment	54
Hire Best.....	55
Quickly, Fancy, Right.....	55
Corrective vs. Preventive	57
Summary.....	58
Chapter 3: From Customer Insights to Internal Requirements	59
Understanding Customers' Needs	60
Partnership and Customer Focus	61
Interview Customers	63
Knowledge Exploration Exercises.....	64
Organization's Response to Customers' Needs	68
From Customer Interview to Organizational Transformation	68
Navigating the Context Map to Find Fit	69
Why Form a New Subdomain?	72
Cost of Introducing a Subdomain	74

TABLE OF CONTENTS

Requirements and Story Writing	75
Ubiquitous Language: What Is It? Why Is It Important?	76
Who Writes Stories?	78
Ubiquitous Language in Requirements	79
Writing Executable Specifications	81
Halfway into Gherkin	82
All the Way into Gherkin	84
Planning Work	85
Prioritized Backlog	85
Feasibility	86
Managing Dependencies	87
Valuable Stories (Verticals)	94
Technical Stories	96
Carrying Out Work	98
Definition of Done	98
Estimates vs. Velocity	100
Your Estimate Is Your Deadline	101
Achieving Predictability	101
Summary	104
Chapter 4: Design and Architecture.....	105
Architecture as a Cross-cutting Concern	106
Definitions and Purpose	106
Is Architecture Dead?	108
Architecture as a Service	111
Partnership with Domain Experts	113
Teams and Microservices	114
Architecture Supports Organizational Growth	115

TABLE OF CONTENTS

Architecture in Analysis and Requirements Gathering.....	116
Upfront Design Supports Gap Analysis.....	116
Knowledge Exploration and Architecture.....	118
Caveat of a Technical Solution.....	118
Architecture Body of Knowledge.....	121
Architecture Landscape.....	122
Buy vs. Build.....	123
Good Architectures	124
Architecture and Technology	128
Using Technology.....	132
Domain Model Kinds.....	136
Life with Anemic Domains.....	138
Layered Software Architecture	140
Microservices	142
Evolving an Ecosystem of Microservices	148
Command Query Responsibility Segregation (CQRS).....	151
Path to Event-Driven Architecture (EDA).....	153
Building Cloud-Ready Applications.....	155
Performance.....	158
Front-End Application Architecture.....	159
Built-In Security.....	160
Databases.....	161
Architecture and Implementation	164
Tactical DDD	165
Evolving Design	165
Writing Domain Layer's Code	166
Consolidate Development Tools and Languages	167

TABLE OF CONTENTS

Architecture for Testable Systems	168
Testable Code	168
Testable Application.....	168
Architecture for Deployable Systems.....	169
Versioning.....	169
Containerization.....	171
Architecture for Maintainable Systems.....	171
Mindset Shift—No Tech Debt.....	172
Working Systems.....	172
Fixing It Twice.....	174
Simple vs. Complex Systems	175
Summary.....	175
Chapter 5: Implementation and Coding	177
Cross-cutting Concerns Related to Coding	178
Professionalism of a Coder.....	178
Put Talent into Important Tasks	179
Continuous Improvement	180
Quality vs. Quantity.....	184
Code Reviews	185
Designing Code	187
Implementation of Architecture	187
Code Design Techniques.....	189
Essence of Object-Oriented Programming	189
Purpose of Design Patterns	193
Implementing Code	195
Tactical DDD Patterns.....	195
Declarative Design in Code.....	211

TABLE OF CONTENTS

Front-End Development.....	215
Exception Handling.....	222
Testing Code	226
Unit Testing.....	227
TDD, ATDD, and BDD	230
Code Deployment and Maintenance	232
CI/CD Before Development.....	232
Planning Deployment When Developing.....	233
Planning Maintenance When Developing	234
Summary.....	234
Chapter 6: Testing and Quality Assurance.....	235
Testing Processes and Principles	235
Who Is For and Who Is Against?	236
Quality Is Team's Responsibility	236
Test Everything, Automate Everything.....	237
Importance of Test Automation.....	240
Test Design and Architecture	244
Test Types	244
Test Case per Requirement	246
Test Automation Design Patterns	248
Test Data Management.....	259
Implementing Automated Tests.....	265
Delayed Automation.....	265
Early Automation	266
Enhancing Deployments with Test Automation	269
Fast Feedback and Compensating Actions.....	269
Continuous Deployment.....	272
Summary.....	273

TABLE OF CONTENTS

Chapter 7: Deployment	275
Culture of Releases.....	276
Why Release Software?.....	276
Unimportance of Releases.....	277
CI/CD—Deployment Foundation.....	280
Continuous Integration	280
Continuous Deployment.....	282
Building Deployment-Ready Applications	283
Developing Deployable Units	283
Ensuring Smooth Deployments via CI/CD	284
Dev–Prod Parity.....	285
Effects of Containerization on Deployments.....	286
Summary.....	287
Chapter 8: Maintenance and Support.....	289
Maintenance-Free Mindset.....	289
Organization's Approach to Maintenance	290
Support-Oriented Organizations	291
Award-Winning Support Teams	292
Who Prevents Problems?.....	292
Maintenance-Aware Mindset	293
Maintaining Applications in Practice	293
Fix Root Cause, Not Surface	296
Building Blocks of Maintainable Systems	296
Summary.....	298
Afterword: Wrap-up	299
References.....	301
Index.....	315

About the Author



Tengiz Tutisani has been in the software development industry for over 20 years. His experience ranges from startups to Fortune 500 corporations. He has held a variety of roles in technology leadership (software engineer, technical lead, development manager, application architect, solutions architect, enterprise architect, and chief architect). Tengiz's broad experience and frequent recognition for outstanding quality and performance have convinced him to teach others unique engineering and architecture techniques. He authored this book to describe advanced techniques for professional software development and architecture disciplines.

About the Technical Reviewer



Tom Graves has been an independent consultant for more than four decades in business transformation, enterprise architecture, and knowledge management. His clients in Europe, Australasia, and the Americas cover a broad range of industries, including banking, utilities, manufacturing, logistics, engineering, media, telecoms, research, defense, and government. He has a special interest in architectures beyond IT and integration between IT-based and non-IT-based services.

Acknowledgments

I want to acknowledge and thank the people who helped me make this book a reality. I am grateful to every person that I mention here.

First, a big thank you goes to every individual who took time to read the final manuscript of this book and wrote a short review, which will be used in various ways to inform the readers about the book's benefits: Dave Black, Jim Hammond, Preeti Baranga, Lasha Kochoradze, Nugzar Nebieridze, and Romana Stasiv.

Furthermore, I want to thank those who provided invaluable feedback about the manuscript by noticing typos or unclear sentences or even proofreading it: Dave Black, Jim Hammond, Gomti Mehta, Preeti Baranga, and Romana Stasiv. I know that all of you went the extra mile to ensure that this book reached its deserved quality. Without you, I am afraid that I would have published text that would make me feel embarrassed in the future.

Additional gratitude goes toward the Apress editorial team, who guided me through turning this publication into a polished piece of work worth taking to the shelves and screens of a broad audience. The Apress team includes Aditee Mirashi, Shobana Srinivasan, Mark Powers, James Markham, and Sowmya Thodur. A special thank you goes to the technical reviewer, Thomas Graves, who provided subject-matter expertise, challenging technical questions, and feedback essential to taking the book to the next level.

Finally, my unmeasurable appreciation goes to my wife, Romana Stasiv, who patiently supported me while I worked on the book, both the initial manuscript and with Apress. I wrote the text once, but I rewrote it about 150 times! Nobody would have tolerated so many long working hours without genuine love connecting our hearts. Also, with Romana's Agile expertise, I received early feedback about many topics in this publication, which helped me pave the road to the finish line.

Preface

Before getting into the book's central chapters, I want to explain why I wrote it to begin with, what you will learn, and how it will benefit you on your path to building software solutions.

Why I Wrote This Book

Looking back at my career in software development, I can see how I ended up writing a book like this. It addresses gaps in the industry that I experienced firsthand, and it reflects who I am—a software development perfectionist (bear with me—I will prove that this is not a bad thing at all).

This work is an attempt to fix problems that always challenged me. I know that these same issues worry many others too.

At every job I have had, I was bogged down by non-readable code, non-practical architectures, a vast number of defects, unclear requirements, unavailable domain experts, monolithic codebases, long deployments (releases), and so on. The business side was not happy, either. Domain experts complained about the quality of software, how it worked, its capacity to handle more users, the cost of developing it, and so forth.

I honestly believe that I have answers to all of these challenges *for those who want to solve them*. I care because I experience these same problems myself daily, and I address them successfully where my capacity allows me to do so. I have developed my process over the course of many years, and it is time to share it with others.

PREFACE

Next, I present a couple of anecdotes from my past that inspired me to write this book. I hope that some readers will recognize their own experience or personality between the lines. This narrative will also explain what kinds of problems we will be solving throughout this publication.

At my first job, I was the only developer on a team. I had the freedom to write code in any way I wanted, and that seemed to be a perfect environment in which to thrive. I developed many applications and components from scratch, and I enjoyed the freedom of improvising in code. Everything was going smoothly until the codebase grew to the point that I had a hard time working with it. I almost wanted to start it all over again! It turns out that many developers struggle with the same issue throughout their career paths.

How can the code that we wrote ourselves bite us back? There must be something wrong with what we do, don't you agree?

Therefore, we will learn how to write code that does not become our problem shortly after we author it.

At my second job, I worked on a team consisting of a couple of engineers, so I had a chance to read code written by other developers. It was a rather complex banking domain, and I needed to learn it fast to keep the job.

When such a challenge faces us, we realize that the codebase can either be our friend or our enemy, depending on whether it helps or blocks us from the goal.

I read a lot of code, but it did not give me any knowledge about the domain. Instead, it confused me more and more as time passed. I could only explain this complexity by believing that the code was not supposed to be understood by people because it was written just for computers. That was a naive thought.

We are humans in the first place, so we should write code for humans. Computers will understand code no matter how we write it. So writing a program for them is a piece of cake. Try to write code that humans can understand—that is a challenge!

Therefore, we will learn how to write code that expresses valuable domain knowledge and is readable by technical learners of the problem space.

I once worked on a project where developers had good knowledge of a problem space, and specialists in the subject matter were also accessible. However, meetings occurred in silos instead of leveraging access to domain knowledge. First, a domain expert would describe a requirement in business terms; next, the conversation would transfer to the developers, while domain experts did not (or could not) participate in the discussion from that point on. Engineers would “interpret” the expectation using technical concepts, patterns, and frameworks, and would use their terminology and tools for implementation. Because of this intentional disconnect, the resulting solution was too technical and often had little or no relevance to how the business worked; instead, it added inconvenience and complexity for business SMEs (Subject Matter Experts).

Therefore, we will learn how to write software that serves both developers and non-developers equally. It is time that we start helping business colleagues instead of complicating their lives.

One more project comes to mind. We could not focus on new feature development because the defects were frequent. We had to jump on them as they appeared due to the high visibility of the application. We would fix errors and then go back to our current sprint (iteration) backlog, and we just knew that the next bug was hours, if not minutes, away. Consequently, new feature development went slowly and was unorganized.

PREFACE

It is a terrible feeling when you cannot focus on something and finish it without interruptions. It is like being tired in an airplane and trying to sleep, but a narrow seat and limited legroom do not let you relax completely. In those moments, you dream about a simple, comfortable bed.

Therefore, it is crucial that we know how to avoid defects and allow ourselves to focus on new feature development instead.

Although many consider bug-free software impossible, we will learn how to reach this objective in real life.

I remember the tired face of an engineer who was tasked with supporting an application in a production environment. He did not even write that software and did not know all the intricacies inside the system. He had to keep bugging an actual development team with questions when a new kind of issue was discovered. He had a handbook full of troubleshooting steps for various types of symptoms that the application could produce. He was hesitant to contact the development team every time because the engineers were "cool guys" who had more important tasks than to support the application (written by them in the first place).

As you can imagine, the mentioned system was a nightmare to run and troubleshoot. In my mind, the root cause was developers' being unaware of production support's pain points and lacking the necessary skills for producing better software that would be easier to maintain.

Would the outcome be different if developers had to run and support their solutions in a production environment instead of somebody else doing it for them?

To tackle the described problem, we will learn how to build software that works in a production environment without a dedicated production support team by distilling ways to develop better programs and maintain them in production with minimal to no cost.

Perhaps every developer has been on a project where developing a new feature is like rocket science. The codebase is so complex that an implementation process is merely a trial-and-error activity: a developer needs to make careful changes to several moving parts while finding more unknowns on the way. Such an increased level of complexity makes work and estimation very difficult.

When software development becomes a discovery process when implementing every task, this hints at problems in how an application is structured.

We will learn about development and architecture techniques for building better software systems that are easier to develop and change.

Another side effect that happens in the mentioned situation is a slowdown in development pace.

Hence, we will also study techniques to accelerate software development processes instead of slowing them down.

I recently encountered a web application that was deployed to three servers to handle many parallel requests. Surprisingly to engineers and their management, the system was struggling to serve only 100 simultaneous users. They tried to add more servers, but even more strangely, it only worsened the situation. As a last resort, the team rearchitected the application for better scalability, but that exposed other bottlenecks, complicating the goal of reaching more users.

When such circumstances happen, you need to admit that your application is not scalable.

Therefore, we will learn how to build software that scales out horizontally by adding more instances of the application when the number of users increases.

These situations just described served as a foundation for the *six primary pillars* upon which this book is built. We will go into more detail in the next section.

PREFACE

For now, I hope that most of you either recognized your experience or felt an urge to solve these kinds of problems at present or in the future.

Meet Effective Software

A computer program done right should prevent situations like those just described. That is what I call “effective software.” We will distill this definition next.

Definition of Effective Software

In this book, I define effective software as that which satisfies the following criteria:

1. Meets users' expectations.
2. Has no defects.
3. Scales out horizontally.
4. Has no dedicated production support team.
5. Accelerates development pace.
6. Doubles ROI per developer, team, and software.

You must remember these items because they form the foundation of this book. If my writing is successful, you should see that each chapter or section targets at least one of these points.

Throughout the book, I will be referring to these items in various ways interchangeably: “six pillars of effective software,” “six elements of quality,” and so on.

I will go into slightly more detail about each of the mentioned elements in the next chapters, thus helping you understand why they are essential.

Where This Book Fits in Among Large-Scale Frameworks

In this section, I will briefly explain where this book fits in when applying large-scale frameworks across organizations.

Enterprise Architecture Frameworks

Some chapters in this book (e.g., “Organizational Structure” in “Part II: Crosscutting Concerns”) solve problems similar to those addressed by TOGAF (Technology Open group Architecture Framework, see [TOGAF]) and other enterprise architecture guides (see [Gartner EA]). I want to explain analogies and differences between popular frameworks in this field and my own approach.

At the time of this writing, TOGAF is the most popular and proper representation of the enterprise architecture field. Therefore, I will only refer to it for simplicity instead of generalizing my comparison with the entire area. Most of the points will apply to other frameworks too.

TOGAF is a leading approach for enterprise architecture, and it has proven to be useful over the decades. Many large organizations choose to use TOGAF for its reputation and all-in-one nature. It covers various topics, from business architecture through governance and execution of enterprise activities.

Various approaches that I describe in this book can be seen as a basis of a modern, lighter-weight, and scalable enterprise architecture framework. I firmly believe that the structures described here will suffice and can be adapted by many organizations without the necessity of opting into heavier-weight frameworks such as TOGAF.

Nevertheless, I do not intend to position this book as an alternative to TOGAF as a whole: the two publications have distinct purposes and *problem scopes*. Therefore, they propose different solutions.

SAFe—Scaled Agile Framework

SAFe (see [SAFe]) stands on several core values, such as Agile, Lean, and Systems Thinking. Overall, this framework helps organizations align their large-scale efforts across the enterprise by treating deliverables as increments for the entire company.

For the described idea to be successful, besides process enhancements, we need technical ownership and transformation in areas such as applications, systems, enterprise architectures, DevOps, and CI/CD, to name a few. The practice has shown that, without such supporting mechanisms, SAFe can quickly become an artificial shell around unsolved technology challenges that fall outside of the Agile framework's focus. My book can fill this gap by providing technical guidelines for implementing high-quality, large-scale solutions.

To outline the compatibility between the two areas of study, here is how this publication fits into SAFe's core values:

- Part II: Crosscutting Concerns will explain how enterprise-wide architectures need to be formed based on the same Systems Thinking competency as SAFe.
- An Agile process, which comprises SAFe's core values, is a possible (although not mandatory) approach when applying techniques described in this book.
- While also being part of SAFe competencies, Lean development is an essential part of this book, such as when I suggest validating product hypotheses before implementing full-fledged solutions.

For all those synergies that I just described, the book in front of you can become a supporting mechanism when implementing SAFe or other organizational process initiatives.

How to Read This Book

First and foremost, you need to look at this book as a guide to implementing effective software. As I explained earlier, when I say “effective software,” I mean a digital artifact that meets the six pillars of quality described previously.

With each chapter, I intend to fulfill specific expectations of effective software. Some sections will target just one issue, while others will target several of them.

You will notice two types of chapters primarily:

1. A section that lays out a problem definition and a solution to it, focusing on one or more of the ideal pillars.
2. A textbook-style chapter that covers a complex or commonly misunderstood subject, which I consider to be a prerequisite for building effective software.

I hope that by presenting this material in such a format, I make it easy to understand and follow in practice.

A higher-level structure of the book (see Figure P-1) will follow the flow of the software development process in some way: we will cover crosscutting concerns as well as all typical stages of the software development lifecycle.

PREFACE



Figure P-1. *The book's high-level structure*

The topic of crosscutting concerns will precede the rest of the subjects. This order is essential because most crosscutting solutions do not fit into any of the typical stages but still require attention beforehand to agree on fundamental assumptions. Afterward, diving into each step of software development will uncover and address various concerns within those areas.

At the end of the book, I have included references to reading materials that support this publication. A link to a reference is a codename enclosed with square brackets. For example, [Tutisani web] is a link to my website, which you can find in the references under the "Tutisani web" codename.

An index with keywords is the last piece of this publication, which will help you find specific terms in the book's text.

Enjoy your reading!

CHAPTER 1

Introduction

In this chapter, I want to examine the fundamental ideas surrounding the building of effective software. Presented topics will range from historical reasons for the industry's problems to decisions of avoiding or accepting ideal solutions. Additionally, I offer a deep dive into the definition of *effective software* that I briefly introduced in the preface.

History of Inefficient Monoliths

Most of the software projects executed nowadays are inefficient. They are slow and complicated, produce defective monoliths, do not meet users' expectations, and so on. What is the reason behind these problems?

Haven't we already learned how to build software? Yes, we have, but *times have changed*.

When I started writing code, hardware was expensive. Companies used to buy a server that had a "beefed-up" processor and memory capacity. Afterward, developers wrote code that had to offer as much functionality as possible. Organizations needed an *all-in-one* solution that had to run on a single powerful machine. If there were not enough resources on a server, an administrator would order additional hardware and connect it to the server machine. There were not many people who used the internet and software, so there was no need to worry about scalability. The world needed monoliths, and developers learned how to build them. Computers did not know how to talk to each other, and so distribution and integration

CHAPTER 1 INTRODUCTION

were not concerns. A single team had to consist of hundreds of engineers because there was no other way to work on a shared codebase; they sank in communication overhead and always stepped on each other's toes. These challenges were a norm back then; *software had to work for any price*.

Today, the situation is entirely different. Servers are so cheap that companies want to scale out automatically in the cloud, opting to pay at the end of each month any amount that ends up in their invoices. In return, there is no need to deal with hardware upgrades anymore. There are so many people using the internet that no single server could handle the load, even if just a small fraction of those users were to visit a website. The only way to handle that load is to scale out, since scaling up has demonstrated insufficient limits. A single application cannot manage all functions, so companies want to build multiple mini-applications (microservices) that work in a connected, distributed fashion. And thus, we have numerous software development teams, each responsible for independent solutions. Scalable and efficient software development teams, processes, and systems have become a winning factor for leading technology firms.

However, as I said in the beginning, most of the software projects still employ legacy processes that are not compatible with modern needs. This gap explains the problem of inefficient monoliths. To solve this issue, we need to learn how to develop software that is ready for the future—effective software.

Why People Avoid Building Effective Software

As we learned earlier in this book, effective software is a high-quality digital product that can maximize return on investment (ROI). If these promises are genuine, who would not want to build a program on such terms? It turns

out, people still hesitate, and I will explain some of the typical reasons for this. I want to present these challenges to encourage you to overcome them and to set realistic expectations.

Before we go any further, you should know that you have chosen not an easy path for yourself, but also not an impossible one. I have a rather low level of risk tolerance, and my suggestions are reasonable to me, which should further assure you that matters are not hopeless at all. Instead, it is my responsibility to tell you what to expect when attempting to build ideal software solutions.

Learning Curve

I have collected the necessary skills to build effective software throughout my career. At the time of this writing, that is more than 17 years.

Admittedly, it took me so long because I was not always sure what I was seeking—I made up my mind on the way. Nevertheless, all the techniques needed to build such a high-quality product are scattered everywhere—process, technology, concepts, patterns, project management, testing, development, architecture, and so on. While this book should help you navigate your way in the universe of endless knowledge and information, it will not shorten the journey drastically. It is just not possible. There is thus a significant learning curve behind the effort of building effective software.

I am glad that you are still going for it, as not everybody does.

Quality over Quantity

Often, companies want to release many features within a short timeframe and so cut corners whenever possible. I call this phenomenon “startup syndrome” since newly formed organizations and teams commonly opt for this approach. These people hope that they will rewrite the entire solution once they have become profitable or showcased their abilities, downplaying the need for the rework or increased number of bugs.

CHAPTER 1 INTRODUCTION

Their plans to recreate applications rarely succeed due to new and pressing needs that follow getting their first paying customers. Consequently, these projects experience further slow-downs since they have built their solutions on quick prototypes, which they planned to throw away or rewrite. Most important, what they fail to realize is that *their approach is not faster than building effective software, on average*. Let me explain the math behind this statement.

When you start practicing effective software development techniques, the first few months are indeed slower and more cumbersome. This weakness is a result of the learning curve that you must get through. After that period passes and as you master advanced exercises, things improve (see Figure 1-1). This turnaround is due to continuous process improvement and the innovation that is only possible with a proper foundation. From my experience, most projects reach this breaking point in a couple of months.

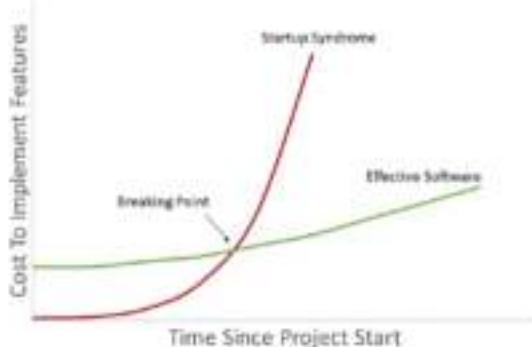


Figure 1-1. Feature development cost with startup syndrome surpasses effective software costs after a couple of months of project's existence

So, you are not giving up quantity for quality as long as your project lasts for at least a couple of months. However, the misconception still exists that effective software requires a sacrifice of quantity to attain quality.

Highly Paid Experts

For the very reason that building effective software requires a considerable learning curve, those who can do it are highly paid and in demand.

Those teams or organizations that lack the necessary expertise but still want to build effective software will need to hire experts and pay their salary. On average, benefits will outweigh the costs when we do so. Unfortunately, edge cases such as hiring an incompetent expert or project scope creep serve as bad examples and discourage attempts. Consequently, to avoid wasting money without results, many organizations and teams avoid building effective software altogether.

Naive Hopes to Survive

Have you ever been in a situation like this? You are driving on a highway, and a radio host is informing you that there is a severe traffic jam one mile ahead of you. The road to the horizon is clear, and you are late for an appointment. What do you do—take the next exit or continue pushing the throttle?

People commonly hope that the news is inaccurate, or that the traffic jam is too far away to affect them. So, they decide to stay on the road. However, a better choice would be to take an immediate exit and seek an alternate route. It is better to be late to an appointment by 15 minutes instead of missing it altogether by arriving an hour late (which will be the outcome for those who stay on a highway in such situations, assuming the radio host has a reliable source of data).

Similarly, engineers and managers involved in software development often cannot believe that things will get worse very soon, even though they keep cutting corners. They experience a fast and smooth development pace only because their projects are so new, and accumulated tech debt does not seem so heavy yet.

CHAPTER 1 INTRODUCTION

These conditions are characteristic of new solutions! Sooner or later, most software projects slow down due to complexity, defects, inability to scale, and so on. This outcome arrives not suddenly but instead gradually, which is what makes it so hard to measure, predict, or even notice. Leadership then calls senior-level engineers for help, but it is impossible to fix the situation without a complete rewrite.

Still, people have naive hopes that this will never happen to them, and so they see no need to build effective software.

These are some reasons to avoid building ideal solutions, but what's on the other end of the spectrum?

Software Development Perfectionism as a State of Mind

First of all, let me say this—I am aware that the world is not ideal and that “perfect is the enemy of good.” I also know that most developers think that software cannot be perfect or worth the investment. I will take my chances and explain why I think you should choose to be a perfectionist.

Is It Crazy?

When you introduce something new and utterly different from everything else, it may seem crazy. I do not want to take too much credit, but it is similar to the effect of inventing electricity or discovering that the earth is round and not flat. Those ideas were once considered crazy. Therefore, if we want to improve, we must give a chance to new proposals.

I do what I teach in this book daily. Nobody thinks that I am crazy. So, it is not so scary to try it.

If you worry that others will see you as stubborn, I assure you, that is not what I am advocating. This book is only a collection of knowledge that can improve your skills. The rest depends on how far you want to take it in practice.

Desire Behind Perfectionism

Perfectionism, as described in this book, suits those people who want to *build software that always works and does not break*. While I consider this to be the mission of every developer, I understand that not everybody shares this vision.

Nevertheless, to better digest the material in this book, you need to adopt the proposed mindset. Afterward, everything I present will make more sense.

Is It Worth It?

A commonly asked question around effective software is whether it is worth the trouble. Is it not easier to build yet another system that works more or less okay, breaks occasionally, and gets fixed when we need it?

From my experience, when techniques to develop ideal solutions become part of our professional habits, building effective software becomes no more difficult than writing any other program. From that point on, we save a considerable amount of time on every aspect of development, and we achieve customer satisfaction by delivering more with less. Eventually, we become the winners compared to other software projects that encounter overhead in every area where we are leaner.

So, yes, purely mathematically, it is worth it. The key is to judge the worthiness based on a long-term advantage compared to other projects and not only based on the volume of short-term efforts.

The objective is also easy to understand; we only speak about six basic ideas.

Six Pillars of Effective Software

When I tried to define what I do daily, I discovered the six pillars of effective software, as follows:

1. Meet users' expectations.
2. Allow no defects.
3. Scale out horizontally.
4. Have no dedicated production support team.
5. Accelerate development pace.
6. Double the ROI per developer, team, and software.

While these pillars express the intent, they can also create confusion, disagreement, and resistance. This split in opinion happens because each of us is unique, and we look at things differently. Some will say that the problems I try to fix do not exist at all; others will say that there is no solution to them, or that I am not solving the right issue.

To align our vision, I want to clarify what each pillar means.

#1. Meet Users' Expectations

This pillar is trying to solve any of the following problems:

- Developers closely follow requirements that the domain experts write, but customer satisfaction still keeps dropping with every release.
- Developers write software by using all the modern tools and technologies, but the program is impractical to use by the end users.
- Developers do not fully understand what domain experts want, and there is a constant challenge in collecting knowledge.

#2. Allow No Defects

For a start, let us agree that defect-free software is a reality. I have often heard that engineers treat defects as a regular thing. At times this is due to not knowing how to avoid bugs; in other cases, they consider finding and fixing mistakes to be the optimal way to produce an application. I will address the former opinion throughout this book (i.e., the means to achieve a defect-free program). The latter view cannot be valid as bugs are nothing but overhead. Specifically:

- Developers need to go through context-switching multiple times. First, when they need to fix a suddenly found bug urgently, and then again when it is resolved, and they need to turn back to feature development. Context-switching is a waste that burns time and resources.
- Every defect must be discovered, documented, prioritized, planned, assigned, investigated, fixed, verified, tested, possibly automated, and tracked to eventual closure.
- Developing new features in a system full of defects is more complicated than doing so without defects. Bugs require special treatment, workarounds, and tricks. Also, broken functionality irritates those who often encounter it—developers.

If not for all this overhead with defects, you could focus more on new feature development. Furthermore, defects create discontent for both the customers and the developers.

- Bugs often mean a broken user experience, which is a frustration for a consumer of a program. It is a negative hit on a company's reputation too.

- Fixing defects is not the most favorite task for developers. They would rather be writing new features. We all understand this very well.

Do not be confused; there can still be defects caused by something other than code; e.g., missing content on a page is an authoring issue if the content is dynamically configurable.

This book will teach you how to avoid *code defects* specifically. It will also provide guidelines on how to prevent other kinds of bugs on a case-by-case basis, but more emphasis will be placed on code defects.

#3. Scale Out Horizontally

I will discuss and address scalability issues in two different contexts—first in software and then in a development process.

With more users and customers, you should be able to deploy more copies of your application and sufficiently handle the increased traffic. That is *horizontal scalability of software*. Incorrect architectural or design choices made earlier can hijack an ability to scale horizontally later.

Vertical scalability would mean to increase server memory again and again until you reach a limit and cannot scale anymore. Hence, horizontal scalability is much preferred as it has no such boundaries.

With a need to solve different business problems, you should be able to form new teams and evolve systems and organizations organically. That is *horizontal scalability of a development process*. Various things ranging from incorrect code design to improper requirements management can affect the ability to scale horizontally.

Vertical scalability would mean increasing the size of a single team until the members of it started spending the entire day on communication overhead, every day. So the horizontal scale is preferred since vertical scalability amplifies costs.

#4. Have No Dedicated Production Support Team

Issues in the production environment often become so frequent that you need to hire a dedicated production support team. This situation seems quite ordinary when it happens.

It is possible to avoid that additional cost overhead by employing proper tactics, which we will discuss in this book.

Let me clarify—I am speaking about a technical production support team and not about a customer-facing support group, which you may still need to understand end users' concerns or to answer their questions in an online setting.

#5. Accelerate Development Pace

Have you noticed how software development becomes more cumbersome and slower over time? That is a typical pattern in most of the companies and teams that develop software nowadays. It does not have to be that way.

#6. Double the ROI per Developer, Team, and Software

If you achieve all the previously mentioned elements, guess what happens?

You get more for the same amount of investment (imagine "buy one, get one free"), or you get the same for less investment (imagine "50% off for members perpetually").

This item is an outcome of applying the approaches described in this book. It has become part of the "six pillars" to explicitly emphasize the business advantage that you gain.

Summary

This was quite an introduction. We glanced at both the hesitations and enthusiasm behind building effective software and what it is that we aim to deliver.

In the next chapter, we will look at commonly overlooked subjects vital to delivering needed outcomes, such as leadership, people, and culture.

CHAPTER 2

Cross-cutting Concerns

Building effective solutions is not only about writing code or architecting systems; this process relies heavily on leadership, operations, organizational structure, and even internal politics. Ignoring these vital aspects is comparable to dismissing the weather forecast when predicting the quality of harvesting season.

Experience has proven that an effective technology transformation has to be driven by leadership. No matter how hard engineers try, their attempts will not suffice against a wall of political resistance. Hence, I will start this chapter with recommendations applicable to leaders and managers.

After leadership accepts the importance of their role in the act of producing high-quality software, it is time to rethink other supporting factors for positive outcomes. The organization's shape and processes within it must aid the optimal flow of information, growth, and scale; this reasoning puts the mentioned topics on our radar. Finally, we will drill into culture and people as they are essential constituents of organizations. This thread of thought guides the structure of the chapter.

Execution, Leadership, Management

Let's first set some expectations for the leadership and management of software engineering organizations in the context of transformation.

Importance of Software Development Transformation

The technological world has changed 180 degrees since its inception. Nevertheless, many organizations that formed during the early days have not changed much since. It has become critical that these companies transform and adapt to the newly established environment. If they cannot revolutionize, their existence will soon be at risk due to their lack of competitive advantage.

The need for transformation is also relevant to building software. To appreciate what has happened in this field, let us recall that a couple of decades ago, engineers did not write unit tests, there was no open source community, and there was no such specialty as "Tester" or "QA." Big waves of breakthroughs have taken place in software development, architecture, distribution, and scalability.

Therefore, it is highly critical that every engineering organization transform and conform to today's standards and demands.

Companies that are just forming should ensure that they meet modern software development standards. Those that have existed for a while need to transform at the earliest possibility.

If transformation does not take place, produced software will not be worth the trouble. You will have to invest in building programs twice as much as your competitors due to lower quality, higher human and technical resource costs, decreased customer satisfaction, and so on.

How to Achieve Software Development Transformation

Transformation is a very complex task. In simple words, nobody comes to their everyday job and says, "We will do things differently from now!" Even if somebody says these words, things do not change overnight. This problem is relevant for any size and kind of organization or team.

Furthermore, the quicker the transformation is, the better the outcome. If it takes a long time, we risk playing a catch-up game forever, because the target keeps moving while we slowly change. In such a case, it also increases costs since our focus is on implementing changes instead of executing an established business model. This stretch can result in losing to competition or having an inability to gain an advantage over others.

The leadership of software development organizations should seek ways to achieve transformation in the shortest time with the practical steps and durable outcomes. Here is one technique that I recommend to reach this objective:

Find an expert or group of experts that can assess your current situation, lay out a transformation plan, and deliver it in a discrete timeframe. Give them the power and right to execute. Follow their guidance and get to the finish line together. Keep the experts accountable for delivering promised results.

This approach is just one of the options, but the focus is on the timely execution of a transformation to avoid getting into the never-ending catch-up game.

Transformation Is for Everybody

At the time of this writing, many organizations are attempting to accomplish an Agile or other technology transformation. Here is how they typically approach this task: Management comes to developers and

CHAPTER 2 CROSS-CUTTING CONCERNS

tells them about “new rules.” In a couple of months, engineering groups learn how to be agile or follow best practices in the industry. Despite this change, these companies do not see significant improvements in their revenue or quality of work. Consequently, management declares that transformation did not make any visible changes, and hence there is no real value under such initiatives. This statement is inaccurate, as there are other explanations behind the described failures.

Let me ask a simple question: What was management doing when development teams were going through the proposed transformation? *Turns out, they were watching from the side!* If changes touch only the bottom of the company and its top continues operating as before, we naturally end up having incompatible parts trying to work together. This mismatch is the real reason for failed transformations!

Therefore, if you want to go through an Agile or other technology transformation, tackle this goal throughout the company. Transform both leadership and development teams, and not only one side of the equation. Transformation is not an experiment for development teams; it is an organization-wide strategic initiative, so treat it like one! If leadership is not ready to change, I recommend not starting the transformation at all; otherwise, do not be surprised if it fails later.

Here is how you should interpret the preceding recommendation for Agile and technology transformations:

- Agile transformation must not be limited to applying modern processes (e.g., Scrum or Kanban) to engineering teams. Instead, leadership must consider implementing a large-scale Agile framework such as SAFe (see [SAFe]), which accounts for transforming leadership too.
- Technology transformation must not be left up to only engineering teams, because typically developers do not have enough power and influence to modernize

solutions across companies. Instead, leadership must actively participate in changes by coaching and mentoring people to follow the new guidelines. An example of such a support is a letter from Amazon's leadership that promoted API strategy when the company needed this improvement across all teams (see [APIE API]).

Ad-Hoc Tasks vs. Process

Building high-quality software takes focused and consistent work that represents a repeatable process. Context switching at unplanned points should be avoided as it often delays deliverables, makes mistakes possible, and decreases employee satisfaction.

Management or other stakeholders often come up with ad-hoc tasks for engineers. They sometimes directly call a developer and ask them to fulfill a specific request. An employee has to put aside their previous assignment and work on this new task immediately. This approach creates context switching, so I want to make managers and stakeholders aware of the undesired outcomes.

Managers and stakeholders are advised to refrain from hijacking an established software development workflow with their ad-hoc tasks. Instead, their requests should go to a product owner or a project manager (or to a person carrying a similar role in an existing process). These tasks will be prioritized to be worked on later, as the schedule allows.

If you work in iterations, then the newly submitted task can be fulfilled in the next sprint. For other processes, it can be queued after the work that is currently in progress.

Hands-on Leaders

Management often steps back and allows their teams to thrive and show what they can do. This attitude helps give opportunities, but it can also put people under pressure if the outcomes are critical to the organization or the employees. If something does not work, now it is their fault, even though they worked hard, while leadership stood aside.

On the contrary, I have also seen managers who want to make all decisions themselves but fail to do so correctly. For example, one manager, who had a great personality but lacked technical skills, would not care which architectural choice was better (he did not understand the importance of architecture at all) and picked one out of thin air. This person put the project on a failure path, and in the end the developers were seen as guilty again. Those engineers who think that the “boss is always right” will probably never agree with the described issue, but such a leadership style still has to be addressed.

Management should take development, improvement, and transformation goals seriously. If leadership wants to get there, it will not happen without their support. This section is a call not just for managers but also for the hands-on leaders who can execute an IT organization roadmap with their teams. They are ready to share responsibility for failure, and they work as hard as their teams do.

While working with a hands-on leader side by side, engineers feel empowered to deliver better-quality software that meets and exceeds customers’ expectations. Also, they are motivated and feel like they are part of a team. If success can only be planned, who can do it better than a manager who influences their entire organization?

Overall Support from Management

Recommendations from the previous sections by no means represent all support that people need from their managers. Leaders should understand organizational goals and aid their teams in getting there efficiently. In this book, I will try to capture all steps of this journey. Management will need to be familiar with this information to help their organizations succeed and, more important, to transform them together.

Besides management, the organizational structure also impacts the subject of this book, so it is our next big topic.

Organizational Structure

In this section, I will draw a line between the organizational structure and the optimal software engineering processes. This link may be non-obvious out of the box. Hence, making it an explicit matter will help address this book's objectives.

A Couple of Important Definitions Upfront

In large, modern IT organizations, we typically have many teams and applications and an elaborate graph of requirements. If we are not careful when forming or managing such environments, we can introduce unintentional overhead that can create complications and slow down a significant portion of the company. In smaller organizations, these same problems can quickly occur with growth if we do not plan properly ahead of time.

Before I clarify and break down the described problem in the next sections, it is essential that you understand the basic concepts of subdomains (see [Evans DDD]), bounded contexts (see [Evans DDD]), and microservices (see [Fowler Microservices]). I will base several

CHAPTER 2 CROSS-CUTTING CONCERNS

important suggestions on these terms as they help define boundaries for applications, systems, and subsystems. In this context, *boundaries are more important than what is inside them.*

Make sure to familiarize yourself with these concepts before we go any further. If you are already familiar with them, then let me quickly recap to refresh your memory:

- A subdomain is a part of an overall problem that an entire organization solves. For example, an established brand can have a subdomain of "Sales" that expresses how it sells its products online.
- A bounded context is a boundary within which a specific terminology makes sense. A meaning of a given phrase can change when looking at it from another bounded context. For example, the word "Order" within the "Sales" bounded context means a purchase request received from a customer, while within the "Procurement" bounded context, it represents an intention to buy parts or equipment from suppliers to compose a product.
- A microservice is a small-size application with well-defined boundaries.

Forming Organizations

Technology organizations often form chaotically—people do not understand what problems their team or an entire company solves, how they relate to each other, which project has a higher priority, and so on. These knowledge gaps create many inefficiencies, such as making non-optimal decisions about systems, subsystems, and overall enterprise

architecture; slowing down or hijacking the company's success; or misrepresenting the employer's interests when delivering technical solutions. These issues can affect many of the mentioned quality pillars, including the ability to meet customers' expectations.

Therefore, form an organization from top to bottom (as in architecture and decomposition of the domain, not as in management). Define an overall company-wide problem domain, then divide it into subdomains, and divide each of them further, until there are no hidden subdomains in any of the defined subdomains (see Figure 2-1). Document and popularize this information so that members of the organization can speak to it.

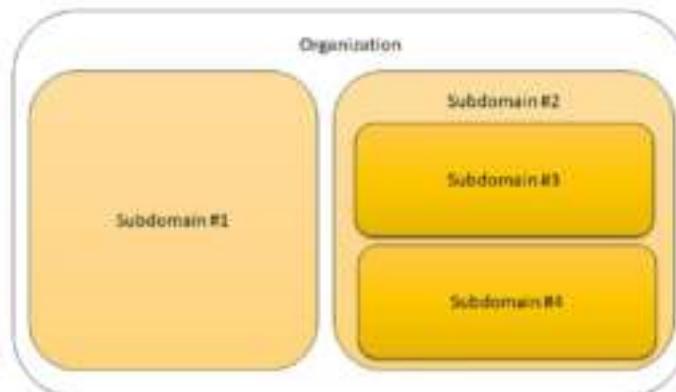


Figure 2-1. *Distilling subdomains within an overall organization domain*

When knowledge about the subdomain hierarchy gets into the hands of the organization's members, they will use it as a tool to solve business problems, find their paths through complex flows of the field, correctly prioritize projects and requirements, and so on. Overall, you will receive higher employee engagement and satisfaction levels. Also, meeting customers' expectations will become more achievable for people in the organization.

CHAPTER 2 CROSS-CUTTING CONCERNS

Do not run this exercise mechanically. Instead, pay attention to the boundaries of each subdomain. Ask yourself critical questions: Is it a well-defined, cohesive problem? Is it an independent domain from other subdomains? Does it serve the needs of a parent (containing or nesting) subdomain?

Also, do not use factors besides business knowledge to identify subdomains. For instance, it is a bad idea to use a project funding model or professional specialties (developers vs. testers vs. project managers) to define subdomains. You need to be solving *business problems*, and all else should serve the same purpose, instead of fitting business objectives under unrelated concerns. Otherwise, the formed organization will be helping the company's internal needs only rather than serving the needs of end users and customers.

Forming Subsystems

In engineering organizations, new software projects often start randomly and maintain an unclear nature throughout their lifetime. Usually, each new project aims to deliver a new subsystem. Still, clarity is missing about boundaries, cohesiveness within its borders, relationship with other subsystems, business value compared to other subsystems, and so on. As a result, the project runs heavier and slower than it could due to unresolved technical conflicts, incorrect implementations that require a rewrite in the early days, and sometimes even duplication of effort among various subsystems.

Therefore, treat each subsystem (both existing and new) as a bounded context. Identify and document existing bounded contexts. Draw a context map (see [Evans DDD]) outlining relationships between bounded contexts as well as their related subdomains. Decide where the new bounded context fits, which subdomain it will serve, and the ties it will have with other bounded contexts (see Figure 2-2). Get in touch with teams that own the existing bounded contexts and ensure that the planned rollout of the new bounded context fits the overall picture (i.e., context map).

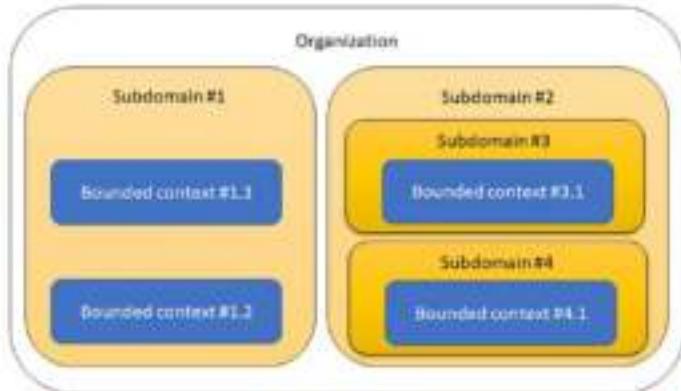


Figure 2-2. Forming bounded contexts within subdomains

A well-understood and documented context map is an excellent tool for software architects and engineers to help solve their everyday needs in typical IT organizations (and ideally, beyond IT as well) while integrating with other subsystems, prioritizing requirements, planning proper implementation strategies, and so on. As a result, you have highly scalable development processes, less complexity and thus acceleration, clarity on relations between business and technical requirements, and so on.

Do not form subsystems just for the sake of having them. Randomly defined systems usually cause confusion, misinterpretation, and misuse among engineers and architects. This randomness will create a slowdown and challenges to meet the users' expectations. Also, such approaches will affect your ability to scale processes horizontally. Choose natural, business-defined boundaries instead of mechanical boundaries. Otherwise, you will introduce complexity, and somebody will have to deal with it.

Forming Microservices

Microservice is a comparably new term that has many definitions depending on the source of information. Historically, it is a refined form of SOA (Service Oriented Architecture, see [Wiki SOA]), and hence some people call it a “granular SOA.” While we all understand that microservices architecture involves building interconnected services, it is easy to miss the importance of defining proper boundaries. If you view a microservice as nothing other than a mini-service and try to apply this pattern to your work, then you will end up building an arbitrary number of services that integrate based on mere technical convenience only. When you form such systems, it is hard to see the point of using this architectural style altogether, except maybe for the sake of the buzzword. No architecture has the right to exist without clear business benefits, and there must be some benefit to microservices too.

Define a microservice to implement a bounded context and maintain a one-to-one relationship between the two. Use these terms interchangeably, and adhere to the same boundaries for both of them. Use this mapping to clearly express a business value behind a microservice—it is a subsystem that solves the problems of a subdomain, just like a bounded context does.

Following the shapes of bounded contexts helps you implement microservices in the right way out of the box because you have the form established and it is well understood by both the business and the technical people. On the contrary, incorrectly defined microservices create all sorts of problems, such as a lack of technical scalability, integration complexity, mental complexity, failing to meet users’ expectations, and so on.

Later in this book, I will come back to the microservices topic from a technical architecture standpoint. In the current section, I only wanted to outline the relationship between microservices and bounded contexts.

Forming Teams

Software development teams often deal with an *overhead* of communication, conflict of interests, shared codebases, shared production releases, and so on. It is easy to notice that the cost comes into the picture when one team has to *share* something with another. On the other extreme, an independent group has the luxury of moving as fast as they want without worrying about dependencies on outside counterparts.

How can we maximize the number of autonomous efforts in an organization that has multiple teams? As it turns out, the solution is easy to explain after we have learned about context maps and bounded contexts.

The trick is to form teams around bounded contexts. Allow a group to own one (which is ideal) or more bounded contexts, but do not allow more than one group to hold ownership of any given bounded context (see Figure 2-3). Help team members understand bounded contexts and subdomains that they own and with which they interact. Ensure that people integrate bounded contexts according to their connections on the context map. Otherwise, the context map needs to change to express necessary knowledge.

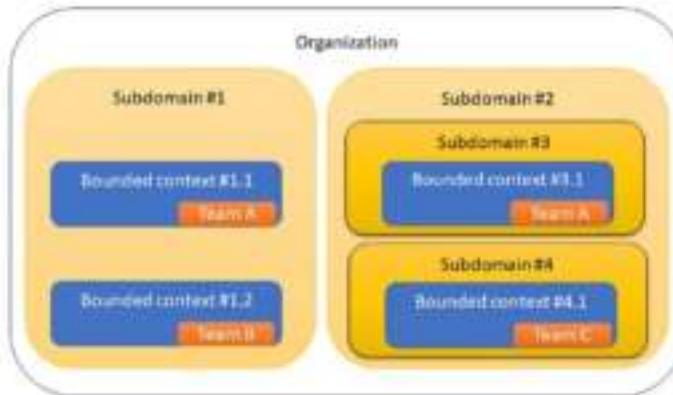


Figure 2-3. Forming teams to own bounded contexts

CHAPTER 2 CROSS-CUTTING CONCERNS

When teams arrange themselves around bounded contexts, there are little or no shared resources, per the design of the bounded context, which allows a great deal of autonomy and acceleration over time. There is nothing more empowering than having the ability to make independent decisions within the owned context and to go as fast or as slow as a group needs to. There is no need to wait for other teams or to fit into the bigger picture. This freedom results in better software systems that scale in isolation and form a sound part of an overall enterprise-wide ecosystem. Teams composed in such a way also demonstrate better scalability from a process standpoint.

You can create problems for yourself if you form groups without looking at a context map. Specifically, you will either miss a bounded context, create a shared issue, or duplicate efforts. Either way, organization-wide outcomes can be impacted negatively.

To put this recommendation in perspective with our quality pillars, you need to make your teams as autonomous as possible to increase and eventually double the return on investment (ROI) per development effort.

Feature Teams

Functional capabilities are what your customers and end users receive, and hence they are the basis of the company's income and success. These items are delivered by *feature teams*, which turns those teams into a vital part of the engineering staff.

Often, the importance of feature teams is discounted, thinking that their job is simple. Instead, it is common to see people believe mistakenly that having platform or infrastructure teams are the sole guarantees of success. This confusion undermines the quality that gets into customers' hands, and it also impacts the ability to meet users' expectations and avoid defects. The thought process behind the preceding precaution must be easy to follow—feature teams work for customers, not infrastructure teams.

Therefore, pull the best talent into feature teams. Increase business knowledge within these groups to deliver the requested functionality with the highest quality to customers. Give feature teams power to be independent and autonomous; award them for positive business outcomes.

Platform and Infrastructure Teams

Definition: A platform or infrastructure team builds a common foundation that multiple feature teams can use instead of building it up from scratch for each capability. I call this common foundation a “platform.” The platform can be a base framework for services, applications, messaging, database access, CI/CD pipelines, or any other infrastructure. “Platform” and “Infrastructure” teams can mean the same or different things from company to company, but they serve a similar purpose. In this book, I will use these two terms interchangeably.

I have often worked with feature teams, and here is what I noticed: *Most custom-built platforms that feature teams use slow them down as opposed to helping them assemble applications efficiently!* Let me distill the reasons behind this phenomenon.

Organizations form infrastructure teams with the intent to help feature teams tackle everyday needs. However, the fundamental flaw of this arrangement is in the relationship between these groups. Ideally, feature and platform teams are supposed to be customers and suppliers, respectively. In reality, feature teams end up conforming to what the platform team dictates due to leadership’s belief that the platform is their salvation.

At the beginning of this intriguing journey, things go smoothly since the infrastructure team operates based on the initial problem statement of feature teams. As time passes, the needs of feature teams change based on evolving business requirements, but the platform keeps marching in the old direction. As a result, the platform’s interface and the expectations of

CHAPTER 2 CROSS-CUTTING CONCERNs

feature teams diverge. To adapt to the platform's interface, feature teams start introducing translations from business into infrastructure language or adding customizations via unnecessary abstractions and intricate configurations. In other words, feature development incurs a technical overhead due to the platform's inconvenience. There is a conflict of interest, and *the platform wins* because it has received so much trust from leadership. Guess who loses—*customers*, as delivering solutions to them takes longer due to the mentioned overhead.

I acknowledge that many organizations still need to build platforms, so eliminating infrastructure teams is not an option. Instead, I want to provide a recommendation for correctly forming platform groups to prevent the problems described here.

Avoid composing siloed platform teams. Instead, establish a partnership between the feature teams and the infrastructure group so that the former is the latter's customer. If possible, members from feature teams must rotate into and out of the platform group to build the infrastructure and gain domain knowledge simultaneously.

Thus, feature teams will feed the platform group and its requirements, and not the other way around. Furthermore, do not force platform adoption among feature teams—this acceptance must happen by itself if there is a need. If the platform becomes useless, throw it away without much hesitation, or rebuild it to be more suitable for efficient feature development; remember, most of the value is still with feature teams.

This approach improves the effectiveness and satisfaction of feature teams with a given platform, and it also increases ROI for platform teams. In some cases, organizations may not even need to maintain a dedicated infrastructure team because it can be sourced from other groups, and this factor will affect costs positively.

Lastly, keep in mind that infrastructure teams are shared resources out of the box. As I emphasized earlier, independent teams deliver maximum ROI due to their autonomy and scale. Since platform teams represent a dependency from other groups' standpoints, they can become bottlenecks within organizations. This precaution is another reason for driving the platform's roadmap by the feature team's needs, as suggested in this section.

Forming Programs, Projects, Requirements, and Deliveries

In many IT organizations, requirements come in weird forms, sizes, and boundaries; they cut through many random systems and subsystems. Therefore, no single team can handle them without significant disruption or ad-hoc project management. It often seems that the way the business is structured is entirely different from how the applications are built. In such environments, implementing medium- or high-complexity requirements is difficult, slow, error-prone, and hard to track due to conflicts of interests, number of dependencies, and lack of overall clarity. It is not a surprise that, as an outcome, organization-wide costs will increase, with more defects, longer development times, and decreased satisfaction due to unmet user expectations.

Therefore, after the context map with subdomains and bounded contexts has been fleshed out, ensure that the requirements management ecosystem follows the same shapes. Form portfolios, programs, projects, and low-level requirements so that each fits within a single boundary, no matter what level it is in a context map. When breaking down into smaller delivery units (e.g., from programs into projects), make sure that each fits within a lower-level bounded context (see Figure 2-4). Track the requirements' flow and their implementation in the same terms with which the context map speaks.

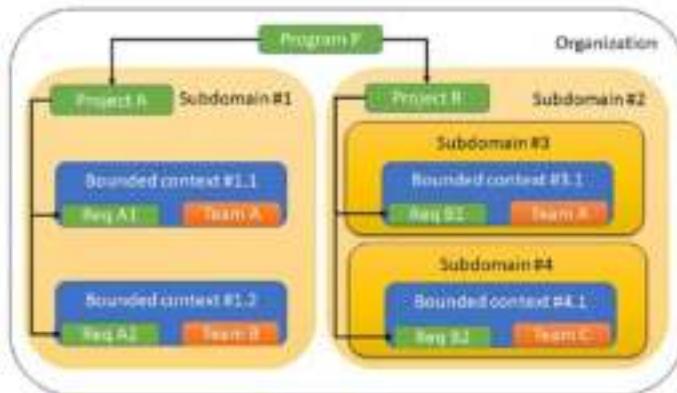


Figure 2-4. Forming programs, projects, and requirements around organization structure

When you follow the boundaries identified by the overall context map, your project management organization will change accordingly. Chasing down interlocked systems and implementing integrations becomes easy. Also, figuring out the system-level architecture becomes straightforward since the requirements will be flowing in the top-down direction from any particular bounded context's perspective. As a result, you will get simplified project management, simplified and transparent system-level architectures, and better knowledge flow within each feature team (since now they understand what all the requirements are about). This outcome, in turn, means that you have produced scalable processes, fewer defects due to fewer unknowns and lower learning curve, increased ROI, and a better chance of meeting the user's expectations since you can focus on it better now.

Organizational Silos

We all know that working in a silo is not ideal. In this section, I will look at the issue from the perspective of optimal software engineering processes and teams.

Horizontal Silos

Horizontal organizational silos can happen between management and their direct reports, at any level in the enterprise hierarchy. The differences in interests and goals cause this disconnect, and it is prevalent in old-style, conventional environments where the subdomains are not formally defined and followed. When such an issue occurs, teams can neither understand the manager's needs nor deliver to the expectations. Many times, the manager wants what is best for the company, and in some cases, surprisingly enough, the manager's focus needs to be adjusted. Irrespective of the actual root cause, the ROI per team is decreased, the unnecessary complexity is present in the process, and meeting the customer's expectations is at risk.

Therefore, make the context map a driving factor for the entire organization's hierarchy. Each manager and their team should fit within a single subdomain, regardless of its depth on a context map. The manager's goal must be to deliver based on owning all of their subdomain's needs instead of only focusing on people management. Emphasize that the subdomain's expectations are of the highest priority because it defines the business problem that the team solves.

This strategic alignment under the context map and its specific subdomain should slowly dissolve and eventually eliminate horizontal silos. As a result, the organization will become more agile and adjust under an overall enterprise roadmap. Furthermore, this approach increases ROI and chances to be successful in meeting the customer's expectations.

Vertical Silos

Vertical silos are usually formed between different specialties and are caused by sub-organizations' covering each of the practices separately. For example, in many companies, there are project management, quality assurance, development, and product organizations. This structure is a

CHAPTER 2 CROSS-CUTTING CONCERNS

critical issue because people from these various institutions form *teams* that need to deliver a *single product*, while they have conflicts of interests between each other. A clear example is “a tester vs. a developer” dilemma when the former is awarded for finding defects while the latter is criticized for the same thing. They are often funded from separate pockets, which further increases the disconnect due to office politics (e.g., a development manager may direct the engineers to avoid helping testers because the developers are “not paid for it”). This conflict degrades the quality of the product because it affects teamwork and employee satisfaction.

So, how do the context map and the subdomains help solve the described problem?

Specialty-based sub-organizations are not subdomains based on already established definitions, because there is no single business problem that each of them solves. Therefore, such institutions should be abandoned and avoided at all times. Instead, all team members within each group must belong to a single assembly—the subdomain they serve. If you need to reach an objective of an experience and knowledge sharing between people of similar specialties, opt into interest-based groups and meetups instead of the obsolete specialty-based sub-organizations.

When every member of each team shares a governing institution, the structure endorses a concept of a *shared goal*, and thus people working in groups deliver better results together. This approach will increase ROI for the entire enterprise and eventually will help meet the customer's expectations with higher-quality delivered products.

Do not interpret my suggestion as a call to deprecate all vertical organizations in a company. For example, HR and accounting will still have to exist within an enterprise, which is fine because members of those sub-organizations are not going to interact and work with the feature teams. As long as vertical silos do not intersect *groups*, there is no problem with them. In other words, every team should fully fit within one and only one parent sub-organization or a subdomain. Furthermore, it will be

helpful if HR, accounting, and other similar institutions are considered to be subdomains. After all, they solve problems that are important to the business and the proper functioning of an enterprise.

Autonomy vs. Reuse

Classic, conventional approaches to an enterprise, systems, and applications architecture suggest *unification* of the building blocks regularly so that we can extract reusable components and leverage them as often as possible. While this is a straightforward technique when described verbally, it requires dedicated effort and consideration that can noticeably slow down and complicate the architecture and development processes. Also, in the end, there is not much benefit gained. For most parts of the organization, leveraging a reusable component means either sacrificing their unique needs or incurring overhead to customize the element per their requirements. Remember, no single interface exists that thoroughly serves all kinds of problems that the enterprise aims to deal with, just like one size will not fit all perfectly. If there were a framework or an architecture that solved everybody's needs, we would all know about it by now, don't you agree?

Therefore, instead of focusing on unification and reusability, focus on autonomy and loose coupling. Independent subdomains, bounded contexts, subsystems, and teams have excellent potential for fast growth and breakthrough, so try to get the most out of this opportunity. Furthermore, encourage tailored and close-to-ideal solutions for each subdomain and bounded context since with autonomy you have such luxury without high costs of conforming. Integrate subsystems and teams with lightweight services and processes by maintaining the preferred loose coupling.

When you achieve autonomy, teams can accelerate when they have such an opportunity without the need to conform to the entire enterprise. Also, the process and software systems become more scalable due to loose coupling and clearly defined integration contracts. Acceleration,

scalability, and simpler processes, in turn, will increase the ROI for running the business. For example, Amazon has practiced this approach internally for many years quite successfully (see [APIE AIAP1]).

The choice between autonomy and reuse is a well-known dilemma in software development and architecture. If you pay close attention to the technology trends, you might notice that independence is taking over reusability year by year. Examples are NoSQL databases (schemaless data duplication over relational constraints and data reuse) and microservices (independent and loosely coupled mini-applications over tightly coupled monoliths). So, autonomy means the future! No surprise that I go for it too.

Let's see how the processes, teams, and everyday work approaches fit into the organizations I've just described.

Processes, Ongoing Efforts, Teams

How does an engineering process affect effective outcomes? What improvements can we apply to this area? We are about to find out!

Agile or Waterfall

While everything in this book can apply to both Agile and Waterfall style development processes, I recommend that you give preference to iterative processes (e.g., Agile). A repetitive approach simplifies adopting the practices from this book, especially higher-level ones that directly influence software development phases. Besides, an agile process delivers additional benefits, such as the following:

- You get to repeatedly apply the same techniques, and thus you get better at them, instead of trying every single thing only once throughout the development lifecycle.

- Defects are discovered in smaller chunks. We can take both corrective and preventive actions more easily, instead of getting stuck in an endless battle between quality and release date (in which case release date always wins, and customers get a low-quality product).
- You can adjust a course of development between iterations instead of continuing to invest in a product that may not be on track to meet the customer's expectations already.

Therefore, to make effective software feasible, always prefer following an iterative process over a Waterfall process.

If you do not have a choice and have to follow a less flexible process such as a Waterfall, be aware that there is still hope. Here are a couple of things to start with:

- To successfully apply techniques from this book, seriously consider having an expert on board and follow the rigorous guidelines set forth.
- Do not give up on the methods that may seem only for Agile at first glance; e.g., practices originating from XP (extreme programming), such as TDD (test driven development) and continuous integration, still can be followed in a Waterfall.

While I will not explicitly differentiate practices that only apply to Agile, you should always try to use every exercise in any process to some degree. Before giving up on any technique, challenge yourself by attempting to apply it in your current process.

Transparent Teams

I have often been on teams where management did not have enough visibility into what the team was doing. They hired project managers and scrum masters to gain control of the situation. In some cases, it turned out that the team members were working hard, and their work was not recognized. In other cases, unfortunately, management discovered that the group was not productive enough since nobody had the right instruments to uncover it.

When there are problems with transparency, it costs money, resources, productivity, deliveries, and quality. If you hire a project manager for micro-management purposes, it costs you one person's salary. If you do nothing, it may cost you more.

Therefore, form teams that are transparent out of the box. Make this a clear expectation and keep all team members accountable for this crucial measurement. Establish instruments to maintain a high transparency level and monitor for changes. Development team leaders should be selected based on these criteria, and they should ensure that the same morale is in place within their groups. Also, ensure that the leadership is transparent too—there is nothing more potent than an example coming from upper management.

As an example of the suggested measurement, keep an eye on the late surprises, such as missed deadlines, increased critical bugs, and too much incomplete carried-over work in the current iteration. In simple terms, bad news—do you hear about them from the team? Is the information reaching management too late to act? Do people let themselves be blocked? A transparent team communicates the problems and asks for timely help, provides mitigation options, and strives to improve. If you see such positive signs, you are on the right track.

When transparency is valued and maintained at high levels, you gain control over the quality of the enterprise-wide deliverables. IT organization is a powerful instrument if it is appropriately structured, and

you have access and visibility into how it functions and how it is driven. *Transparency puts you in the driver's seat* and lets you maximize your ROI per developer, team, and software.

Managing Work in Progress

One of the most critical contributors toward the low quality of software, slow development pace, and missing of customers' expectations is context switching. Humans are not good at multitasking, although many people claim to be gifted with these abilities (this explains why text-and-drive is regularly attempted).

Let us look at some situational questions to understand the problem behind context switching:

- If a single engineer works on several cars in the factory at the same time, do you agree that this raises the risk of mistakes in each of them?

Not sure? Let me try another one:

- Do you feel safe when your dentist works with another patient too while working with you, and switches back and forth between you two?

I am sure that most of us would want to avoid such situations.

Similarly, when a software engineer works on several requirements simultaneously, there is a risk of mistakes' being made in each of the efforts. If the organization uses this same principle throughout the entire software development lifecycle, you have hidden risks not only in development but also in planning, architecture, testing, deployment, maintenance, and so on.

Therefore, enforce the rule that every engineer is supposed to work on one assignment at a time.

When engineers understand the importance of limiting the size of work in progress, you minimize waste, context switching, overhead, and the chances of making a mistake. If they are blocked with the assignment that they have started, prioritize unblocking them. Always let them finish the started work instead of switching to the next item. For example, if a programmer is waiting for a coworker for the code review, make it a rule that the code reviews are the high-priority tasks for each team member so that they unblock others first.

One for All, All for One

When we work in iterations or milestones (which is relevant to Waterfall too), we expect *concrete results*, and not just work in progress. Individuals and teams are often put under pressure to deliver more within any time interval. Groups accomplish this by working on many parallel features at the same time. As a result, when you look inside a single team (e.g., sprint backlog), you see many work items in progress. The overall picture is distorted, tracking the increments is hard, and forecasts are meaningless. Without predictable, well-defined processes, you risk too much by not knowing enough about the future.

Besides the mentioned disadvantages, due to their chaotic elements, the situations just described result in low employee engagement, lousy team play, and ineffective knowledge flow, which again negatively affects business outcomes.

Therefore, set up teams so that they work on each requirement together. Shift the focus from having more things in progress to delivering at least something. Encourage group members to help each other at all costs. Prefer to demonstrate one completed feature instead of having many of them in progress by the end of the milestone to eliminate possible waste, accelerate the feedback loop, and simplify forecasting future performance.

This technique is the basis for building a highly organized technical team where everybody works for the same goal and is ready to go the extra mile to deliver a high-quality product. Such environments also improve knowledge and skill share among the group members because everybody gets to help.

Transformation from Inside

Many good ideas, techniques, and approaches do not find their way into software development teams due to the technical complexity that accompanies them.

For example, I have often been on teams where engineers claimed that writing unit tests would slow them down and decrease ROI. I have also heard many times that implementing a full-stack story (i.e., one that includes both back-end and front-end tasks) is technically impossible within a single iteration. Agile and other improvement-based processes suggest both of these techniques, but not too many people know how to accomplish them. Thus, the teams do not practice them often, which negatively impacts the software quality and deliverables.

When there is pushback from developers, how can a non-technical person execute the positive change? As it turns out, you only need a working example.

Find, recruit, and retain influential engineers who can support the technical side's process changes. Let that talent lead by example and demonstrate to their peers that the technical challenges can be overcome by showing and explaining the concrete and practical ways to do that. Have at least one such designated person per team and watch how the resistance to making positive changes decreases. Keep the experts accountable for outcomes.

CHAPTER 2 CROSS-CUTTING CONCERNS

For example, in the situations described earlier, an experienced engineer could explain how the unit tests benefit her by accelerating instead of slowing down. It is also possible to showcase that developing full-stack stories is technically feasible and helps with the quality of deliverables.

I took this technique from my experience, so I know it works. I have played the described supporting role many times with excellent results. Often, my teams only needed a single example to start practicing something exciting.

Such transformations don't happen in a vacuum; they need a supportive culture.

Culture

We have reached a section of the book that I have debated with myself about many times because it is about people, behaviors, and culture. When I speak about these matters, it is easy to feel slightly uncomfortable because I'm walking on a thin line connecting the emotional with the adequate. Therefore, I want to warn you that you may find topics that are not often discussed. What can I do; if I am after such a goal as improving everything, I will also have to uncover some unspoken truths.

Professionalism in Software Development

I once wrote an article about everything that I thought had gone wrong in the software development industry (see [Tengiz WWWSD]). At the time, I did not quite understand what I was trying to achieve with it. A couple of months later, a colleague of mine sent me a link to a recording of a fascinating talk by Robert Martin about "Professional Software Development" (see [Bob PSD]). When I watched the video, I finally realized that I had been complaining about a shortage of *professionalism* in software development.

It is hard to define *professionalism* in software development because programming, unlike other fields such as health care, does not qualify as a *profession* that would have a rigorous set of procedures and rules. Professionalism, if you think about it, is a procedure. If you do not follow procedures in development, the impact is not apparent until later in the game, and thus best practices and recommendations are often prematurely dismissed. On the other hand, if you make a mistake in health care, results may be life-threatening, and thus procedures are strictly followed. Software engineering is not there yet. It will take a while before programming turns into a real profession from an activity with no rules or responsibility. The book in front of you, like a few other publications, is an attempt to course-correct the situation.

Let us start by defining *professionalism* in our industry using simple words so that it is accessible to everybody, and we all can follow it:

Professionalism is when you do what you claim to be doing in accordance with the standards and ethics of the profession.

If we claim to be producing software that works, then the resulting program *should work*. If we insist that our system is horizontally scalable, then it should support more users by merely multiplying application instances.

I am not asking you to be a tyrant, but you need to keep people accountable for their promises. We cannot continue accepting code that is buggy, half-working, and causes nothing but frustration to its users. To fix the situation, we need to set a specific course of action in the organization's roots.

Encourage people to be professionals and take accountability for their actions as well as the outcomes of those actions. Spread the word about this cultural shift and adjust expectations throughout the organization. Recruit new hires accordingly.

I am lucky to have been part of companies where this cultural shift took place. This change positively affected the organization's financial results and the growth and engagement of people within it. Delivered software was of higher quality, and customer satisfaction grew as a result.

Without a mandate for being professional, you risk turning your business into a joke, decreasing end-user satisfaction and losing to your competition.

Trust

At times, the team's transparency is not at the right level, and management doubts whether productivity is at its max. When that happens, leadership starts enforcing micro-management strategies such as the following:

- Breaking down assigned work into the smallest possible pieces
- Verifying the accuracy of estimates down to minutes
- Double-checking that everybody has taken assignments to fill up exactly 40 hours per week (not leaving a minute to go to the restroom)

While I understand the concern of management, such approaches have never helped. Instead, the additional overhead of micro-management always causes unnecessary pressure and complications for both teams and supervisors. With that comes an increase in cost and risks for mistakes, which eventually affects development pace and even customer satisfaction.

Therefore, establish trust with the team. Compile a list of committed work at the beginning of an iteration or a milestone (which is typically accomplished via the dedicated planning sessions). At the end of it, come back and check the status. Let team members manage their time in between. Increase transparency and let the group feel safe to speak up when there is a risk of not meeting deadlines.

When there is trust between management and the team, transparency will increase naturally, which will allow an organization to avoid late surprises of missed timelines. Such a friendly environment increases employee satisfaction and the feeling of being appreciated for contributions, which positively affects the quality of delivered work.

Delegation of Responsibilities

Without delegating responsibilities, the process cannot scale, and we risk making more mistakes or slowing down due to overhead. For example, if leadership needs to micro-manage every single task, they cannot be freed up to focus on higher-level strategic decisions. So, we need to solve this critical and often underestimated issue.

When it comes to delegating responsibilities (such as a manager assigning a mission-critical task to a direct report), we must admit that not everybody can do this job. Some employees never talk even if they have an essential point to raise. Others never argue if you ask them to do the wrong thing. These behaviors are caused by differences in skills, experience, background, past, or even cultural origins. Irrespective of any reasons, we cannot expect everybody to be reliable enough to execute a given mission. At the same time, some people enjoy working on a daily routine without asking for more responsibilities, and we need them too.

Therefore, seek reliability as a personal characteristic. Recognize team members for having this attribute and use it as a measurement for performance assessment. Watch how this key indicator improves over time as all team members come to understand its importance. When there is reliability, delegated responsibilities will be fulfilled without sacrificing quality, cost, and customer satisfaction. When you identify team members who stand out with the highest degree of dependability, give them kudos and better opportunities. Keep designated team members accountable for their promises and assignments. If this goes well, leadership will be free to focus on higher-level strategic decisions while delegating as necessary.

When delegation is not a risk or a hassle, the entire organization becomes more mature, agile, and easy to control. The process becomes scalable, and upper management can make strategic decisions with the expectation that the organization underneath will follow and transform accordingly. This flexibility can turn into a business advantage over the competition, so never underestimate it.

Identifying Talent

It is an unfortunate phenomenon that outstanding software engineers are in the minority because the list of skills to master is not short. Once you become a senior developer, then you start seeing shortcomings in programs that others *rarely* notice, *because you are in the minority*. This gap creates a conflict of interest, disagreement, frustration, and stress, not to mention apathy between coworkers and destroyed team spirits at times. While it is not a comfortable journey for these unique individuals, what does an organization have to do to turn such situations from a weakness into an advantage?

A famous quote of Steve Jobs answers the question very well:

*"It doesn't make sense to hire smart people and tell them what to do; we hire smart people so **they** can tell us what to do."*
(emphasis added)

Therefore, recognize the minority and find ways to understand whether you are dealing with a skilled engineer or just a strange personality—the two can look similar. It does not make sense to have a strong talent fight endless small fights if she could be an asset for more significant wins. Give this person the right to improve things and monitor progress. Always keep them accountable for their promises and actions.

I have personally worked in environments where talent was not recognized and empowered. While this did not make strong engineers very happy, it also worked against the company's interests. Meanwhile, when organizations recognized unique talent and put them in charge of improvements, the quality of deliveries increased, and a positive cultural shift occurred since this was a clear example and an encouragement for others.

This technique is a low-hanging fruit, a hidden gem that leadership often overlooks. Do not get into the mentioned traps—make the most of your talent.

Speaking of talent, while I highlight the senior-level engineers in this section, I also want to mention that junior-level programmers are another talent you have that you need to nurture and grow into the professionals I asked you to seek. You can source your more robust engineering needs from within, which is another win-win formula.

Relaxed Team Atmosphere

The combination of stress and pressure is the first reason that people stop liking their jobs. Often, unhealthy conditions result from management's being too cautious about time spent "not working but talking." When put under pressure, group members can hesitate to discuss even technically essential details or ask questions when unsure about the best approach to solve a specific problem. That is how stressful conditions produce non-optimal solutions, raise the risk of mistakes and overhead to rewrite, and increase overall delivery cost. These issues also affect customer satisfaction if non-optimal or buggy code gets into their hands.

Therefore, recognize the need to bond within the team. Do not count only work hours and do not think of a casual conversation as a waste of the company's money. Instead, build a relaxed, healthy team environment that encourages open communication, working together for the same goal, respecting each other, and going the extra mile.

An excellent example of a more relaxed approach to the team's work is how Google encourages its engineers to spend 20 percent of their time on side projects (see [Inc GSP]). Such creative environments originate with the realization that people can connect and innovate together, even if they do not work on the same team but are given enough freedom to bond.

Work–Life Balance

While previously I directed leadership to relax their controlling behavior over developers, in the current section I want to encourage engineers to seek and implement work-life balance techniques. If you ignore this advice, you will quickly feel that you burn out faster than you grow, and your job will stop being attractive. Such imbalance impacts both personal and business outcomes, so it is worth covering a couple of relevant recommendations.

Although “work-life balance” may sound like a relaxation technique, in my mind it goes multiple ways: balancing work, balancing life, and balancing the ratio of both elements.

An example of *work balance* is employing techniques that optimize your focus and productivity, such as Pomodoro Technique (see [Wiki PT]). The basic idea behind this approach is taking regular quick breaks between continuous work efforts, which maximizes the amount of work done throughout a day.

An example of *life balance* is putting away your work laptop during lunch break and enjoying a mindful conversation with a colleague or a spouse. Ignoring this simple recommendation can create a feeling of stressful work, which is entirely avoidable.

The technique of *balancing work and life* refers to a controlled approach where you plan your day or week to deliver your professional commitments (i.e., balanced work) while also leading a fulfilling personal life (i.e., balanced life).

Candid Feedback

Compliance, diversity, and inclusion are essential measures to organizations. When interpreted inaccurately, these metrics can negatively influence how people provide and perceive candid feedback. While afraid to sound offensive or unfriendly to others, many individuals often keep quiet even when it is crucial to speak.

For example, here is one situation that I witnessed: A remote team member showcased newly developed functionality from his *large* computer monitor. When streamed to a smaller screen, the picture became so tiny that we could not see every UI element. Attendees could not understand whether the feature was following the original requirement. It was only a matter of asking the presenter to adjust the screen size, but nobody dared to do so while trying to be polite and kind. Instead, the feature was accepted as-is by stakeholders. Later, those same attendees discovered several defects in earlier demonstrated capability because it turned out different than it sounded during the demo. This hesitation to provide immediate feedback delayed feature deployment to a production environment and added overhead of showcasing it again after a fix was implemented. I bet the company CEO would not want that to happen again because this episode was caused by a mere misunderstanding of the company's compliance policies.

Therefore, encourage everybody to provide candid feedback when necessary; the earlier, the better. Educate people that this input must be treated as helpful advice to improve software quality and has nothing to do with how we treat each other.

Early feedback is a powerful instrument to increase customer satisfaction and deliver better-quality software. This technique is similar to asking customers ahead of time whether a planned product would be helpful and useful to them, instead of working on something that was not going to be used at all and would be a waste of time and money.

Change of Mind

The best solutions are designed in open-ended conversations. However, due to job insecurity or personality, many people approach discussions with a mindset to push their ideas and not accept any alternatives. When that happens, we witness a one-sided conversation where the outcome is decided upfront, shutting down the possibility of innovation and further

CHAPTER 2 CROSS-CUTTING CONCERNS

improvements. It is in the organization's best interest to promote inclusion and ensure that *objectively* the best idea wins since that matters from a customer's satisfaction standpoint.

Therefore, approach every dialogue with keenness to change your mind. Be prepared to listen and understand every opponent's thoughts before declining it. Who knows, maybe you are about to hear the next big thing.

It is not a change of mind that I am asking for; it is *preparedness* for it that makes a difference because you are open-mindedly considering every possible option. If your idea is the best, it will win an argument no matter what. You can take something more from a conversation treated as a dialogue and not just a debate.

When an organization employs these described techniques systematically, people feel more appreciated and heard. The overall result improves, and customer satisfaction increases.

Social Aspect of Engineering

Four decades ago, software development looked quite different than it is today:

- A single engineer could craft an entire (small in today's measures) website.
- Everybody had to use low-level programming languages with machine instructions, so there was not much to express or read in code.

Things have changed since then:

- Several teams often work on various parts of a large-scale solution.
- High-level programming languages have emerged that allow writing of human-readable code blocks.

This evolution indicates that expectations, tools, and organizational structures have matured for solving more intricate business problems. Nevertheless, based on multiple case studies (e.g., see [RG CCS]), one of the primary reasons for failing software projects is nothing other than *complexity*. To explain this phenomenon, let me try to distill what complicates software solutions.

As long as the program repeats the corresponding business domain's concepts and shapes, naturally its complexity must not surpass the intricacy of the problem space itself. Therefore, if a solution has become more complicated than the domain it serves, perhaps the connection between them is missing. How do we establish a link between these elements? The answer is straightforward—via *communication* between domain experts and engineers! If this social aspect is absent, developers will base solutions on technical terms that have no value to domain. As two areas continue to diverge, the technical complexity of implementations increases, eventually leading to failed projects.

Therefore, add a social aspect to the development process—encourage conversations between engineers, product owners, analysts, and stakeholders. Software that solves business problems does not have to be complicated and awkward but instead must follow a shared business language. Set this as an expectation. Hire experts, if necessary, to demonstrate techniques for tackling complexity.

I will explain the essence of shared business language in more detail in later chapters and refer you to specific reading materials. For now, let us consider that definition to be out of the scope of this section.

As communication between technical and non-technical sides improves, software solutions will adhere to shapes of a business problem. Consequently, you will avoid dealing with unnecessary added complexity in programs, allowing you to accelerate when others typically slow down.

Complexity as Job Safety

Many engineers believe that their services will be unnecessary if software development becomes more straightforward or accessible to non-technical people. This thought is written in every engineer's mind on a subconscious level. When I try to pitch techniques to decrease complexity, some developers openly complain that I am trying to make their jobs unnecessary. They claim that implementing software the way I suggest requires less-advanced computer science skills. I disagree with that statement because of two reasons:

1. If engineers do not need to solve one kind of problem, they will finally have enough time to address other issues requiring attention.
Specifically, skills and techniques to decrease complexity and increase maintainability are far less known than those to develop low-level complex code blocks. Furthermore, these skills are not easy to master and are far more rewarding. Therefore, my approach is not a threat to job safety but rather is an opportunity to go in a new direction.
2. Even if something else (e.g., AI) replaces our jobs, there are still plenty of things to do in this world.
I love my job, but I can not resist technological breakthroughs. Either we transform and pave the road, or others do it, and we conform to them anyway.

While introducing an artificial and unnecessary complexity can give engineers a feeling of safety, it increases the cost of development and thus shrinks ROI. Also, more complexity means more chances for defects and more difficulty in achieving customer satisfaction, not to mention more overhead. It adds to other key indicators, such as an ability to scale, too.

Therefore, beware of engineers who introduce complexity without necessity. Find people interested in building systems that require minimum or no developer's intervention to work without defects and scale as an enterprise grows. Do a sober judgment of decisions that come from technical personnel by asking questions about future costs and maintainability constraints. Always prefer to develop applications that are self-sufficient as it will significantly decrease price in the long run.

Notice that with this section, I am not suggesting you avoid complex implementations entirely. Instead, I recommend avoiding *unnecessary* complications that could add overhead to a business or process. While a non-technical person might prefer to build the simplest thing possible at all times, the proper choice must address long-term quality expectations such as maintainability, extensibility, and scalability. This statement clarifies why this book's recommendations sometimes seem complicated, but they help build simple-to-use and beneficial solutions to business needs.

Team Spirit

We often underestimate the importance of building a real team spirit. When a group works together openly, transparently, and in a highly collaborative fashion, they solve problems faster, produce higher-quality software, and increase engagement. They make better decisions due to inclusion. Little things that seem unimportant can derail team spirit and amplify costs and consequences. So, we must be sensitive to every tiny element and work on it to improve. Here are a couple of examples that can disrupt team spirit to give you an idea of patterns to monitor.

1. Engineers are sitting wearing headphones and not hearing when you say their name: If I (developer) need to walk to somebody's desk and wave my hands before I am allowed to ask a question,

CHAPTER 2 CROSS-CUTTING CONCERNS

it affects my comfort zone. I will soon stop doing so and start making assumptions. Some assumptions will turn out right, and *some will be wrong* and will cause overhead. It is hard to deal with personalities that are not social. Not everybody feels comfortable asking other team members to stop listening to their music for a moment. I can do it once a day, but not once every hour. Situations like this hinder open communication in a team.

2. Designated team leads or architects are trying to play an "ivory tower" role. While they have an important role, these people need to follow their responsibilities. They must present their thoughts and open up a discussion to ensure that they have covered all scenarios without missing critical use cases. When "ivory tower" is present in the environment, developers (especially those with opinions and skills) may feel unappreciated and excluded, which ruins many benefits of teamwork.
3. Developers often prefer to sit in their cubicles instead of sharing an open space. I understand that this is a pain point for many people, but if we all sit behind our walls, we will be individual contributors and not a *team*. Members of a single group must see and hear each other so that they readily share knowledge and experience.

Therefore, monitor and study teams and their behaviors. Identify and resolve issues that negatively impact team spirit. Do not underestimate the importance of little things. The rule of thumb is to have a cozy, open, and transparent atmosphere in the team.

Keep It Fun

Having fun at work is a vital part of engaged, satisfied, and dedicated personnel. Without this element, our daily duties become “just a job” that lasts “from eight to five.” When the excitement is absent, talent turnover increases, domain knowledge of teams lowers, and the quality of delivered solutions drops consequently.

Therefore, take care of engineering culture and keep it fun. The book in front of you stands on an idea of perfectionism, but remember that it is up to you how far you want to take it. Some teams strive to be their best, and perhaps they will gladly consider every technique from this publication. Other groups will need to consciously choose what they want to tackle to maintain morale and positivity. No matter what I ask you to do, do not let it ruin your organization’s fun.

One exercise that is an example of engineering excellence but requires careful consideration before practicing is pair programming. If implemented dogmatically, pairing among developers will create unnecessary tension. I have seen environments where companies ask their employees to follow this technique at all times, regardless of the usefulness or relevance of pairing. You need to realize that people need personal time, and asking them to spend an entire day talking to a colleague without a break might be too much!

An alternate way to practice pair programming and keep having fun is to do it on-demand (when somebody asks for it) or within a required threshold (e.g., five hours per week).

I enjoy pair programming as long as I know when it will happen and how long it will take. I believe that such a compromise can be a golden medium between constant pairing and forgoing this exercise altogether.

Culture is people and exists for people, so now we will discuss how to bring the right people into the organization.

Recruitment

In previous sections, I covered expectations that need to be met by managers, engineers, and other members of IT organizations. However, the efforts of all these people will have little or no effect if recruitment does not control the talent flow into the enterprise. That is the aspect I will be covering in this section.

Supporting Role of Recruitment

The hiring process is an essential element that decides who gets into the company. This mechanism significantly impacts overall organizational success because acquired talent creates products that get into the customer's hands.

It does not make sense to keep fixing what we have at the end if there is a leakage at the entry point—problems will not decrease, and our solution will not have any long-lasting effect. As a result, we cannot gain full control over the quality of deliverables, and customer satisfaction may suffer.

Therefore, recruitment must be considered a strategic process for acquiring the kind of talent that the rest of the organization desires. Create customer-supplier partnerships between recruitment and other organizational operations and divisions, thus ensuring that talent flow from outside meets demands inside the company.

Once this strategic partnership is established, you can further focus on improvements within the engineering organization. When there is a need for new hiring, this pre-established relationship with recruitment will ensure that you can meet the demand.

It is impossible to list in advance all talent qualities you want to look for, but it will happen throughout this book. As I will be addressing specific techniques for developing effective software, I will define the particular needs for talent. In the current section, I only wanted to ensure that recruitment is ready to work for other divisions of an organization when their demands arise.

Hire Best

Our subconscious belief says that the world is not perfect, and all competent engineers already have their dream jobs. It is also an unfortunate reality that companies hire people solely based on quantity rather than quality. Even though headcount math works against me in this context, I am confident that a couple of talented engineers can deliver more within any timeframe than a dozen incompetent developers. It is a mistake to give up on finding the best talent and settle for average because this decision will be reflected in costs very soon. Delivered software will have more defects, it will turn out to be unmaintainable, and, most important, our customers will notice the difference.

Therefore, look for and hire the best talent—those who take software development as part of their personality and not just a job, who want to produce the best programs possible and never settle for less. People with such an attitude can turn engineering organizations into well-performing, high-profit institutions that correctly solve business needs and meet or exceed customer expectations with high-quality deliverables.

Just give it a try, and you will see that such an investment decreases your overall costs due to higher satisfaction rates both within and outside the company.

Quickly, Fancy, Right

Finding the right engineering expertise is the first step in the puzzle of optimizing an engineering organization's performance. Often, delivered solutions seem to be what we wanted, but we still see improvement opportunities. If we already have a good talent pool, then what is wrong?

It turns out that strong engineers can be further categorized due to differences in personalities, attitudes, skills, and willingness to adapt to change. When we understand the details behind this intricacy, we can adequately manage human resources as well as plan workload and

CHAPTER 2 CROSS-CUTTING CONCERNS

decision-making distribution among employees. Otherwise, while the right talent is present, we are not leveraging it in the best way possible, which will negatively impact the quality of delivered solutions and thus customer satisfaction too.

Here is a classification of well-performing engineers that can help with making better hiring decisions:

1. *Do work quickly.* These engineers are lightning-fast, and they believe that their skills improve with each feature that they develop. However, they usually miss the point that after a certain number of lines of code, "how" matters more than "what." In short, this group of people delivers a lot with mediocre quality (quantity over quality). Eventually, produced software starts breaking due to increased overhead of complexity and bugs; an alternative approach's importance becomes apparent.
2. *Do work fancy.* These engineers always rush to use modern frameworks and technologies, regardless of whether it is the right thing to do for the system at hand. Every tool that does not fit a job well adds to the overhead of maintenance, increases costs, and eventually uncovers a flaw in these developers' approach. It seems these people are not driven by the best interest of the software, but rather by the best interest of their career. Maybe that is why some people jokingly call this approach a "resume-driven development."

3. *Do work right.* These engineers start with the challenge at hand and make it the center of their decisions. Their solutions naturally fit business needs without incurring unnecessary overhead, by adjusting technology for the problem and not the other way around.

Therefore, when recruiting or interviewing new talent and candidates, seek attitudes described in the third group. Consider other groups as force multipliers if necessary. Make sure that decision levers are in the hands of the right folks. As always, keep people accountable for their solutions.

Corrective vs. Preventive

One of the main reasons for the high number of defects in every software project is the lack of proper care to prevent them from repeating. This phenomenon occurs because developers often want to fix the issue and move to the next one immediately. This approach seems reasonable at first sight as they are trying to minimize cost; i.e., the number of hours spent solving defects and not implementing new features. However, because of rushing, engineers often do not put proper solutions in place, and issues repeat after some time. Eventually, teams develop more features and patch more repeatable defects; at some point, the number of open bugs amplifies (similar to complex interest formula). Ultimately, fixing repeating errors that pop up at various places will surpass time spent developing new features.

Fortunately, some engineers understand this, and they properly care for defects by fixing roots rather than scratching the surface. Having at least one such person on each team is an absolute necessity to decrease the number of bugs and increase ROI per developed software.

Therefore, when recruiting or interviewing new talent and candidates, look for those special ones who can prevent problems rather than patch them temporarily. Such engineers are rare but utterly essential. Make sure that these developers have a say in software quality and maintenance directions.

Summary

This long chapter proves that many cross-cutting matters affect the efficiency of software development and delivery. We looked at vital topics of leadership, organizational structure, processes, culture, and recruitment. I have laid a solid foundation to jump into the next chapter and see how software engineering organizations can turn customers' expectations into working products.

CHAPTER 3

From Customer Insights to Internal Requirements

How do you meet your customers' expectations without asking questions or listening to their needs? How do you avoid bugs without thoroughly analyzing requirements in the first place? How do you deliver results without proper planning?

These questions point at typical improvement opportunities within the area of requirements gathering, analysis, and planning. My experience tells me that there are no shortcuts around these activities.

The process of value creation usually starts with a dialogue between customers and non-technical representatives of an organization that fulfills clients' needs. When the requirements finally reach engineers, they will respond with questions, confusion, or a request for help. To make this cooperation effective, we must set certain criteria for requirements gathering to avoid the overhead of rework and communication.

Besides enhancing the process of upfront analysis, there are specific patterns and practices that I will suggest for non-technical activities to support engineering and architectural tasks. This dependency is another reason to talk about the presented topics.

Thus, we arrive at the ultimate goal of this chapter—enhancement of processes and practices that apply to requirements gathering, analysis, and planning. These recommendations will provide vital support for the successful delivery of ideal, working solutions to your customers. To turn the suggestions into reality, product management, related business units, and engineering leadership must be responsible for executing or supporting the needed improvements.

Let's start by understanding the customers' needs.

Understanding Customers' Needs

Here, we will focus on customers of a software development team or an organization. My goal is to help you identify end users' needs and to create and align the company's values accordingly.

First of all, let us agree on the terminology that I will use in this section and throughout the book.

In my vision, *end users* are *customers*; therefore, I use these words interchangeably. This group of people has interest in utilizing the software product that you produce because it solves their needs. Customers can be direct (your clients), indirect (your client pays you to develop software for their consumers), or external or internal to your organization. A customer may pay you a salary to build software, or they may compensate you for using a finished product that you provide. Irrespective of payment scheme and business relationship, it is crucial that you remember this distinction—customers are *end users* of the application. If somebody pays you to build an app, that does not make them customers unless they are also the program's end users. While this separation helps you understand the techniques described next, it also allows us to delineate between essential and unimportant aspects of software development.

Sure, your clients are essential to you if they pay for your services even if they do not use the final product. Still, this fact is irrelevant from software development standpoints such as quality, scalability, maintainability, and product viability, which are the focus of this book.

Again, this book will refer to end users as customers and try to meet their needs irrespective of who pays for software development. However, satisfying the expectations of *paying clients* is a different problem, not necessarily related to engineering practices as such but more to contract negotiation and business relationships. Understandably, these concerns fall outside of this book's scope.

Partnership and Customer Focus

Engineers often mistakenly think that domain experts (including business analysts and product owners) are customers. Therefore, the team focuses on meeting the expectations of SMEs (Subject Matter Experts) rather than fulfilling the needs of end users. While it may seem to be the same thing, there are slight differences that can be critical success factors for a developed product.

For example, I once asked an end user to provide feedback on an application that I had developed with my team. I was surprised to discover that the user only used half of the features of the software.

After silently watching the user's actions, I concluded that the program was not optimized for his daily tasks and caused him to perform many unnecessary actions to get to the result. If not for this visit, I would be under the impression that we had developed a state-of-the-art product that everybody enjoyed using, while that was far from the reality.

Such an outcome resulted from engineers' following requirements formulated by a product owner without a single thought that end users might want something else. We had effectively wasted our time and the company's money.

CHAPTER 3 FROM CUSTOMER INSIGHTS TO INTERNAL REQUIREMENTS

Therefore, domain experts are not customers, although they can be well qualified to represent customers' needs. Instead, SMEs are partners of engineering teams. I recommend establishing an equal, mutual relationship between domain experts and developers so that they can work together to achieve a common goal—satisfy customers by meeting their expectations.

When domain experts are not mistaken for customers, there is a better atmosphere for open collaboration between SMEs and engineers, which increases the chances of developing a better product for end users. If you have conquered this mental technique, then you are ready to take it one step further.

This confusion between domain experts and end users exists because development teams are often kept out of customer interviews and user experience research activities. Therefore, all they see are domain experts representing end users, and thus it is easy to confuse the two. If engineers do not understand real customers, they will build programs that are not for real ones, and therefore will fail to meet actual end users' expectations. This failure can negatively affect the company's reputation, financial performance, and overall success on the market.

Therefore, turn development teams into customer-focused groups that work for customers' needs and not only for the needs of the company. Allow team members to attend customer interviews, or at least to see recordings of research activities. Listen to their insights and seek hidden gems that domain experts might have missed. Everybody in the company (including developers and domain experts) will have their unique way of seeing and explaining the problem behind customers' feedback. Organizations need to leverage this diverse set of interpretations and approaches to compile a final list of requirements that optimally address expectations.

Teams that focus on customers' needs are more likely to deliver closer to actual expectations and with better quality, which will result in increased customer satisfaction in the long run.

Interview Customers

When we have a novel idea, it is tempting to jump into the software development process and build something amazing to see if it works out as we envision. One thing that we naturally underestimate is the cost of developing a program. If the final product turns out to be not so valuable to probable customers, we will regret all that expense for nothing. Besides failing to meet users' expectations, we will incur unnecessary costs for the organization.

Therefore, try stepping out of your comfort zone as early as possible. Speak to your customers and test the market (dedicated domain experts or other authorized individuals can do this). Determine what they want and when it will be applicable. Prioritize your work (backlog) accordingly, even if the order of tasks does not match a previously planned strategy. It is much cheaper to conduct customer interviews before full-fledged development starts; do not fall for the temptation of developing software first and testing the market afterward.

This advice can sound like going to the front line and exposing yourself instead of safely building something awesome; I get it. However, this technique is practical and doable and is considered normal by numerous modern companies. There are plenty of options to conduct customer interviews: you can hire a dedicated research company for that, pay probable customers for their candid feedback, post on public or focused forums, or even ask your friends. If you try this approach once, you will realize how much time and money it can save.

Although this section will seem applicable only to newly formed startups, it is common to see large organizations trapped by legacy processes. Company executives usually gather to guess (officially—*forecast*) market needs and fund projects for developing software systems of significant size and complexity. Such processes may work if you are lucky, but progress will be at a slower pace, and the rate of success will be low. The practice has shown (e.g., see [Ries LS]) that experimental and learning-based product development techniques show improvements faster than conventional approaches do.

CHAPTER 3 FROM CUSTOMER INSIGHTS TO INTERNAL REQUIREMENTS

Note that this section does not prescribe any specific form of customer interview, and it does not even have to be an *interview*; although I will be using this term throughout the book for clarity. I encourage you to research various ways of learning the market and then pick one that works for your unique needs. The point of this exercise is to gain knowledge *before* building a product and not the other way around (unless that is a part of the experiment for obtaining necessary insight).

Knowledge Exploration Exercises

Now that you have collected customers' needs or feedback, it is time to conduct *knowledge exploration exercises*; i.e., to digest learning and turn it into domain expertise. In simple words, that means SMEs within your company need to understand, live, and breathe end users' needs. This step is essential for the following reasons:

- **Constant access to customers is rare.** Usually, you collect learned insights or end user asks and act on them to implement software that meets demand. When you are done working, you need to take the product back to customers and see how well it performs. To be successful with this attempt, you need to have access to domain knowledge to guide you in the right direction, while instant customer engagement is limited. Knowledge exploration exercises solve this challenge by letting you house knowledge and domain expertise within your organization, closer to SMEs and development teams.
- **Building a business is on you.** Although end users are the most critical stakeholders of software, only SMEs within your organization can apply business elements

to customers' needs, which is essential for any for-profit enterprise (i.e., finding ways to make money). An unavoidable prerequisite for applying monetization to product is turning customer feedback and expectations into domain knowledge—a task performed by knowledge exploration exercises.

There are a couple of ways to implement knowledge exploration exercises: *Event Storming* (see [Brandolini ES]), *Model Exploration Whirlpool* (see [Evans MEW]), and *Whiteboard Modeling* (i.e., modeling at a whiteboard). While I expect that readers will learn about these techniques on their own, I want to explain the fundamental differences between two approaches that I have found effective—Event Storming and Whiteboard Modeling.

- Event Storming focuses on discovering *domain events* (see [Fowler DE])—occurrences that are important in the problem domain and recognized by domain experts. This approach is especially helpful when your exploration focuses on process and flow rather than concepts and associations.
- Whiteboard Modeling focuses on discovering *domain entities* and the *complex relationships* between them that domain experts understand. This approach fits best when you are exploring nouns (i.e., concepts, the "what") in knowledge and how they relate to each other, while a process or a flow may be relatively straightforward and less involved.

Next, I will describe the phases of knowledge exploration. Each part will distill the general concept and provide interpretation in terms of two concrete approaches—Event Storming and Whiteboard Modeling. For clarity of the presented material, I will delineate two exercises from each other, but that does not mean it is the only way. You could mix

them since there is no need to constrain yourself with using just one exercise. Remember that knowledge exploration is a way of conducting an *experiment*, and thus you are allowed to try anything that you think works.

Now we are ready to look into the stages that compose knowledge exploration.

Stage #1: Casual Discussions

This stage is as simple as this: all parties involved in the software development process will get in the same room and talk about learnings from customer interviews. The invited audience will include at least a development team (software and test engineers, scrum masters, project managers, business analysts, product owners, etc.) and internal domain experts (business representatives who are interested in implementing software or new features). Do not rush to write extensive documentation or create prototypes—those things will only give a false feeling that decisions are set in stone; you will be inclined to follow what is written or shown instead of experimenting until the best option shows up.

During this discussion, different sides of the delivery chain (development vs. business) get to know each other and learn how to communicate. It may be a bit awkward if you are doing this for the first time, but eventually it will become a regular part of the process and will start delivering valuable outcomes.

With both the Event Storming and Whiteboard Modeling, this stage is identical: Discuss what you heard from end users, and encourage and feel encouraged to state various interpretations of the same feedback. Talk until you conclude that everybody understands the learnings in the same way; a shared vision of a solution—although not very clear—starts to show up on the horizon.

Stage #2: Free-Formed Sketches

You need to start expressing your thoughts more visually now that you have reached some verbal consensus about customer expectations.

In Event Storming, attendees should grab sticky notes and write domain events on them. Write every notable occurrence that comes to mind, irrespective of whether somebody else has already mentioned it, regardless of whether you think it is an event at all or not. Forget about chronology and only stick them to the board. We will sort them in the next stage.

In Whiteboard Modeling, you grab sticky notes or sharpies and start placing (or drawing) various concepts (nouns, entities) on the board. Again, do not bother about uncovering duplications or distilling relationships; the next stage will take care of that.

Continue discovering free-formed sketch elements until you are ready to connect various parts of the land to assemble the entire knowledge landscape. This moment will come naturally when participants of the exercise have fleshed out everything that reached their minds.

Stage #3: Charting the Landscape

The goal of this stage is to organize various elements placed on the board so that everything together makes sense.

In Event Storming, you need to move sticky notes around so that they are ordered chronologically. Optionally, discover *commands* (reasons) and *aggregates* (state changes) that cause event occurrences. The result is a whiteboard full of sticky notes that describe the flow or business functionality in terms of events, commands, and aggregates (see [Google ESI] for example).

In Whiteboard Modeling, start connecting various parts with lines (associations) to depict either composition (containment) or aggregation (awareness). Optionally, discover aggregate boundaries and rules of

CHAPTER 3 FROM CUSTOMER INSIGHTS TO INTERNAL REQUIREMENTS

eventual consistency too. The result can be a UML (Unified Modeling Language) class diagram that depicts essential components and their relationships corresponding to domain knowledge (see [Google WBMI], for example).

Repeat this and previous stages until the entire visual flow makes sense for attendees of the exercise.

At the end of this entire activity, knowledge captured on a wall or a whiteboard will represent your target solution. If necessary, take a snapshot of the whole visual outcome so you can refer to it later. If you have such a luxury, keep the wall occupied with stickies and drawings until the planned product is released so that you can make adjustments with new learnings along the way.

Now you can say that you have some understanding of the customers' needs; it is time to respond to them.

Organization's Response to Customers' Needs

As you remember, earlier in this book I suggested forming an organizational structure as a *hierarchical context map* (see "Organizational Structure" from Chapter 2, "Cross-cutting Concerns"). This section will base solutions on that concept, so make sure to read the mentioned parts of the book before proceeding further.

From Customer Interview to Organizational Transformation

Sometimes, customer interviews hint at minor fixes within a product; in other cases, new insight has a significant impact on existing organizational structure. Let us consider a finding that crosses a single development team's competencies, such as a necessity to alter a high-level strategy

or a program. When such a discovery puts our project upside-down or demands to change almost everything the company does, we are tempted to believe that findings are inaccurate or irrelevant. By doing so, we may stay in our comfort zone for a while, but we risk missing users' expectations, increasing a sunk cost, and losing to the competition. If we do not transform now, we can *achieve failure* in response to our hard work.

Therefore, insights from customer interviews must be converted into strategic action items and applied to an enterprise's hierarchical context map. Recall that I suggested forming the organizational structure from top to bottom; information also flows in the same direction throughout its existence. Following the same principle, we must apply changes at a given point in a hierarchy and transform the structure underneath. In other words, transformation must be top-down and not bottom-up.

It is easier said than done to transform a functioning part of an enterprise based on a newly found insight, but that is the whole point of being a successful, agile organization. When market conditions change, you need the flexibility to adjust; the more often you do it, the more comfortable it is next time. After each successful transformation of the enterprise, you win against the competition, so it is worth the trouble no matter what.

Navigating the Context Map to Find Fit

After customer interviews, we often form a new requirement that demands a strategic adjustment of the context map. In some cases, we decide to establish a new subdomain on a map, while in other cases an existing subdomain needs to take on new or alter its current responsibilities. Before making this important decision, we need to identify the subdomain of interest. This exercise can be challenging, so I will try to give you a practical way of solving it.

A hierarchical context map looks like a graph where any parent node has children underneath, and any child node has only one parent above it (see Figure 3-1).

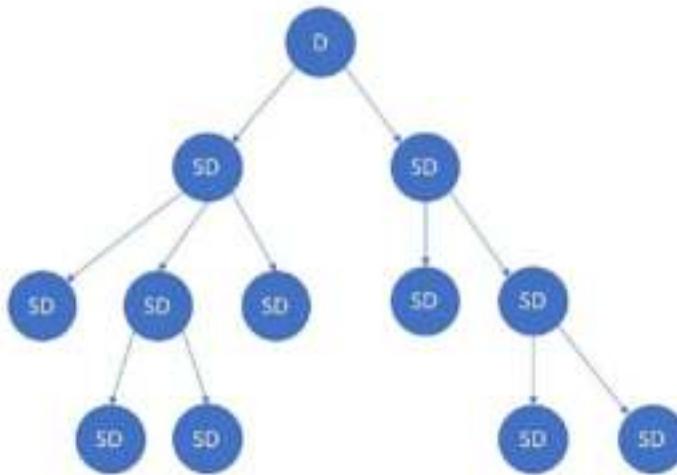


Figure 3-1. Hierarchical context map represented as a top-down graph (D=Domain, SD=Subdomain)

Let us agree that a jump from parent to child or vice versa is a *vertical* move while jumping between siblings under the same parent is a *horizontal* move. With this basic terminology in place, here are questions that you need to ask yourself repeatedly by looking at the context map and requirement at hand until you find the appropriate subdomain:

1. Is it a high-level or a low-level functionality (or, *what level is it?*)? Answering this question will help you navigate the hierarchical context map vertically. If a requirement is higher-level than a given subdomain on a map, go up; otherwise, move down. Your task is to find a subdomain at the right level with competencies *within* (i.e., its nested subdomains) that stand close to the requirement. Focus on these subdomains—both parent and its children—and move to the next question.

CHAPTER 3 FROM CUSTOMER INSIGHTS TO INTERNAL REQUIREMENTS

2. What are the problems that subdomains in focus solve, and how do they solve it? To answer this question, you will need to learn something about the given subdomains and the relationships between them. Knowing each subdomain can help you identify the possible owner of a new requirement or possible integration; knowing relationships between subdomains can help you navigate horizontally to understand the flow of data and invocations between siblings. While doing circles with these exercises, you will naturally decide where the new requirement fits.
3. Where are boundaries around existing subdomains? Think from a problem space standpoint and analyze the question both objectively and pragmatically. Understand the problem as a business understands it and not based on mere technical implications such as microservices or databases. This brainstorming should help you decide whether any of the existing subdomains can own the capability, or if you must roll out a new subdomain.

There are two possible outcomes from the described navigation exercise:

1. You decided that the new responsibility fits into one of the existing subdomains. In this case, there is no change necessary to the context map except perhaps documenting the new responsibilities. For example, if you have a "Payments" subdomain

that solves Visa credit cards' problems and you need to support the MasterCard payment option, you probably would extend the "Payments" subdomain rather than rolling out a new one. This decision path is straightforward, and I will not go into further detail about it.

2. You decided that the responsibility needs to be owned by a new subdomain. For example, if you have a "Payments" subdomain and your organization wants to roll out its custom payment method (e.g., reward points), that would not fit into the responsibility of "Payments" but would rather be a new subdomain. The cost behind this path is higher due to the need for a new bounded context, a new team, and possibly a new microservice. So, you may be tempted to avoid overhead if possible. Forming a new subdomain can be a hard decision to make without sufficient arguments. Therefore, I want to depict some of the benefits that you gain by doing so.

Why Form a New Subdomain?

The primary reason for introducing a new subdomain is when a business defines and sees a new functionality as separate from other existing capabilities. As we discussed before, I recommend forming an entire enterprise around the shapes of the problem domain. Naturally, it does not make sense to deviate from this approach for a single subdomain. If we decide to enhance an existing bounded context when a new one makes more sense, we risk introducing complexity for domain experts, who seek

to achieve technical simplicity. Besides, with this step we are putting more responsibilities on a development team that implemented the chosen subdomain's bounded context. Nobody minds more work, but when it is an entirely different business problem with a separate set of requirements and stakeholders, there can be hesitation. Also, additional requirements will compete with the existing backlog's priorities, and something will get delayed.

There is another competing force that can tempt us to go against introducing a new subdomain. Traditional knowledge behind business and management practices teaches us to leverage resources we have instead of creating new systems to maintain. This practice fails in software system development, especially with scalability in mind. If you explain a basic idea of microservices to a business person who is always focused on resource cost optimizations, you will not go too far with the concept of introducing a new program for every piece of business capability. Since the described preference goes against a new bounded context and its subdomain, it automatically encourages building a monolith that will not scale when necessary.

Therefore, after identifying a new problem that you need to solve, conduct a context map analysis. Decide whether targeted features belong in any of the existing subdomains of the organization. If you find a natural fit, then go for it. If not, then consider introducing a new subdomain, a bounded context, and a microservice as necessary. Redraw the context map with a new subdomain and keep refining it until the overall big picture makes sense from an organizational standpoint. When the context map starts expressing the enterprise's strategical intention, begin implementing a new bounded context as a microservice.

Cost of Introducing a Subdomain

Introducing a new subdomain means you must build a new team and a new microservice (new team is optional as we earlier agreed that a single team can own more than one subdomains); afterward, you will need to deal with processes and integrations between systems. Although you may think otherwise, this approach is not as costly as it sounds. Here is why:

1. Complexity is involved when you do this exercise for the first time, because you are trying something new that you have not done before. Over time, if you keep introducing new microservices when appropriate, you will accumulate the necessary skills to do it quicker, and this approach will not seem so heavyweight. If you notice that you are introducing a new microservice far too often, go back and learn the definition of microservice again. Maybe you are defining new *services* and calling them *microservices*. The problem is not confusion in terminology but in overhead when you try to apply microservice patterns and governance practices to each service. I cannot go into too many details about microservices in this section, but you can refer to other chapters of this book that speak about this topic more (start with Index).
2. Let us assume that you add a new feature to the existing subdomain where it does not fit naturally. You probably do this as a short-term approach and intend to separate new functionality into a microservice when it matures over time. However, separation is not easy in the future because there are drastic differences between patterns applicable

within a microservice vs. *across* microservices.

For example, recall that each microservice has its own database. It is almost impossible to split the database into pieces when trying to isolate microservices. Because of complexities like this, separation may never happen. You will have to deal with overhead caused by tech debt throughout the entire product lifecycle, which far outweighs the costs of avoiding it from the beginning. Therefore, if you decide to introduce a new subdomain, you minimize long-run costs rather than incur overhead.

3. Many outstanding companies practice rolling out *properly formed* microservices repeatedly, and their success is proof they made the right choice. For example, Amazon has an enormous number of microservices. Also, they have built their entire organization around teams that own microservices (see [Infoq MSA]).

Once you redraw the context map or decide where new capabilities fit, it is time to start writing requirements, which I will discuss in the next section.

Requirements and Story Writing

In this section, I will refer to a *story* and a *requirement* interchangeably since they both denote an *ask* that the software development team needs to implement. While Agile practitioners create stories and non-iterative processes call them requirements, this section aims not to emphasize process differences but rather to focus on techniques for writing effective asks for engineering groups.

Ubiquitous Language: What Is It? Why Is It Important?

Before we go any further, I encourage you to become familiar with the concept of Ubiquitous Language (see [Evans DDD]); otherwise, some pieces of advice that I give may be hard to understand.

Usually, writing a program is like dealing with a foreign language: developers write code, and non-technical people (e.g., domain experts) have a hard time understanding it. At first sight, everything seems as it is supposed to be. We have domain experts on one side and engineers on the other, and they all do what they are good at. Domain experts know the problem that needs to be solved, and engineers come up with a solution. If you follow this process, as simple as it sounds, you will describe a problem in one language and solve it in another. Maybe this outcome seems fine to you because we compared programming to a foreign language, but consider these complications:

- When a domain expert speaks, engineers need to deal with a constant mental overhead to translate it from *business* into *technical* terms to implement a new requirement or change existing behavior in the program. This burden causes a slight slowdown throughout the entire process and thus incurs costs. Furthermore, since it is a *translation*, there is a chance of misinterpretation and missed requirements as the origin of information (i.e., domain expert) cannot validate the correctness of the translation instantly.
- However, when developers discuss implementation details upfront, domain experts often go into “ignore” mode because they do not understand a word of what they hear. Due to this disconnect, even if something is wrong in implementation from a *business* standpoint,

SMEs will have a hard time identifying the shortcoming until somebody decides to translate the crux of conversation for domain experts in *business* terms, or worse—until the product is ready for demonstration. This phenomenon again causes slowdowns due to possible rewrites and can impact customer satisfaction with incorrectly built software. It is often easier to assume that implementation is correct rather than keep translating back and forth every single time, and this assumption results in mistakes.

"All these complaints are valid," you will say. "But what is the point if code is still written in a foreign language?" What if I told you that you could minimize translation overhead by slightly adjusting the approach? Specifically, you need *to find and adopt a shared (a.k.a. ubiquitous) language understood by both domain experts and software engineers*. Both sides need to contribute to reaching this objective:

1. Domain experts need to adjust their communication language to make it easier for engineers to follow it in code and the entire solution. We will discuss this technique shortly in the next section.
2. Engineers need to put in effort to follow business language. We will review this approach in later chapters of this book when we talk about software development and architecture.

In the next couple of sections, I will explain tweaks you can make to natural language when writing requirements so that developers can adhere to it more easily. Presented techniques will progress from the simplest to more advanced levels for clarity and completeness of the topic.

Who Writes Stories?

In some cases, domain experts stay away from writing stories and let technical teams take ownership of it. When this happens, naturally, we end up having a backlog formulated in technical terms because developers speak *technical* language, and not *business* language. This approach does not seem so bad in theory until you suddenly realize that there is not much that domain experts can understand in their product's requirements. As a result, we have problems such as the following:

- There is no sense of ownership or progress toward a goal.
- It is hard for a non-technical person to ascertain whether backlog is delivering what the business asked.
- There is a risk of missing end users' expectations and thus jeopardizing customer satisfaction due to possible confusion about the intent of stories.
- Quality is at risk due to inadequate implementations and increased number of mistakes, fixing which takes additional iterations.
- The cost of overall delivery goes up due to the mentioned kinds of overhead.

Therefore, all functional requirements should be formulated and owned by domain experts (i.e., product owner, business analyst, a business representative, etc.). Let the technical team implement these requirements as they are written instead of rewriting them in technical terms.

Note that this advice only applies to functional requirements. It is fine to use technical language when describing tech debt or other non-functional expectations. Nevertheless, even a technical requirement must clarify the business value that it delivers or allows the company to reach.

When domain experts take ownership of functional requirements, any non-technical person who understands the problem domain can read stories and measure progress toward upcoming or future releases at any point in time. Besides eliminating the issues described previously, this clarity will remove mental overhead associated with translation and improve knowledge flow between technical and non-technical team members.

Ubiquitous Language in Requirements

As we agreed before, both domain experts and engineers need to form and follow a shared ubiquitous language in their communication, writing, and code. In this section, we will look at basic techniques to practice a universal language when writing requirements. This is an essential step toward success as written requirements are what engineers commonly receive and follow. If we can write acceptance criteria using a ubiquitous language, then we can achieve our initial goal—to write both stories and code with the same vocabulary. With the right tools and techniques, program implementation can follow a ubiquitous language.

Ubiquitous language is not the same natural language that we use in our speech, although it is very close to that. We need to apply minor but necessary tweaks to spoken word to turn it into ubiquitous language. Let us look at one common issue that calls for course correction to give you an idea of what it takes to form a shared language.

Commonly, a problem space (i.e., an area or field of work) has many synonyms meaning the same thing (e.g., "Customer" vs. "Client" or "Employee" vs. "Team Member"). Domain experts have a habit of spontaneously replacing words with their synonyms when explaining requirements to the development team.

However, as I mentioned previously, engineers need to bring implementation code as close to business language as possible, so when implementing an ask stated using interchangeable synonyms, the program

CHAPTER 3 FROM CUSTOMER INSIGHTS TO INTERNAL REQUIREMENTS

will also use different words to denote the same thing. This ambiguity creates additional mental overhead because now every engineer needs to know the glossary of synonyms to read the code or to speak to domain experts. A ramp-up process also becomes harder because every synonym needs clarification. Besides, when speaking to domain experts using different words to mean the same concept, it often becomes necessary to verify that everybody says the same thing as we cannot afford a misunderstanding or misinterpretation. These problems hint at the necessity to tweak natural language slightly instead of using it as-is when writing requirements.

Therefore, domain experts need to be familiar with the idea of a ubiquitous language to form it with the engineering team. Practice picking one term out of all synonyms to denote any single concept; these selections will form a unified subset of the language that domain experts use to produce requirements. Document both unified terminology and their synonyms in a glossary (e.g., a Wiki page) to make it accessible to all interested parties. When an unfamiliar concept appears in conversations, revisit the lexicon, and add the new word to the right set of terms. Treat unified terminology as your ubiquitous language and ask engineers to follow it as closely as possible. Pay attention to discussions among developers and domain experts to spot warning signs in the form of deviations from established language.

When requirements are written using a unified subset of business language (i.e., ubiquitous language), understanding the backlog becomes more effortless than before. Now, the reader can build mental connections between various work items to digest information faster. There will be less confusion and necessity to clarify stories or codebase due to ambiguous terminology in it, and therefore the development pace should improve slightly.

For the exhaustive definition of a ubiquitous language and techniques for forming it, please refer to the respective reading material (see [Evans DDD]).

Writing Executable Specifications

An executable specification is a requirement or acceptance criteria written in an almost natural language that can be executed by computer as-is to verify the desired functionality. In simple words, *a non-developer can write an executable specification, and a machine can assert the program's compliance with expectations*. Note that a developer's involvement is necessary to convert the text of expectations into corresponding function invocations.

The benefit of executable specifications is in the expressiveness of acceptance criteria and the ability to verify and see results of checks in an almost natural language instead of dealing with code that means nothing to non-technical people.

Here is a process distillation for writing executable specifications:

1. A domain expert formulates acceptance criteria using a predefined, almost-natural, and slightly structured language. This step is necessary because it is technically more convenient to connect code with structured language than unstructured.
2. A developer writes code that associates essential verification programming instructions with every line of acceptance criteria. In simple words, this step translates expectations into a program.

Let us look at one of the most popular structured languages for writing executable specifications—Gherkin (see [Gherkin]). I recommend using Gherkin because of the following benefits:

- Gherkin enforces only minimal constraints over natural jargon, and thus you can stay very close to ubiquitous language as long as you follow a small set of simple rules. For example, you need to start every new line

of acceptance criteria using one of the predefined keywords ("Given," "When," "Then," etc.). You can find more detailed guidelines in the official documentation (see [Gherkin]).

- If you carefully follow Gherkin language rules, the resulting text will qualify as an executable specification. A developer can take the final document and connect it to code using one of the Gherkin-based test execution frameworks. Since Gherkin has been around for a while, there are many choices for accomplishing this goal; necessary efforts are not more than what it takes to write an automated test for the same acceptance criteria, while long-term benefits far outweigh the alternative. It is worth mentioning that I authored one such framework, Xunit.Gherkin.Quick (see [Tengiz XGQ]), for those developers using .NET Core or .NET. If you are using any other technology, I am sure that there is a similar framework for your choice.

Halfway into Gherkin

I have often seen a product backlog that showed signs of *partial* Gherkin usage. For example, since it is considered a clear way of writing requirements, domain experts often express their needs by using Given-When-Then keywords. This approach already delivers benefits over using a casual, unstructured natural language, such as the following:

- It forces one to split every acceptance criteria into three distinct elements—a context or precondition (Given), an action performed by a user or a system (When), and an expected outcome (Then). Without Gherkin, this separation is often unclear.

- It helps spot requirements that are too big and are candidates for breaking down (we all know that smaller stories are simpler to track, develop, test, and deliver to customers). When we have more than a couple of criteria (too many Given-When-Then trios), we know that requirement is probably asking for too much in one go. On the other hand, if we write acceptance criteria as a paragraph, it is harder to say where to draw the line of an adequately sized requirement. It is easy to write a lot of text and think that you merely explained a single story, while you may have gone too far by squeezing dozens of various needs in a single paragraph. Given-When-Then trio makes measurements more explicit.

Therefore, instead of writing requirements as a chunk of casual text, use Given-When-Then keywords to articulate needs. Spot how many pieces (Given-When-Then trios) you end up having. If there are too many fragments in any given story, consider breaking it down.

By applying this straightforward technique to your story-writing exercises, you will simplify development, testing, and deployment processes, as well as tracking progress toward the next release.

I call this approach "halfway" because the full-fledged Gherkin syntax goes far beyond the usage of Given-When-Then keywords. While you may be enjoying the benefits of such a primitive use of Gherkin jargon, I encourage you to consider going all the way in. There will be more about this advancement in the next section.

All the Way into Gherkin

Many development teams are halfway into Gherkin usage like I described in the previous section. Should they consider going all the way in and using the full syntax of Gherkin? Is the learning curve worth payback? It is easier to decide if we look at additional benefits that using full Gherkin syntax provides compared to being halfway in:

- A full-fledged Gherkin discourages putting When after Then, but you probably do not know that if you are halfway in. What happens when you violate this simple rule? You are back to where we started—too many requirements glued together as a single expectation (they look like one requirement because there is only one “Given” context). If you go all the way in, such silly mistakes will disappear.
- If domain experts do not understand full Gherkin grammar, they use it inaccurately. For example, while the “When” keyword is supposed to mark an action, it is often mistakenly used as a precondition. Thereby, code that tries to implement half-Gherkin text as a specification introduces technical awkwardness and confusion. Engineers and testers deal with complexity overhead when they return to this strange code block, but they hesitate to change it because code needs to follow specifications carefully. This overhead might seem marginal, but consider accumulated cost over time for both engineers and testers.
- The most significant downside of not going all the way into Gherkin is that you do not have executable specifications. When writing automated tests, engineers need to deal with translation overhead

between specs and code, or they need to rewrite specs in the right way. A half-Gherkin does not qualify as an executable specification because all frameworks that parse Gherkin text expect full Gherkin syntax and not half of it.

Therefore, if you are halfway into Gherkin usage, consider learning and adopting it all the way. The learning curve is not significant, but it pays back with many advantages, as described in this section. If you can do half Gherkin, you can do full Gherkin too!

Wrote your requirements? Great! Now, let's create a plan to implement them.

Planning Work

In both Waterfall and iterative processes, teams conduct planning sessions to decide what they will work on next. In this section, I will focus on techniques that make the planning exercise productive and most effective.

I may use terms from Agile (such as Product Backlog or a Story), but I do not necessarily prescribe following Agile processes over Waterfall.

Prioritized Backlog

When a team finishes its assignments, it is time to pick the next task from the product backlog. The simplest way to quickly grab something is to take it from the top of the list. However, when the process is not mature, there may not be a list, or it may not be sorted by priority. In such a case, a developer may need to check with a business analyst (or product owner) to locate the most critical task. If the analyst is not available, the engineer will need to either wait or pick a job based on their judgment, which may not be ideal. Waiting means more time before the next feature is released,

while guesswork implies that an important task may get delayed. Besides, such situations may hint at other problems such as unclear product roadmap or poor backlog management.

Therefore, it is the responsibility of a business analyst (or product owner) to maintain backlog items sorted by priority. At any point, the development team should feel comfortable picking a top-most task from the list without the overhead of additional questions, wait times, and delays to features.

This issue usually seems unimportant, but you will improve several strategically essential activities, such as roadmap planning and continuous flow of work, by solving it. Eventually, you will also save development time and accelerate the delivery pace slightly. Such small improvements will accumulate over time and work in your favor.

Feasibility

Achieving a continuous flow of development work is vital, as it means no delays or unnecessary interruptions. When the backlog is prioritized, the team can keep coming back to it to pick the next task when there is free capacity. What happens next is also paramount: We need to ensure that work gets carried out successfully and without blockages. If there are minor questions in the middle of development, that is not a big problem. On the other hand, it is a warning sign when a developer discovers that the component she is working on is supposed to invoke a service that is not yet ready. It is also not good news if a task does not fit in an allotted timeframe (sprint or milestone), because then we risk breaking a promise or not delivering to expectations.

Therefore, we need to take work item feasibility seriously. Before applying efforts to any task, the development team is responsible for ensuring the delivery of work within a given timeframe without significant glitches. Exceptions do happen, and it is not a catastrophe as long as it stays an exception. In most cases, though, the team needs to meet established expectations.

CHAPTER 3 FROM CUSTOMER INSIGHTS TO INTERNAL REQUIREMENTS

There are various techniques to measure work item feasibility upfront; one of them is *INVEST* (see [AA INVEST]), which I like to use as a basis because it focuses on the most critical criteria for a task. Perhaps each team should start with a strong foundation like *INVEST* and add other items on top.

To shortly cover the meaning of *INVEST*, it is an abbreviation of five expectations for each story before we consider it feasible. The following is a distillation of terms in *INVEST* from the Agile Alliance website:

- “I” = Independent (of all others)
- “N” = Negotiable (not a specific contract for features)
- “V” = Valuable (or vertical)
- “E” = Estimable (to a good approximation)
- “S” = Small (to fit within an iteration)
- “T” = Testable (in principle, even if there isn’t a test for it yet)

While these expectations seem straightforward, they are not so easy to achieve due to either technical implementation or project or product management challenges. For that reason, I will cover several topics in this book that will help you stay true to all five expectations for work items in your backlog.

Before going any further, please familiarize yourself with the *INVEST* principles since I will be referring to them later on.

Managing Dependencies

How do we make every story independent in practice? It seems easier said than done when we see it in *INVEST* criteria. It is indeed challenging from all sides of the equation—development, process, and product management. Effectively, independence means understanding and

managing dependencies for each story ahead of time so that we can measure their feasibility and complete them without hiccups. To tackle this challenge, let us look at techniques to manage dependencies in an organization.

Parent Subdomain Prescribes Child Subdomain Dependencies

Before discussing techniques for managing dependencies, let us recognize a benefit that we get for free by following the organizational structure from Chapter 2 of this book.

In a hierarchical context map, every parent subdomain is aware of integrations (and thus dependencies) between subdomains or bounded contexts underneath. That is a fundamental guiding principle of the organizational structure and corresponding framework suggested by this book. As we agreed in respective sections, all requirements flow from top to bottom in an enterprise. As integration is also a requirement, it will be predefined on the parent subdomain level before reaching any child subdomain and corresponding development team for implementation (see Figure 3-2).

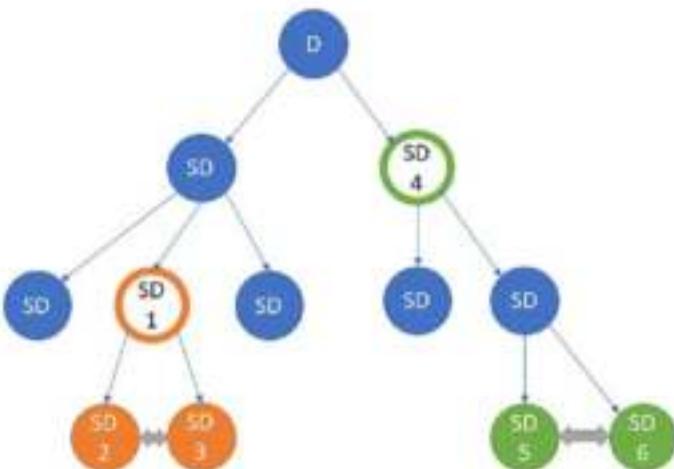


Figure 3-2. Subdomains SD2 and SD3 integrate due to their direct parent's (SD1) needs; subdomains SD5 and SD6 integrate due to their grandparent's (SD4) demands. In both cases, ancestors define integration needs for their descendants. Note that this figure does not show all horizontal dependencies among subdomains; however, there is always at least one parent domain that governs or dictates any given horizontal integration

Therefore, a hierarchical framework for structuring an organization eliminates the complexity of discovering dependencies. There may be exceptions that will redefine a context map due to unforeseen discoveries. However, usually, each subdomain can expect that dependencies and integrations will be predefined for them by their ancestors.

For example, suppose that we have introduced a "Rewards Payments" subdomain under "E-Commerce." How can we discover which other subdomains of the organization need to integrate with the newly added capability of Rewards Payments? Based on my last suggestions, you would want to ask the parent subdomains' stakeholders or SMEs. Perhaps this vertical path would take you to a specific ancestor subdomain (e.g., "Organization"), which imposes the following requirement: *for every fulfilled order, a customer receives rewards points for future purchases.* As

you can see, this statement prescribes an integration between “Orders” and “Rewards Payments,” which must be indirect descendants of the “Organization” domain (see Figure 3-3).



Figure 3-3. Organization domain prescribes integration between its indirect descendant subdomains—Orders and Rewards Payments

Classify Dependency

While parent subdomains handle the discovery process for the child subdomain’s dependencies, the descendant subdomain’s team is responsible for managing its development technicalities and processes afterward. To be effective at this task, we need to *classify* dependency.

Each dependency (and thus integration) has two bounded contexts (and corresponding subdomains) on its ends. We need to understand who is who in this relationship. In other words, we need to say what *kind of relationship* we have between given bounded contexts.

A hierarchical context map, besides enlisting subdomain integrations, must also specify relationship kinds between them. This information will be accessible to each bounded context’s team when necessary. Depending

on relationship kind, different dependency management techniques will apply from the bounded context's team standpoint, mainly depending on the direction and mechanism of invocations between systems. Please refer to Domain-Driven Design (see [Evans DDD]) to learn about relationship kinds between bounded contexts and their management strategies.

I assume that you have some understanding of relationship kinds between bounded contexts from DDD; if not, make sure to familiarize yourself with them. Here they are listed just as a refresher:

- **Partnership:** Bounded contexts are equally important to each other.
- **Shared kernel:** Bounded contexts share their models, either partially or wholly.
- **Customer-supplier:** There is an agreement between bounded contexts about the strategical importance of one over the other.
- **Conformist:** One of the bounded contexts conforms with another.
- **Anti-corruption layer:** A bounded context integrates with another but tries to minimize the impact of integration.
- **Open-host service:** A bounded context with which many others integrate.
- **Published language:** A unified or well-known model, language, schema, or protocol often combined with open-host service.
- **Separate ways:** No integration at all
- **Big ball of mud:** Unorganized structure and integration between bounded contexts; non-optimal boundaries of bounded contexts themselves; no bounded contexts

Except for "Separate ways," all other kinds of relationships will need to be managed as a dependency from a bounded context's standpoint.

Dependency Goes First

It can be a straightforward idea that an organization needs to implement dependencies before building dependent systems. However, various teams commonly violate this simple rule.

Let us look at the problem from two different angles—an organization and a team—to realize the challenge and cover techniques for addressing it.

Organization-wide deliveries often span multiple teams and projects and depend on each other. Deadlines for delivering the entire program are set in stone, and thus project managers are forced to squeeze project delivery plans into predefined timelines. This pressure results in puzzle-like project charts with tightly adjacent dependent features with a high risk of failure due to the necessity to succeed in *everything*. If anything slips on such a delivery plan, the entire program may fail or get delayed. Such approaches sacrifice the quality of delivered products and hurt customer satisfaction because they encourage one to cut corners when something derails.

Therefore, when forming a large-scale program plan, always put some buffer into it. Abandon traditional all-or-nothing project management practices that have tightly adjacent delivery milestones. Instead, decompose work into smaller, more manageable, and preferably independent pieces and keep a time buffer between them.

This safety net ensures that a slight delay in an earlier feature will not cause trouble to subsequent tasks. Thus, overall program delivery plans are more likely to succeed due to minimized risk in the roadmap.

Software development teams often decide to align their efforts so that upstream and downstream groups start working simultaneously. This trick may sound like a good idea to deliver more in a shorter time, but in many

cases it leads to delayed work and unmet expectations. There is a lot of unknown and risk of late findings in software development that are the root cause of unanticipated latency among upstream systems.

Therefore, if you have a dependency on another development team's delivery, let them finish the work and avoid starting your efforts until you see the necessary capability deployed and working correctly. Do not plan to begin your work ahead of time by assuming that the required functionality will be available before you need it. Instead, once you have verified delivered capability, plan your work in the nearest future. In rare cases, you may see that you could have trusted the dependency team's estimates, but in most cases, things will happen that are out of upstream's control. While waiting for dependency, do not consider yourself as being blocked—there must be other backlog items that you can work on.

By following this technique, you will explicitly stay true to your plans as you will not attempt to work on high-risk items. Similarly, every other team in an organization can avoid delays caused by dependencies. Higher-level program tracking becomes straightforward as you have a solid plan and metrics for deliveries from each group. You can finally predict the deadline for programs spanning multiple groups without worrying about the accuracy of each team's estimates. Remember, every precise prediction is another success in meeting expectations by delivering based on given promises.

Dependency Is Functional. Integration Is Technical

As we discussed earlier, the hierarchical context map needs to depict dependencies between subdomains. Domain experts handle this work while defining business-driven parent subdomains. When it comes to implementing lower-level subdomains using bounded contexts and microservices, those same domain experts often feel responsible for driving the integration's technical implementation. Matters get worse when software architects step back from technical decisions and let business SMEs do this work. Such process disorder results in building non-optimal software systems in terms of maintainability, scalability,

performance, and so on. For example, a domain expert may prefer to integrate microservices via direct database access instead of API calls. While this choice is faster and simpler to implement, it is considered an anti-pattern since it tightly couples applications to each other and negatively impacts the maintenance and scalability costs in the long run.

Therefore, let domain experts define dependencies between subdomains since it is part of business knowledge. Let technical staff decide how to integrate systems since technical knowledge is necessary for making optimal choices in this area. Create a culture such that both sides respect each other's decisions as each acknowledges their own field of competency.

Valuable Stories (Verticals)

As you remember, in the INVEST criteria, "V" stands for "Valuable." This section will try to adjust the understanding of this expectation by technical and non-technical members of a development team. Ultimately, the solution lies in a compromise for both sides of the equation.

Domain experts tend to focus on higher-level deliverables that stretch at an organization level. Therefore, requirements that business representatives write often expect too much in one go and span across long periods (longer than an iteration in Agile or a milestone in Waterfall). When a development team receives such a big ask, they need to break it into smaller chunks somehow to allow delivery piece by piece. Product owners and business analysts demand to receive "all or nothing" since that is what they need to get, and tracking a low-level breakdown of requirements does not make sense to them. Not seeing any other way, the engineering team decides to develop *technical* pieces of functionality that will work together at the end. One example of such a non-functional breakdown is building one horizontal layer at a time (e.g., backend first, front-end next).

This workaround sounds smart at first sight since it solves the challenges facing both business and technical personnel, but it causes several problems, as follows:

CHAPTER 3 FROM CUSTOMER INSIGHTS TO INTERNAL REQUIREMENTS

- Tracking technical pieces of delivery does not make much sense to the business. Having no other way to follow progress, leadership starts focusing on people's time management (i.e., micro-management) instead. Furthermore, if we have too many work items in play, metrics stop making sense to stakeholders. This uncertainty decreases team transparency and jeopardizes the predictability of deliveries.
- When the team completes the final piece of work, and domain experts see a product, their feedback may ask for a significant change. The problem is that this event happens too late in the game, so fixing mistakes may cost too much. If feedback relates to the foundational part of the functionality, we may need to rewrite the entire product. As a result, there is a risk of increased cost due to the stretch in the feedback loop.
- If we break the solution into technical pieces, the development process will have to run without a domain expert's feedback for a long time and thus without an opportunity to apply a ubiquitous language to implementation. This outcome is unavoidable because technical components are not domain-driven, and therefore they do not promote communication between technical and non-technical members of the team. We already discussed the importance of ubiquitous language elsewhere in this book, so you have an idea about implication.

Therefore, avoid splitting a business requirement into technical components at all times. Instead, divide expectations into smaller functional pieces so that each part makes sense to domain experts. This process is a

negotiation and requires that both sides be ready for a compromise for better outcomes. Product owners and business analysts need to be more flexible to drill down into functional requirements and break them into pieces that they can track and understand independently. Developers need to be ready to cooperate and help domain experts in the process of defining feasible boundaries. Afterward, engineers need to keep communication open to successfully apply a ubiquitous language to implementation.

When a large requirement is split into smaller functional parts rather than technical pieces, tracking progress becomes meaningful from a business standpoint. As a result, transparency and accountability are optimal for the work that is planned or is in progress. This technique will positively affect overall organizational delivery pipelines, communication, and knowledge sharing across team members.

Technical Stories

In Agile and other processes, an idea of *evolving design* is trending, which is often confusing to developers. While you cannot predict the entire product roadmap and thus cannot design for every use case upfront (i.e., the crux of the evolving design), it is often mistakenly understood that technical design (architecture, style, patterns) will evolve *by itself*. With such a belief, developers do not invest enough time in technical discussions and decisions, which negatively affects the quality of delivered programs. Architecture is half-baked or non-existent, and the application cannot meet non-functional expectations such as quality, performance, scalability, and adequacy of architectural choices. Awkward, uncommon structures and constructs emerge that are hard to understand or explain to other engineers. This outcome means longer ramp-up times for new joiners and an unnecessary complication earlier in the software lifecycle; thus, it increases costs.

CHAPTER 3 FROM CUSTOMER INSIGHTS TO INTERNAL REQUIREMENTS

On the other end of the extreme, I have seen business analysts or product owners acting as architects and making technical decisions about the program's structure. Worse, solution and application architects explicitly hand over such decision levers to the business side. Would you go to a dentist to treat your stomach? Then what makes you believe that domain experts can solve architectural problems? Architecture should indeed be carefully hand-picked under the problem domain. However, without technical expertise, the choice will be suboptimal: implementation will not meet non-functional requirements (i.e., availability, scalability, and maintainability). Left with the responsibility to make both functional and technical decisions, business representatives usually focus on domain-specific tasks, and technical needs receive no attention. This outcome results in technical debt accumulation and future costs when the team needs to address debt. Meanwhile, the quality tanks and customer satisfaction is at risk. Only late in the game, technical personnel realize that they need to take ownership of design and architecture. However, it is almost always impossible without a significant rewrite or re-architecting of software.

Therefore, it is vital to understand the necessity for a technical backlog besides functional tasks. Engineering team members need to take ownership of design and architecture for delivered software. Product owners and business analysts need to defer technical decisions to members who have corresponding skills. Engineers and architects need to assist business functions by assembling supporting technical requirements. Essentially, you must establish a partnership between product and technical personnel to achieve business benefits without sacrificing technical characteristics of quality, maintainability, scalability, security, performance, and so on.

When the responsibility for suggesting and composing technical requirements is on engineers and architects, you have assurance that the delivered program will be of high quality, scalable, maintainable, and so on. Therefore, you are avoiding a cost overhead for unnecessary re-architecting or rewriting in the future when you carry out the work, speaking of which...

Carrying Out Work

Let's now look at vital techniques to ensure we can efficiently carry out the work we have planned.

Definition of *Done*

Any reworking in a software development process is overhead that adversely affects the cost of delivered programs, customer satisfaction (due to unforeseen delays), quality (due to squeezed deadlines), organization's overall performance, and reputation. To avert these outcomes, you should avoid rework at all times.

Let us look at typical development situations that cause rework:

- A developer needs to stop working on the current story and go back to the previous story's code because the QA team just reported a bug in it. This situation creates context switching, which delays the delivery of both features.
- A developer needs to rewrite part of code that did not pass code review. This scenario uncovers time spent unnecessarily on a program that had to be changed later on.

We may not be able to predict such scenarios before they happen, but it is straightforward to avoid them *after* we have encountered them at least once.

Therefore, learn those frictions that cause rework in a team and systematically brainstorm how to avoid them in the future. Mitigate adverse outcomes by planning ahead of time actions that help prevent issues.

Compile a list of activities that need to happen before starting to work on the next iteration, milestone, or significant chunk of work. These repeatable

actions are usually called Definition of Done (DoD). Follow DoD in every iteration or milestone. Before you call it “done,” each requirement needs to meet the criteria of DoD; until then, it is NOT done. Track rework to ensure that DoD helped and, if not, refine DoD until it becomes useful for a given team.

Here are the DoD criteria that you could incorporate to avert the previously mentioned situations:

- To avoid the developer's context switching to go back to the previous story due to a bug, we could agree that the developer needs to team up with the QA engineer until testing the story is complete instead of starting a new work. This technique goes hand-in-hand with the approach of finishing one story at a time as a team, which I suggested earlier in this book. Alternatively, developers could have access to test cases to ensure that the delivered code passes all acceptance criteria defined by QA team members even before the testing process starts, which could avoid bugs altogether.
- If you base code reviews on agreed-upon conventions and design guidelines, you can avoid the majority of subsequent rewrites by aligning the entire team to the same coding rules.

These items are just an example of applying DoD, which can go far beyond these simple problems. With every team, challenges are unique and need addressing in a way that works for the group. Either way, when DoD is in place, you can eliminate rework, optimize the cost of development, and increase the quality of delivered software.

Estimates vs. Velocity

When project managers try to predict future development pace, it is common to base the calculation on estimates. When actual delivery times do not match with initial estimates, project managers and other leadership representatives often get nervous, which in turn creates pressure for development teams. In such situations, engineers feel tempted to cut corners and meet promised deadlines regardless of what it takes—sacrificing quality, accumulating tech debt, and so on. Decisions like these increase the overall cost of development as they delay delivery of the entire project due to newly introduced tech debt and complexity. Is that the influence that project management is supposed to have? Does this outcome align with the business's vision for achieving customer satisfaction? What is the purpose of estimates then?

Use estimates as a capacity gauge only. That is, the total of provided estimates should roughly equal available time within iteration, milestone, or release. This simple approach helps you plan future workloads. Avoid turning estimates into deadlines; otherwise, you will only face the problems described previously. Instead, predict the next deliveries based on average historical velocity and set stakeholder expectations accordingly. Observe the development pace of a team instead of enforcing timelines.

What is velocity? In Agile, you can track total story points delivered in an iteration; in the Waterfall, you will need to sum up the requirements' T-shirt sizes.

The described technique is sufficient for predicting future deliveries, which means meeting deadlines, producing high-quality programs, and increasing customer satisfaction. Besides, you get all these benefits without creating unnecessary pressure for the development team.

Your Estimate Is Your Deadline

As I mentioned before, under tight deadlines, developers start making odd decisions to cut corners and meet estimated timelines. For example, it is common for team members to decide not to write unit tests in order to fit in the allotted timeframe. Although this behavior creates a feeling of accomplishment short term, it negatively affects the overall cost of completing the project due to accumulated tech debt, low quality, and even poor customer satisfaction.

In the previous section, I suggested that project managers and other responsible people should refrain from enforcing deadlines based on original estimates. In some environments, that option is not viable, and we are back to stressful workloads. When project management is trying to enforce tight deadlines, what can developers do?

Developers need to understand that deadlines enforced upon them are based on estimates they gave in the first place! To avoid tight deadlines, provide estimates that let you comfortably deliver commitments. Always remember items from your Definition of Done and account for all of them. If unsure, increase projection instead of hoping for the best-case scenario. Remember, if you cut corners now, it will slow you down in the future, so prefer to be late once instead of being behind at all times.

When engineers start providing safe-side estimates, overall predictability will improve due to fewer slips in delivery dates, and pressure will also be minimal.

Achieving Predictability

The core of customer satisfaction is *predictability* because then we deliver based on our promises, and we are in control of our future as an organization. From the development teams' standpoint, predictability is the ability to *precisely* say what they will achieve in a given timeframe. It is one thing to predict velocity, but it is another to be confident in our predictions—that is predictability!

CHAPTER 3 FROM CUSTOMER INSIGHTS TO INTERNAL REQUIREMENTS

As I mentioned earlier, predicting future velocity is straightforward:

$$\text{predicted velocity} = \text{average historical velocity}$$

Now, let us understand what can make this forecast unreliable. As the formula shows, the predicted number is derived from past performance. Since future velocity is only a calculated value, we must seek problems in the input; i.e., the previous throughput.

Various factors affect teams' performance, which can make average velocity inaccurate. If we minimize such aspects, we will be able to capture the correct velocity and improve predictability.

Let us look at some examples that might distort the accuracy of measured velocity:

- **Overtime work.** If team members work more than the usual number of hours, then velocity will go up. Since overtime burns people out, the mentioned effect is only temporary—eventually, productivity tanks and we are back to missed deadlines. Therefore, I recommend that overtime work be discouraged.
- **Rotating people across teams.** Management commonly thinks that rotation is the best way to educate everybody about everything. Indeed, an engineer moving from one group to another will learn many little things about various programs. However, from a stable velocity standpoint, the rotation process will ruin each group's delivery pace, where the moved person works temporarily. An exception is if the rotation process's impacts are accurately measured, but that is complex and almost impossible from my experience. Besides, rotation can achieve knowledge breadth but not depth in any of the programs.

Therefore, preserve the membership of development teams to achieve stable velocity and thus predictability. It is fine to rotate team members if the predictable delivery pace is not a priority, so you need to weigh the trade-offs.

- **Fluctuating workload.** I have worked in companies where the size of the workload changed based on company politics or process inefficiency. For example, some companies allocate budgets to expand teams without assigning more work. In other cases, non-productive grooming sessions do not yield a new set of backlog items. If there is nothing to work on, or if the workload only occupies a part of the team's capacity, velocity will be lower than it could be. Eventually, looking at average historical speed will not provide an accurate input to predict the future. Therefore, if it is within your control, allocate enough workload to fill the development team's capacity so as to improve predictability.

Keep monitoring the situation in your team and find other issues that cause velocity to be inaccurate. Do not be confused though—delivery speed *can fluctuate* over time under normal conditions too. I am only asking to ensure that *measured throughput* is accurate and reliable.

When average velocity is sustainable, then you can use it to predict future development and delivery pace. If you monitor this important indicator throughout the entire organization, you will be successful in managing product and portfolio roadmaps across the enterprise, a sign of mature agility and future-readiness. Most important, delivering based on your promise, setting the right expectations, and subsequently meeting those expectations become feasible.

Summary

We have covered a lot in this chapter, from customers to knowledge exploration exercises, organizations to context maps, requirements to ubiquitous language, and predictable deliveries. Now it's time to think about design and architecture as critical steps for connecting the needs with the implementation.

CHAPTER 4

Design and Architecture

I once read that *architecture is an opinion*. If we trust this basic definition of the subject, it becomes difficult to define anything exact within this space because everything I say can be just my opinion. With the same argument, it is effortless to argue about anything that I say, because your idea can be different from mine. Therefore, I want to claim right from the beginning that everything you are about to read in this chapter of the book is strongly opinionated; that is by design.

What assures you about the validity of my opinions? I feel comfortable backing my architectural style and techniques with my experience building real-life projects for corporations of various sizes and industries. I wanted to clarify this matter to distinguish my work from random information that has no value. Therefore, rest assured—everything you read in this book has been tried, and it works, and thus I am sharing my experience with you and other readers.

Besides my experience, another kind of certification for the provided information is the established knowledge found in software architecture. I will refer to additional reading material so you can learn more about specific subjects; you can do that for both verification and education purposes.

CHAPTER 4 DESIGN AND ARCHITECTURE

As I clarified earlier in this publication, I do not plan to give you yet another book that explains every known technique from scratch. Instead, I will cover topics that help make the best use of *already available knowledge* in this area. For example, instead of explaining what a particular design pattern is, I prefer to give you a link to the origin of that information; meanwhile, I will only address use cases in which the technique is useful or harmful.

I will also guide you through techniques that are less known and come only with experience while combining several patterns to achieve the desired result. I believe that such an approach to learning can be a good breakthrough for a software engineer or an architect.

In my opinion, software architecture is a supporting topic for all phases of a typical software development project. Therefore, I intend to discuss issues in this chapter from various angles, such as cross-cutting concerns, analysis, design, coding, testing, deployment, and maintenance. I hope that such a structure will make knowledge more applicable to real-life work that you do professionally.

Let's start with cross-cutting concerns in software architecture.

Architecture as a Cross-cutting Concern

Let's discuss how architecture impacts cross-cutting concerns addressed earlier in this book.

Definitions and Purpose

Before we go any further, let me review a couple of vital definitions on which the rest of the content relies.

Software Architecture

Software architecture has many definitions, all of which are correct to a certain extent. You can visit Wikipedia (see [Wiki SA]) or other articles on the internet for an explanation of this term. I have also published an article on this topic via my website (see [Tengiz WHATISA]).

In this section, I want to draw your attention to the most critical aspect of the definition: *Architecture is something that you cannot easily change after implementation, but it is still a vital supporting mechanism for the system's function.* By saying so, I am trying to warn you that poor architectural choices made earlier cannot be easily undone later in the project lifecycle. At the same time, they can hinder and complicate both the implementation and the usefulness of the deliveries. This limitation is what makes the role of architecture so critical to success. Inadequate architecture will fail you; a robust structure will serve you.

Software Architect

As I said before, *architecture is an opinion*, which makes an architect an *opinionated person*. Opinions do not exist in a vacuum; they originate from people's minds. It is hard to argue which view is wrong or right without knowing right from wrong. That is why architecture has a body of knowledge that can shape or influence a considerable number of decisions. Since this knowledge's breadth is quite extensive, it takes time, curiosity, and talent to learn it to the extent that one can use it to fulfill desired design goals.

All of this can sound like yet another definition of software architecture. However, I intended to show qualities that a software architect needs to possess. The most important is to have a mastery of the knowledge area so that the many moving parts of theory do not contradict each other and can apply to problems in practice. It takes a certain kind of person to do this properly, with persistence, so that coworkers and partners can rely on provided solutions.

CHAPTER 4 DESIGN AND ARCHITECTURE

This role has various definitions over the internet and in books, so I encourage you to go through several of them for a taste of what the role is. Also, be sure to check out my article describing who a software architect is (see [Tengiz WHOISA]).

Regarding professional responsibilities, I recommend that an architect be sufficiently hands-on with code and engineering practices to the extent necessary for guiding teams. This familiarity is necessary to ensure practical, doable architectural solutions. If an architect is not hands-on, then suggested implementation paths may not be feasible and will stay on paper. In some cases, a skilled senior engineer or a technical leader can help an architect bridge the gap with developers. When such a person is absent, an architect must have sufficient knowledge and comfort level with code and implementation. A divide between architects and engineers undermines the entire value of software architecture. Therefore, avoid this pitfall to ensure maximum return on investment (ROI) for your software projects.

At a bare minimum, an architect must be a hands-on modeler and an effective communicator. This person can convert a business and systems architecture vision into design documents and diagrams, which the engineering team then uses as guidelines for implementation.

Is Architecture Dead?

In the modern world of software engineering, *evolving design* has become popular. Many engineers interpret “evolving design” as a development process without upfront investment in design and architecture or a product that gets its structural shape by itself. And if so, you can rightfully ask a question: Is architecture dead if we do not need it anymore? Let me provide my thoughts.

By definition, architecture cannot be dead. It is always there in every software solution, whether we want it or not, intentionally or unintentionally. It is something that is sewed into the roots of

implementation and cannot be changed easily. Therefore, *architecture always exists*. Now, let us answer the second part of the question: Can we build software without upfront design and architecture?

I noticed that many software engineers believe that design is supposed to evolve without any special attention to it. Undoubtedly, *some kind of* design will develop, but will it be the *proper one*? I think that Brian Foote has answered this question well in his article about Big Ball of Mud (see [Foote BBOM]). The referenced paper speaks about the fate of software projects when architecture, structure, and rules are non-existent, and matters become wild. It is evident that without proper control and guidance, the result will be far from what we needed and will not allow the fulfillment of the goals of the system without significant overhead. Therefore, we can conclude that it is *possible* to build software without paying attention to architecture, but it is *not a good idea*.

Let me pose one more interesting question: What makes people wonder whether we need software architecture? The answer is behind the movement from Waterfall to Agile methodologies.

We used to follow phase-gated development in Waterfall processes where design and architecture occurred before development and coding activities. When some teams switched to iterative Agile methods, they thought it was good to drop all steps before development. They forgot to check the difference between Waterfall and Agile, which is as simple as follows: *Agile is running Waterfall phases in iterations*. The Waterfall problem was not that we had too many steps, but rather that each stage happened *only once* throughout the entire project lifecycle. Agile does not mean that we keep only development and coding activities from Waterfall times, and everything else is not relevant; instead, we do *all the same exercises* repeatedly in each iteration. Here is a breakdown of every phase from Waterfall as it appears in Agile:

CHAPTER 4 DESIGN AND ARCHITECTURE

- Analysis and requirements gathering turned into customer interviews, grooming, refinement, and planning.
- Design, architecture, coding, and testing exercises kept their old names, and they happen in every iteration.
- Deployment turned into a release of a product increment.
- Maintenance is an ongoing effort that feeds back into the backlog of tech debt and routine tasks.

As you noticed, architecture is still necessary as it is part of the process. If you do not do design exercises in Agile, then you are not doing Agile.

Therefore, we can conclude that architecture is not dead in modern software development methodologies.

I encourage you to learn about this important topic from a couple of valuable articles: "Is Design Dead?" by Martin Fowler (see [Fowler IDD]) and "Software Architect's Role in Agile Processes" by myself (see [Tengiz SARIAP]).

Brainstorm Before Coding

Often, developers jump to coding without much thinking. Later on, when a technical challenge or inconsistency shows up, the team needs to go back and rework implementation. *This is an expensive mistake* that increases sunk cost and tanks ROI.

You can avoid these circumstances by thinking through implementation ahead of coding. Put a process in place that encourages the team to brainstorm solutions before jumping to code. This course correction may not be as detailed as purposeful architecture and design, but it is a good starting point.

Based on simple math from my experience, an upfront design that takes a day or less can prevent rewrite of code that takes a week or more. Throughout the project's lifetime and as the codebase grows, the advantage of upfront design will increase since the rewrite would impact more parts of a more extensive software system.

Reuse Knowledge

Architecture acts as a knowledge repository for solving various design challenges; therefore, it is essential to maintain this repository throughout the project's lifecycle (and beyond if necessary).

This advice usually means to systematically diagram or document the behavior and structure of existing systems and keep communication channels open between architects and engineers to take maximum advantage of the partnership between them.

While architectural diagrams and models support implementation and code, information flows the other way around, too—from developers to architects—which lets us enrich our knowledge repository with lessons learned for future utilization.

Architecture as a Service

I have been practicing software architecture for a while, and I have tried many approaches to working with development teams effectively. At times, I attempted to micro-manage engineers by giving constant guidance and feedback for every little detail that they implemented. This approach proved to be too cumbersome, time-consuming, and discouraging of creative thinking in developers. On the other extreme, I have tried to completely disconnect from engineering teams by only drawing high-level system diagrams and presenting them to the technical audience without any guidance on how to implement proposed solutions. This tactic resulted in systems that did not conform to an architectural or enterprise

CHAPTER 4 DESIGN AND ARCHITECTURE

vision and solved problems in a less than optimal fashion. An ideal level of engagement with development teams varies from one group to another, and it is somewhere in between the two models that I described.

I currently believe that *software architecture is a service* to other functions of IT organizations such as development, testing, product and project management, and application support teams. This service is available for consumption, and it delivers the best value when the consumer is *interested in leveraging* the offered service. To effectively practice this approach, the following key initiatives must take place:

- The role of software architecture must be clearly understood by organizations and functional areas that might need input from architects.
- A partnership must be established between software architecture and other functions, which fosters open, two-way communication and dialogue. This approach ensures that problems needing the architect's attention are uncovered as necessary and addressed in a professional, supporting manner.
- Instead of micro-managing software engineering and other practices, the architecture team must open up their help as a service by relying on others to reach out for opinions, guidance, and help. Explain to other groups the possible risks that can occur if software architecture is not incorporated while solving critical enterprise-wide issues involving development, integration, or other functional or non-functional challenges.
- Instead of staying in a silo, architects should make themselves available to provide competent responses and attention to every question or interaction

approaching the architecture team. This openness encourages continued open dialogue and partnership between architecture and other business areas that deliver software systems.

I hope that I was able to clarify the meaning behind “architecture as a service.” In my opinion, this approach is the most optimal way of successfully applying the best practices of software architecture to existing problems.

Partnership with Domain Experts

Definition: I will frequently use the term domain in this section and beyond. By this term, I refer to a boundary of a problem space that we can consider in isolation from other subjects. Another way to look at this term is to tie it back to the domains that come from domain-driven design (see [Evans DDD]) and underpin the organizational structure I presented in previous chapters (see “Organizational Structure” in Chapter 2, “Cross-cutting Concerns”).

While software architecture may seem to be an extremely technical domain, its sole purpose is to help business. After all, software delivered by the efforts of architects and engineers must serve business needs.

Earlier in this book, I spoke about bridging the gap between engineers and domain experts, so this concept will not be new. In this section, I want to emphasize the vital role of architecture in this initiative.

I believe that the best software architects put business ahead of technology; they solve business problems first instead of focusing on technical aspects. Then, they connect the final solution with problem space. Your guidance should primarily be dictated by *how a business runs* instead of *what the technology does*. I will distill this architectural approach later in the book. In the current section, I ask to have domain experts available for partnering with architects.

CHAPTER 4 DESIGN AND ARCHITECTURE

In many cases, I have seen software architectures fail, not due to lack of the architect's skills but rather due to the inability to get hold of domain experts. Many architects rightfully believe that design must serve business, but they have extremely limited or no access to domain experts to make that idea work. Instead, they have to take abstract expectations of domain experts and execute them by applying technical concepts and translations from requirement to implementation. In other words, the desire to implement solutions properly is not enough—we need the organization and leadership to be aligned and to help architects be successful in solving business problems. Ideally, the leadership team must emphasize the importance of a partnership between architects and domain experts so that architects do not bump into the wall while trying to gain access to domain experts.

If a domain expert is available to the architecture team, it is time to hold architects accountable for solving business problems with solutions that help run the business instead of complicating it. Since I will be touching on this aspect in many places throughout the book, I will not go into more detail now.

Teams and Microservices

Earlier in this book, I explained that development teams need to group around microservices. The architecture team needs to align the boundaries of microservices to business problems. This step is essential for aligning an organization with a modern, loosely coupled, microservices-based architecture.

Carving the landscape starts by defining business problems (business architecture). Next, architects adhere to identified boundaries when building software systems as microservices (information systems architecture). Lastly, engineering teams will group around microservices to implement deliverable applications (see "Organizational Structure" in Chapter 2).

Architecture Supports Organizational Growth

Software architecture needs to play a supporting role not only when defining a baseline of the landscape in an organization, but also throughout changes that the enterprise faces.

Software architects are accountable for keeping system-wide architecture in a state that effectively serves business and engineering needs. With this mission in mind, architecture must evolve as an organization grows. Therefore, the architecture landscape needs to be examined and restructured regularly to ensure that, at any point, it solves the problem optimally.

One example that perhaps explains the crux of this technique is splitting teams. As an enterprise widens, there is often a need to bring more engineers to the group to deliver more features without sacrificing deadlines. As we all know, above a specific headcount, the overhead of communication and coordination gives diminishing results. Therefore, the team is split into two or more parts. What often happens is that the architecture is kept as-is by putting responsibility on multiple teams to work on the same codebase, service, or application. This solution is hugely suboptimal as it again puts a burden on engineers to manage dependency and communication overhead because they are sharing technical resources. Thus, the problem is not solved at all.

This point is where architecture needs to step up. When considering breaking into teams, brainstorm and find ways to also split the respective architecture (but continue governing all resulting pieces). A single microservice might require a division into two or more microservices. As a result, each microservice must be owned by one team, as I previously suggested (see "Forming Teams" in Chapter 2). *Responsibility for recommending and guiding this change falls on software architects.*

How about exercises that happen before the architectural work starts? One such example is requirements analysis. How can the two activities support each other's success? Let's find out.

Architecture in Analysis and Requirements Gathering

Architecture can enable requirements gathering and analysis in various ways. This section will discuss ways to take advantage of this.

Upfront Design Supports Gap Analysis

In previous sections, we agreed that upfront design is a way to achieve architectural alignment and the application of best practices to a solution before coding starts. Another benefit of this exercise is a technical gap analysis, which is the subject of this section.

Usually, when domain experts present a new feature request, the engineering team asks clarifying questions to ensure a proper understanding of expectations. After resolving queries to a certain extent, the requirement starts to make sense. Now we can roughly estimate the level of effort needed to complete the work and deem the need as being ready for implementation. The next step is taking on the work item and implementing it using code.

It turns out that many gaps in requirements or technical challenges are discovered long after development starts, not before. Typical resolutions to these issues are asking additional questions and redoing work if necessary to course correct. *A rework is an unplanned cost overhead and an additional risk of delayed feature delivery.* Therefore, let us discuss ways to eliminate the hazard.

A straightforward solution to the preceding problem is to think through implementation details before coding starts. In response to this advice, I often hear—"But we already do that, and it is not helping." *Well, do it again, and do it better.* Let's face it: this is a challenging effort because it requires analytical thinking, an environment for an open dialogue, curiosity, knowledge of the problem statement, and the right skillset to

document findings. As soon as you have the proper instruments at hand, you can predict every intricacy of implementation without actually writing a single line of code. Is it too much churn for a marginal benefit?

A comprehensive analysis can be accomplished within up to 10 percent of the entire effort's estimate, based on my experience. In contrast, a rewrite can add another 100 percent of work. I know it is hard to predict the rework frequency, but I can tell you that it is quite common in the development process.

Another benefit of upfront design is its help with functional analysis. If you conduct this exercise in partnership with product management, you can spot feature contradictions and problems ahead of time.

Doing It in Practice

Let me distill a few practical techniques and prerequisites for upfront design, as follows:

- Hands-on modelers (typically architects) must partner with product management and engineering to understand requirements and provide the team with necessary guidelines. Awareness of the end users' expectations is also beneficial for this task.
- Upfront design results in a document that depicts the intricacies of specifications and implementation. The resulting design document is not a blueprint but rather a conceptual model. This artifact can be prepared using UML (Unified Modeling Language) tools (e.g., [WhiteStarUML]) and whiteboard modeling (see "Knowledge Exploration Exercises" in Chapter 3).
- The upfront design exercise usually occurs as a joint session between representatives from engineering (e.g., architect) and product management (e.g.,

business analyst). This session is where you discover and reconcile gaps in requirements. If possible, you create a design document as a conceptual model and a guideline for subsequent implementation and coding efforts.

Knowledge Exploration and Architecture

Earlier in this book, I discussed various kinds of knowledge exploration exercises, such as event storming and whiteboard modeling. While these techniques help bring implementation closer to a domain, they also serve as technical guidelines for architecture and design activities. Let us review both these exercises from a technical viewpoint.

When conducting event storming sessions, the outcome is a repository of events, commands, and aggregates. These concepts directly map into application building blocks with the same names. Specifically, events discovered during event storming will turn into events of event-driven architecture (EDA; see [Wiki EDA]). Commands are an integral part of both EDA and CQRS (Command-Query Responsibility Segregation, see [Fowler CQRS]); and finally, aggregates are one of the essential building blocks of model-driven design (see [Evans DDD]).

When conducting whiteboard modeling sessions, we discover business objects and the intricacies of their relationships that make up the domain model at hand. These elements directly translate into model-driven design building blocks since they are nothing but entities, value objects, and associations between them, all together forming aggregates.

Caveat of a Technical Solution

As a practice, software architecture comes with an interesting caveat, which I want to discuss in this section.

Elsewhere in this book, I brought to your attention a problematic disconnect between domain space concepts and implementation components. I also explained how important it is to keep implementation close to business terminology to lower the complexity level and increase ROI long term.

Since both architects and software engineers are technical people, it is a typical situation that they introduce the described gap instead of bridging it. If such a problem occurs on a project, then having a software architect might complicate matters by contributing to complexity instead of lowering it. Therefore, I suggest that you keep an eye on warning signs and be ready to act with course correction.

One clear indication of the problem is the technical nature of conversations during meetings related to requirements. For example, if you follow Agile processes, look for talks between engineers during grooming and planning sessions that are unclear to domain experts. Understandably, there will be some level of technical complexity involved in every solution. Still, business representatives should be able to grasp the fundamental idea since it involves nouns and actions in business terms. It is a problem when the technical conversation is entirely unclear for domain experts because it lacks elements of the ubiquitous language.

Think about the technical solution as a combination of two layers: one that is clear to domain experts and is based on shared language and concepts, and another that is technical, and only engineering team members understand it (see Figure 4-1).

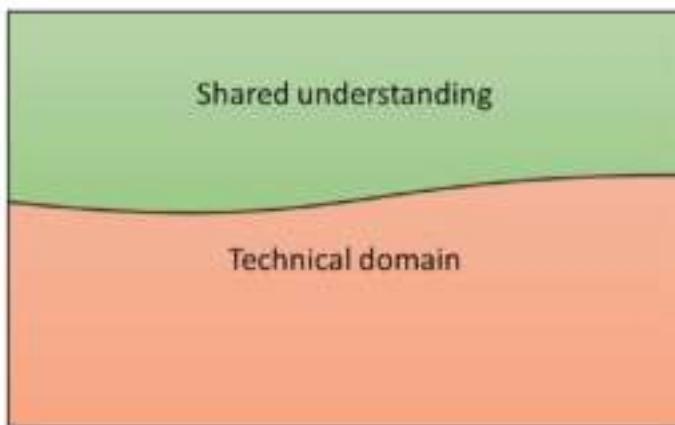


Figure 4-1. Shared and technical language layers

The layer of shared understanding consists of technical building blocks that directly translate into domain terminology: components, modules, class names, entities, value objects, aggregates, events, and even business services. All other elements of the solution might be hard to explain to business people (e.g., message bus, database, or API), and they form the technical layer. To manage solution complexity and maximize ROI long term, architecture and an architect must attempt to shrink the technical layer's size and maximize shared understanding. However, the overly technical nature of discussions between engineers indicates that the technical layer is more substantial than the shared one. A careful software architect who understands the criticality of the problem must be involved in the project to steer the situation to the right side.

If you apply a controlled approach to technical discussions and designs, the technical layer will shrink, and the boundaries of common understanding will expand. As a result, communication, understanding, and knowledge transfer between domain experts and technical team members will improve and positively impact software development pace, quality, and ROI.

When the requirements analysis concludes, it is time to build out the architecture guiding the implementation. This work involves many patterns and principles, so the time is right to discuss them.

Architecture Body of Knowledge

In all other parts of this book, I emphasize the importance of software architecture to achieve effective software solutions. Now I will dedicate some time to explain what software architects need to master in technical terms.

Expect that this section will get technical with deep-dives as necessary into topics that I think are vital for every successful software architect.

As always, I will refer the reader to the material available on the internet instead of explaining a known topic. I will only describe concepts from scratch when I feel that a specific viewpoint provides additional benefits by clarifying common confusion points or conveying my position around a particular topic.

One more general note: In this section, I have collected topics that I consider to be worth your time and attention. Remember that this is my opinion, and thus it is subjective. I can say openly that these topics have helped me many times while architecting software systems, so I am giving them to you from that perspective. I believe that these recommendations can be critical deciding factors for your success, too.

As a result of my subjective preference, some topics that you may be fond of might be missing from this section. Take that as a shortage in my experience, and feel free to give me feedback.

Architecture Landscape

I want you to start thinking about architecture as the shape of the landscape. Start from a very high-level view where you have the entire enterprise broken down into problem spaces; then, for each problem space, you have one or more systems fulfilling the need, just as islands sit in an ocean. Go to a whiteboard and draw your enterprise, organization, or department. Connect lands where systems integrate (e.g., see Figure 4-2).

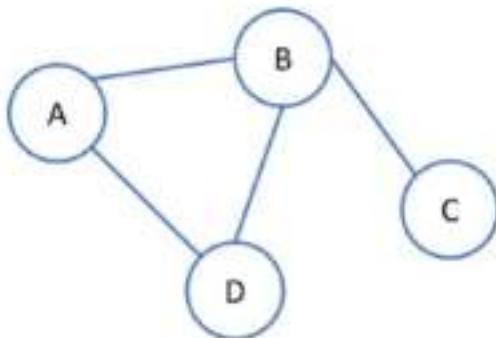


Figure 4-2. Architecture landscape example consisting of four integrated systems

Lines connecting islands represent data flowing through domains, which often prescribes usage or implementation of individual applications. Business typically speaks the language of this data, its attributes, and operations, through which information transforms, slices, and rejoins.

Building out every enterprise happens similarly: understanding business and data, and putting solutions around these assets.

If you stay at this level of the thought process for a while, it becomes irrelevant how exactly each software system is implemented, how difficult it is to build such systems, and where their boundaries are. All that matters is that the entire landscape of solutions fulfills the end-to-end business needs.

Buy vs. Build

Once you have mapped your organization's business landscape, the next question is how to implement each identified system. Two possible choices are buying versus building.

If you are at a small enterprise, buying off-the-shelf software may be an attractive option. Where you need to be careful is the product's ability to integrate with the rest of the landscape elements. Therefore, buying a program from a given vendor must be a conscious decision considering time to market, price, interoperability, integration, and replacement costs.

Remember that a product that is written for the entire segment of the business will not give you a competitive advantage. Therefore, you must compensate for that shortcoming with your ability to execute your unique business idea. Maybe uniqueness is not relevant in your problem domain, and so there is no need to worry about it.

For example, if you are buying a franchise from an established company, you may end up using their software solutions and running business precisely as prescribed by the terms of the licensor. Building a custom solution for only your franchisee instance would not be an investment worth making. Therefore, you do not need custom-built software in such a case.

If the preceding conditions for buying a commercial off-the-shelf solution are not about you, you probably need to consider building custom software.

Here is a critical point: From a high-level view of the enterprise landscape, *it does not matter* how you implement each plot. Either buying or building should fulfill the business need that an isolated island serves. The choice is a tradeoff. A custom-built software solution can adjust its shapes as a problem space changes over time or as companies pivot throughout its existence to remain competitive on the market. With commercial off-the-shelf software, you must give up such flexibility for the sake of cost and time to market.

CHAPTER 4 DESIGN AND ARCHITECTURE

The rest of this chapter—and the rest of this book—assume that you chose to build a custom software solution. I went with that assumption only because it is the book’s topic, and not because I discourage buying off-the-shelf solutions. *There is nothing wrong with either choice.* I have seen and worked with companies that had all commercial off-the-shelf software solutions connected to run an entire business without a single custom-built tool on their landscape (except the integration mechanisms between the systems). Such ecosystems also fall under the field of architecture, but they are outside of the current book’s scope.

Good Architectures

How would you describe *good architecture*? Where would you draw a line between pragmatic and ideal structures? In this section, I will try to prove that good architecture equals *ideal architecture*.

I understand that the world is not ideal, and thus an ideal architecture sometimes is not feasible. However, those cases are more like exceptions rather than a common situation. I want to encourage every software architect and engineer to consider tradeoffs when sacrificing an ideal solution for interim or short-term values. I am not suggesting you give up short-term benefits. Instead, I am trying to say that in most cases you can achieve immediate goodness without an architectural sacrifice. Sometimes technical people hurry to cut corners to reach a specific goal and do not acknowledge that sacrifice is unnecessary. If you try to explore this idea, you will discover that adhering to ideal solutions can be a reasonable and not a crazy thought.

An architecture with sacrifices will bite back at some point. Each sacrifice means technical debt, which slows down planned work and jeopardizes the quality of a product, as we all know from publications such as *Phoenix Project* (see [Kim Phoenix]). Would you want a car with a known issue in its engine? Manufacturing and other fields have already

learned that we should optimize the workflow and throughput by finding bottlenecks. When delivering software solutions, architecture is the backbone supporting the overall effectiveness. It can impact any other delivery element, so it's worth investing our attention in this area.

Give yourself another chance, reconsider your choices, and always go with the right approach. You know when you are cutting corners, so you can stop it from happening. Otherwise, your decisions will come back to you as trouble later, reminding you of past mistakes. I had such a feeling often when a technical debt caused a problem, and I wished I could turn back time. I think most technical decision makers could recall at least one such occasion in their careers. We all have a choice to not let it happen by standing up for good architecture.

Doing It in Practice

Practice has shown that cutting a corner or sacrificing an ideal architecture is never a good idea. I will describe two relevant examples from my experience.

Some time ago, I worked with a company that had many legacy systems. My task was to build a new application that would integrate with existing programs by adhering to event-driven architecture (see [Wiki EDA]). Reaching this objective was challenging for two reasons:

1. Legacy systems did not publish or subscribe to events; all they did was expose API endpoints to retrieve and manipulate data batches.
2. The technical staff was convinced that a new system had to follow the same approach to fit into the existing ecosystem.

CHAPTER 4 DESIGN AND ARCHITECTURE

I knew that batch processing was not the right choice for building event-driven applications (see [Tengiz BPVSIP]), so I wanted to avoid this approach altogether. However, it was tempting to give up and go with the legacy way of doing things since everything followed this standard. How would you solve such a challenge?

Here is an answer: Generally speaking, you need to build a solution in the desired way and then implement an anti-corruption layer (see [Evans DDD]) to shield yourself from conforming to legacy systems.

I decided to build a new system based on events and single-item processing. Afterward, I introduced translation from batch processing results to individual item-specific events within the anti-corruption layer. This segregation of responsibilities allowed combining the ideal single-item processing with the legacy ecosystem of batch processes (see Figure 4-3).

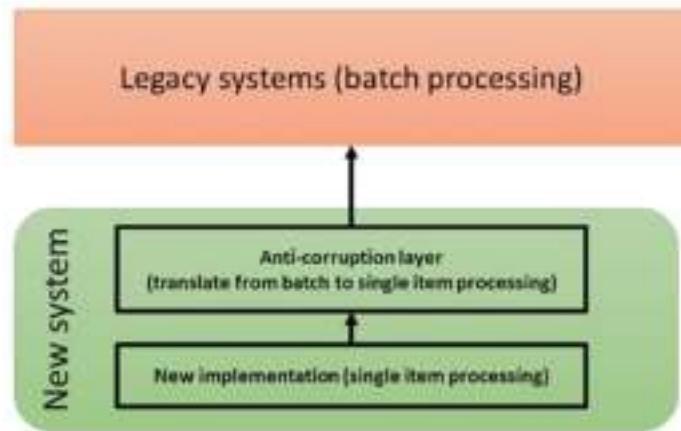


Figure 4-3. Shielding a new system from a legacy ecosystem via an anti-corruption layer. The ACL is responsible for receiving batch results and dispatching events describing each changed item at a time

The preceding example demonstrated that sacrifice was unnecessary, even though most engineers and architects thought there was no way around it. Cutting the corner was tempting, too, as it seemed to fit well into the existing landscape.

On another occasion, I worked at an organization with a history of building hundreds of databases but not many applications. When a new requirement came from their clients, they would always implement it as a SQL job to meet tight deadlines (i.e., a scheduled command execution within a database). At the same time, this organization was trying to evolve into modern microservices architecture (see [Fowler Microservices]), which heavily relies on messaging and API-based integration. Although SQL jobs were the easiest to implement from a short-term perspective, they could not work with events or APIs. Therefore, the old approach was setting the company farther away from their long-term objectives. As soon as the leadership realized the root cause of prolonged technology transformation, they discouraged making a technical sacrifice. Furthermore, project managers reconsidered deadlines to accommodate building ideal solutions and avoiding legacy approaches. In this example, the key was to soberly decide between the two options and choose the one that better aligned with the strategy or long-term plans.

It Is Okay to Postpone Decisions

I have encouraged you to make decisions for the sake of proper architecture when you are on the fence. However, sometimes it is hard to say what an appropriate way of going forward is. In those cases, avoiding conclusions or picking a random path may be the right approach. I suggest that you read my article about postponed decisions (see [Tengiz ADPD]) to be ready for such situations.

Architecture and Technology

Definition: This section will use the terms domain and technology, by which I mean two parts of the solution—the domain layer (if we speak about code) or the problem domain, and the technology supporting it (anything that is not a domain, but is necessary to implement it).

Technology plays a big part in thinking of or building software architectures. I want to discuss these two elements in this section to align our views.

True Value of Software

Let us think for a moment about what makes a software system a valuable asset. One straightforward characteristic is the system's use: a program that meets users' expectations and solves designated problems is invaluable. For commercial solutions, profitability also matters. Let us dig into this area more since enterprise systems, the primary focus of this book, have monetary value most of the time.

In the simplest form, we have the following formula for profit:

$$\text{profit} = \text{revenue} - \text{cost}$$

This formula tells us a fundamental fact: You can sell commercial software, but unless revenue covers costs, it will not help your profitability. Also, consider all definitions of the equation, such as the cost of lost or degraded reputation (e.g., you might have heard about a 2023 Southwest Airlines failure due to technical debt). A big part of this book pays close attention to cost reduction in software development for precisely this reason.

Examples of costs are price to build software, to maintain it, and to host it. Without carefully considering factors like these, you cannot claim that the program is profitable or commercially valuable.

For example, an SaaS (Software as a Service; see [Wiki SaaS]) can produce \$100 in income per month. Is it a valuable asset? Probably yes, since it delivers revenue. Let us add to the story that its implementation requires \$1 million, and it costs around \$200 per month to host it in the cloud. I think those additional inputs make this SaaS not so valuable anymore.

Therefore, the real value of the software system is based on usefulness and its development and maintenance costs.

This conclusion is an important milestone before I continue to subsequent sections. Next, we will see how technology choices impact the value of the delivered software.

Is Technology King?

I often hear that “technology is a king” in software architecture. I need to disagree with that. Technology comes and goes as it becomes stale. It cannot be king in the game, because it frequently loses value.

When technology becomes stale, and we want to replace it, we usually rewrite software systems. That action is a significant investment that drops down ROI for any software development project. If you need to rewrite a program every couple of years, that cost overhead drags down the system’s overall value. Recall the previous section to understand my point—every cost overhead negatively affects profit.

If you followed my thought process, we could agree that technology is not a king; instead, it is a risk that can bring down ROI for software development projects. While I will explain how to deal with this technology aging issue later, let me first show you the real king.

Domain

My suggestion is that *domain is king*. Let us look at what I mean by that.

CHAPTER 4 DESIGN AND ARCHITECTURE

The domain is how business works and runs; it is what makes us unique and is our competitive advantage on the market. In software systems, it is an area of codebase that holds business logic. That part of the program is the most valuable asset of the software system.

Let us go back to the definition of a valuable software system that I presented earlier: *The real value of the software system is based on its usefulness and its development and maintenance costs.*

Is the domain useful? Yes, it is. It is the only piece of code that talks about our business and competitive advantage rather than technology, which is a temporary occurrence. Does the domain require continued development and maintenance efforts? Not more than the evolution of the business that it serves. If the problem area is already well developed, and we are not changing business processes, the software's domain layer is probably timeless. While technologies come and go, the domain stays as it is. It is the least amount of investment that delivers the most value.

What does this conclusion mean for software engineers and architects?

- Always isolate domain layers from infrastructure and concrete technology code via class libraries, packages, and programming against interfaces. This separation helps us see the domain without being distracted by anything else. Also, we can replace everything else (including technology) and keep the domain. So, separation is key.
- Isolation also helps us achieve the durability of the domain layer's code. For example, if the system is built in a vanilla C# programming language, we can keep it as-is when runtimes (.NET), libraries (FCL), frameworks (Entity Framework), or databases (SQL vs. NoSQL) evolve. Vanilla C# code does not need to be modified as long as it does not depend on technologies that need to be upgraded. In a worst-case scenario, you

might need to change project properties to target a new compiler, which will be it since more modern compilers are compatible with older versions. A similar approach takes place for Java and other languages too.

- When technology changes, you need to implement interfaces again, targeting newcomers to the area (e.g., query NoSQL instead of querying SQL). Those changes happen outside of the domain layer's code. In other words, interface implementations stay unchanged within the domain layer, while you alter their implementations residing outside. This point proves again that technology comes and goes, while the domain remains as-is.

These conclusions are depicted in Figure 4-4.

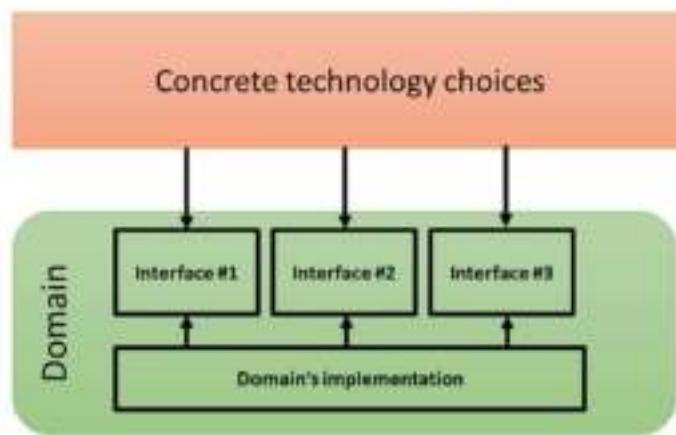


Figure 4-4. Domain isolated from technology choices via interfaces (direction of arrows describes usage, reference, or awareness)

Using Technology

If you have followed my thoughts so far, then we agree that technology must be a second-order question after the domain. Nevertheless, technology rarely leaves the picture entirely. Any software system relies on technology to an extent, and we cannot ignore that fact. Instead, we need to master techniques of proper usage of technology to support software systems that we deliver. As Chinese military strategist and philosopher Sun Tzu suggested, you must know your enemy (see [Tzu TAOW]).

In this section, let us discuss approaches and cautions related to technology selection and usage in software solutions.

Vendor Lock-in

Vendor lock-in or vendor lockdown are terms to describe how it is hard to replace one vendor with another. It is easy to get into this trap if we do not stay sober when building architectures around various external vendors' technologies. Always ensure you have the ability to opt-out of a given vendor's services. What will it take to achieve independence should you decide to do so? Without such considerations, you can hinder growth opportunities for software solutions and thus for the company.

An example of a typical vendor lockdown is the use of Amazon Web Services (AWS), which is heavily utilized by thousands of companies to run their solutions in the cloud. Challenges face these consumers when they decide to switch to another cloud vendor (e.g., Microsoft Azure) if the pricing offer is better. Another set of companies may prefer to host their solutions on their own premises and struggle to make this move because their code is intertwined with Amazon's platform. Both kinds of challenges describe vendor lockdown.

Here are some examples of using Amazon's offering against your long-term benefit only to find yourself locked into the vendor:

- Tightly coupling with Amazon's proprietary services (such as SNS—Simple Notification Service) throughout an entire codebase instead of abstracting notification capabilities via interfaces. This warning relates to both the usage of Amazon's code packages (e.g., Nuget) and hardcoding their API addresses. When the time comes, refactoring the entire codebase is more laborious, while implementing a single interface would be straightforward.
- Utilizing Amazon's proprietary cloud web interface as an IDE (integrated development environment) by writing and running code via those screens exclusively (e.g., Amazon Glue and Lambda services support this technique). When the time comes, you will discover that you cannot quickly turn such code into an independent application as it can not execute outside of Amazon's cloud environment.

Both examples lead to vendor lockdown. I am not advertising against Amazon, but I want to increase awareness of the consequences that a particular way of cloud consumption brings.

Unavoidable Level of Technology

You cannot entirely ignore the existence of technology, no matter what. When designing systems, you usually remember the programming language that you will use for the application as well as the storage that will hold data, which is technology already. Furthermore, these aspects affect how you architect the program or set non-functional expectations.

CHAPTER 4 DESIGN AND ARCHITECTURE

Nevertheless, I recommend limiting the effect of technology choices on your architectural decisions as much as you can. I briefly mentioned this concern earlier, but I want to clarify further to avoid confusion.

Here is a universal rule of thumb: It is alright to assume that you will use a specific programming language (C# versus F#) and its paradigm (object orientation versus functional programming). If you can stop at this selection and abstract away everything else, you will do a perfect job as a software architect. By abstraction, I mean that you must program against interfaces. This technique is always feasible, no matter what you are trying to build. When the time comes, it is easy to substitute a new implementation of the interface, so you are future-ready.

Here are a couple of examples for this technique:

- A repository interface can abstract storage or database.
- A service contract or a proxy interface can abstract service invocation.
- An interface can abstract even current date and time for testing purposes (interface can be mocked and injected when running tests or executing in a production environment).

Abstractions Can Help

I briefly touched on abstractions as a way to ignore concrete technology selections and avoid vendor lockdown. Programming against interfaces has other advantages as well, as follows:

- If you need to upgrade the technology version behind an interface, you can do so without significant refactoring of the application since the impact is isolated and measurable.

- If you want to alter technology usage (e.g., to utilize a new version of the API), you only need to implement an existing interface differently.
- Technology does not pollute the domain model as the two parts stay separated by interfaces. This approach separates responsibilities, and it allows the identification of essential pieces of the solution from everything else.
- If you are not sure which technology to use, just put it under an interface and go with the most straightforward option. Come back to the choice later and substitute a more comprehensive implementation as necessary.

Beware of Technologists

Being surrounded by technologists can bring you back to the foundational problems described in this chapter. These people are believers that technology is king, and they usually dumb down the entire solution under a given tool. That imposes risks by incurring unnecessary overhead of complexity. Delivered solutions center around how the technology works instead of focusing on how the business needs to run. This path is a trap that can lead to you losing a competitive edge on the market, so make sure that you notice it early enough to course correct.

Consolidate Technologies

Every enterprise needs to be mindful of the technology choices that they make. While it may be exciting for engineers to use new technology for every new project within the company, this experimentation gives diminishing results long term. Specifically, you will need to maintain

multiple technologies and increase the learning curve for newcomers who might need to touch several projects while staying with the organization. Such overhead can go against the company's objectives to optimize processes and resource utilization, and it can decrease ROI from software development.

Therefore, look for opportunities to consolidate technology choices for both existing projects and new initiatives. Replace proprietary options with more wide-spread tools and follow existing conventions instead of introducing new players. This simple practice should bring down unnecessary overhead and costs for the learning curve, maintenance, licensing, and support of technology products used in the company.

Domain Model Kinds

Previously, I stated my preference for putting domain above technology and infrastructure concerns when implementing software systems. However, we can express domain in multiple ways, and not all of them have the same quality characteristics. Therefore, in this section I want to provide more guidelines on possibilities.

Rich Domain

A rich domain model is the preferred form of implementing domain layers in software systems. There are many good explanations of rich domains in the literature (see [Evans DDD]) or on the internet (see [Tengiz Modeling], [Tengiz HTDBO], and [Tengiz IOM]). For that reason, I will not explain it in this section.

A rich domain model helps achieve maximum benefits out of delivered software systems because this approach tackles complexity in code and therefore brings down maintenance and enhancement costs in the long run.

Anemic Domain

An anemic domain model is considered an anti-pattern in the software architecture area of knowledge. It sits on the opposite pole from the rich domain. You need to know what it is to avoid it at all costs, so please make sure that you learn about it in the available literature (see [Evans DDD]) or on the internet (see [Fowler ADM]).

The anemic domain model is one of the primary enemies of long-running software projects. It promotes mixing technology with a domain or spreading knowledge thin throughout the entire codebase of the application. Therefore, programs implemented in this manner will not help software developers to quickly gain domain knowledge or feel comfortable with making changes to the existing implementation. As a result of this shortcoming, affected software projects will show signs of a slowdown and low quality, which decreases ROI and customer satisfaction.

Big Ball of Mud

The term *Big Ball of Mud* (see [Foote BBOM]) describes software solutions that are hard to understand, maintain, enhance, or run. These programs give diminishing returns on investment, and therefore I recommend avoiding this style of development and architecture at all times.

Domain-Driven vs. Data-Driven

This terminology is another way of describing a preferable approach from a less preferable one. Rich domains result from domain-driven designs, while anti-patterns arise from data-driven models (data-driven can be good or bad, depending on the context. See [Tengiz DDGOBI]).

Other Anti-Patterns

There are undoubtedly other anti-patterns of domain modeling of which you need to be aware. Since variations and their explanations stand close to each other (just like Anemic Domain and Big Ball of Mud), I will not mention all of them—it is not even possible. I only want to warn you against anything that is not a rich domain model.

Use your judgment when deciding to go with one or another domain modeling principle, but please familiarize yourself with both pros and cons of any given approach.

Life with Anemic Domains

While I advertise rich domains, the reality is that most of the software systems I deal with have anemic models. I am sure that this is a typical case around the globe, so we must know how to live with this reality. There are a couple of techniques that have helped me throughout my journey of working with anemic domains, and I want to share my experience with you.

Firstly, a rich domain takes advantage of strong object-oriented principles. It is a violation of these basic rules that makes a model anemic. *Therefore, I conclude that anemic domain models qualify as procedural code.* In simple words, this anti-pattern shows itself via a lack of encapsulation and abstraction. Code deals with data values by moving them from one object to another without designating responsible entities for the consistently changing state. Repeatedly applying this approach results in having many components that modify data and pass it around, which over time turns into containers of spaghetti code (see [Foote BBOM]). Therefore, another conclusion is that *spaghetti code is a characteristic of an anemic domain model.*

When a large codebase is built using an anemic domain model, there is not much that you can undo. At times, it makes more sense to follow anti-patterns because otherwise you risk introducing another style of development into the codebase, sacrificing consistency and unification. Arguably, it is easier to maintain a single way of doing things (even if it is an anti-pattern) than having to deal with two distinct styles of development (half anti-patterns and half done right). *Therefore, turning only a part of an anemic domain into a rich domain can give diminishing results due to mental and maintenance overhead.* There must be something in between the two extremes.

From my experience, a middle ground is an expressive code, intention-revealing interfaces, and declarative design (see [Evans DDD]). It is easy to understand why I picked these techniques: you cannot do much with spaghetti but to add a tasty dressing, chop it into digestible pieces, and organize these bits. While the resulting solution is far from a rich domain, it still gives benefits such as readable code and language close to domain specifications.

For example, if we have a spaghetti code that does about a dozen different tasks in a row, we can chop the method into those pieces and give each extracted part its name from business domain vocabulary. Next, consider implementing a specification pattern (see [Evans/Fowler SPEC]) or another technique of declarative design. Technically, we represent *operation* as an object rather than encapsulating *state* into an object (which would be the case with rich domains). So, instead of having spaghetti in code, you end up having a set of small objects, each representing a step of the operation. A caller function puts them together by either fluent interfaces (see [Fowler FI]) or adaptive models (see [Fowler RTAAM]).

Once you finish the preceding exercise, give yourself some applause: you just gained benefit from spaghetti code and increased ROI for the codebase by simplifying somebody's life on a team of engineers. Declarative design helps shorten the learning curve, express the domain, and bridge the gap between technical developers and specialists of the problem space.

Layered Software Architecture

I think you have heard about layered architecture of some kind. Nowadays, most applications are built by utilizing layers. If you are new to this concept, make sure to become familiar with layers by reading about this topic (see [Wiki MTA]).

Layers help separate concerns in a codebase by providing guardrails to engineers when implementing requirements. *Guardrails* means built-in guidance, less confusion, and more time to focus on more important things such as building and delivering features. Also, layers are part of an established language that helps communicate across technical personnel any plans for implementing code or placing individual components under specific packages or projects of the solution.

Because of the mentioned values, I recommend that you think about layers before you start writing code. When you have a plan in place, you are free to start development because you have established sustainable guardrails.

Popular Application Layering Patterns

Layers are crucial to application development so much that most of the existing application frameworks rely on layers by either implementing a given layering schema or even taking its name. For example, Microsoft's MVC framework, which exists in many variations, relies on the MVC layering pattern; i.e., it tries to split the codebase into model, view, and controller layers (see [Wiki MVC]). Other application frameworks, such as Microsoft's WPF or WebForms, rely on or recommend using layering patterns such as MVVM (see [Wiki MVVM]) or MVP (see [Wiki MVP]).

If you decide to use an application framework that does not recommend using a specific layering schema, give it some thought on your own. Ensure that you have a plan about your future layers because it will impact other values, such as having an isolated and clearly defined domain boundary.

How Many Layers?

What do application layering schemas have in common? The number of layers. It is always three!

This characteristic is not a silver bullet, and there can be rare exceptions to this rule. However, it is a rule of thumb to keep the number of layers down. Depending on your preferred approach, you might have up to five layers, but that is a maximum. If you have more than five layers, then the chances are that you are overthinking the solution or inventing layers that should not exist. While creativity is a good thing in general, in this case it works against effectiveness and productivity. Specifically, you introduce new and uncommon terminology and language that others will not understand or will not accept. In the best-case scenario, your unique approach will become a new learning curve that slows down development and requires specialized training. The software development industry has already gained enough knowledge and experience to claim that you will never need too many layers. Therefore, consider adhering to the expertise of the sector before inventing a new layering schema.

I have written about layers and precautions related to them in two articles, which you will find on my website (see [Tengiz GTLAA] and [Tengiz LSA]).

Other Popular Layered Architectures

Earlier, I mentioned a couple of popular layering schemas. In addition to those patterns, I also want to recommend that you become familiar with Onion Architecture (see [CG OA]) and Hexagonal Architecture (see [Wiki HA]). These two patterns are slightly different from previously introduced ones due to the following:

- Onion Architecture can result in five layers as opposed to the most commonly encountered three layers.

- Hexagonal Architecture is a pattern that helps look at other layered approaches in a more structured way: on a higher level, from a so-called “ports” and “adapters” standpoint.

Microservices

I have finally reached one of my favorite topics and the basis for many passionate discussions nowadays—microservices. I believe that this architectural trend will be mainstream for many years to come, so I want to address it in this section.

Clarification of Terminology

I have often heard from technical people that a microservice means a small service. This assertion is not necessarily accurate. Here is why:

- While *micro* means small, in the case of microservices, this only refers to a size relative to previously popular SOA (Service Oriented Architecture) services. Historically, the motivation of the microservice architecture was to define finer-grained systems than the previous generation of SOA services. The corresponding bounded context drives the actual size and boundaries of a microservice, and this chopping exercise is not a “micro” level at all.
- As we all know, a “service” often means an API with endpoints, which we can invoke from the program’s code. Although a microservice sounds like a service, it does not necessarily need to be an API. In this case, *service* means “capability.”

Make sure that you understand these clarifications to avoid falling into the trap of designing poor microservices.

Implementing Microservices

As I briefly explained before, a microservice is an implementation of a bounded context from DDD. It is time that we delve into the technical intricacies behind this statement.

A microservice is a cohesive implementation consisting of all components that solve a specific business need. These components can include APIs with their endpoints, messaging middleware, long-running processes, jobs, console applications, UI applications, a database, and so on. Indeed, a single microservice can have one or many instances of all these components and still be a microservice. This richness is what I meant when I said that a microservice does not need to be small in size. Elements forming a microservice solve a single business capability. That common goal is what unites participants of a microservice under one boundary (see Figure 4-5).

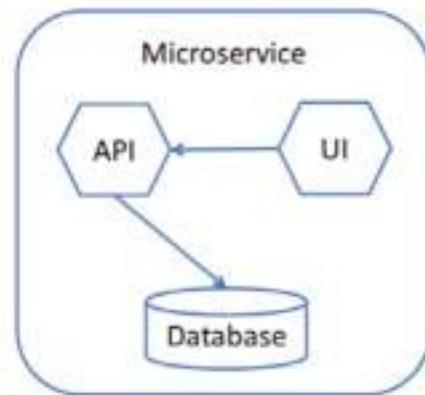


Figure 4-5. A typical microservice example consisting of a database, API, and UI

For example, a “shipping” microservice can include API endpoints to get shipping cost estimates, submit shipping requests, and retrieve the latest shipment status. This same microservice will probably have a database that stores all shipment requests and their states. Additionally,

an administrative UI application can allow uploading of all supported zip codes into the shipping microservice's database. All mentioned components *together* form the "shipping" microservice.

Integration

Microservices integrate through their APIs or messaging layers (the latter usually happens within enterprises). All other approaches, no matter how strictly you maintain the contract, have weaknesses and side effects. Directly accessing a database or other internals of another microservice is an anti-pattern.

Restricting microservice integration through their APIs has proved to be effective in companies like Amazon (see [APIE AIAP]), so make sure to read success stories about this technique and take lessons learned.

Autonomy and Scale

Technically, each microservice is an independently deployable unit. This fact makes the entire ecosystem of connected services more reliable. As long as integration contracts are maintained and followed, it is possible to swap out one microservice with a newer version without affecting others.

A separately deployable unit also means the development processes and autonomous teams are independent. This benefit decreases the overhead of unnecessary alignment and coordination when working on shared codebases.

Furthermore, an independent deployment unit can scale better than a monolith can due to granular resource usage. For example, a shipping microservice might require more parallel instances than payments as more users will reach the shipment options page before even paying.

Apart from a technical scale, you can achieve better human force multiplication with autonomy. For example, processing payments can be a more complex and vital domain than shipments, so you could increase the critical team's size without impacting other groups.

Once you gain benefits from autonomy, do not allow it to weaken. For this, ensure that any given microservice is owned by a single development team to avoid stepping on each other's toes. I already recommended this technique earlier in this book.

Scale Out

Scaling horizontally is the preferred way of doing force multiplication. Due to its autonomy, you can scale any microservice independently from others. A horizontal scale is achieved *within* the microservice while maintaining integration contracts with other systems.

For example, to scale a service within a microservice, the load of incoming requests needs to be distributed across all instances evenly, which requires infrastructure setup. A URL that consumers invoke should not change when additional instances of a service are added to the network. You can do this by putting all instances under a DNS name, a load balancer, or a reverse proxy. Containers and their orchestration systems take a similar approach, so no matter how exactly you host a microservice, this model can always help (see Figure 4-6).

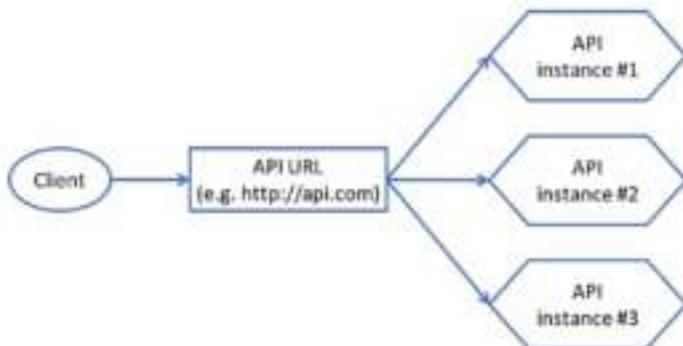


Figure 4-6. Scaling API horizontally without affecting consumers, by directing traffic to multiple instances

Reusability vs. Duplication

Before I explain the problem, let me clarify the terminology as follows:

- *Reuse* refers to the fact that a single microservice is the owner of data and behavior related to a specific business capability. All other microservices need to call the designated service and only the designated one to acquire data or execute related actions.
- *Duplication* is a situation where the same data and behavior are present within multiple microservices. Implementation of each such system is more straightforward in isolation since all necessary data is immediately available, and no integration is needed.

These two extremes often compete when designing microservices. Let us discuss tradeoffs and understand how to employ each technique at the right time.

Microservices are reusable components by nature. The built-in scalability of a microservice ensures that it can handle increased loads when necessary, and thus consumers can rely on its availability without duplication. Despite all these promises, not everything is reliable; network latency, lost connections, internal server failures, and retry strategies come to mind. How do we deal with reliability within the ecosystem of connected microservices? How do we avoid the collapse of the entire ecosystem due to errors within an isolated microservice?

The answer to this dilemma is that reusability has its limits, and we need to acknowledge them. Regardless of availability levels, do not rely on the successful execution of every request. A service that looks healthy can fail to process the request and respond with a "500" (internal server error) status code due to code defect. To overcome these challenges for critically essential requests, plan for delayed processing, and let users submit their input into an application.

How do we take data of another microservice that is down and not lose it? We simply keep it within our microservice temporarily and retry processing later. This pattern often causes discomfort to engineers because it suggests duplicating foreign data. Things are not as bad as they sound since this approach is a combination of two well-known patterns: *CQRS* (see [Fowler CQRS]) and *eventual consistency* (see [Wiki EC]). The first technique recommends splitting commands from queries, while the second one teaches to delay data transfer between systems. In a nutshell, this is what I suggested to do.

To summarize, *if the reliability of request acceptance and processing is critical, design systems for intentional data duplication and delayed execution.*

I want to clarify an important point when taking the suggested route: Always remember which microservice is the *system of record* for the data you hold. This acknowledgment will affect how you design request processing and data storage components. For example, if you are generating payment requests and cannot submit them to the respective service, remember that a unique identifier of record (such as payment reference number) still needs to be generated in the payments microservice. Keep room for that and do not assign identifiers yourself; you are not the system of record; you only hold data temporarily.

Since we weigh reusability versus duplication, the same tradeoffs apply to the engineering groups that build systems. Each microservice's team should have the autonomy to make decisions and implement components that they need without being tied up by other teams' work to reuse some parts. It is preferred to give each side autonomy to accelerate by duplicating efforts instead of tying them down to reuse components in every possible case.

Based on where the industry moves, overhead between teams caused by coordination needs costs more than an obligation to maintain duplicate components. Boundaries of microservices already provide separation to minimize the intersection of groups, so duplication should not be so bad. Mainly, it is a matter of acknowledging that both implementation and development processes need to be conducted differently from the past.

Evolving an Ecosystem of Microservices

As I mentioned earlier, microservices are an evolution of SOA. SOA services enable the structuring of entire enterprises around them. Similarly, microservices can evolve into a connected ecosystem within or across organizations. Therefore, it is utterly essential to know the basic principles of building microservices. I recommend that you check out resources online that teach the concept of microservices and related patterns (e.g., see [mslo]).

In this section, I want to cover a couple of vital aspects of building and evolving an ecosystem of microservices. Specifically, how should we build UI in front of existing services? Do we go with monolithic UI (see [MS MUI]), client-side UI composition (see [mslo CSUIC]) or server-side page fragment composition (see [mslo SSPFC])? Since each of these approaches is valid in certain cases, I want to discuss them in more detail by covering their implementation intricacies.

Monolithic UI Backed by Microservices

Monolithic UI (see [MS MUI]) is the most straightforward approach for teams that build microservices. However, from an architecture standpoint, I would recommend staying away from this pattern whenever possible. You will incur communication and integration overhead between a team that builds a monolithic UI and all other groups that create backing microservices. You have a single point of failure and a bottleneck in the face of the UI team. While you may smooth out integration overhead, conflicts of interest will always get in your way: The UI team tries to build the optimal user experience. In contrast, backend services teams want to implement technically optimal API endpoints. Although the combination of these two objectives sounds like an ideal solution, underlying driving forces pull implementations in different directions, making it difficult to integrate or deliver optimal solutions.

For example, a backend microservice team might decide that they will return a complete data set, and UI could do necessary sorting, filtering, and paging afterward. This technique makes sense if we look at API development in isolation—the resulting endpoint would be highly reusable for all kinds of consumers. Now, look at the situation from the UI team's standpoint: They want to make the user experience as lightweight as possible, without fetching a single unnecessary data bit. Querying a complete data set and then narrowing it down in the browser goes against the objective of the optimal user experience. Additional data sitting in the browser's memory might jeopardize the goal. In this conflict of interest, one or the other side needs to compromise.

I do not doubt the feasibility of a compromise. Instead, I am worried that the arrangement will be based on secondary factors such as the relationship between teams instead of more vital aspects, such as the effectiveness of the user experience. Therefore, when possible, avoid following a monolithic UI pattern to avert adverse effects on customer satisfaction.

Client-Side UI Composition

This approach (see [msio CSUIC]) can be a golden mean between universal and easy solutions. Each microservice-building team is still responsible for rendering its front-end, and there is a common platform that simplifies this objective with compatibility and reuse.

For example, a framework or library (such as Angular or React) can help define a common platform and patterns for shaping the base component structure; next, each team adheres to established standards and expectations by delivering their components. The common platform ensures that every element can successfully plug into an overall skeleton.

Perhaps you will need a *temporary* platform team for building a shared skeleton. Down the road, those feature-specific groups that need additional capabilities in the base structure can take responsibility to put

functionality in place. As I explained earlier in this book, I have rarely seen platform or infrastructure teams delivering real value. They quickly lose the context of feature teams and are unable to be effective in a silo; hence, I suggest only to assemble a temporary group with such a mission.

Server-Side Page Fragment Composition

This pattern (a.k.a. Micro Frontend) is the most sophisticated approach to integrating microservices for UI composition. Each team would need to master advanced front-end engineering skills to deliver on this objective. While this approach is ambitious, consider your talent consciously before jumping into work. To make this pattern more attractive, this approach is the most universal.

Each microservice needs to expose an API endpoint for each page or widget with data. All dependent services consume these endpoints instead of developing UI themselves. In simple integration scenarios, we can redirect the user to the page of the microservice. In more advanced cases, we may need to place a widget of a microservice on a separate page. Let me explain both examples.

Suppose an e-commerce website has both shipping and payment pages. On the payment page, you can change the shipping method as it affects the order total. We can achieve this experience by implementing a server-side page fragment composition. The "shipping" microservice entirely renders the shipping page. When the user continues to the next step, the browser redirects to the payment page, which the "payments" microservice generates. The necessity to display shipping options on the payments page hints at a dependency between the "payments" and "shipping" bounded contexts and thus microservices. "Payments" are downstream, and "shipping" is upstream. The "payment" microservice asks the shipping service to render the shipment widget for the payment page. The shipping service's designated API endpoint will return the requested component either as an HTML fragment or as a JavaScript code

block. When a change to shipping options is detected, the widget informs the payments page about this event, which changes the order's total amount based on the new selection.

Without going into more detail, you can imagine how intricate this implementation can be to support interaction between the payments page and the shipment options widget. I am not even touching the topic of consistent styling and design between the widget and the container page. This additional complexity is what makes the approach less desirable for teams without enough experience or expertise.

As I suggested before, this pattern is a universal way of building an ecosystem of microservices, so I recommend considering it when possible.

Command Query Responsibility Segregation (CQRS)

While I do not intend to explain the CQRS pattern from scratch (read [Fowler CQRS]), I want to give a couple of hints for implementing solutions while applying this pattern. Keep the following in mind:

- Combine CQRS with rich domains for the best results. I earlier explained what a rich domain is, and I now want to recommend that you use both patterns when applying the CQRS pattern to the solution.
- Query objects (a.k.a. read models) are also part of a rich domain. While these sound like data containers, they are a result of extraction from domain entities. In other words, domain entities are a source of truth when generating query objects out of them.
- Query object composition is a domain concern. If your application relies on the read models to display data on a screen, then probably the correctness and

consistency of those models are essential for your software system. Such a vital problem needs to be part of the domain whenever possible.

- Represent three classes of problems in the domain layer—entities, query objects, and the transformation operations between them. Highlight ones that have critical importance to success. Usually, when you go with the CQRS pattern, it is probably the case that all three are vital for your solution.

Let me distill everything that I suggested to see how to do it in practice.

Employ Rich Domain to Generate Read Models

A rich domain will guide you toward having entities and value objects in your codebase. Those objects are a source of truth at any given time. However, since you will consider applying the CQRS pattern, I assume that stale read models are acceptable to display on a page. Therefore, query objects are returned by API endpoints so that pages can bind to the response. When you separate a query object to keep stale data in it, you are entering the world of a tradeoff and special treatment: while query objects can be outdated, they still need to be *correct* at a specific point in time.

For example, if the query object holds flight details, data must be accurate, even though it represents stale information; i.e., a *consistent state in the past*. Since the correctness of the read model is essential, we need to use proper modeling techniques for aggregating the model. A couple of patterns to explore in this area are intention-revealing interfaces, declarative design, fluent interfaces for readability, and value objects representing various data bits that help shape the final read model.

Avoid Vendor Lockdown

Avoid depending on the concrete platform to generate query objects. For example, I once saw an implementation where read models were a result of a SQL database job. Such solutions represent vendor lockdown, obviously, and they will become a rework overhead if you decide to switch storage technology in the future. This additional effort is technical debt that decreases ROI from the solution.

Instead of relying on a concrete database or vendor technology, make sure to write code using a rich domain model responsible for generating query objects. Such code blocks are vendor-independent and can port to another storage technology when needed. Besides avoiding vendor lock-in, an additional benefit is that transformation written as program code can be unit-tested and can turn out to be more comfortable to maintain and extend over time. Therefore, the approach that I suggested in this section will improve the overall quality of delivered programs.

Path to Event-Driven Architecture (EDA)

As you will notice, the software architecture industry is moving toward ecosystems of microservices. Each microservice serves a specific business capability and therefore manages related data elements. If we look at the entire set of connected systems under medium- or large-size enterprises, it is natural to ask questions about data consistency across an organization.

CAP theorem (see [Wiki CAP]) suggests that we need to give up either real-time consistency or availability as we cannot have both at the same time. Most modern organizations have realized that the availability of services is of utmost importance, so they consciously sacrifice real-time consistency. *I highly recommend that you consider doing the same whenever possible.* If you follow this path of thought, you will need to explore eventual consistency (see [Wiki EC]) and event-driven architecture (EDA; see [Wiki EDA]).

CHAPTER 4 DESIGN AND ARCHITECTURE

In simple words, EDA helps achieve eventual consistency across the ecosystem of microservices. Each state change in a given microservice is communicated to dependent services by publishing an event that describes what happened. Microservices that subscribe to a given event will react to messages by applying changes to dependent data elements. The received event will explain what happened and to which entities within an upstream microservice. A downstream service can retrieve additional information about the event from upstream via further API invocations if necessary.

If you apply the described technique across the entire enterprise, you will see a graph of connected services and their data elements that react to changes in a chained fashion. At any point in time, all data elements are almost consistent with the possibility of a short delay (see Figure 4-7). You need to target this state!

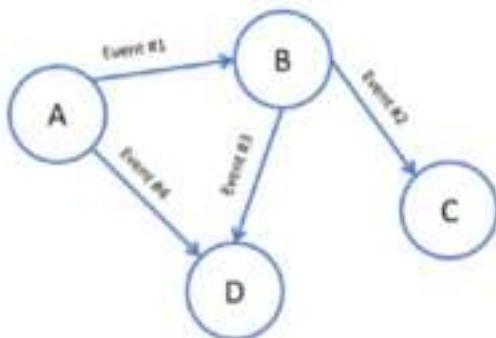


Figure 4-7. Microservices connected via event-driven architecture (EDA) when various events are sent between connected systems

From a business standpoint, EDA helps discover important events and flows that involve multiple microservices and their respective data elements. In many cases, your domain experts can understand each event's meaning and can describe them in knowledge exploration exercises such as event storming. These benefits make a strong case for applying EDA to implementation when possible.

Building Cloud-Ready Applications

In this section, I will go over a couple of essential topics for building applications that need to run in the cloud.

Twelve-Factor App

A twelve-factor app (see [12factor]) is a methodology for building applications for the cloud out of the box. I highly recommend that every software architect be familiar with this set of principles.

Containerization

Containerization (see [Wiki Containerization]) is a way of designing and running an application as an independent unit that behaves the same way irrespective of the environment around it. In other words, you build a container once, and it produces the same results every time you deploy it. This behavior is a remarkable achievement for application quality as it eliminates most of the environment-specific technical issues that cause unplanned work and delay planned feature deliveries.

Besides quality improvements, containerization enables horizontal scaling, which is crucial for every modern software system.

A practical way of applying containerization to an application is to learn and utilize one of the container platforms such as Docker (see [Docker]). The good news is that all container platforms follow the same principles, so it is not hard to understand the overall idea if you try only one of them.

I will not go into further details about containerization. Now that you understand its benefits and have an idea of where to start, you can take it from here and work on this notion on your own.

Avoiding Cloud Vendor Lock-In

In earlier sections, I raised a warning about experiencing vendor lock-in if you are not careful with cloud hosting offerings. However, at the same time, I am encouraging you to get your application into the cloud. How do you achieve both these objectives at the same time? There are a couple of strategies that you can take.

The first and most straightforward strategy is to play safe and go to the cloud only with containers. In other words, apply containerization to your application before taking it to the cloud. This technique ensures that your solution will not become tightly coupled with proprietary cloud hosting infrastructure, and thus you can switch to another vendor down the road. The trick is that the container works and runs the same way in any vendor's cloud.

Another approach is not so straightforward but still doable. The solution will require learning a specific proprietary offering and designing your application carefully to minimize impacts. The key is to layer your program so that cloud-specific mechanisms are invoked from the thinnest edge (outermost) layer, which calls into other platform-agnostic code. This way, all your layers except a single layer can be ported to another vendor when necessary.

For practicality, here is one example to clarify this advice: Amazon's lambda function can be implemented as a C# project (or Java package), which references other class libraries and invokes methods in them. A project that is specific to lambda should not have much logic in it except to call methods in vendor-agnostic class libraries (see Figure 4-8). Suppose you decide to go to another cloud provider in the future; in that case, you can easily replace the lambda project with another entry-point project that calls into the same vendor-agnostic class library's methods. On another extreme, if you write code directly in the lambda function's console on AWS pages, you will end up being tightly coupled with other proprietary capabilities.

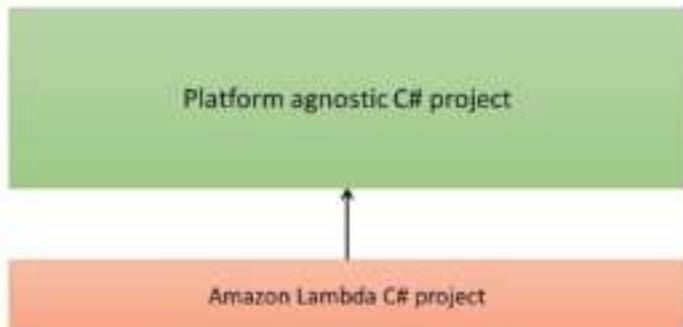


Figure 4-8. Amazon Lambda C# project is a thin layer that does nothing but invoke methods from platform-agnostic C# libraries

Dev-Prod Parity Locally

Dev-prod parity is one of the principles of the twelve-factor app (see [12factor]). I decided to mention this pattern here as it becomes highly critical when migrating applications into the cloud. Developers will have to spend too much time on maintenance and replicating issues found in production if you do not have proper tools and infrastructure to quickly achieve this goal. As a result of this overhead, you will notice decreased ROI from the development process.

To address this challenge, you must have a way of quickly standing up a dev environment and running code locally for debugging purposes. Some will say that it is not possible because the system is running in the cloud. My advice goes back to the previous sections when I recommended being careful with the vendor's proprietary offerings. If you followed my hints, it should not be hard to run and debug your application locally, almost exactly as it runs in the cloud.

Let me build this advice on the previous example of a lambda function project. You can quickly develop another thin layer as a console application that calls into the same vendor-agnostic class library method that lambda does in the cloud (see Figure 4-9). This technique will allow the running of code locally and debugging it.

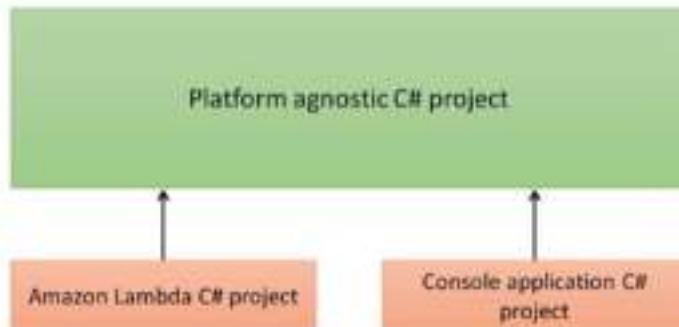


Figure 4-9. Console application project is another thin layer that does nothing but invoke methods from platform-agnostic C# libraries to imitate the behavior of lambda execution

Performance

While software architecture has always been an exciting topic for me, I often saw a conflict of interest between performance concerns and architecture. With more mature technology choices and architectural styles for selection today, I now have better answers for those who worry about performance while architecting solutions.

Performance vs. Clean Design

When we write code, many times performance and clean design become competing factors. For example, batch data retrieval and processing might finish faster than processing a single item at a time, although the latter arguably results in cleaner code. Which way should you go?

I feel that with so many ways to scale applications, questions like this become simpler to answer. You would go with single-item processing and scale components out that are responsible for executing work. With a sufficiently sizeable horizontal scale, single-item processing will be at least as fast as batch processing.

Therefore, do not sacrifice clear design for performance; instead, find architectural alternatives for solving performance issues. Do not declare a loss without consciously understanding all your options, as that is a typical scenario that I often witness.

Let me clarify: Performance is an important topic, and you should not ignore it. Instead, you should find a proper compromise between performance and architecture.

Designing Components That Need to Be Fast

The previous section addressed decisions between performance and clean code. Sometimes, a given component needs to be fast, irrespective of how its design looks. In that case, separate critical algorithms from the domain layer via abstraction (i.e., interface) and optimize the extracted code block. Remember that most optimal code is not readable, and it should not get in the way of business logic.

Such components must be an exception to the earlier described rule, which suggested that a horizontal scale should address most issues.

Front-End Application Architecture

Nowadays, most UI applications run in a browser as a combination of HTML, CSS, and JavaScript code. With an increased focus on the web front-end, it is vital to know the role of software architecture in this comparably new area of engineering.

Before browsers became so powerful, desktop-based applications were the mainstream for developing front-end programs. Back then, architecture had its influence by splitting the program into traditional layers such as UI, application services, and domain. Somehow, when browsers attracted the attention of engineers, software architecture stayed in the back seat for a while, which negatively impacted structures that evolved on the front-end.

Here is one indication of the mentioned problem: Up until now, I have encountered front-end applications that do not have domain layers. *In my opinion, this is a critical shortcoming that needs resolution; otherwise, the increased complexity of UI applications negatively impacts ROI from development. It slows down processes and drops quality.* Without going into more technical details, make sure to read my article about this problem (see [Tengiz AJSAA]).

The lack of proper software architecture techniques on the front-end is also caused by the absence of the architect role for UI applications. Usually, most engineers working on web client-side code prefer to focus on aspects of UI and UX design (HTML and CSS) rather than architecture (design patterns and layers). Most of the UI architects that I have met were not strong enough in JavaScript coding, not to mention design patterns and architecture styles, so they could not lead architecture-related discussions. I expect that the meaning of a UI architect's role will change over time, and things will get better. Meanwhile, I consider the front-end side of engineering a vast untapped territory that screams for improvement.

Therefore, I ask every software architect to consider UI-related structures seriously and take ownership of architecture in this direction. For those architects focusing on the front-end, my message is that UI is not only HTML and CSS but also engineering and architecture, so I encourage this group of individuals to step up and learn and evangelize architecture for front-ends.

Built-In Security

Security is one of those concerns that comes to mind after every other software engineering aspect, such as coding, testing, and deployments. For that reason, when a security team or automated software check detects numerous vulnerabilities, there is not enough time to address them. Hence, organizations often take shortcuts by deferring resolution to a later time. Deploying insecure software to a production environment means

jeopardizing important information and risking the customer's trust. You can decide if this path is worth the trouble, but I highly recommend against it.

Therefore, there is a need to address security issues found in software systems before taking them to production environments. How do we achieve this objective without sacrificing timelines and cost?

The answer to this question is to use an iterative approach. When you architect the system, do so *with security in mind*. It is easier to address smaller chunks of vulnerability issues in each iteration than to deal with all of them at the end of the project's lifecycle. Make a security check part of the Definition of Done and challenge yourself about squeezing it into smaller iterations or milestones. With this approach, it will be unnecessary to sacrifice the safety of customer data for approaching deadlines, which ensures less unplanned work and tech debt in the long run and hence increased customer satisfaction.

While this section focused on security, the same principle (design with it in mind) applies to many other aspects, such as data privacy, compliance, standards based on the industry's needs, and so on.

Databases

The vast majority of all applications store their data in a database of some kind. Therefore, software architecture must address storage concerns apart from the guiding structure of programs themselves. I follow this objective in the current section.

Database Technology Is Irrelevant to Design

Earlier I discussed the importance of avoiding vendor lock-in. Part of that effort is abstracting the database so that the domain and other layers are unaware of concrete storage technology. *Interestingly, this abstraction also*

results in improved design as it guides you to focus on the core domain model and its intricacies instead of coupling and limiting design decisions with any given technology.

Here is another side effect that occurs due to the unnecessary coupling of domain and storage technology. I have seen many software solutions where architects and engineers were concerned about *impedance mismatch* (see [Wiki IM]), so much so that they dumbed down the domain layer under the database schema. When you do this, you introduce an unnecessary burden of translation between problem space concepts and implementation as it adheres to database concepts and not to business ones. This anti-pattern goes against the objective of putting the domain and code closer to each other, which we already discussed. Therefore, avoid this trap at all costs. Design your domain objects based on problem space concepts and not based on the way you store them.

In-Memory Identity Value Generation

A design that ignores database technology must take ownership of some of the concerns that the database can do. One example is an identity value generation during insertion.

We all know that a database can take care of assigning the next available incremental number to a row, but that means the design relies on storage's ability to do so. Also, there is a temporal coupling between insertion and the need to receive and use the identity value of a newly created entity. Therefore, consider generating identity in code explicitly and assigning it to a new object instead of waiting for the insertion operation to complete. This approach is not only technology-agnostic but also a more scalable technique due to the eliminated need for database interaction.

One way of generating and assigning a new identity value to an object's property is by using a globally unique identity (GUID)—an available technique in every programming language.

Application Database vs. Data Warehouse

It is critically important that you distinguish between two kinds of databases by their role in the organization—application data store and data warehouse. In short, the application database is *owned* by a single software system that reads and writes to the storage; meanwhile, a data warehouse *gathers* information from multiple databases and allows reading and writing to it in a more uncontrolled or flexible fashion.

If you dismiss the preceding advice and instead combine the two kinds of storage, here are some problems that you will face:

- If you write to a storage a bit of data incompatible with the program reading from it, your actions can cause application issues.
- Every change of the database schema by application developers requires coordination with data warehouse experts as there is a risk of an impact on warehouse operations, such as replication and reporting. This overhead causes an unnecessary slowdown in the development process.
- Open access to the application's database gives a confusing message to viewers of data, which might result in throw-away work. For example, a warehouse engineer might decide to write into the table, hoping that it will show on the UI. At the same time, the application might have an in-memory cache in place that avoids reading database records for long periods, which turns inserted data into no action. Such small disconnects in knowledge and expectations accumulate and turn into an unnecessary overhead of communication and waste. The idea is that only the application owning the database knows how to write data to it.

Therefore, distinguish the application database from the data warehouse and separate them from each other. Limit the application's database for read-only access for everybody and everything except the owning program. A data warehouse is a shared resource, so treat it as a bucket where everybody works together.

Versioned Databases

Application databases evolve in iterations, just like applications that read and write to these storages. For that reason, we need techniques to upgrade the database from its current version to the next one when we deploy the corresponding application's new version to a given environment. This approach produces *versioned databases*.

I will not go into in-depth detail about an implementation of this technique, but I encourage you to read my articles on this topic (see [Tengiz VDB], [Tengiz VSDB], [Tengiz STVDB], and [Tengiz MVDB]).

Is the architect's job over with the fleshed-out architecture plan? I think it is just starting because the engineers are about to work on the implementation.

Architecture and Implementation

Software code is an implementation of the architecture. Therefore, architectural guidance is necessary during the development process.

In this section, I want to briefly touch upon important software architecture topics that affect coding and implementation.

Tactical DDD

While the strategic area of domain-driven design affects the higher-level structure of the entire enterprise, the tactical part of DDD is a vital technique for implementation, helping to manage codebase complexity and to bridge the gap between developers and problem space SMEs (Subject Matter Experts).

Tactical DDD is a complex and rare skill that requires the purposeful education of engineering teams. While acting as a software architect throughout my career, I always ensure that I set clear expectations with developers about learning and following DDD in code. It makes a big difference when an architect explicitly sets a high bar for engineering excellence with practices such as model-driven design and DDD.

You can learn more about tactical DDD in a famous big blue book (see [Evans DDD]).

When applying this advice to real-life situations, initiate technical discussions with engineers and use tactical DDD terms when explaining solutions. Listen to their responses and try to align language around proper patterns and practices. Shifting mindset is the responsibility of those senior team members who know advanced topics such as DDD and modeling.

Evolving Design

The term *evolving design* refers to the practice of architecting software systems gradually and in iterations. How do you turn this exercise into reality in development teams? One way to achieve this objective is to make designing a solution part of the Definition of Done. There must be an expectation that the team understands the importance of structural coherence and consistency before jumping to code.

It is a significant breakthrough when the team sees *domain as a model*, and there is at least one hands-on modeler in the group. The existing model serves as an incarnation of domain expertise and an enabler for

deep learning, knowledge crunching, and continuous improvement. The resulting code reads like a functional requirement that allows developers to correlate business needs with implementation, be more productive on projects, and minimize disconnect with problem space SMEs.

The next level of modeling mastery is when learning is kept in a live design document serving as a shared language and distilled knowledge. Hands-on modelers keep improving this artifact to continue enhancing the bridge between problem statements and implementation details.

Writing Domain Layer's Code

As I explained elsewhere in this book, complexity is the leading cause of failure of many software projects (e.g., see [RG CCS]). The root cause of this problem lies in every solution's heart—a poor domain model, or sometimes a total absence of it.

The first step to address this issue is in the architecture and modeling exercises I described earlier. The next important part is coding and implementation. This stage is where the usefulness of architecture and modeling is put to the test. If you forget about the model as if it were only a mental exercise and nothing more, then code will become spaghetti and follow the anemic model. When this effect occurs, the benefit of the model and architecture is diminished by complexity overhead and a gradual slowdown in delivery pace.

Therefore, adhere to a model when writing a program. Essentially, code is the realization of the model. Put the primary engineering focus on domain model implementation by focusing the best talent around it. Avoid the typical trap of directing the smartest people to build infrastructures and not domain layers. The primary value lies in the core of implementation, while technology choices and infrastructure only support the domain layer.

Consolidate Development Tools and Languages

Architecture concerns usually arise at a higher level than development. As a result of this effect, code-level matters such as choice of tools and languages lack attention. This situation decreases overall organizational agility and readiness for changes.

For example, during a sudden strategic reorganization, can you ask an engineer from one team to move to another and be productive from day one? Try to ask a Java programmer to write JavaScript code. Since these two languages and their corresponding frameworks are different, the shift will require a considerable learning curve. Would this change be more straightforward if both teams wrote code in Java?

If we look at a small company with a single engineering team, technology and tool choice are probably not so critical, and it can be anything that meets the organization's goals via price and effectiveness. Also, there is no consolidation necessary if there is only one team.

The importance of consolidation increases in large enterprises: They have multiple teams and software products, and they want to be agile and ready for sudden changes.

Therefore, when in a large enterprise with multiple teams, always look for consolidation opportunities in tools and programming languages to achieve maximum agility and adaptability.

Typically, you can unify the choice of programming languages, frameworks, technologies, patterns, concepts, architectural styles, processes, and methodologies.

An architect has successfully considered the implementation details, but is the system testable to support the proper quality assurance activities? This question is the next dilemma for the software architect and the next topic for us to cover.

Architecture for Testable Systems

The quality of software systems is one of the most critical metrics that increases customer satisfaction by decreasing the number of defects and accelerating development pace. Everybody understands this simple statement, but many times the connection between quality and system architecture is not clear.

It is a proper software architecture that enables adequate testing for quality. Therefore, in this section I want to cover a couple of important topics about structuring applications for better quality assurance.

Testable Code

Code is testable if you can write tests that check the correctness of components without modifying implementation. You can achieve a high degree of testability by programming against interfaces. Ensure that you are familiar with such concepts as SOLID (see [Wiki SOLID]) and dependency injection (see [Fowler DI]). Classes and components that depend on interfaces are testable because you can mock up the abstracted implementation when running tests and assert the operation results after it runs.

Testable Application

Testability above the code level is another design objective to achieve. Not every component or feature is easily testable out of the box. The selected architectural style should consider this challenge when guiding the structure of the entire system. An architect should think about how quality engineers will test the program instead of leaving it up to testers. If the system's correctness is hard to check by either automated or manual tests, something needs to change in the application and not in the test framework.

For example, if you are building a front-end application, testers might find it hard to identify an element on the screen. It is probably a good idea to add an ID or another suitable attribute to every item on the screen that the automated test could pick.

It helps you better understand the challenges of quality assurance if you consider test automation as another engineering effort and not as work done in a silo outside of development. Furthermore, engineers can better solve testability objectives if they are the ones who write automated tests.

A testable application has quality characteristics worth taking to the production environment. Let's see how software architecture supports deployments.

Architecture for Deployable Systems

Deploying software systems is a task for DevOps and CI/CD processes. Interestingly, it is easier to deliver some programs to the hosting environment than others. What makes the difference? Besides the configuration of deployment pipelines, the architecture of applications also affects results. Therefore, I want to discuss a couple of topics in this section that you need to consider when architecting software. These points will either enable or hinder your ability to deploy applications quickly and adequately.

Versioning

An application needs to *evolve in versions* as a cohesive unit that includes both code and the corresponding database. You should install or uninstall a version when necessary to go to the next or previous working snapshot. To achieve this objective, you must plan for the evolution path of the application.

CHAPTER 4 DESIGN AND ARCHITECTURE

Earlier in this book, I briefly explained the necessity to version the application database, so I will not cover that topic again. This time, let us focus on the versioning of the application.

It is easy to forget about the evolving nature of an application in the middle of development. You add new features to the system and modify existing capabilities, almost like a routine. When you try to take the program to a production environment, you realize what is there currently and what you are trying to do with it. If you do it right, your application should not have any downtime that affects customers who try to place an order or explore your products online. If the move to a new version is complicated or confusing, your customers might notice glitches, impacting their satisfaction. It is easy to avoid complications if you are ready for production deployment at all times.

The first step for readiness is to mentally realize that *deployment is going to happen no matter what*. Next, force yourself into building chunks of self-contained features that move a system from one working state to the next working state; i.e., from one version to the next. *Repeat this exercise for every change*. If each small increment achieves this objective, then the entire release should also guarantee evolution to the working state, or backward if something goes wrong. This shift of mindset is a fundamental breakthrough that puts you on the right track to having successful product releases. As you noticed, one simple idea behind this process is versioning.

Therefore, plan for versioning when architecting and implementing software systems, as this technique can be the deciding factor in whether your production deployment is successful or a failure.

I recommend that you look for additional reading material about versioning of various deployable units, such as for database, API, and other parts of your systems. For example, see [Tengiz VDB], [Tengiz STVDB], [Tengiz VSDB], [Tengiz MVDB], and [RAPINET Versioning].

Containerization

I briefly mentioned the concept of containerization (see [Wiki Containerization]) earlier in this book. This time, I want to reiterate that properly applied containerization can make your deployment strategies more successful as it puts you into the mindset of building a working product, as follows:

- Everything that works on a local machine will work in a production environment, too (while the environment, such as network calls, may be different, the container is the same).
- Every version of an application is a package that goes to various environments without modification, which ensures the quality of each deliverable unit once it reaches the production environment.

An often overlooked activity—maintenance of the software systems—must be at the forefront of the architecture, so let's talk about a few critical topics in this area.

Architecture for Maintainable Systems

This book promises to eliminate the need for a production support team. The current section is not a contradiction of the mentioned objective but rather a couple of supporting techniques to reach the goal of having highly maintainable software.

Mindset Shift—No Tech Debt

I have been on many engineering teams, and I have often heard various statements that point at non-optimal solutions. Here are a couple of examples:

- Ideally, it must be done differently, but we cannot do so to meet deadlines.
- We will need to rewrite the solution one day anyway, so let us follow the flow and not disrupt the process with refactoring or improvements for now.

I understand that the shift will be difficult, but somebody needs to stop this madness. If you are a software architect or want to be one, you need to take quality seriously. Do not cut corners, do not accumulate tech debt, do not let things happen that complicate matters of architecture, implementation, or any other phase of the software delivery chain. Start with architecture and keep avoiding tech debt at all times. If you have to cut corners, document compensating technical tasks, and ensure that they get into a workstream soon.

Working Systems

While it may not be evident, being a software architect does not only mean drawing diagrams and providing high-level vision statements. Besides other responsibilities, a software architect needs to employ structural thinking to deliver a *working product*.

By “working product,” I do not mean only a product that works without bugs and meets expectations; I mean the one that works without production maintenance! Gather engineering talent in development teams and set a goal of running the program only with the group’s help. As a result of this exercise, if the squad turns into a maintenance team, you cannot deliver new features. So it is of utmost importance to learn

and implement structures that optimize, minimize, and often eliminate routine maintenance. This shift allows developers to focus on new feature development and turns the application into a system that *always works*.

I know that the preceding statements can seem vague, but the problem is that the solution varies on a case-by-case basis. To make matters a little bit simpler, I will provide a couple of examples. This approach should prove that architecture can solve maintainability concerns as long as you allocate time and attention to it.

Common Maintenance Tasks

One common problem of highly transactional e-commerce systems is horizontal scaling when load increases during the Black Friday season. Some companies have maintenance teams that run particular housekeeping tasks to prepare applications for upcoming traffic. Manual activities include procurement and allocation of additional virtual machines and deployment of an application onto new servers. This activity can be eliminated or at least minimized with containerization and microservices. Containerization ensures that the program can be easily multiplied into many instances when needed. Microservices help to abstract the scale size behind the program's boundaries so that the service consumers will not need to know when you scale up or down.

Another common maintenance task is data import into the database from old legacy systems. Usually you can employ database administrators to create data replication between various stores. The problem with this approach is that it does not let you continue evolving the database without affecting replication. Worse, replicated data does not follow the consistency rules of the application, which poses a risk to the program's stability. The solution is to implement proper integration for systems by employing event-driven techniques to let data flow between applications automatically.

And the last example that comes to mind is the cleanup of accumulated log files to free up the server's disk space. I have seen dedicated personnel opening the folders and deleting the log files manually and systematically. Why can't the application automatically delete log files when they cross a given threshold? This simple technique eliminates the need for a maintenance team that monitors disk usage and the number of records so as to take action accordingly.

Fixing It Twice

Let us recall how development teams fix bugs after finding them in a production environment. Some tasks keep coming back, but it is hard to notice for engineers—their job is to fix things. It is the responsibility of the software architect to say that *fixing something twice means we are doing it wrong*.

If you continue fixing repeating problems, then you have an overhead that you should eliminate. This redundancy means more time spent on maintenance and less on feature development. Easily maintainable systems should allow you to flip the focus to the delivery of new features instead. Therefore, I will give you a couple of pieces of advice to avoid fixing the same thing twice.

For every bug fix via a code change, unit tests must be written to ensure a permanent resolution. If a developer makes modifications that can reproduce the same defect, the respective unit test will alert you about it.

Repeating maintenance tasks need to be automated. If you had to clear log files twice when they reached a certain threshold, perhaps it is time to implement an automated deletion script that will run on schedule.

Always fix the root cause instead of patching the surface. For example, if invalid data needs to be detected, where do you verify input—on the UI screen or in an API endpoint? The UI screen might seem sufficient to validate data and not let it flow to API endpoints. If you do not fix API validation simultaneously, a malicious user can still get inconsistent data into the system's database. Therefore, you need to correct the root

cause—i.e., the API—to disallow invalid data submittal. Once you take care of the mentioned part, then fix the UI screen too so as to have a better user experience by validating input in the browser without an unnecessary roundtrip to API endpoints.

Simple vs. Complex Systems

It is vital to understand that a system is simple if running and maintaining it in production is easy. Do not confuse these criteria with the simplicity of implementation. A given system might be built using simple and straightforward patterns, but it might be hard to deploy and run. Usually, correlation goes another way: building high-quality, maintainable systems is harder for engineering teams as it requires mastering and adhering to more advanced patterns and practices.

To objectively gauge simplicity versus complexity, look at architected systems from outside, not only from inside.

If you take somebody else's software and run it in production, you will realize whether you are dealing with an easily maintainable application. Build your systems so that they look easy for those who need to deploy and run them, and do not settle for less.

Make sure to read my article on this topic for more information (see [Tengiz SACISD]).

Summary

This chapter looked at software architecture from various angles, such as cross-cutting concerns, requirements analysis, coding, testing, deployment, and maintenance. There is always something we can do with architecture that supports these areas, and the impact is a two-way street. We also discussed the architecture body of knowledge, covering many essential patterns and structures that will help you build effective software solutions.

Next, we will dive deeper into coding and implementation.

CHAPTER 5

Implementation and Coding

In this chapter, I want to cover a core exercise of software development—coding. You can expect that the majority of the presented sections will be technical in nature.

Every engineer writing a program has to deal with various activities directly or indirectly related to coding: professional culture, testing, deployment, and maintenance. Therefore, I will also touch on those topics to avoid missing any important aspect affecting success.

With every bit of advice or recommendation, I am trying to put your software solutions one step closer to the ideal outcomes that I promised earlier: delivering on expectations, zero defects, horizontal scale, minimal to no support, acceleration, and increased return on investment (ROI). Sometimes, this link might not be visible, but it is always there. Generally speaking, more robust coding and engineering skills result in higher quality, working products, which positively affects critical indicators such as customer satisfaction and ROI.

Let's start with the cross-cutting concerns, which indirectly impact coding activities.

Cross-cutting Concerns Related to Coding

Although coding may look like an entirely isolated technical activity, in reality, multiple non-technical factors affect the efficiency of this exercise. In this section, let's look at such aspects.

Professionalism of a Coder

Earlier in this book, I talked about professionalism, and I described it as follows:

Professionalism is when you do what you claim to be doing in accordance with the standards and ethics of the profession.

Now that we are speaking about coding, let us agree on the actions that developers need to take to be professionals. I am not trying to play a blame game, even though it might seem so; instead, I want to influence positive changes. I am a hands-on engineer, and I have met many kinds of specialists, so I know what I am talking about.

Developers often write in their resumes skills that they do not have; they also claim techniques that they do not follow. We must stop acting in this way by either setting the right expectations or improving our skills.

One example is a popular test-driven development technique (TDD; see [AA TDD]). Every other engineer has this in their profile, but they fail to demonstrate it in action after they join a company.

Another example is writing unit tests. I have met many engineers who said that they wrote unit tests until somebody noticed that they did not.

There is no excuse for claiming the mentioned techniques falsely, as they are easy to master. You can take action and become a professional by learning skills that you think you should have.

Therefore, I encourage every developer to keep their promises and do what they say. It is the negligence of this promise that creates a negative image of engineering. Let's be professionals!

If you want to learn more about things that require improvement in the software development industry, read my article on this topic [Tengiz WWWSD].

Put Talent into Important Tasks

If you are a development manager or a leader, perhaps you know who the most skilled engineer on a team is; but how do you ensure that they work on an essential piece of delivery? Let us find out.

As we agreed earlier in this book, the domain layer is the most valuable part of software solutions. Ironically and contrary to this statement, it is a common misconception that business logic is the least important aspect of applications. Because of this misunderstanding, most talented developers often work on infrastructure instead of business logic. Such an approach puts many software projects in jeopardy due to increased complexity that doesn't get much attention until the project is facing severe problems down the road (see [RG CCS]).

In other cases, organizations understand the importance of their domain layers, but they fail to recognize where this area is within the solution. To avoid such a trap, try to spot areas related to bottlenecks or most bugs; identify the most complicated part, or which components impact your revenue. By asking the right questions and attacking challenges without hesitation, you will be investing time and resources into the most valuable tasks.

For example, let us discuss UI-heavy software solutions, also known as single-page apps (SPA; see [Wiki SPA]). On such projects, I have often seen one developer working on a front-end and several others working on a backend. Bugs keep occurring on the front-end, and it soon becomes a bottleneck. Organizations live with this situation since backend engineers do not feel comfortable writing HTML and CSS. These companies do not realize that a *considerable part of their business logic lives in the front-end*, requiring immediate attention, contrary to other types of programs

CHAPTER 5 IMPLEMENTATION AND CODING

that have domain layers on their backends. If teams facing this challenge cannot redirect the majority of their talent to work on a front-end, something is wrong with the implementation or resource allocation.

As a possible solution to the described circumstances, I would ask backend engineers to work on a front-end. Alternatively, I would try to rearchitect an application so that the business logic is accessible to more developers than just one front-end specialist. I will not go into more detail since it goes beyond this section's scope, but it must be clear that there are multiple options.

To conclude, you should always try to identify where the essential business logic lives within your solution. Next, find a way to let more engineers focus on that critical piece of the program.

Continuous Improvement

While I advertise the feasibility of an ideal software, it is essential to clarify that *perfection comes in iterations*. With the first attempt, you might not be able to deliver the best-quality application, but you need to keep trying. It helps if we look at this process as an *iterative improvement*.

Developers deliver work in chunks naturally, either as separate tasks or as milestones. Therefore, their engineering mastery sharpens over time by providing better-quality features and improving earlier work. With better quality, bugs will not leak into production, keeping customers happy, which is a valuable exercise.

It is crucial to encourage and welcome a spirit of continuous improvement in engineering personnel. Start with advocating for the right decisions, customer focus, and regular learning of modern development practices. Examples of such techniques are TDD, BDD, ATDD, pair programming, refactoring, design and architecture patterns, and so on.

Continuous Refactoring

Continuous improvement (CI) is a general term consisting of multiple exercises, one of which is continuous refactoring. This technique is especially helpful for a codebase's highly critical areas, such as business logic.

Naturally, the first attempt at domain modeling is not the final version, because the resulting code is only an initial draft of the expressed knowledge. As the business evolves to meet the market's demand, so too does the shape of the model change. Therefore, domain models go through waves of refactoring and rewriting naturally. When we learn something new about a business, we come back to the code and try to codify the insight. Therefore, every enhancement to the domain model results from a knowledge evolution rather than a mere technical improvement.

In domain-driven design (see [Evans DDD]), this technique is called "Refactoring to deeper insight." As a result of repetitively applying this exercise to a codebase, every team member will benefit from cleaner code that helps them understand the problem space. When developers have such a toolset at hand, they are more successful at meeting the expectations of the business and its customers as their knowledge grows faster with the evolving domain model.

Limit Works in Progress

I think you have heard the advice to limit your works in progress. People tend to either mechanically follow or dismiss this direction due to the unclear motivations behind it. In this section, I want to explain the underlying reason for this recommendation so that you are productive when implementing such measures.

Firstly, more works in progress means fewer things done and longer feedback loops. As you can imagine, these consequences are not attractive from either the quantity or the quality standpoint of deliverables.

CHAPTER 5 IMPLEMENTATION AND CODING

For example, let us look at a typical situation in a development team that works in iterations. There are only a couple of days left until the end of a sprint, and several items still have not been touched. One of the engineers becomes available to pick up new work. He decides to act smart and start working on all remaining items simultaneously to finish all of them before iteration ends and avoid carryover. At the end of the sprint, all those work items are still in progress. Sound familiar?

In this example, an enthusiastic developer did not deliver any of the pending items. It would have been better if the engineer had taken a single small task and finished it in the remaining couple of days instead of starting all of them and not finishing anything. Delivering one item is more advantageous because the team would demonstrate and complete one more task by the end of the sprint. This difference might not seem significant at first glance, but it helps the team receive early feedback about work and deliver completed features sooner to customers. It is such small wins that accumulate over time and turn into a competitive advantage.

Another reason for limiting works in progress is to identify bottlenecks in a workflow. How is that possible? Kanban (see [Wiki Kanban]) relies on this technique to expose steps in the process that need attention since they clog deliveries. Precisely, you can apply a "work in progress limit" ("WIP Limit" for short) to every stage of flow and then let the team continue working as usual. You might soon notice that stories (or cards) cannot move forward from a certain point if there is not enough space for them in the next step due to its WIP limits. Therefore, when such circumstances happen, you have identified a bottleneck that needs immediate attention (e.g., see Figure 5-1).

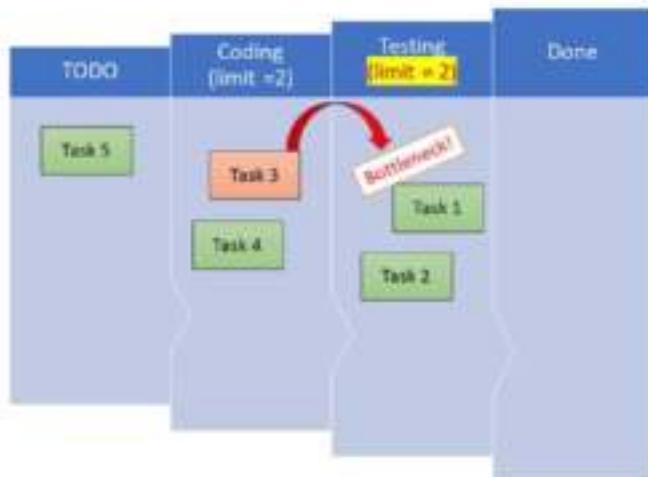


Figure 5-1. WIP limit helps identify a bottleneck in a testing step where the limit has been reached, and no more items can move into this phase

Some ways to resolve the described situation would be to assign more developers to the problematic stage or decrease the incoming load; after all, the chain is only as strong as its weakest link. Going back to my original point, we would not be able to diagnose such a critical issue in a workflow without limiting works in progress.

The final reason for limiting your workload is based on a fundamental of human nature. People are not good at context switching since our brain delivers the best results when we focus on one thing at a time. When we take on several simultaneous tasks, we are frequently switching our context, which is unnecessary overhead that can impact results.

Quality vs. Quantity

I have often heard interesting comments from developers, such as the following:

- Our team does not write unit tests, because we don't have time for them.
- Our product owner does not want us to refactor a problematic functionality since she prefers to spend that time on new feature development.
- We have technical debt because the business asked us to deliver something quickly to production.

These statements all point in a single direction: developers believe that business prefers quantity over quality. Let us discuss this hypothesis.

If you followed the idea behind this entire publication, you probably know what low quality means: more bugs, technical debt-related overhead, slowdown, weak customer satisfaction, and decreased ROI. Do you believe that business wants these outcomes? I doubt! The problem is that the business does not understand the consequences when they make these choices. Therefore, what we have is not a preference of quantity over quality, but rather an *uneducated decision*. If you want your business representatives to decide whether you need to write unit tests, you need to ensure that you explain to them the consequences and not only the actions.

For example, if you feel that the unit tests are a must, but the product owner does not see value in them, clarify that without them, there will be bugs that you will not notice but customers will, and their satisfaction will drop; don't forget to mention that you will work slower too, since you will deal with a buggy code, or will need to be overly careful not to break anything. If the outcome does not change (which I doubt), you have one less reason to worry if anything goes wrong.

I want to touch on a more problematic side of this question: expecting technical decisions from non-technical people, an example of which is a domain expert choosing whether engineers need to write unit tests. Do you think it is wise for any specialist to ask others how to do their job well? *Sometimes, we need to do the right thing without asking questions to avoid problems down the road.*

While developers are responsible for following engineering best practices, they need to balance this mission with functional feature development. Although these two objectives might sound conflicting, one of them supports the other. Specifically, adherence to technical excellence simplifies and accelerates the development of business functionality.

Therefore, if you are a developer, I encourage you to take ownership of decisions such as writing unit tests, refactoring, or avoiding technical debt. Do not pose this question to business. Sometimes, there is no need to ask, because it is not the right question for business to answer. If you need to justify work to improve quality, think about the consequences if you do not put in this effort, and you might find the necessary answers.

Understandably, you will only have the mentioned choice if the decision-making levers are in your hands. Otherwise, you must encourage leadership to read the chapter about cross-cutting concerns in this book (see Chapter 2), where we discussed empowering engineers to do the right thing. Putting the mentioned decision power into the right hands is part of that mission.

Code Reviews

In today's world, code reviews have become an integral part of engineering. I want to discuss a few fundamental topics in this area.

Code review is not a tool for criticism or offense. It is instead a way of assuring quality and mentoring each other. Remember this statement if you are submitting your code for a review or glancing at somebody else's code.

CHAPTER 5 IMPLEMENTATION AND CODING

I have met developers who hesitate to ask for code reviews because they are afraid of negative feedback. However, some engineers try to prove their supremacy through this exercise. Somebody needs to be wiser, and it seems more natural if we start with people who receive criticism. No matter what you hear back, it would be best to look at feedback as your opportunity to learn and grow. If you demonstrate such a positive attitude, you will be an example of doing it right.

If you fear submitting your code for a review, sometimes it helps if you put yourself in the reviewer's shoes. Try such simple things as *reviewing your own code* by pretending that somebody else wrote it. A healthy self-criticism can help you spot coding errors that will attract corrective feedback from others. If you practice self code reviews, you will learn to fix your errors ahead of a review by others. As a result, you will start delivering better quality code and will deserve positive feedback and compliments during peer code reviews.

Meanwhile, if you are responsible for reviewing code for others, try to prioritize this activity. Remember that a developer waiting for a code review is either blocked or will multi-task by starting a new work in the meantime. We already spoke earlier that multi-tasking is not advisable as it goes against limiting works in progress. Therefore, a better alternative is a faster code review feedback loop. It is also more beneficial if you unblock developers waiting for you rather than deliver other tasks and keep the rest of the team clogged.

Lastly, if you are reviewing code, please *read it* and *do not hesitate* to provide candid feedback. I have often met developers who mechanically approve code changes or do not feel comfortable asking for corrections. Both these approaches diminish the value behind code reviews, and perhaps it would even be better if you do not perform this activity altogether. As I emphasized earlier, code review is an instrument for mentoring each other, the primary constituent of which is candid feedback.

Coding is not only about writing code but also about following a specific design or architectural style. Let's connect the two areas next.

Designing Code

Do you think coding is just coding? Well, think again! In this section, I want to look at coding from an architecture standpoint because there is a lot of structure in code, and it is beneficial to glance at this aspect briefly.

Implementation of Architecture

Earlier in this book, I directed software architects to look at code as an implementation of an architecture. The same advice goes toward developers: a program is most valuable when it accurately implements a guiding structure of the corresponding architecture.

This direction fits logically in the notion of the top-down information flow, which we discussed earlier. When we identify business capabilities, we align overall enterprise architecture under them, usually via microservices and respective teams. Once we achieve this high-level alignment, the next step is to go one level below and align code under architecture. Just as architecture serves business capabilities, code needs to help the guiding structure by implementing it and keeping it up-to-date with learnings (see Figure 5-2).

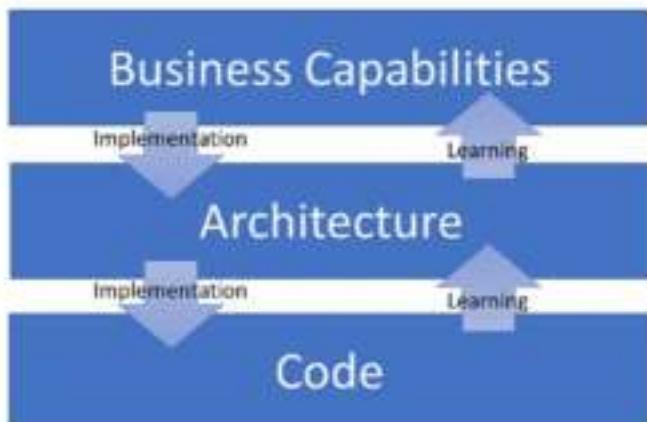


Figure 5-2. *Code is an implementation of architecture, while architecture implements business capabilities*

To achieve the described objective, I want to set some expectations for developers writing code.

It is the responsibility of every engineer to understand and follow the provided architecture and design documents. While I am not asking one to adopt every single direction blindly, developers must understand the value behind alignment. Perhaps there is a good reason for any suggested structure for the application, even if it does not always make sense. When there are doubts, developers need to ask clarifying questions. Most important, both architecture and code are supposed to achieve a common goal *in tandem*—increase quality and deliver based on technical and non-technical expectations.

If a program's code looks nothing like the planned architecture, there must be a critical gap or a dysfunction somewhere in the processes, tools, or resources. Make sure to find and resolve such issues in a timely fashion when you spot them. Such a course correction will ensure optimal outcomes from a development and architecture partnership.

Code Design Techniques

I already spoke about the value found behind upfront design exercises before jumping to code. In this section, I want to extend the same topic from a coding perspective.

While it may seem that only an architect is responsible for designing solutions, developers can also take on this task and even gain benefits by doing so.

Firstly, any engineer can do an upfront design of code (while still adhering to the architecture guardrails) without the help of an architect. For instance, you can use a UML (Unified Modeling Language) tool or a whiteboard for drawing class diagrams to decide on the usage of particular design patterns. It may seem challenging to plan implementation ahead of time, but it becomes easier with each new attempt. After all, UML is not hard to learn or read.

Another helpful technique can be C4 modeling (see [C4Model]), which helps one to think about the architecture in a unified, structured fashion and develop a common vocabulary about systems and their documentation.

Besides upfront design, UML can also help with analyzing an existing codebase and learning its intricacies. If you find yourself looking at a complicated part of an implementation and wondering how every piece works together, consider creating a visual representation of those elements and their relationships. As you distill components and connect them where appropriate, you should notice that things look much simpler now. Many of us prefer a visual way of learning, so use it as an additional force to understand code and plan your work.

Essence of Object-Oriented Programming

I want to dedicate this section to object-oriented programming (see [Wiki OOP]) since I believe that it is the primary direction in enterprise software coding and development. While functional programming is advantageous in certain situations, it is not as convenient or naturally useful as OOP. The primary reason behind such a preference is the way people tend

CHAPTER 5 IMPLEMENTATION AND CODING

to think of or express domain knowledge—via nouns and verbs, which directly translate into objects and their behavior. This factor makes me believe that OOP will continue being a preferred choice for business or enterprise software development for the foreseeable future.

Confusion About OOP

There is frequent confusion about OOP. People believe that they write object-oriented code as long as they use an object-oriented programming language. Unfortunately, this is not necessarily true. It is quite possible, and very common too, for people to write procedural code while using an object-oriented programming language (e.g., C#).

A typical example of procedural code that developers often consider object oriented is a sequence of steps to fulfill a particular operation. For instance, when placing an order, an application needs to charge a credit card, decrease inventory levels, and send a confirmation email. One way to implement this functionality is to write three classes, each fulfilling one step of the overall process. Afterward, we can compose (i.e., inject) those three classes into one master class to invoke them in a sequence (see Figure 5-3).

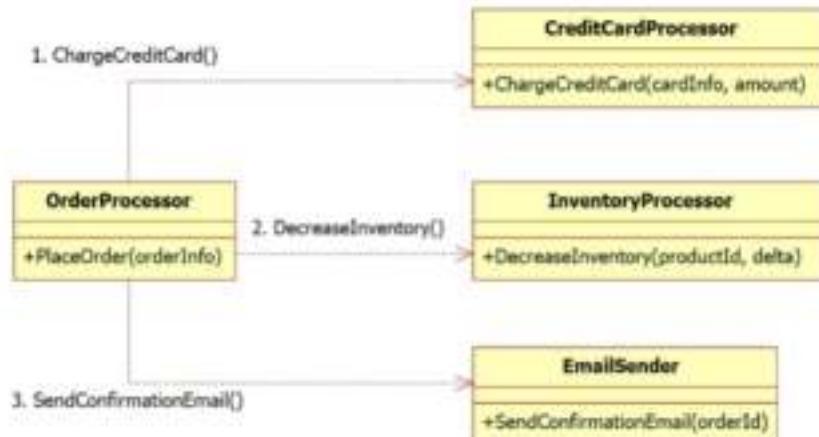


Figure 5-3. *OrderProcessor* invokes methods on *CreditCardProcessor*, *InventoryProcessor*, and *EmailSender* classes

The described implementation looks like object-oriented code indeed: We have three objects, each has its behavior, and we compose them via one more object. Arguably, what we see is not object oriented. To understand my arguments, I want to explain what OOP really means.

OOP Distilled

As you remember, object-oriented programming stands on top of four primary principles:

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

Out of these principles, I want to speak about *encapsulation* specifically.

Every properly designed object needs to encapsulate *state*. While some objects are purposefully stateless, there must be objects that do *have a state* to encapsulate in any enterprise software. If there are no such objects in a codebase, perhaps we are dealing with an anemic domain model (see [Fowler ADM]), which is yet another violation of OOP principles.

In the previous example of placing an order, we are dealing with stateless objects that do nothing but execute procedures. We could have written the same code as a set of methods calling each other, without even putting them in their own objects (see Figure 5-4). This point proves that the previous example showed a procedural and not an object-oriented code.

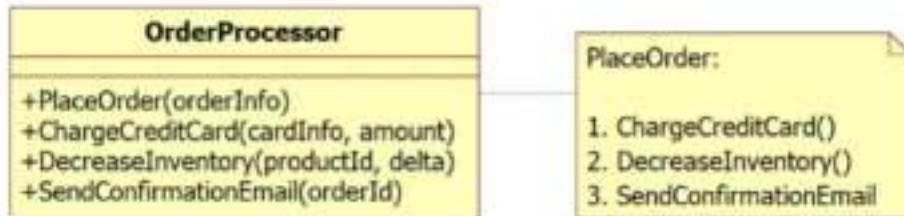


Figure 5-4. *PlaceOrder* method of *OrderProcessor* calls *ChargeCreditCard*, *DecreaseInventory*, and *SendConfirmationEmail* methods, all of which are defined within the *OrderProcessor* class

Now, let us brainstorm how one could implement the mentioned example in an object-oriented fashion. Perhaps charging a credit card and sending an email would need to stay as stateless procedures because they are calls to outside services. As for decreasing an inventory—it sounds like an alteration to an object's state. If we can express this step by encapsulating a state change in an instance of an object, we will prove that we are leveraging OOP. While there are many ways that we could implement such a task, I have provided a simplistic version of it (see Figure 5-5).

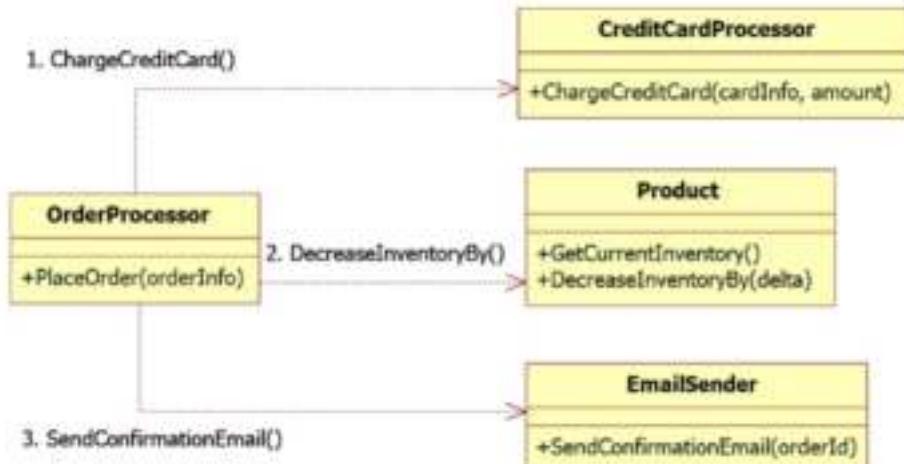


Figure 5-5. A stateful *Product* object has replaced a stateless *InventoryProcessor* object to leverage OOP

As you can see, `Product` is a stateful object that leverages encapsulation, and thus the solution qualifies as object oriented.

Purpose of Design Patterns

While writing code, developers consistently solve various kinds of problems. In some cases, they need to decide how to connect objects, while in other instances, the challenge lies in expressing a specific type of behavior. These are discrete kinds of problems. Many engineers have found repetitive patterns of questions, as well as answers to them. That is why design patterns (see [Wiki SDP]) exist—to solve repeating types of problems in a standard fashion instead of reinventing the wheel each time.

Another reason to follow and apply design patterns is to establish a common vocabulary and knowledge among developers. For example, if I tell you that I implemented a retry mechanism via a *decorator* pattern, that already communicates my solution's essence. Alternatively, you could suggest to your colleague to change the implementation of a *strategy* that sorts customers on a screen. Since these two examples rely on pre-existing patterns (decorator and strategy), explanation and cooperation have become more efficient.

Every knowledgeable software developer needs to know a basic concept of a design pattern and a handful of concrete patterns by their name and purpose. When writing code, you should commonly use SOLID principles (see [Wiki SOLID]) and Gang of Four patterns (see [Gamma GOF]). As you gain more experience, I would expect that you will become curious about higher-level concepts such as Patterns of Enterprise Application Architecture (see [Fowler POEAA]) and Enterprise Integration Patterns (see [Hohpe EIP]).

Composition vs. Inheritance

Various design patterns suggest solving a given problem via either composition or inheritance. There is a widespread opinion in the software development industry that composition is preferred over inheritance. I believe that these two directions can coexist. Furthermore, you must understand both options to determine the optimal solution to every situation.

I will not go into more detail on this topic, but I suggest that you read my article about it (see [Tengiz CVSI]).

Opportunities for Patterns

Some coding situations are apparent candidates for the application of design patterns. Other cases might seem inappropriate for such structures at first sight. I challenge you to apply design patterns to uncommon situations before you declare them “just code.” Otherwise, you may give up on an opportunity to improve code readability, clarity, and quality by employing a guiding structure.

For example, one aspect of implementation that patterns rarely touch is *logging*. This part of code is only a method invocation to write a line of text into the target storage (e.g., a text file). From this standpoint, patterns seem unnecessary. However, if you look deeper into the essence of logging, it solves a significant problem—making an application maintainable. When things go wrong, we look into logs to understand what happened. Therefore, logging is at least as vital as any other code block of a program. If patterns exist to improve implementations, then why not apply them to such an essential code block as logging?

If you look for patterns related to logging, you will find a couple of interesting topics. One of them is about structured logging (see [TW SL]). This approach turns log messages from noisy strings into data with insights about the application’s activity.

Another appealing pattern related to logging is Domain-Oriented Observability (see [Hodgson DOO]). This technique makes logging a part of domain-related concerns, helping develop cleaner designs and purposeful logging standards.

I hope that I have clearly expressed an often-overlooked opportunity to apply design patterns. I am sure that you can discover many other exciting use cases that will benefit from the rigor and clean structure of design patterns.

That is it for now, but we will return to design patterns in the next section when discussing hands-on tactical knowledge for implementing code.

Implementing Code

We've reached the meat of this chapter—coding itself. How can we effectively carry out coding? Which patterns are most valuable? Are there any lessons learned that are especially helpful for this area of work? We will discuss all these points in the current section.

Tactical DDD Patterns

This section is about tactical DDD patterns, which you can learn from *Domain-Driven Design* by Eric Evans (see [Evans DDD]). It is unnecessary to explain them here if there is such a fantastic book out there already. Instead, I plan to draw your attention toward intricacies essential for a successful application of the mentioned patterns. I will do so by dedicating a brief section to various concrete constructs from the specified group.

Therefore, a prerequisite for reading subsequent sections is to learn the basics of tactical DDD patterns.

Entity

Entities are business objects. A common confusion exists between entities and data transfer objects (DTOs). To avoid misconceptions in this direction, please read my article about this topic (see [Tengiz HTDBO]). In short, entities typically encapsulate behavior by ensuring the consistency of an internal state. However, a DTO is always consistent, so no particular encapsulation is necessary.

Entities That Look Like DTOs

Can an entity look like a DTO? It absolutely can. This kind of design is applicable if an entity's state stays consistent by changing any single property atomically. When you deal with such DTO-like objects, there is no need to invest in a state encapsulation since it is consistent no matter what. Encapsulation is only necessary when we want to ensure an atomic transition from one valid state to another, such as through a method invocation to alter multiple fields simultaneously.

Active Record Entities

Active Record (see [Fowler AR]) is a pattern where we treat a database-persisted object as a business entity (see Figure 5-6).

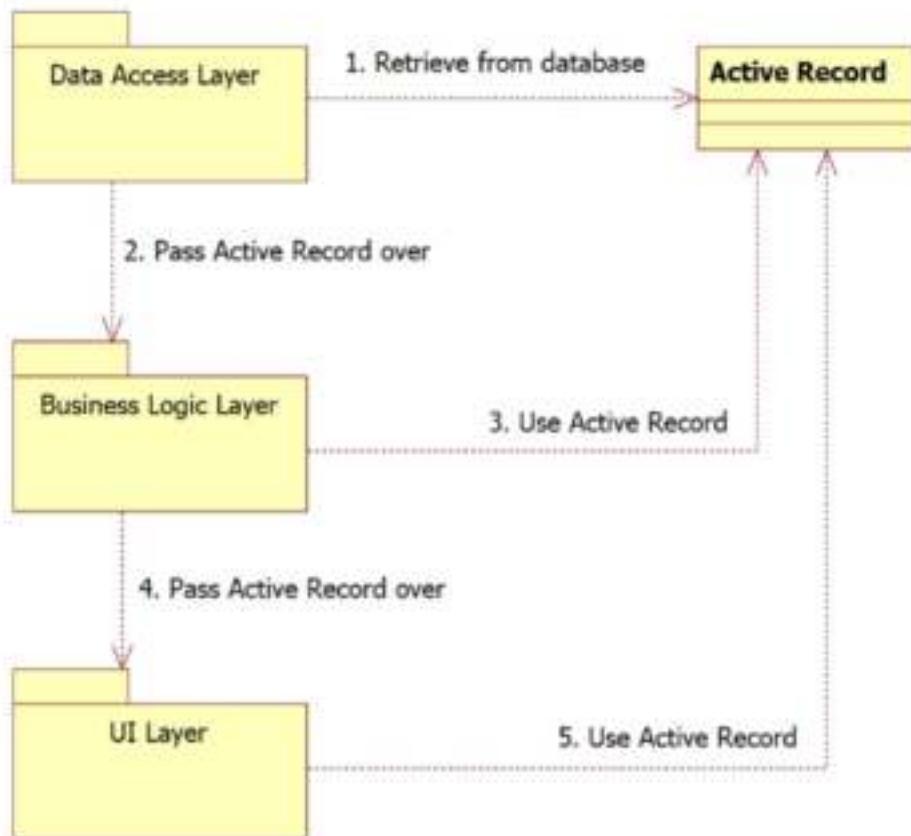


Figure 5-6. Active Record is a shared database object that all application layers use

The motivation behind Active Record is the convenience of reusing a single database object across all application layers. This object serves data access and business logic layers for two distinct purposes—database access and domain modeling. This technique creates a sense of efficiency as it eliminates the need to define and maintain two separate structures.

For basic software projects, Active Record might seem the way to go. However, beware of its limitations before deciding to do so. Let me explain my caution.

CHAPTER 5 IMPLEMENTATION AND CODING

As your codebase grows and the business logic evolves, you will naturally start feeling how the database schema behind Active Record limits your modeling abilities. For instance, if you use relational databases, your Active Record entity will be either extraordinarily normalized or flattened due to database design considerations. However, rich domain entities follow different values, such as repeating shapes of business concepts rather than database table schemas. These two objectives are typically incompatible and cause complications when evolving a project. This limitation will accumulate into a considerable burden over time, so you must account for it.

If you want to avoid this design challenge, consider separating business entities from database objects. I understand that you may worry about maintaining two distinct structures. Still, when they start to naturally diverge throughout a project's lifetime (see [Wiki IM]), you will appreciate this advice due to its flexibility (see Figure 5-7).

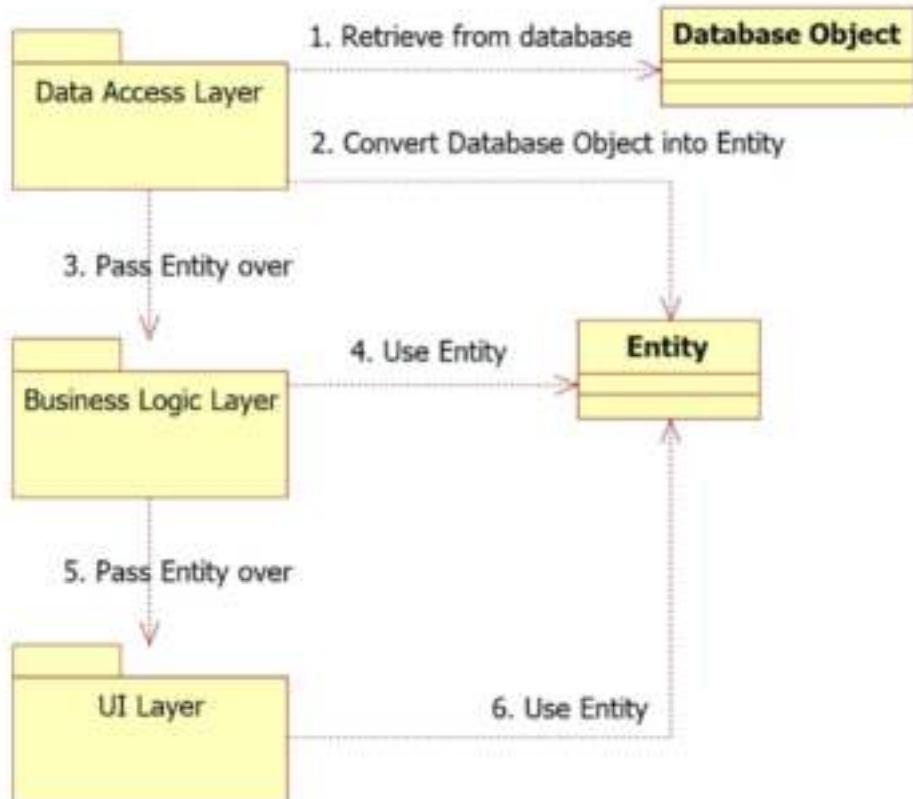


Figure 5-7. Separating database object from business entity due to impedance mismatch

Therefore, avoid the Active Record pattern for complex domain models and instead separate database objects from entities to better serve two distinct purposes—data access and domain modeling.

Module

Two independent software patterns organically received the same name—module. It is better if we distinguish between the two approaches to avoid confusion.

CHAPTER 5 IMPLEMENTATION AND CODING

The first “module” pattern is also known as “plugin” (see [Fowler Plugin]), which relies on substituting a component’s or entire layer’s implementation with another one. This pattern is not what tactical DDD describes. To avoid confusion with the module, I suggest that you call this pattern “Plugin.” The essence of this structure is seen in the uniform interface between various Plugin implementations, which warrants substitutability (see Figure 5-8).

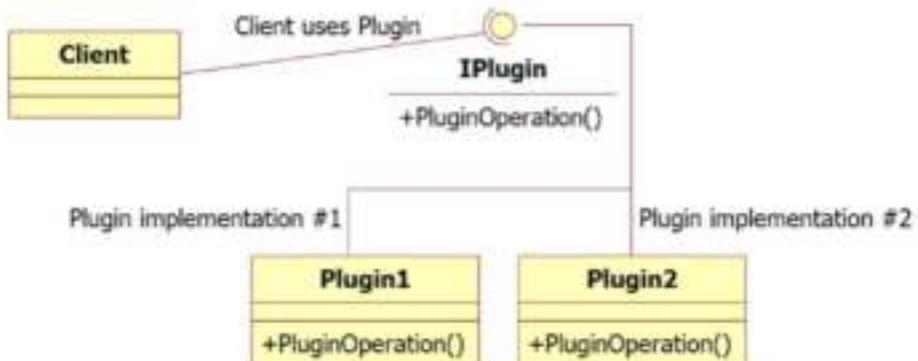


Figure 5-8. Plugin design pattern with two implementations of the same Plugin interface

The second “module” pattern (see [Tengiz MSA]) is about dividing an application into logical boundaries, which usually are finer-grained than application layers. This pattern is part of tactical DDD. Throughout this publication, when I mention “module,” I will be referring to this structure (see Figure 5-9).

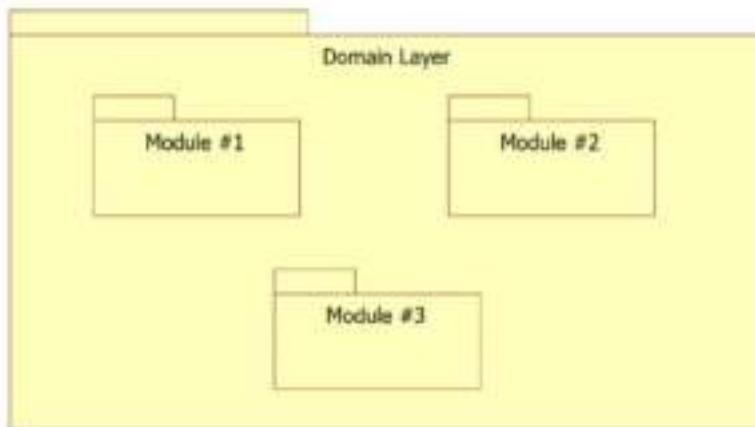


Figure 5-9. Modular architecture that divides a domain layer into logical boundaries

Enforcing Modular Architecture

There are not many programming languages that support strongly typed modules, so you must decide what you are going to do about it.

For instance, C# does not have sufficient built-in mechanisms to implement modules, specifically the following:

- Namespaces do not enforce rigorous modular boundaries and usage, such as avoiding circular references.
- Class libraries could act as modules, but then you end up having too many of them.

In short, I have concluded that C# is not capable of implementing DDD modules optimally. Therefore, it is crucial to think through this caveat if you consider following a modular architecture without the right tools.

Repository

A common confusion about repository is its role—whether it holds database objects or business entities. Let me clarify without further ado that the latter is the intended use of this pattern: repository is storage of aggregates that consist of entities and value objects. This explanation clarifies another common misconception: We should have one repository per aggregate root and not per entity or a database table.

An essential role of a repository is to abstract away persistence concerns when designing domain models. You can defer storage-related decisions to a later point and focus on a business layer first. In the next sections, let us discuss a couple of typical design challenges that cause hesitation when trying to follow this advice.

Relational Database Mismatch with Repository

Let us assume that we designed a domain model first, and now it is time to store aggregates in a relational database. When inserting an aggregate into a table, it is tempting to create relationships with existing entities in another table. In other words, while two aggregates holding the same information are supposed to have two separate copies of that information, relational databases advise us to reuse the same rows for both relationships. These two design guidelines go against each other. Which way should we go?

I recommend that you *violate standard rules of relational databases* and store two aggregates entirely separate (see Figure 5-10).

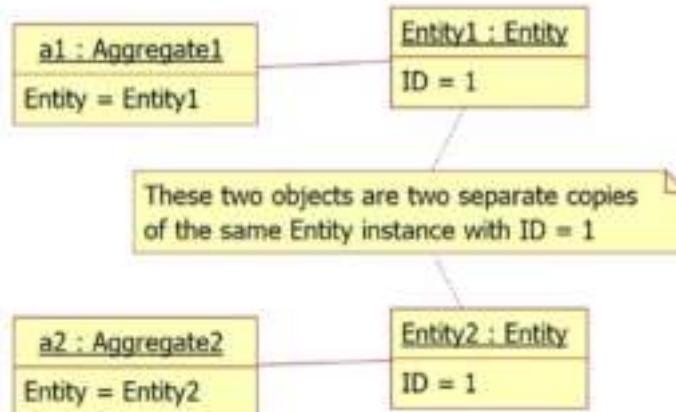


Figure 5-10. Two aggregates holding two separate copies of the same instance of an entity

If you try to adhere to relational rules instead, you will face more pressing issues without knowing it. Let me distill the problem.

A relational database asks you to reuse the same entity instance in a database from two separate aggregates (i.e., two aggregate rows refer to a single entity row via their foreign keys).

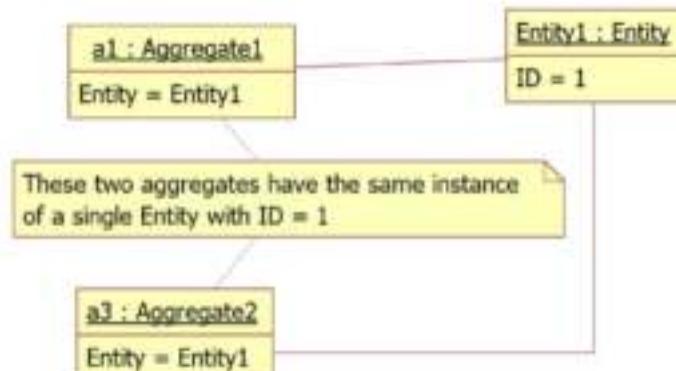


Figure 5-11. Two aggregates share the same Entity instance

CHAPTER 5 IMPLEMENTATION AND CODING

In this situation, if you try to update the shared entity row via one of the aggregates, the other aggregate will also change as it also refers to the same row. Recall that a transaction in DDD modifies only one aggregate at a time! We just violated this principle due to the object design that relational databases encourage.

The moral is as follows: Tactical DDD is a discrete way of implementing domain models, and you need to honor it if you want to succeed in it.

Non-Relational Database Mismatch with Repository

Let us assume that a given repository interface allows searching for an aggregate by matching an expression. For instance, in C#, you could express the criteria as a lambda function, as follows:

```
var customer = _customerRepository.Find(c => c.LastName.  
StartsWith("T"));
```

When designing a domain model, you can write code against the described repository interface without worrying about its implementation. When the time comes, we face an interesting question: How do you implement the searching function of a repository against the non-relational (NoSQL) database? Most NoSQL stores allow data retrieval by key only, so querying customers whose last name starts with "T" can be tricky.

The answer to this dilemma is in *minimalistic repository interfaces*. When designing domain models, assume that a repository can only have the simplest operations that any database can handle, specifically the following:

- Creating a single aggregate. For example, `repository.CreateCustomer(customer)`.
- Querying a single aggregate by key. For example, `repository.GetCustomerById(123)`.

- Updating a single aggregate. For example, repository.`UpdateCustomer(customer)`.
- Deleting a single aggregate by key. For example, repository.`DeleteCustomer(123)`.

Such an approach solves the described problem because any kind of database (relational or non-relational) supports these simple operations. Since search by pattern-matching is no longer a method on a repository interface, there is no challenge of implementing it.

If you only had access to such simple functions on a repository interface, how would it influence your domain modeling approach? If you can answer this question, you are one step closer to implementing properly designed business logic layers.

How About Reports?

After reading the previous section, you might wonder: How should we implement operations that allow searching for objects by attributes other than keys? While you can get along without such queries in a domain layer, reports commonly need the flexibility to traverse a collection of rows, so the suggested trick will not work.

The answer to this question is in a combination of CQRS and event-driven architecture, which turns reports into aggregates. Specifically, the following:

- Compose data necessary for a given report into a single CQRS read model and treat it as an aggregate.
- When a user wants to view the report, retrieve *entire read model aggregate by its key* and display it on a screen.

CHAPTER 5 IMPLEMENTATION AND CODING

- To filter report's rows based on user's input, narrow down the aggregate's content in memory after retrieving it from storage and before displaying it on a screen.
- Keep the report-related read model up-to-date by employing event-driven architecture: subscribe to events that tell you when to refresh the aggregate and regenerate the report's content accordingly.

As you can see, this technique still deals with a single aggregate by its key, so we have successfully applied the same minimalistic method to this uncommon situation. Therefore, we can conclude that the described approach is sufficient for advanced search scenarios, such as reports.

Factory

Factory pattern exists in two flavors in Gang of Four (see [Gamma GOF])—*Factory Method* and *Abstract Factory*. Tactical DDD introduces the notion of a *Factory*, which is not the same as either of the previous patterns.

One key difference in DDD's Factory from Gang of Four's counterparts is that the Factory of tactical DDD is not substitutable. Instead, it is part of a business logic layer as a helper method since the construction of an aggregate is a domain concern. On the contrary, Gang of Four's factories are usually substitutable: We need to derive from base classes or interfaces representing an Abstract Factory or a Factory Method to override their implementation.

Warning About Gang of Four Factory Patterns

As I explained earlier, factories of Gang of Four (GoF for short) are substitutable. Typically, we implement an interface or a base class to return a concrete type of an abstract product. In other words, each implementation of a GoF factory works with one specific product type (see Figure 5-12).



Figure 5-12. Abstract ICarFactory returns abstract Car object, while concrete SedanFactory returns concrete Sedan object

As an example of a common misconception, I have seen implementations of a GoF factory pattern take arguments describing what kind of product the caller needs. For example, an Abstract Factory following such a design looks as follows:

```

Car car = factory.GetCar("Sedan");
//---- OR ----
Car car = factory.GetCar<Sedan>();
  
```

In this code snippet, the caller specifies what kind of implementation it expects from a factory. This design diminishes the value of a *substitutable factory* since the calling code has already decided what exactly it needs, and thus it does not rely on an abstraction. Furthermore, what is the purpose of a factory if you tell it which concrete product it needs to construct? If your code knows about concrete implementations of an abstract product, you could instantiate them directly without a factory, as follows:

```

Car car = new Sedan();
  
```

Therefore, we conclude that the GoF factory should not take type arguments; otherwise, it becomes an anti-pattern because it stops having a purpose.

CHAPTER 5 IMPLEMENTATION AND CODING

A proper way of using factory is to rely on its abstraction of a product (i.e., Car):

```
Car car = factory.GetCar();
```

If we want to opt into Sedan implementation, we should substitute factory implementation by SedanFactory. The client's code will not need to change since it relies on an abstract Car, which is the essence of smooth substitutability.

Command

The command pattern is not an official part of tactical DDD. However, I decided to cover it here as I consider it to be an indivisible aspect of this area of study. Specifically, commands become handy when dealing with multiple aggregate updates in code.

Consider a scenario where a given domain event affects multiple related entities, such as when a stock price change reevaluates a given investor's holdings. To implement this functionality, we need to subscribe to the mentioned event and update the prices of *all holdings* that possess the stock in question. Recall that in DDD, we do not modify more than one aggregate in a transaction. With such a limitation, how would we implement the described scenario if each holding were a separate aggregate?

One way to overcome the challenge is to dispatch multiple commands from a single price-change event handler, each requesting an update to one related holding. In that case, we can handle each such command separately and modify the affected aggregate without touching others. By doing so, we successfully adhere to the limitations that DDD imposes (see Figure 5-13).

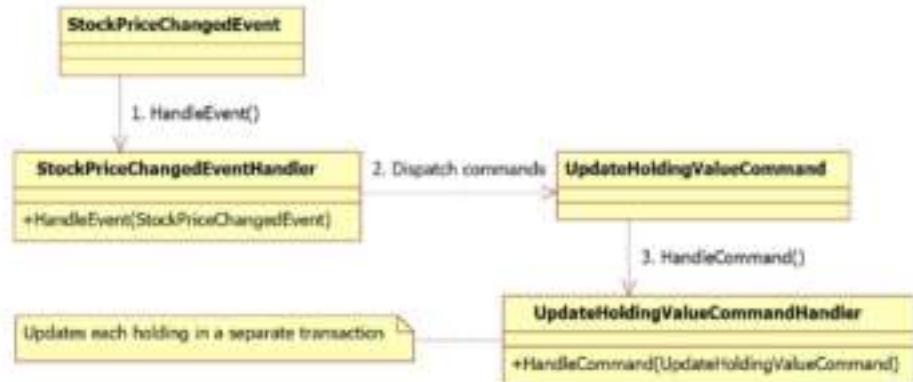


Figure 5-13. Command pattern allows handling updates of multiple aggregates, each in a separate transaction

As you noticed, this solution relies on a *command*, and thus I see it as an essential part of tactical DDD.

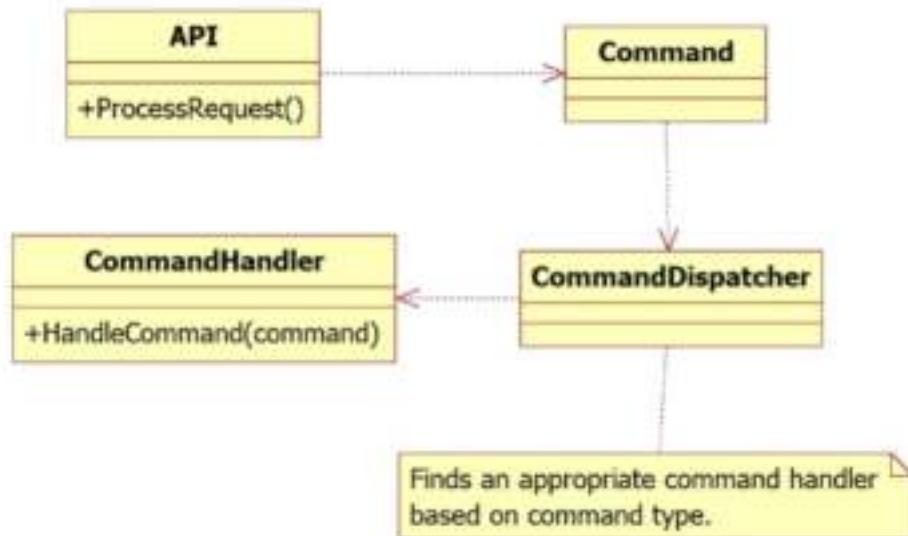


Figure 5-14. API sends a command to a command handler via command dispatcher

Warning About Commands

One common use case of a command pattern is handling requests that arrive at an application such as an API. Typically, there is some kind of dynamic dispatcher between a request processor and a command handler. This dispatcher examines the received command type and finds an appropriate handler to fulfill the request.

There are several motivations behind such architectures, as follows:

- Abstract message routing decouples request processing from command handling.
- We can substitute an abstract message router with a new implementation if necessary.

While I understand these motivations, I disagree with the solution because it is unnecessarily complicated for the given criteria. A simple method call on an interface (see Figure 5-15) would deliver the same effect with a couple of added benefits.

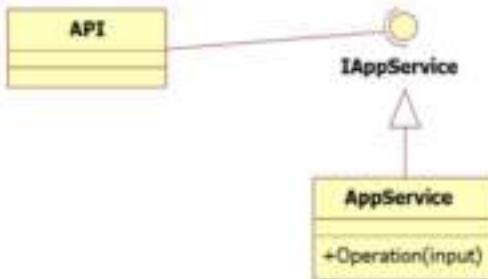


Figure 5-15. API calling an interface method to fulfill a request

An essential difference between the two described techniques is that the second interface defines concrete methods for fulfilling API's needs, as any typical dependency injection would do. I prefer this approach because it has the same benefits as the previous one:

- A service interface is sufficient to decouple request processing from an implementation.
- Since service is an interface, we can substitute a new implementation if necessary.

This solution avoids the unnecessary overhead of finding an appropriate handler for every command instance, unlike the message router that relies on this technique. Furthermore, the dispatcher is not truly substitutable because conceptually it ties together a particular set of command handlers in tandem. Replacing such a router would be impractical because of the necessity of handling so many variations of requests. However, a simple injected interface reveals the design's intention more clearly because every request is naturally paired with precisely one way of handling it.

To conclude, abstract message routing does not seem to be necessary in most cases. Therefore, go with simple solutions unless required to do it differently.

Declarative Design in Code

Perhaps you have already heard that *engineers spend more time reading code than writing it*. While there is nothing wrong with being curious and reading code, this observation hints at the complexity of interpreting and understanding existing code before making modifications. Therefore, it is every developer's job to simplify this journey for themselves and their colleagues. We can achieve this objective by sharpening skills that directly affect how we write code. Among such techniques, declarative design and intention-revealing interfaces are absolute must-haves.

CHAPTER 5 IMPLEMENTATION AND CODING

If you asked me to explain what these two terms mean in my own words, I would say the following:

- Code that I wrote must sound like a requirement that I was implementing when writing it. That connection makes learning more accessible and brings the reader up to speed faster.
- Another way of assessing code's quality is to pose a question: Will I like the code I wrote today if I return to it in a month or two?

The majority of codebases that I have seen in my career fail to meet these two expectations.

Perhaps you are curious about how exactly to achieve a satisfactory level of mastery. I would suggest starting with simple code blocks that do regular data modifications and try to rewrite them to distill more knowledge.

Here is an example of a code that updates an investor's holding values based on a new stock price. We will try to improve this method with the mentioned techniques later. For now, read through the code snippet and think of ways to make it more readable:

```
public void UpdateHoldings(Stock stock)
{
    var price = GetPrice(stock.Id);
    if (price != stock.Price)
    {
        var ids = GetHoldingIds(stock.Id);

        var updates = new List<HoldingUpdate>();
        foreach(var h in ids)
        {
            updates.Add(new HoldingUpdate { h, price });
        }
    }
}
```

```
        UpdateStockPrice(stock.Id, price);
        UpdateHoldings(updates);
    }
}
```

I understand that optimizing such a simple code snippet may seem like overkill, but I figured that it is easier to learn with basic examples.

Several warning signs catch my eye in the preceding code block:

- `GetPrice` is not an intention-revealing name. Without looking inside this method, it is hard to say what it retrieves—a price of a stock or of a product?
- The `updates` variable holds a list of arbitrary data containers, which almost looks like magic (i.e., the design is not declarative).
- The entire content of the method does not clearly explain to a reader what is happening. I would need a couple of additional minutes to figure out what I was seeing. This redundancy amplifies as the method grows over time. Therefore, a rewrite to cleaner code would help until it's too late to change it.

Let me show you how I would rewrite the preceding code to achieve readability and express intentions:

```
public void UpdateHoldings(Stock stock)
{
    var existingStock = GetStockById(stock.Id);
    if (existingStock.Price != stock.Price)
    {
        existingStock.CopyPriceFrom(stock);
        var stockHoldings = GetStockHoldings(existingStock);
```

CHAPTER 5 IMPLEMENTATION AND CODING

```
foreach(var holding in stockHoldings)
{
    holding.UpdateValueByStock(existingStock);
}

SaveStock(existingStock);
SaveHoldings(stockHoldings);
}
```

Here is why I think that the second code snippet is better than the first one:

- All variable names now clearly express intention.
For example, there is no more `h` variable to denote a holding.
- I replaced `GetPrice` with `GetStockById` to retrieve an entire Stock aggregate instead of only acquiring its price. In subsequent lines, I will rely on encapsulated behavior inside this aggregate instead of holding its price in a decimal variable and having to deal with the logic procedurally (i.e., without encapsulation).
- I got rid of the `updates` variable and instead introduced `stockHoldings`, a list of `Holding` aggregates. This change also allowed me to clearly express the intent of updating the value of a holding due to a change in a stock's price.
- Overall, the method better expresses its intention without a need for a reader to reverse-engineer the meaning from technical details. In other words, I tried to shift focus from implementation to business functionality, which always helps when tackling complexity.

One thing that I kept without a change is a modification of multiple aggregates in a single business transaction (stock and holdings). Perhaps you spotted that part as a violation of a tactical DDD recommendation. I did not risk making this move, because it would require more drastic changes, such as the introduction of domain events. For this section, improvements that I applied will suffice.

To conclude, both declarative design and intention-revealing interfaces are easily accessible techniques for developers, even without significant refactoring. Therefore, I highly encourage you to use them whenever possible to improve ROI for every code block you write.

Front-End Development

Earlier in this book, I spoke about my view of front-end development: It seems slightly behind the backend's maturity from an architectural standpoint. This phenomenon happens partly because on the front-end, there are such elements as styling and web design. These areas require specialized knowledge, which automatically puts qualified designers above experienced engineers. Because of such a shift of focus, the best coding and development practices do not get close to the front-end. I respect those designers who help us build beautiful and responsive user experiences, but naturally they cannot cover all exposed fronts. As a result, coding, development, and architectural patterns are not so popular on the front-end.

One proof of this is an observation that most front-end engineers are not good at JavaScript programming, but they feel very comfortable with HTML markup and CSS. While I acknowledge the necessity of web design, I think JavaScript programming is an essential part of front-end development. Therefore, we need to solve this shortage.

CHAPTER 5 IMPLEMENTATION AND CODING

In my mind, there are two possible ways out of the described issue:

1. Front-end engineers get better at coding and architectural patterns, which would allow these specialists to cover the mentioned gap.
2. Front-end development as a practice emerges into two branches—web design and JavaScript engineering—which would enable respective specialists to focus on separate areas of work. Due to front-end frameworks, these two groups can join their deliveries by binding designed and stylized views to coded JavaScript models.

If we can accomplish one of the preceding objectives, we can maximize ROI from front-end engineering by tapping into techniques such as the following:

- Applying best coding practices to front-end. Design patterns and structures were never meant to be limited to a backend; thus, this breakthrough would be quite logical.
- We can evolve front-end architectures for complex applications by introducing layers, domain models, and other advanced structures necessary for high-quality technical solutions.

Since we are speaking about the front-end, I want to discuss a couple of essential design concerns that emerge on this side of things when coding.

Front-End Repositories and Services

By now, you should know from tactical DDD what a repository pattern is. If you decide to evolve your front-end's maturity at some point, you will want to introduce repositories too, just like for the backend. However, I want to discuss this design decision ahead of time since it comes with caveats.

As you will remember, a repository is the storage of aggregates. Usually, this component does not validate input objects and does not have any logic besides persisting and retrieving entities. In other words, a repository only holds your objects until you need them back, similar to a collection or an array.

On the client side, matters look slightly different: a front-end repository typically sends data to a backend via an Ajax call (an asynchronous invocation that does not cause a page refresh), which may or may not pass validation before being inserted into a database. This limitation is intentional since we want to prevent malicious requests from being submitted to the backend if somebody decides to bypass the front-end. With this consideration in mind, a question arises as to whether the front-end repository is a valid pattern. Specifically, is the front-end repository only a data holder if it also validates input via Ajax calls? The answer is yes, as long as your JavaScript domain model validates objects *before* submitting them into the repository. Assuming that front-end business logic encapsulates the same validations as backend endpoints, the repository will have the effect of a simple collection. This statement is true even though the backend still validates the input, but it never fails due to the front-end's prior checks (see Figure 5-16).

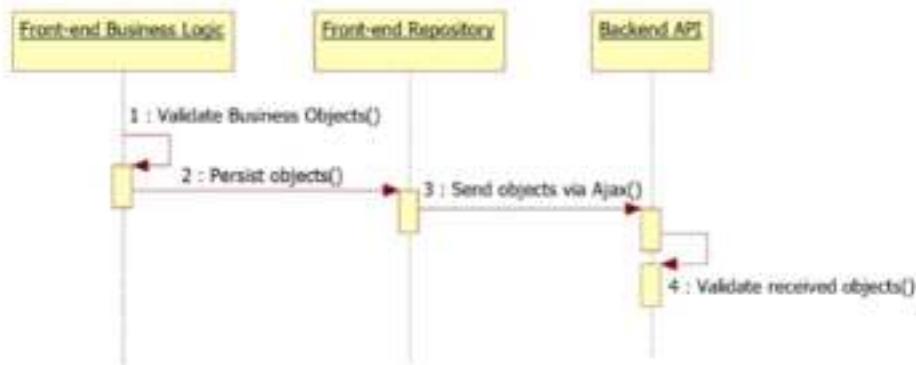


Figure 5-16. Front-end business logic validates objects before persisting them via repository; subsequent validations on the backend pass seamlessly and result in successfully storing objects

CHAPTER 5 IMPLEMENTATION AND CODING

How about cases when the front-end wants to rely on a validation logic that is on a backend? In that case, to stay true to patterns, you need to design this interaction by representing the backend as a *service* interface and not a repository. Similar to how we invoke remote services to execute operations, the backend plays that same role for the front-end. Therefore, in such cases the repository pattern is not applicable since the service interface is a better choice (see Figure 5-17).

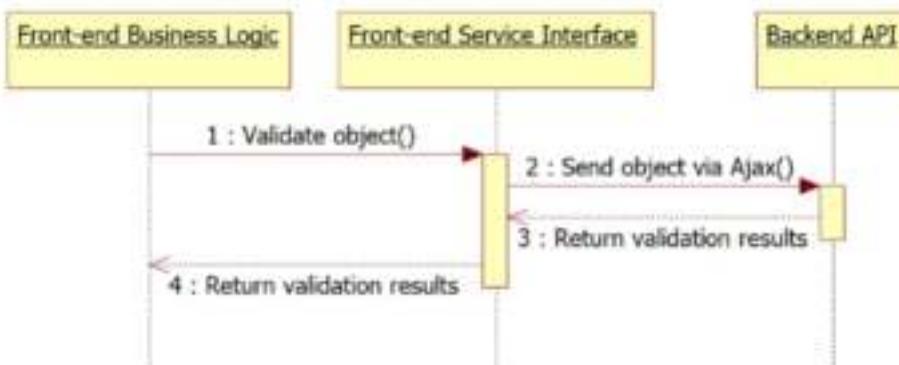


Figure 5-17. Front-end business logic asks a service interface to validate an object; subsequent validation on a backend returns execution results to the front-end

Hosting Static Assets for Front-end

I want to cover another important topic within the area of front-end engineering—hosting static assets.

Static assets require special treatment on a front-end for optimal performance of an application. Specifically, if a given asset is a piece of content that does not change between deployments, it is common to put it on a CDN (content delivery network) and serve from there. One example of a static asset is a CSS file, which we can include on a page by specifying a full path under the CDN server.

The technique that I just described is trendy nowadays, but it comes with overhead. Developers have to reference CDN files in local development, which is a hassle due to the need to republish content to the CDN after every change. I want to address that gap in this section.

Here is my suggestion: Serve static content from CDN only for production environment pages. When developing locally, it makes more sense to reference local files. As you change the contents of those files, you will not need to keep publishing content to the CDN repeatedly, which will improve your development experience.

When deploying an application to an environment that relies on a CDN's content, automatically push assets from a build folder to CDN servers and substitute file references accordingly. You can automate this step via CI/CD pipelines to avoid error-prone manual uploads and code alterations (see Figure 5-18).

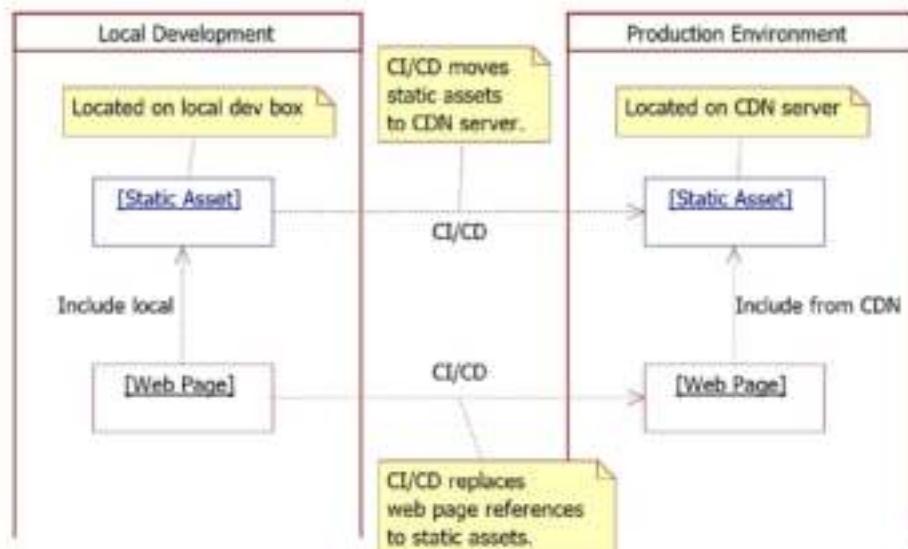


Figure 5-18. CI/CD automatically uploads static assets into the CDN and fixes content references in web pages accordingly

Developing for Various Form Factors

Modern web front-end applications need to be usable on all devices and screen sizes; such capability helps these programs maximize their user bases, which is an excellent business benefit.

At a high level, there are two distinct techniques for rendering application screens on various form factors. I think it is worth briefly reviewing these approaches, both for knowing them and for making the right choice based on your use case.

Responsive design (see [Wiki RWD]) helps instruct elements on a screen to change their positions relative to other components based on current form factors. This technique is useful if you plan to display the same controls on all devices, but their placement must be slightly different. For example, you could design a responsive page for rendering address information. While you could align city, state, and zip code on a single line on large desktop screens, it makes sense to show these bits of information under each other on mobile devices. This trick is possible due to responsive design instructions within CSS rules.

Adaptive design (see [Wiki AWD]) is handy when the program's screens differ drastically across form factors, making it more logical to rebuild an entire display instead of altering element locations only. An example of an adaptive application could be a sophisticated accounting software that would not be convenient to operate on mobile screens. On small form factors, it might be a better idea only to display the current process status in a read-only mode. At the same time, the page allows the manipulating of information on wide monitors. Applications usually implement adaptive designs by detecting current device dimensions and swapping one page with another.

I cannot stress enough how critical making the right choice is. I have seen a tendency to go with a responsive design in all cases, which will only complicate matters and introduce unnecessary overhead.

For example, in a typical scenario, a separate UX design team composes screen mockups, and engineers implement them later. An often-overlooked caveat in this situation is that the UX designers do not use responsive libraries when drawing screens. Consequently, mockups can be entirely different between various form factors. Meanwhile, many front-end developers religiously follow responsive designs, so they do not even suspect that an adaptive approach would be a better choice. Resulting “responsive” pages end up having multiple versions of screen content, half of which are hidden in each form factor.

```
<content>
    <!-- Content for small screen goes here.
        (hidden on large screens)
        Actual markup is omitted from example -->
</content>
<content>
    <!-- Content for large screen goes here.
        (hidden on small screens)
        Actual markup is omitted from example -->
</content>
```

This trick might sound like a good idea, but it has negative impacts on both performance and the developer’s experience. The former is an outcome of needing to download a significant amount of markup, while half of it will not even be visible to an end user. The latter is a problem when a developer tries to change something on a page and unintentionally impacts a hidden portion of the display. Therefore, it must be clear to begin with that adaptive design should be the preferred choice.

I recommend that you become familiar with both approaches and carefully choose one based on the best fit instead of automatically picking either technique.

Keep Pushing the Limits

As I said, the front-end does not enjoy mature development practices like the backend does. However, there is still hope that matters will improve because many engineers are passionate about this field. If you are among such enthusiasts, here are a couple of recommendations to consider for leveling-up the front-end development:

- Become a master of JavaScript. It is a powerful programming language that has a lot to offer besides commonly used and straightforward operations.
- Learn about object-oriented programming and use it when writing JavaScript code. You will discover that this paradigm is becoming more and more popular (e.g., with TypeScript), so this is your opportunity to get ahead of the game.
- Learn about design patterns and layered architecture and use these techniques daily.
- Write unit tests for JavaScript code. Over time, consider writing tests ahead of code via TDD, ATDD, or BDD techniques (I will cover these topics later in this book).

If you are looking for a quick jump-start, make sure to check out my article about advanced JavaScript application architecture (see [Tengiz AJSAA]).

Exception Handling

Perhaps exception handling is one of those topics that deserves a lot of attention but gets none. Most of the exception handling techniques that I have seen in real-world applications catch everything and then either continue execution with a possibly corrupt state or throw an error to kill

the program entirely even if it is not necessary. I know that such code blocks are often written based on the developer's gut feeling rather than a clear understanding of the exception handling guidelines. Therefore, in this section, I hope to clarify a few critical matters around this topic.

Why Catch Exceptions?

A basic rule of thumb is to catch *only* those exceptions from which you can recover. All other exceptions need to bubble up the call stack hoping that an appropriate error handler will catch them. If you want to catch all exceptions regardless of your ability to recover from them, try to answer the following question: What would you do with an exception that indicates memory shortage or stack overflow? There is no way out of those errors. If you attempt to handle these kinds of exceptions, perhaps your handling block will cause more trouble than there is already. Therefore, if you cannot recover from a given type of error, do not catch it.

There is one exclusion from this rule: It is fine to catch all exceptions if all you do is log them and let them bubble up. In this case, you are not trying to recover from an unknown situation, but instead are leaving a footprint that allows others to investigate what has happened later.

Why Throw Exceptions?

When you write a method, you must think of it as an independent component. To its caller, this unit of code is a black box that takes input as arguments and gives the output as a returned value. It saves the developer's time when they can use an existing method without ever reading what is happening inside it. You, a method's author, can provide this convenience by designing your code block against a *contract*. A contract is a piece of the necessary information that the caller needs to utilize a method: inputs and outputs.

CHAPTER 5 IMPLEMENTATION AND CODING

There is one more component of a contract that we often overlook—*exceptions*. If you carefully think about it, an exception is indeed another kind of output from a method. If you do not inform a caller about possible errors that a given unit of code produces, how will they know without reading your code? Therefore, besides taking inputs and returning outputs, plan to throw exceptions that describe error situations about a method, and document them. Try to be precise with this exercise, as every developer who calls the method will want to rely on this information.

Specifically, consider the following points:

- If your method explicitly throws an exception, document it as part of the method's contract. Throw distinct exceptions to allow the caller to uniquely identify one error situation from others so as to apply different recovery steps.
- If your method calls other methods, find out all exceptions that they can throw and decide whether you can recover from them. If you can do so, then write a recovery code. Otherwise, you need to bubble those exceptions up (optionally, after logging them if this is the right place to do so), and then you need to document them as they become part of your contract.

Law of Demeter for Exception Handling

I suggested earlier that a method calling other code units needs to handle exceptions that the nested invocations cause. If you continue applying this design approach to every method in a hierarchical call stack, you end up handling errors of *direct invocations* in every method's body (see Figure 5-19). This technique directly correlates with a well-known Law of Demeter (see [Wiki LOD]). This pattern suggests that every component is aware of its *direct dependencies only* without further concerns of internal implementation details such as deeper-level invocations.

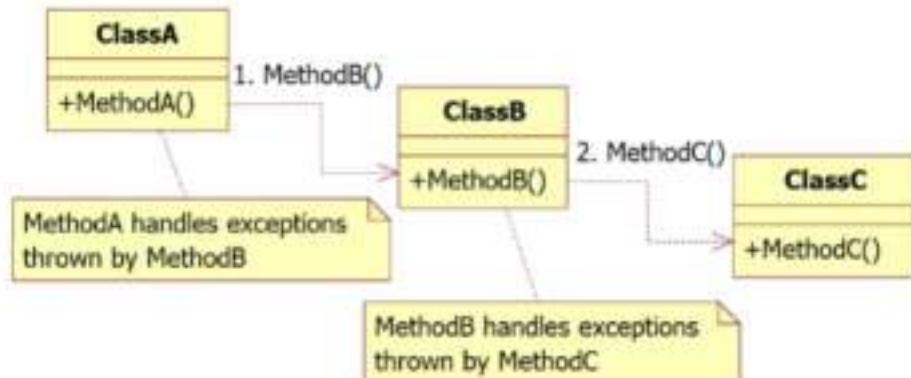


Figure 5-19. In a hierarchical method call stack, each method is responsible for handling exceptions that their direct dependency throws. MethodA is not aware of exceptions thrown by MethodC.

It is a good idea to familiarize yourself with the Law of Demeter to master exception handling accordingly since there are many similarities between the two techniques.

Handling Globally Unhandled Exceptions

There will be exceptions in your program that you will not be able to handle. Some of them will come from ill-designed or poorly documented methods, which you should treat as general kinds of errors as you do not have a way to handle them. In other cases, you will intentionally let the exception bubble up the call stack since it has a critical nature (e.g., `StackOverflowException`—what can you do about it?). Who catches these errors that crash applications and leave no trace?

Fortunately, every runtime and platform has a way to catch unhandled exceptions. Typically, in these code blocks you log an error and let the application die. There is not much more you can do about such occurrences. When somebody decides to investigate the root cause, the log that you generate will be helpful.

CHAPTER 5 IMPLEMENTATION AND CODING

For example, in C# .NET, you can log globally unhandled exception types in the following manner (the code is overly simplified for demonstration purposes):

```
AppDomain.CurrentDomain.UnhandledException += GlobalErrorHandler;  
//...  
  
static void GlobalErrorHandler(object sender,  
UnhandledExceptionEventArgs e)  
{  
    logger.LogError("Unhandled exception occurred: {0}",  
    e.Exception.GetType().Name);  
}
```

After discussing exception handling, I am confident we have covered the essential implementation-related topics. Next, let's see how we can test the code that we have written.

Testing Code

Performing application quality assurance via user interface (UI) and service-level tests has become an integral part of software projects. While this stage is responsible for catching defects and testing systems against user expectations, its cost makes people seek alternatives.

The problem lies in the low efficiency of high-level tests for quality assurance: executing a single intricate scenario against a complex system might take hours, if not days. For instance, a process of order fulfillment typically takes a couple of days from order receipt to shipment. Hence, testing this scenario through a service endpoint or a UI interaction is a lengthy process. If a given system has multiple such functions, you may need to allocate a considerable amount of time to testing. This necessity

extends the length of the feedback loop, delivery cycles, and perhaps customer wait-time. Therefore, it is essential to find more efficient ways of checking the quality of software solutions.

The described issue is the reason why the techniques that target *code-level testing* are so popular. Tests that developers can run against code are much more efficient than those for high-level quality assurance. For example, while executing a single scenario via UI-targeted tests can take hours, you can run thousands of unit tests in a minute.

You can see this difference in efficiency between the two approaches by looking at a well-known testing pyramid (e.g., see [Vocke TPTP]). Unit testing is at the bottom of the shape due to its speed and granularity. Service and UI tests sit on the upper portion as they take longer to run while providing end-to-end feedback.

If you are a developer and want to master code-level testing techniques, you should learn about unit testing, test-driven development, acceptance test-driven development, and behavior-driven development. We will talk more about these exercises in the next sections.

Unit Testing

In this section, I want to cover a couple of topics related to unit testing. I will assume that you are familiar with this technique, so please read about it beforehand (see [Fowler UT]).

Unit Test Coverage Fallacy

When development teams start practicing to write unit tests, they often set a target for coverage in percentage. For example, it is common to hear that a goal is to reach 80 percent unit-test coverage. I suggest that we take that number with a grain of salt. I will clarify the reasons next.

CHAPTER 5 IMPLEMENTATION AND CODING

It is effortless to achieve measurable coverage if you understand how unit-test runners calculate the final number. Most (if not all) tools consider that a given code block is covered if a unit test reaches the location of this block during test execution. Anybody who knows this detail of implementation can trick the tools. One can write a unit test that does nothing but invoke a method in multiple ways, which will be enough to bump a coverage indicator to 100 percent for this code block. I have seen many unit tests using this trick, intentionally or unintentionally, to generate a false feeling of safety. Therefore, I do not take the mentioned measurement seriously: if it looks good, that does not mean anything.

However, I want to clarify that with *well-written* unit tests, achieving 80 percent is a valuable milestone. Please do not shoot for this number just for the sake of seeing it. Instead, do it for improving and assuring the quality of your code blocks.

How Many Unit Tests Are Enough?

If 80 percent coverage is not necessarily a target, how much is enough for writing unit tests?

I recommend you *never chase any percentage indicators*. Instead, focus on quality and an engineering culture that nurtures technical excellence. A professional developer will always attempt to unit-test *every single line of code* without ever wondering about coverage metrics. I admire such specialists and respect organizations that create such empowering environments. It is not about numbers; it is about quality.

Here are a couple of techniques that I recommend to make the most out of unit testing:

- Maximize the code coverage of business logic. Tests targeting this area serve as both quality checks and executable documentation. Also, domain layers are the most valuable part of an application, so it must be the center of your focus.

- If you want to cover every single line of code, practice TDD, which ensures that no code exists without tests. This technique asks you to write a test first and then code to ensure the test passes. In other words, you will not write code for which you do not yet have a test, and hence the entire code is covered by tests out of the box.
- Make sure to write unit tests for both backend and front-end code. Fortunately, various frameworks are targeting every programming language, so take advantage of them. Commonly, teams write unit tests only for backends, but the front-end does not have to be an exception from this exercise.

Multiple Asserts in a Single Unit Test

A unit test consists of three parts—"arrange," "act," and "assert." It is a common misunderstanding that you can only have one assert statement in a test method. If you fall into this trap, you will end up duplicating both "arrange" and "act" across multiple tests, which only differ by "assert." Such a code duplication increases maintenance costs as you will need to keep similar tests in sync. Therefore, let us agree that a unit test can have multiple assertions as long as it tests a single "act" statement.

Here is an example of a unit test written using Xunit with multiple assertions, and it is still a well-designed test:

```
public sealed class Address_Parse_Should
{
    ...
    [Fact]
    public void Interpret_String_With_Unit_Number()
    {
        //arrange.
        ...
        //act.
        ...
        //assert.
        ...
    }
}
```

CHAPTER 5 IMPLEMENTATION AND CODING

```
var addressString = "1600998 Georgia Circle, Unit #8,  
Atlanta, GA 12123";  
  
//act.  
var address = Address.Parse(addressString);  
  
//assert.  
Assert.NotNull(address);  
Assert.AreEqual("Unit #8", address.Line2);  
//TODO: add more asserts to test the same output object.  
}
```

The shown code rightfully contains multiple asserts since it deals with a single distinct scenario, defined as a unique “act.” Furthermore, it is alright to add more asserts to this test method if they also relate to this exact scenario.

Also note that in the preceding example, the first assert checks a prerequisite that must be positive before subsequent asserts. If we did not have an assertion against the null instance, the chances are that the second assert would fail with “NullReferenceException.” Thus, the first assert changes the behavior from “NullReferenceException” to “AssertException,” which is much more developer-friendly when it occurs.

TDD, ATDD, and BDD

After you have mastered writing unit tests, the next step is to advance your skills via TDD, ATDD, and BDD. I will explain the crux of these exercises and clarify how they relate to each other.

TDD Is Faster Than Unit Testing

Who would think that writing unit tests before you write code (i.e., test-driven development—TDD) allows you to release code faster than writing unit tests after writing the code?

The basic idea behind the mentioned hypothesis lies in running unit tests to verify functionality instead of instantiating an entire application. Since the former approach is more efficient and cost-effective than the latter, TDD wins over writing unit tests after code is complete.

Without going into more detail, I want to refer you to my article online about this topic (see [Tengiz WTDDIFTUT]).

Commonalities Between TDD, ATDD, and BDD

I decided to cover these three exercises together because they have something in common: TDD, ATDD, and BDD are all techniques for *writing tests before writing code*. Each of these exercises solves the task from a different angle or level.

TDD (test-driven development) focuses on testing each public method in a codebase. Each of these targets may or may not correlate directly with any given business capability because tested units stand at a lower level conceptually. Typically, developers follow TDD while writing *unit tests* for code that they produce.

ATDD (acceptance test-driven development) focuses on writing *acceptance tests* before implementing functionality that would meet given acceptance criteria. Usually, acceptance tests attack capabilities from a higher level than unit tests, and thus they might not catch all fine-grained edge cases the way that unit tests can. However, acceptance tests give more precise feedback on whether an application meets given business criteria. Typically, both developers and quality assurance engineers can practice ATDD since both specialists could write acceptance tests. A developer writes these kinds of tests via integration or component testing; a QA engineer can practice ATDD for integration, systems, UI, API, or end-to-end tests.

BDD (behavior-driven development, see [AA BDD]) is a unique way of practicing ATDD when we *mix a human-readable language with acceptance test code*. An example of such language is Gherkin (see

CHAPTER 5 IMPLEMENTATION AND CODING

[Gherkin]), which you can combine with one of the existing frameworks to write acceptance tests (e.g., see [Tengiz XGQ]). Typically, both developers and QA engineers can practice BDD in the same way that they exercise ATDD.

Next, let me put the coding in the often overlooked yet important context of deployments and maintenance.

Code Deployment and Maintenance

While deployment and maintenance may be the last things developers consider while writing code, they are vital for successfully delivering your solutions to customers. Let's briefly talk about these topics in the context of coding.

CI/CD Before Development

I will talk about CI/CD (continuous integration and continuous deployment) elsewhere in this book. In the current section, I only want to suggest that development teams implement CI/CD pipelines as early as possible. I recommend this approach because delivering solutions is more efficient when CI/CD pipelines are already in place. Without this infrastructure, development teams will need to manually install an application every time to every unique environment. When unplanned issues occur while deploying a system, developers apply ad-hoc fixes to hosting environments or external configurations. Guess what happens when they try to deploy the application to another environment? Nobody remembers which exact ad-hoc repairs are necessary to have the system up and running. Eventually, every environment turns into a special case that is hard to replace. This outcome limits the agility of an organization since solutions become tightly coupled to concrete resources.

CI/CD practices and technologies exist to avoid problems such as these by exercising frequent and automated builds and deployments. Therefore, if the mentioned benefits are valuable to you, it is better to start by planning CI/CD processes before building an application. Upfront investment in the right tools and techniques will ensure long-term success and increased ROI from deploying and running your software systems in all environments.

Planning Deployment When Developing

In earlier sections of this book, I suggested that you treat every code change as a separate version of an application. When coding a solution, you need to adhere to this advice in various ways on a case-by-case basis. I want to provide a couple of practical examples for those readers who prefer the distillation of information around this area.

If your code change requires altering a database schema, prepare a script that makes that modification and consider it part of the same version as the code change itself. To follow this advice in practice, you can either write scripts manually or do it automatically via database migrations (e.g., .NET's DB migrations) or tools such as Redgate SQL Compare (see [Redgate SC]). You can also find other similar solutions that better fit your taste.

If your code change relies on a certain kind of data in a lookup table of a database, write a script that generates it. Alternatively, write documentation that explains how to get the right data set into the database as a prerequisite.

Generally speaking, you need to brainstorm and find ways to turn your code change into a self-contained package.

As a last resort, always compile a document that contains installation instructions if your code change assumes that a specific resource exists. Do not forget to document the steps to uninstall it as well, including the rollback scripts or instructions for related components, if any.

CHAPTER 5 IMPLEMENTATION AND CODING

You might not need any of these tricks if your application is pure code and nothing else, as in this case, both installation and rollback are a matter of replacing binaries. In all other cases, the preceding advice will help you be successful with production rollouts and rollbacks.

When the time comes to deploy your application, you should have all the necessary instructions and scripts ready ahead of time. This preparedness will reduce mistakes and improve the quality of deliverables.

Planning Maintenance When Developing

Like the previous suggestion, developers need to consider ways to maintain an application when it runs in production. This objective is the basic idea behind having proper logging and monitoring of a program.

Indeed, you cannot account for every possible scenario, but usually it is feasible to brainstorm and ensure that you deliver maintainable applications to the production environment. Doing so does not require any notable cost overhead but will significantly improve customer satisfaction.

Summary

In this chapter, we discussed topics that will improve the efficiency of coding and implementation activities, such as the professionalism of developers, knowledge of OOP and multiple code design patterns, hands-on coding-related subjects including tactical DDD, testing via test-first approaches, and coding in the context of deployment and maintenance. There were plenty of areas that developers should find helpful to improve their daily effectiveness when delivering solutions.

But does the solution delivery end with coding? Of course not. We need to talk about testing and quality assurance now.

CHAPTER 6

Testing and Quality Assurance

Performing quality assurance on software products is an essential phase of building applications that meet or exceed customer expectations. How you execute this exercise can make the difference between the failure and success of this book's mission—delivering working solutions.

Producing high-quality applications is the responsibility of the *entire team*; you can't leave it up to only quality assurance engineers. Furthermore, as you will hear me recommending in subsequent sections, developers need to learn and practice in this area as much as testers do.

Therefore, do not ignore this chapter even if you are a software engineer; every development team member will benefit from reading this part.

We will start with testing processes and principles to cover areas that support real-life testing activities.

Testing Processes and Principles

Before we delve into the technical aspects of testing and quality assurance, let us talk about the processes, principles, and other supporting ideas that make this exercise more effective, efficient, and beneficial to help you deliver high-quality solutions to your customers.

Who Is For and Who Is Against?

A prevailing opinion in software development teams conveys the dysfunction that I want to discuss in this section:

"Engineers build features and testers break them."

Developers tend to ignore what testers do or say because they see it as hindering success. By doing so, developers end up neglecting insights for building better software, which is the real objective of quality assurance.

As testers see and feel the disregard for quality measures, they start questioning developers' credibility in building high-quality applications. This reaction not only negatively impacts team spirit, but also further aggravates the ignorance that developers demonstrate. Looking at this situation from the side creates an impression that testers are protecting quality, and developers do not care about it. Worse, this perception can slowly become reality and turn into a culture that is hard to change after a while.

An extreme peak of this anomaly is when *testers reward each other for finding bugs* while developers struggle to handle a flood of defects. Sound familiar? Such circumstances indicate a lack of trust, team spirit, and cooperation between two vital functions of the value delivery stream—engineering and quality assurance. We shall try to promptly fix this dysfunction as it derails customer satisfaction and organizational success due to increased bugs and decreased return on investment (ROI).

Quality Is Team's Responsibility

"If you want to go fast, go alone; if you want to go far, go together."

—an African proverb

In most cases, the previously presented anti-pattern starts from leadership approaches that need course correction. Unfortunately, I have met both quality assurance and engineering managers who aggravate the problem by directing either side toward competition instead of cooperation.

For instance, a QA manager might reward testers for finding bugs instead of guiding them to work with developers on quality improvement. Do not get me wrong—I prefer that testers find bugs before our customers do; however, without cooperation with engineering, defects will keep existing, and rewards will not help. That is where my concern is.

On the other hand, development managers might advise engineers to stay far from quality assurance, which segregates two responsibilities instead of encouraging a joint focus on quality.

Therefore, try to dissolve silos instead of enacting walls. Establish a partnership between engineering and quality assurance and stop rewarding those who find defects. Consider bug discovery and resolution to be the entire team's responsibility instead of putting it on a specific role. This mental adjustment increases attention to quality at the development stage and improves collaboration between various functions when they find a broken functionality. If you succeed with these cultural shifts, you have a chance of restoring team spirit. This outcome is essential to delivering high-quality solutions to customers while also improving employee satisfaction and engagement.

Test Everything, Automate Everything

If you want to build high-quality software, prepare to *test and automate everything*. I know that this advice might sound like an unwise investment, but I will shortly explain the reasons behind it. Afterward, I will give you hints for doing it all in practice.

Why

Low quality standards have a rippling effect on an entire process of software development, starting from engineering teams and ending with customers.

If a developer makes a code change, the chances are that something worked before, but now it is broken. An engineer needs to know about this problem as soon as possible to take immediate action for course correction. Lack of such feedback necessitates longer wait times and undesirable context switching, which impact efficiency and deliveries. Furthermore, if there is no way to detect issues, broken features can even end up in the customer's hands.

A similar challenge occurs when you elevate your program from one environment to the next; for instance, from test to staging. You need an ability to quickly assert that everything works as it did previously since data or configuration differences can affect the system's functionality. If technical team members miss such glitches, your customers will undoubtedly find them during the next elevation from stage to production.

I know that many teams try to fight such problems with manual testing, but it is not optimal in the long run. As a software product grows, the repetition of manual regression tests becomes cumbersome, time-consuming, mechanical, and error-prone. Such inefficiency begs for automation, which delivers the benefit of repeatability and a fast feedback loop for both developers and testers.

How much automation is enough? It depends on whether you can say what frustrates your customers and what is acceptable when it cracks. If you do not anticipate adverse outcomes from broken capabilities, maybe you do not need to worry about automation. In other cases—which represent the majority—it is critical that *everything functions always*; hence, you will need to automate every single feature.

While it may sound like a lot of work to automate everything, subsequent positive outcomes are worth the hassle. Imagine having almost real-time indicators that tell you when something is wrong, and you can make informed decisions about immediate next steps. Customers notice such operational differences, and they reward you by sticking around for better products and experiences. Isn't this the best prize that you can have?

How

Let us assume that we agreed on the necessity of testing and automating everything. A more challenging question is *how* to achieve this level of assurance with the right balance of resources, time, and effectiveness. On the one hand, nobody wants to deal with maintaining thousands of automated scripts that break so often that testing and further automating new features become impossible. On the other hand, a test suite that covers only a fraction of all scenarios is easy to maintain but not good enough to trust its results. How can we gain the benefits of both worlds at the same time? We must automate everything, and do so in a sustainable fashion! Let us see how.

A crux of my recommendation lies in a test pyramid that I mentioned before (see [Vocke TPTP]). Simply put, I would describe the trick as follows:

- All tests form the same pyramid, so we need to look at all levels of tests together and not in silos. You cannot reason about the relative number of the unit and API tests if you do not treat them as part of the same suite.
- We must strive to write more lower-level and fewer higher-level tests since the former are cheaper to implement and maintain. Furthermore, we must apply such analysis to an entire test suite with all test levels instead of splitting suites by function; i.e., unit tests owned by developers versus higher-level tests retained by testers.

- A test suite still needs to cover all scenarios as it would without the pyramid. The only difference in such an approach is a preference toward lower-level tests whenever possible. For example, if a unit test can indicate a broken feature, there is no need to automate it via higher-level API or UI tests.

If you carefully follow the preceding steps, your test suite will have significantly more unit tests than API and UI tests, decreasing maintenance costs. Simultaneously, as the test suite still covers all scenarios, you will have a reliable indicator that can alert you when something is wrong.

Importance of Test Automation

The test automation process is responsible for quality assurance and fast feedback, which are vital prerequisites for building ideal, working software solutions that meet customer expectations. Despite this correlation, many organizations and teams treat test automation as extra work rather than as a mission-critical operation. A couple of observations from my experience prove this point, as follows:

- There are usually silos of development and QA functions. Supposedly, the former represents the engineering power, and the latter is responsible for building and maintaining automated scripts. Subsequently, these two groups do not share knowledge and best practices, which dramatically diminishes ROI from test automation.
- QA work often gets less funding than development, as it is considered less critical for business outcomes. As a result, automation often falls behind engineering due to a lack of resources, talent, and time.

- Developers frequently look at the automation codebase as a black box that is not interesting to them, and only QA engineers are responsible for understanding it. Because of such prejudice, when testers become a bottleneck, there is nobody to help them out, and automation starts falling behind the development.

Due to these pain points, test automation fails to deliver optimal results for quality assurance and fast feedback. As this gap is alarming, I want to explain how to prevent or solve the described problems and dysfunctions.

Who Owns Test Automation?

First things first—*entire teams own test automation* because it benefits everybody equally. An organization's engineering culture must encourage assembling automated scripts as a team instead of dealing with them in a silo.

I have often heard developers complain that testing is not their job. This point is fair since QA engineers have better skills to do this kind of work. However, automation is not about testing; it is about building mechanisms that inform us when something is broken. Automated scripts are as much code as any feature that developers produce, so there is no difference from this standpoint. Therefore, engineers must not hesitate to work on an automation codebase. If you see concerned developers due to such expectations, it is time to explain that *automated tests help engineers receive fast feedback and release working products faster*. If there were no testers on a team, would the group decide to give up on such a tremendous benefit?

While I encourage developers to write automated tests, I expect testers to compose test cases that serve as instructions for automated scripts.

Automate in Time

Now that I have explained the concept of shared team ownership of test automation, it becomes easier to explain how you can automate every feature without falling behind the development pace.

As soon as a team decides to work on a set of requirements, testers can start writing test cases while developers start to build code. When test cases are ready, testers can start automating them.

A common challenge with this approach is having a limited number of QA engineers on a team, which soon turns automation into a bottleneck. How do we handle this shortage? If you recall from a previous section, automation is not the sole responsibility of testers anymore, so you have an entire team at your service! When one of the engineers finishes developing a feature, they can take on automating test cases to fill the gaps that QA engineers cannot cover.

A less optimal but typical approach would suggest starting to develop new features as developers become available instead of letting all team members write automation. However, such a process only increases the burden on testers instead of mitigating a bottleneck. Besides, as we discussed earlier, starting a new development would accumulate *work in progress*, which would delay feedback loops, increase context switching, and negatively impact quality and ROI.

Therefore, instead of keeping development and testing efforts separate, consolidate and optimize the entire team's workflow holistically.

An ideal outcome from the described technique is a *joint team effort* for delivering working, fully automated features along with test cases. If you see this effect, you are on the right track!

Apply Best Practices

As I expressed earlier, the automation codebase is as much code as the application that it tries to test. This statement is fundamental when thinking about possibilities and opportunities in this area of work. Specifically, best coding practices and design patterns apply to the automation codebase just as they apply to any other code.

I will cover automation design patterns in more detail later. In this section, I want to encourage you to check out that information. Never underestimate the importance of standards and best practices for automation scripts, because they are an essential part of successful software projects. If you want to have a reliable mechanism that alerts you to problems, you need to invest in building one. Applying design patterns and practices to the automation codebase is one such investment that people often overlook as they do not realize that such things exist or are relevant.

Blur the Line

Perhaps you have noticed a pattern in my recommendations—I am presenting test automation as an addition to development work, instead of separating one activity from another. I am trying to blur the dividing line because it makes implementing high-quality solutions easier and more practical.

Interestingly, many modern software development companies try to make this same shift by recognizing that both testers and developers are engineers. I agree with this trend, and I recommend that you also start considering the entire team's work holistically. In a nutshell, if all members of a group can help each other when implementing or automating features, you will gain much more agility against piling up work in progress and toward delivering more high-quality features with fast feedback loops. These benefits are too much to give up, so do not dismiss this advice lightly. By eliminating functional silos in your teams, you can achieve optimal ROI from development and test automation!

Earlier, we discussed design and architecture for coding and implementing software solutions. Is there a similar area of knowledge for testing as well? Let's find out in the next section.

Test Design and Architecture

As an organization matures into test development and automation, they start to see the need to apply architectural elements to this discipline, just as with the application code. Therefore, I will discuss this view in the subsequent sections.

Test Types

Besides the test levels that we discussed earlier, a knowledgeable QA engineer also needs to distinguish between various *test types*. Unfortunately, this area of study might be a bit confusing due to many authors' perceiving and explaining this topic disparately (e.g., see [ATSN DTOTS] and [GFG TOST]). Irrespective of which terminology you prefer to use, you need to be familiar with most test types in order to apply appropriate techniques to various situations.

I am sure that you can learn this topic on your own since there is a vast amount of information online. In this section, I will only cover a common confusion in distinguishing between two popular test types.

Integration vs. Functional Tests

An automated integration test script typically queries or posts information into a screen or an API and then verifies results via a backend such as a database. This technique ensures that a front end exposes data precisely as it is present in a backend.

Meanwhile, a functional test is responsible for checking the functionality of a screen or an API without any backend. Because of this, testers commonly think that there is not much value or scope in this approach since it does not have a way to assert the correctness of outcomes. As a result, functional tests cover only those basic scenarios that do not rely on a backend; in other cases, functional tests look like integration tests because they validate the results against the backend. These anti-patterns come down to a misunderstanding of what a functional test is, so I want to clarify this subject.

A functional test is a vital technique for system verification. It asserts functionality from the end user's standpoint instead of concerning itself with the implementation details of a backend. On the contrary, an integration test only checks the technical correctness of an implementation and not what the user expects to see; *these two objectives are fundamentally different*. For clarity, let me demonstrate what happens if you only have integration tests and no functional tests.

Suppose that a screen displays today's deals from a database. A marketing department expects to see special offers on Black Friday, but they notice that prices are not as low as needed. When they complain about it, testers check the automated test suite, which consists of integration tests, and find no failures indicating something is wrong. Indeed, the screen precisely displays what is in a database, which is enough to pass an integration test. It turns out that somebody forgot to update database entries in the morning! Which test is responsible for catching such issues? If we only rely on integration tests, we will never suspect that users do not see deals that they should see. Therefore, we miss a more appropriate kind of a test for covering this use case—a functional test.

How do we know what users need to see if such information is not in a database or anywhere else except in somebody's mind? As an option, you can store expected content in a static location accessible to functional tests, and verify a screen against that instead of comparing the screen to

database records. Yes, this test will need to treat the screen as a black box, and that is by design. After all, users also look at our applications as black boxes without worrying about their implementation intricacies. Since functional tests mimic user behavior, a black-box approach is entirely appropriate.

Test Case per Requirement

A new test case often originates from a new requirement. However, every story is not necessarily a separate scenario, contrary to what most testers assume when structuring test suites. I want to distill this advice by looking at a basic example.

Suppose a development team received a request to implement a sign-in functionality that ends with displaying the current user's phone number on a screen. This requirement resulted in a new scenario that testers automated. In a couple of weeks, the same group received another request to display a user's address on the same screen beside their phone number. Testers repeated the usual approach and added another scenario to cover this new story. As a result, they have a test suite consisting of two test cases:

- Test case #1: Sign in to an application and verify that a dashboard displays a signed-in user's phone number.
- Test case #2: Sign in to an application and verify that a dashboard displays a signed-in user's address.

Do you spot the duplication between these two scenarios? Arguably, a more efficient way of automating the second story would be to extend the existing test case instead of introducing a new one.

I know that there are two distinct schools of thought around this concern, and some might disagree with my view. Here is why I believe that the suggested approach is more advantageous:

- If two test cases only differ with what they verify, perhaps they are targeting the same scenario; hence, there is an opportunity to merge them. This technique yields a shorter test suite execution time and improved maintainability since it avoids code duplication.
- Interestingly, earlier in this book I suggested merging unit tests for a similar reason—when “arrange” and “act” steps matched. Thus, this technique relies on the same well-known coding guideline as lower-level unit tests do.
- In practice, many requirements ask to *slightly modify an existing functionality* that is part of an existing scenario. Hence, instinctively, it makes sense to follow this guideline instead of forcing a new test case creation mechanically.
- In the long run, since many stories will alter existing capabilities, any particular test case will correspond to a system’s behavior rather than a specific requirement. This correlation is helpful when understanding and verifying the system’s functionality instead of fixating on a particular user story that resulted in a test case. After all, a test suite needs to represent an application’s capabilities instead of describing a product backlog!

For all mentioned reasons, you must avoid the mechanical creation of a test case for every requirement. Instead, reconcile each new story with an existing test suite, by either extending or updating it.

With these conclusions, the previously shown example of test cases turns into a single scenario, which looks like this:

- Test case #1: Sign in to an application and verify that a dashboard displays a signed-in user's phone number *and* address.

Test Automation Design Patterns

As I emphasized earlier, test automation is an engineering concern, and thus design patterns and best practices apply to this area as well. By learning and employing design patterns, you will enjoy their typical benefits—clean and efficient code, leveraging knowledge instead of reinventing it, a shared vocabulary, and so on. Due to these positive outcomes, every automation engineer must know several fundamental design patterns, which I will cover next.

Crux of Test Case Patterns

A design pattern often has a *shape of abstraction* that it represents. For example, a repository pattern looks like storage of things. Likewise, if I wanted to describe a test case pattern, in my mind it would look like a scenario that this test case covers. This simple idea is the crux of two models that I will cover next.

Gherkin Test Case Pattern

If your team employs Gherkin language (see [Gherkin]) when formulating scenarios, perhaps it makes sense to model your test cases around this dialect too.

For instance, a Gherkin scenario text might look like this:

Scenario: Successful sign-in

```
Given I stand on the sign-in page
When I enter username 'JohnSmith'
And I enter password 'p@$$w0rd'
And I hit a sign-in button
Then the dashboard page shows up
```

How can we model a test case pattern to mimic such a scenario?

Easy! Imagine an object that has five methods on it, each implementing a corresponding step of the scenario. This construct is a *Gherkin Test Case design pattern* (see Figure 6-1).

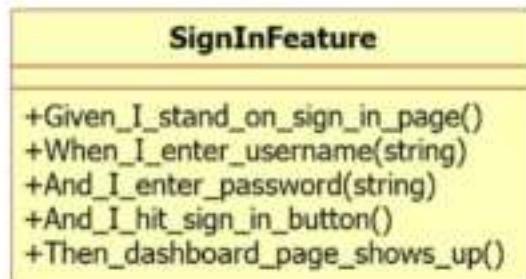


Figure 6-1. Gherkin Test Case design pattern: each scenario step has a corresponding method implementing it

To execute code from the Gherkin Test Case object, you can utilize one of the specialized frameworks that parses Gherkin text and map your object's methods to scenario steps (e.g., see [Tengiz XGQ]). Such structures help you implement the described pattern by letting you focus on test automation instead of language-parsing routines.

Scenario Object Pattern

If you use the popular Gherkin language for writing scenarios, perhaps the Gherkin Test Case pattern is your best bet. However, more often than not, test scenarios do not rely on Gherkin, and hence the described design does not apply. Instead, when writing test cases, we often end up structuring them as our tools or conventions govern.

For instance, if you use the Microsoft Test Management (MTM) system to define test cases, you will notice that it does not follow the Gherkin format. Instead, MTM guides us to define a testable scenario via sets of "action" and "expected result." Under such circumstances, we would have written the previously shown scenario as Table 6-1 shows.

Table 6-1. *Sign-in Scenario Follows the "Action" and "Expected Result" Format*

Action	Expected Result
Open sign-in page	Sign-in page opens up
Enter username 'JohnSmith'	
Enter password 'p@\$\$w0rd'	
Hit sign-in button	Dashboard page opens up

It is hard to fit such a scenario into the Gherkin Test Case pattern since there are no "given," "when," and "then" steps. If you try to employ the Gherkin Test Case anyway, you will need to decide how to map the shown six statements into an object that would otherwise have five methods, as I presented earlier. I recommend avoiding such attempts since it will complicate the automated test approach and corresponding code due to forcefully fitting a solution to a problem. Instead, let me introduce a design pattern that closely mimics this test case.

Scenario Object is a general-purpose design pattern representing a *test case of any shape* as an object. This structure is a generic form of the Gherkin Test Case.

For the preceding example of a test case, Scenario Object will have six methods, each corresponding to either an action or an expected result in the table (see Figure 6-2).

SignInFeature
+Open_sign_in_page()
+Sign_in_page.opens_up()
+Enter_username(string)
+Enter_password(string)
+Hit_sign_in_button()
+Dashboard_page.opens_up()

Figure 6-2. Scenario Object pattern: each sentence from a scenario has a corresponding method implementing it

The final piece of the solution is to execute the Scenario Object's methods in the necessary order. You can achieve this goal via a simple invocation chain from a test method or a generic code that dynamically discovers and calls methods.

Page Object Pattern

Automated test cases frequently verify web page functionality via frameworks such as Selenium (see [Selenium]), which allows working with DOM (Document Object Model) in an object-oriented fashion.

This practice has shown that it is more convenient to write automation code if we encapsulate Selenium's objects via a pattern such as Page Object (see [Fowler PO]). This construct is a simple object that wraps around the page, and we can use it from test methods instead of directly working with the screen's elements.

I have seen various interpretations of the Page Object pattern, and I have noticed that a certain kind of misuse has become increasingly popular. I want to warn you against such a pitfall.

Predominant Misuse of Page Object Pattern

When engineers try to implement the Page Object pattern, they often overlook that this construct relies on *encapsulation*. As a result, most Page Objects that I have seen violate this principle and expose Selenium's low-level objects via properties or other members (e.g., see Figure 6-3).

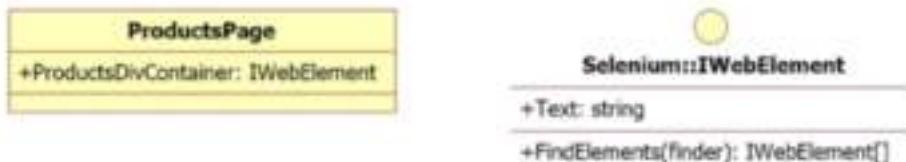


Figure 6-3. *ProductsPage* class implements the Page Object pattern. This class exposes Selenium's low-level *IWebElement* instance via the *ProductsDivContainer* property

Here is an example of using such a poorly designed Page Object to verify that product names properly display on a screen:

```

// `page` variable represents Page Object.
IWebElement productsContainer = page.ProductsDivContainer;

ReadOnlyCollection<IWebElement> productDivs =
productsContainer.FindElements(By.ClassName("product-div"));

for(int i = 0; i < productDivs.Count; i++)
{
    IWebElement currentDiv = productDivs[i];

    //verify the product's name.
    Assert.Equal(expectedNames[i], currentDiv.Text);
}
  
```

As you can see, this test code briefly uses a Page Object to acquire an IWebElement container; the rest of the code traverses DOM to verify page content via low-level implementation details without utilizing the Page Object. Such an approach sounds like a good idea: it simplifies matters within the Page Object because it only needs to expose a single container element, and the rest of the complexity lives somewhere else. The problem is that, by doing so, we add complexity overhead to *every test method* that decides to use this Page Object. Such a preference is naive since we have sacrificed a lot for little gain. Also, we have violated the encapsulation principle of OOP for Page Object.

There are a few additional problems with the described anti-pattern that are worth mentioning, as follows:

- Test code has become tightly coupled with the implementation details of Page Object and thus has fallen into the trap of vendor lock-in that I discussed earlier. If we ever decide to replace Selenium with another, similar framework, we would need to refactor every test method with access to low-level details such as the IWebElement interface of Selenium.
- Since Page Object does not encapsulate access to information, test methods will have a large amount of duplicate code that identically traverses data. For the preceding example, such duplication will be necessary when we want to loop through the same HTML elements elsewhere. Furthermore, if page structure changes, we will need to rewrite all those test methods instead of fixing code only within Page Object.

CHAPTER 6 TESTING AND QUALITY ASSURANCE

- The code that I showed earlier is not intention-revealing because it speaks about HTML elements rather than articulating a business logic. Automation code is a great candidate for expressing domain knowledge since it serves as both a verifying and an educational tool. However, because we have decided to expose implementation details from Page Object, we have encouraged test methods to forget about business logic and focus on page technicalities.

Now that I have listed issues related to this anti-pattern, let me demonstrate how I would fix it, in Figure 6-4.

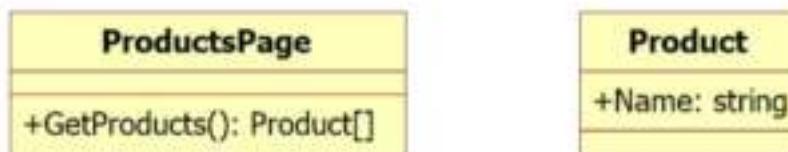


Figure 6-4. *ProductsPage Page Object encapsulates access to Selenium's IWebElement and instead exposes access to custom Product class instances*

As the diagram demonstrates, Page Object does not expose low-level details anymore; by doing so, it encourages us to write cleaner and more intention-revealing test methods:

```
// `page` variable represents Page Object.  
ReadOnlyCollection<Product> products = page.GetProducts();  
  
for(int i = 0; i < products.Count; i++)  
{  
    Product currentProduct = products[i];  
  
    //verify the product's name.  
    Assert.Equal(expectedNames[i], currentProduct.Name);  
}
```

In the preceding code, `Product` is a custom class that exposes the `Name` property. Page Object initializes `Product` instances when we invoke the `GetProducts()` method instead of giving access to `IWebElement`. Notice how this refactoring has improved the readability of the test method! We do not need to parse through page intricacies such as the hierarchy of DIV elements or their structures. Instead, we can focus on a simple verification of product names that the screen displays. Besides, we have eliminated all other mentioned problems that were inherent to the original misuse.

Therefore, when designing Page Objects, never expose implementation details that relate to Selenium or other HTML DOM abstractions. Instead, encapsulate low-level structures within Page Objects and make only domain-specific information accessible to the outside world.

Test Object Pattern

Perhaps you will agree that the Page Object pattern is much more comfortable and cleaner to use than dealing with the screen's HTML structure directly. The limitation of this technique is in applicability—it is only useful for testing pages. Would it not be nice if we could apply the power of encapsulation to other situations as well? For instance, think about API test scenarios: Do we have to work with HTTP clients and protocols instead of business logic? While Page Object helps us to work with pages in an object-oriented fashion, there has to be something that aids with everything else. Keeping this objective in mind, I want to generalize Page Object into the *Test Object* pattern.

Test Object wraps the testing target and hides its implementation details (see Figure 6-5). After that, the outside code can leverage Test Object's members to interact with or verify the target's behavior in an object-oriented fashion.

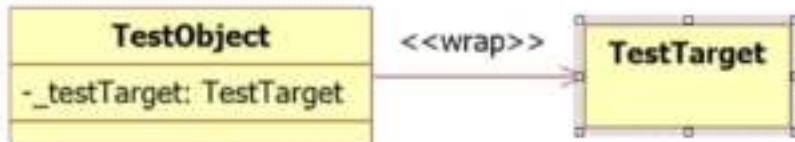


Figure 6-5. *Test Object design pattern: encapsulate access to test target*

For instance, if a development team builds an API for external consumers, an engineer can automate verification of delivered endpoints via the Test Object pattern. The result is an object that encapsulates interaction with the API via methods that speak the business language. Let me demonstrate this pattern in action with an example of API for products in Figure 6-6.

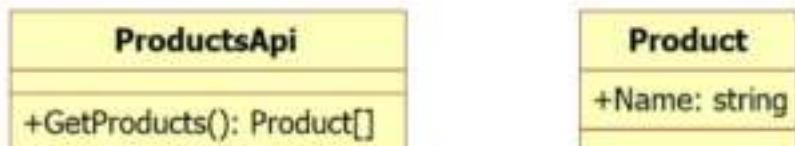


Figure 6-6. *ProductsApi is an implementation of Test Object pattern that wraps interactions with an API of products*

Instead of entangling with implementation details, test methods can express domain-specific intentions when asserting the correctness of API behavior, as follows:

```

// `api` variable represents Test Object.
ReadOnlyCollection<Product> products = api.GetProducts();

for(var i = 0; i < products.Count; i++)
{
    var currentProduct = products[i];

    //verify the product's name.
    Assert.Equal(expectedNames[i], currentProduct.Name);
}
  
```

Notice how the preceding example nearly repeats the previously shown code snippet for Page Object. This similarity is by design since Test Object is a generic case of Page Object.

Layered Test Architecture

We have covered two vital groups of test automation patterns:

1. Gherkin Test Case and its generic counterpart—
Scenario Object
2. Page Object and its generic counterpart—
Test Object

Now it is time to speak about layering these patterns to develop a robust automation codebase. To achieve this objective, let us chain the structures from high-level to low-level, similar to how application layering works: test methods should interact with Scenario Objects, which should utilize Test Objects (see Figure 6-7).

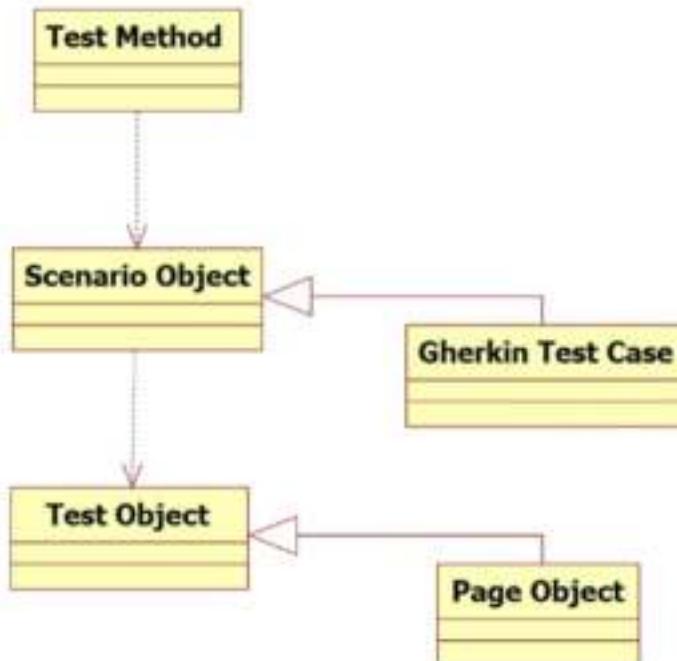


Figure 6-7. Layered test architecture: *Test Method* interacts with *Scenario Object*, which communicates with *Test Object*. *Gherkin Test Case* and *Page Object* are examples of *Scenario Object* and *Test Object*, respectively

The beauty of these layers is in the separation of concerns. Each horizontal slice has a distinct responsibility, making each piece clean, easy to read, and easy to build.

Ubiquitous Language in Test

In previous sections, I emphasized the importance of using business vocabulary in test automation. I must clarify now that I was hinting at using a ubiquitous language (see [Evans DDD]), which is a technique applicable to both development and testing.

If you want to maximize ROI from writing and automating human-readable test cases, learn about Ubiquitous language, and utilize it. This advice is applicable for all test automation phases and layers: writing test cases, test methods, Scenario Objects, and Test Objects.

Test Data Management

Test data management refers to creating and maintaining data within systems on which the test execution relies. Let us look at various test types and their corresponding data management techniques.

Unit Test Data

Unit tests have the luxury of mocking up dependency interfaces to ensure that a given test method runs in a repeatable fashion. If there were no way to substitute dependencies with fake objects, unit tests would need to rely on real services and databases to have data that meets the expectations of a particular scenario. Thanks to mocking, managing test data for unit tests has become straightforward.

For example, to unit test an API controller that retrieves a product, we only need to mock up a repository that is the source of such information (see Figure 6-8).

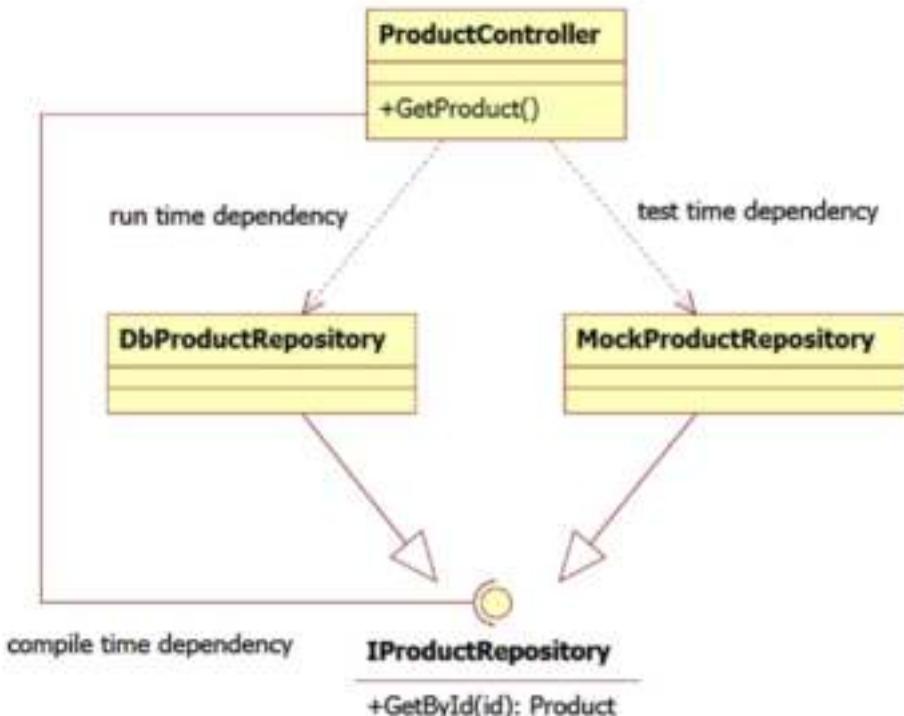


Figure 6-8. *ProductController depends on **IProductRepository**, which we can mock to unit test the controller's **GetProduct()** method*

Managing test data for integration, functional, and end-to-end tests is not as straightforward as it is for unit tests because these constructs rely on real sources of information by design. Let us discuss related solutions one by one.

Integration Test Data

Typically, integration tests query an API or a screen and compare results with a backing database. Therefore, your goal is to ensure that specific storage has the expected dataset.

One way to achieve the described objective is *to assume* that the storage has necessary data; i.e., if this precondition is negative, the test case fails.

```
//First, assert that test data exists.  
var testData = GetTestData();  
Assert.NotEmpty(testData); //Fails if test data is missing.  
  
//Next, verify test case assuming test data exists.  
//... omitted from example.
```

Why don't we ignore missing test data or treat it as a successful test run? That choice could give us a false positive and not catch possible issues with the application's behavior. However, now we have another problem: we might see failing tests that do not necessarily mean that something is wrong; they only say that the test data is missing. While I understand that you might not like this technique, false negatives are still better than false positives: at least we will not take the possibly broken program to production.

If this approach does not satisfy your expectations from a robust test automation standpoint, consider an alternative: each test method can create test data when it starts running and clean it up at the end of execution.

```
//First, create necessary test data.  
CreateTestData();  
  
//Next, verify test case, assuming test data exists.  
//... omitted from example.  
  
//Finally, clean up test data.  
CleanupTestData();
```

The latter is always my preference among the two solutions, but it comes with the added complexity of implementation and feasibility constraints. If the target database or application does not allow the creation of test data programmatically, then we have to revert to the former technique.

Functional Test Data

In an earlier section, I explained the differences between integration and functional tests. If you read that section, perhaps you can predict my solution for functional test data management.

Since functional tests verify the system's behavior from an end user's perspective without touching the backing database, the only way to manage the respective test data is via additional metadata. Specifically, automated tests will need to parse a file or another static source to retrieve test data and compare it with the application's screens or endpoints, as follows:

```
//First, read test data from a dedicated source.  
var testData = ReadTestDataFromFile();  
  
//Next, get actual results from screen or API.  
var actualData = GetScreenData();  
  
//Finally, compare actual data with expected test data.  
Assert.Equal(testData, actualData);
```

This solution might not seem robust to you since it will break as soon as the application's behavior changes. However, that is by design. If the system's capabilities are evolving, the corresponding test dataset should also change. Furthermore, you must strive to replace the test data ahead of upcoming modifications to ensure that tests warn you if the expected program changes do not occur.

End-to-End Test Data

When testing a complex scenario that spans multiple systems, managing test data becomes overly complicated because we need to apply test data management techniques to numerous applications. This complexity is why people frequently avoid automating end-to-end tests. If you are still keen to challenge this status quo, I will give you a couple of recommendations.

There are two choices for end-to-end test data management—centralization or decentralization. While either of the two options can work, I recommend you pick one and apply it consistently across teams.

A centralized approach suggests having a single test data management system where each team registers its expectations from other applications. At the same time, each group uploads test data into the same central system based on requests from other counterparts. Simply put, each team understands the expectation that they need to create and keep test data always available in the mentioned central repository. Automated tests query the test data repository before querying systems and use that information for corresponding verifications (see Figure 6-9).

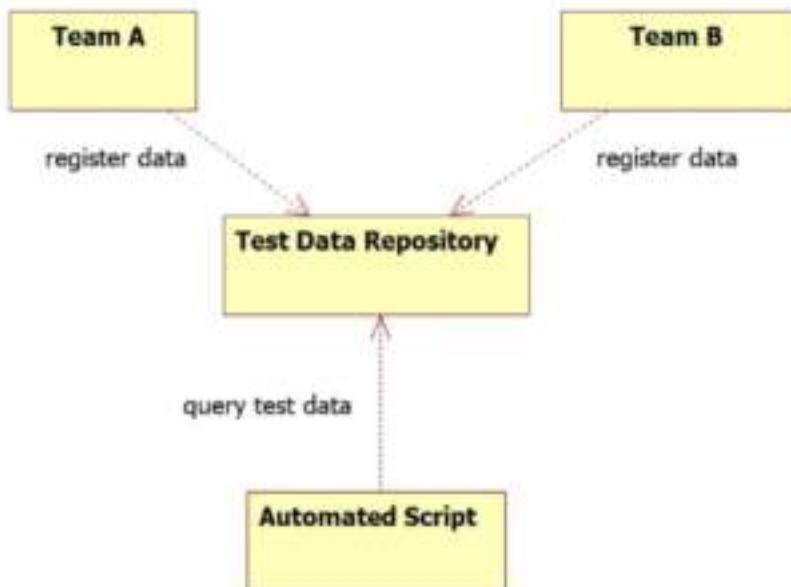


Figure 6-9. End-to-end automated script queries test data from a centralized test data repository where all teams upload their data

A decentralized approach suggests altering the architectures of systems to ensure their testability. If you remember, I briefly touched on this topic when covering testable software architectures, and this is when it becomes relevant. Simply put, each application exposes dedicated API endpoints that allow the creation of test data. Automation scripts utilize these endpoints to ensure that necessary information is available in a system under test before querying it for an assertion (see Figure 6-10). While this technique is universal and robust, it requires much more work than other options, so pick it after seriously considering your limits.

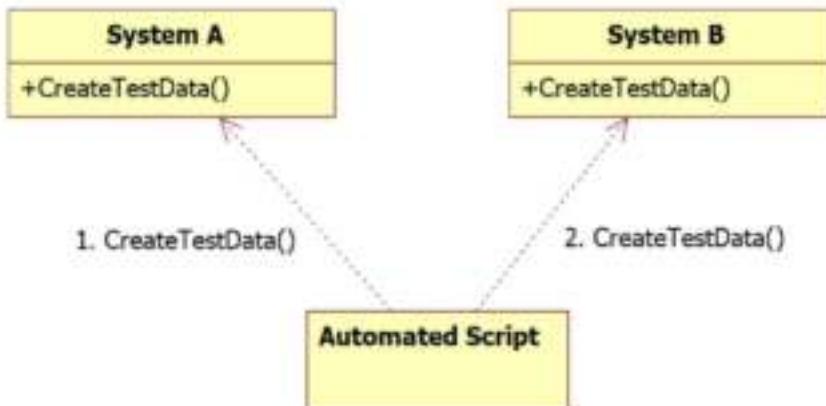


Figure 6-10. End-to-end test script creates test data in each system via dedicated API invocations. Afterward, the script continues execution by assuming the existence of newly created test data

Now that we have discussed the test design and architecture topics, let's see how we can apply them to test implementation activities.

Implementing Automated Tests

Most teams prefer to write automation scripts after development is complete; only a few know the techniques to tackle this exercise in parallel with development. I will explore both options in the next sections under delayed and early automation topics, respectively.

Delayed Automation

A delayed test automation process is straightforward. First, developers build a new functionality; afterward, testers write test cases and automate them. Since the automation takes place later in the flow, I call it *delayed*.

CHAPTER 6 TESTING AND QUALITY ASSURANCE

The simplicity of the described approach makes it most popular among development teams. However, you must know that there are downsides to this process:

- When testers start writing test cases, they often discover edge cases that developers missed during coding. This finding takes work back to engineers until they resolve the issue and redeploy the application to let the testers verify it again. As you can imagine, this back-and-forth movement yields unnecessary context switching and a rework of implementation, which delays delivery dates and decreases ROI.
- In an iterative or milestone-based processes, test automation might not fit in a timeframe that resides between completion of development and final release date. Consequently, teams will try to accelerate testing by cutting down the number of test cases, which puts the quality of the application at risk. Alternatively, engineers might decide to skip the automation of numerous written scenarios, which eliminates the opportunity to get fast feedback when things go wrong. Hence, neither of the approaches is optimal.

You can avoid or resolve these issues if you consider early automation, which I will cover next.

Early Automation

Early automation is a technique of shifting the step of building automated scenarios to occur earlier in your flow. If you succeed with this technique, you can start writing automation scripts at the same time as development and complete both activities almost simultaneously. Such a shift

will resolve both of the described challenges that occur with delayed automation. Let me explain the details of the solution and its benefits.

First, you need to write test cases earlier than development starts, which is not difficult. The only prerequisite for this exercise is to have clear acceptance criteria well ahead of development. The outcome of such a small shift is tremendous: developers can examine test cases ahead of implementing a feature, which ensures that *code will pass quality checks without rework*. Consequently, you have eliminated waves of unnecessary cycles, context switching, frustration, and delays that would come after finding defects or missing edge cases. At the same time, team spirit improves as the entire group works together to deliver a high-quality product instead of focusing on a routine of bug discovery and fixes.

Now, let us talk about starting to automate test cases before or in parallel with development, and finishing it only slightly later than feature implementation. A logical technical challenge is the absence of a page or a resource that we are trying to test: How do we verify something that does not exist yet?

To answer this question, let us recall the test architecture layers that I suggested to use—a trio of a test method, Scenario Object, and Test Object (see Figure 6-7). Interestingly, out of these layers, only Test Object depends on the final test target; all others model higher-level concerns, and they do not access the application's screen or a resource directly. Hence, we can implement test methods and Scenario Objects without waiting until the development is complete. However, we need to do something about Test Object because it directly depends on a tested resource.

Since Test Object *encapsulates* access to a real test target, its public interface stays the same irrespective of its implementation details. This design factor works well to overcome the challenge of early automation: We can build empty Test Objects and use them from Scenario Objects even if the tested resource is not ready (see Figure 6-11).

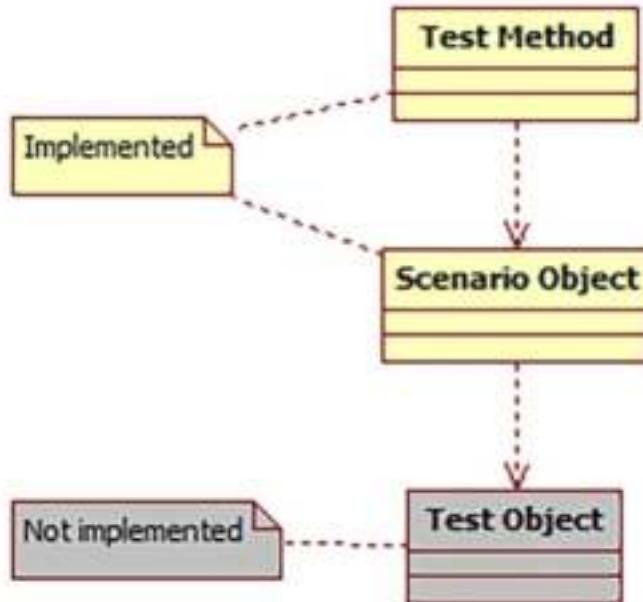


Figure 6-11. Layered test architecture consisting of *Test Method*, *Scenario Object*, and *Test Object*. Out of these layers, only *Test Object*'s internal implementation is pending

Since the tested resource is not available, *Test Object*'s members either stay empty or fail with an error indicating a pending implementation. Scenarios that try to query these incomplete members will fail until the targeted resource is ready for testing, which is the expected outcome since it avoids false positives.

As soon as the test target is implemented, you can finish low-level details of the *Test Object*, which is enough to run automation scripts successfully. This step should not take a significant amount of time since most of the automation—that is, test methods and *Scenario Objects*—are complete at this point. Hence, an overall automation effort finishes at almost the same time as development.

Such a shift of timeline for automation can be a deciding factor between delivering a working product on time versus an unnecessary sacrifice of quality or missed deadlines. Therefore, you must understand and practice this approach when you can.

The test automation code described has many benefits; among them, it can enhance deployments for effectiveness and efficiency. Let's find out how.

Enhancing Deployments with Test Automation

Automation delivers the obvious benefit of quick feedback, as I have emphasized throughout previous sections. Next, I will explain how these insights can strengthen deployment processes.

Fast Feedback and Compensating Actions

When a development team makes a code change and merges it into the main branch of a repository, the chances are that something will go wrong. If we want to avoid negative surprises, test scripts need to run automatically against the new version to verify its validity. If everything looks good, then there is nothing to worry about; otherwise, corrective action is necessary. Let us distill every step of the described process and see options for implementation.

As you remember, continuous integration (CI, see [Fowler CI]) is responsible for verifying codebase correctness after every change. This mechanism is a natural candidate also to run automated scripts and take corresponding compensating actions. There are two constituents to this solution—extracting feedback from test results and taking steps to resolve side effects. I will distill each of these steps.

I suggest to split automated feedback generation into phases: first, run fastest tests (i.e., unit tests), then essential tests (i.e., smoke tests), and finally, the rest of the suite by priority. This procedure should report failures at the end of each step to provide the fastest feedback per test category. Consequently, as soon as the CI is complete, you will have verification results that describe what went wrong after the last code change. Now it is time to act on this information.

How can you compensate for a negative test run result? Ideally, you should attempt to roll back changes that broke a build. Do you remember how I insisted on treating every modification as a version in earlier chapters of the book? If you did well with that direction, perhaps rolling back will be straightforward. Alternatively, if you have containerized your application, that could also help with stripping out breaking changes. Your automated tests can run against an isolated container rather than the containers being deployed to a real server. In case of failure, you can kill such containers without impacting any environment, which will be an equivalent of rolling changes back.

Fast Feedback Accuracy

While fast feedback is essential for the ability to react quickly to breaking changes, it is also vital that you improve the *accuracy* of this insight. I have often seen test suites that give fast feedback, but the result points to problems in automation rather than in the application's code. Such a construct diminishes the value of quick feedback since it creates an unnecessary and false feeling of urgency when there is no reason to worry. I want to cover two concepts that can help implement reliable test suites—flaky and frenemy tests.

First, let us discuss *flaky tests* (see [TTWI FT]). These constructs intermittently fail or succeed for reasons that have nothing to do with the application's behavior. Flaky tests typically randomly give false positives or negatives; for this reason, you should avoid creating them.

For instance, suppose an integration test tries to locate a test data in a database; if it cannot find the necessary information, it passes. When the data appears for the second round of verification, the same test can fail if it discovers that the application misbehaves under such circumstances. This test is flaky since it is unreliable and randomly gives a false feeling of safety. One way around this issue is to fail the test when necessary data is missing, which at least will avoid false positives. If you want to go for an even better solution, ensure the existence of test data via one of the techniques that I described earlier when I spoke about integration test data management.

Another example of a flaky test is generating a random input to verify screen or API behavior. You might discover that an application reacts positively to one sample of random data and negatively to another. Such a test is also unreliable since it might give false positives early and true negatives too late, leaving you without a chance to fix them. To avoid such flaky test behavior, brainstorm positive and negative use cases and split a particular test method into separate scenarios, allowing each to focus on a concrete input without randomization.

Next, let us talk about *frenemy tests*. These constructs help identify *where exactly* the problem is; they target concrete suspects to detect failure or exclude options from investigation and accelerate troubleshooting. Therefore, make sure that you have frenemy tests in your test suite.

For instance, if your application utilizes an outside service for a specific scenario, the chances are that this exact use case will not pass verification when a dependency has temporary problems. When you receive such automated feedback, you might start investigating your application's correctness while the real issue is within dependency. Frenemy tests help in such situations: they do nothing but verify the health status of concrete suspects, which is the outside service. When you see a negative verification result, you can spot that the frenemy test is also flashing red, which tells you where the problem is. Such insight can be precious and relieving when you do not know where to start investigating.

Continuous Deployment

One area that people frequently discuss with continuous integration and automated tests is continuous deployment (CD, see [AA CD]). The basic idea behind CD is it provides the ability to take new features to your customers instantaneously as long as an automated verification gives a positive signal. While I believe that such an ideal outcome is possible, I want to warn you about a few pitfalls.

First, automated tests are *not useful* for verifying positive user experience (UX), such as accessibility or look and feel. Consequently, positive fast feedback might give you a false indication of success when there is no broken code, but there is an unpleasant experience. Specialized tools exist that verify mentioned edge cases, but they are less known, rare, and complex. Without such advanced mechanisms, the CD can generate more trouble than benefits, so be cautious about this limitation.

Another implicit side effect of CD is the possible absence of business verification. It is easy to interpret CD as a mechanical elevation of code to production as long as all automated tests pass. However, this rule fails to cover situations such as a missing test case or misinterpretation of a requirement. In both these cases, an automated test suite will pass, but the outcome can be a broken or unwanted functionality in a production environment.

Described challenges exist partly because of *continuous deployment's* being a comparably new term, and many of its edge cases are still not tried. Also, you will find different interpretations of this discipline in various literature or online resources, which adds to the confusion. All these factors should inspire caution when implementing the CD. My suggestion would be to learn its pros and cons before deciding to go all the way into the CD.

When choosing between blindly following or discarding CD processes, I recommend a pragmatic approach: utilize tools that take artifacts out of a CI build and let you *manually elevate* them through environments

as soon as the new version passes corresponding automated and business verification tests. This way, you take advantage of both sides of the solution. While CI automatically verifies a build's correctness via test suites, a human eye ensures that the new release meets business, accessibility, and UX expectations. After both these steps are complete, it is safe to deploy your application to a production environment.

Summary

We have concluded our exploration of testing and quality assurance of applications. We touched on processes, architecture, implementation, and deployment. These topics are essential when delivering effective software that works flawlessly and meets user expectations. Speaking of delivery—let's see what it takes to deploy your application to the environments accessible by your end users.

CHAPTER 7

Deployment

If you have succeeded in coding effective software solutions, you might think deployment sounds like an easy task. What could be so hard about moving your application from one place to another?

This view is simplistic, as the deployment process' intricacies can ruin your success in a matter of seconds. For instance, imagine an incomplete release of a website that does not allow customers to place orders because the corresponding page is missing. For many businesses, this situation would be a disaster.

A well-planned release can be a competitive advantage as you will get software products into customers' hands consistently and without glitches. Therefore, deployment is another vital area requiring high-quality solutions and is worth covering in this book.

This chapter will focus on streamlining deployment processes and on mechanisms that are essential to building and delivering high-quality, working solutions. You have likely already seen most of these topics elsewhere in this book. Such correlation is by design: I believe that all steps of building applications must run with a release in mind and, therefore, must fulfill the duty of preparing systems for smooth deployment. From this perspective, the current chapter will revisit and recap all those measures to connect the dots of an end-to-end software delivery chain.

Culture of Releases

Before we discuss deployment activities, I want to touch on some simple but essential topics around releases. Revisiting these fundamentals can trigger or foster a mindset that helps people succeed with deployments that achieve positive business and quality outcomes.

Why Release Software?

Deploying software solutions to various environments is a common activity. It comes with a heavyweight artillery of processes, principles, and tools, such as DevOps, CI/CD, cloud, containerization, and so forth. While trying to master all these instruments, we often forget what we are trying to achieve and for what reason. As a result, deployment processes tend to focus on mechanical problems that these tools pose instead of on delivering actual value with all releases. Therefore, let us recall the following:

We deploy applications because we want to provide benefit to customers via working software. Before getting into the technicalities of deployments, I recommend that you recognize this underlying intention.

We can break the preceding statement into multiple objectives, which will clarify why I insisted on particular approaches earlier in this book, as follows:

- Each requirement that we tackle must have a clearly defined value proposition; otherwise, we might be releasing an unwanted capability. We achieve this objective via customer interviews, knowledge exploration exercises, ubiquitous language, and so on.
- When implementing a modification to a program, we need to successfully transition the system from one working state to another. This directive reminds you to treat each change as an independent version.

- To gain and maintain a competitive advantage, we must frequently iterate the preceding steps without much overhead. This suggestion ties back into continuous improvement and value creation—from customer interviews to repetitive code refactoring and deployments.

Unimportance of Releases

I have worked with many organizations and engineering teams. If you asked each of those groups whether a release is a big deal, you would get different answers. In some cases, deploying to production is a huge event, while it went unnoticed in other cases. How do you prefer to look at this activity?

If a release is a big deal, perhaps you are incurring unnecessary overhead in your process! My concern is not that you need to order a pizza for multiple people while they all try to get your application out. The trouble is that you cannot release the system without so many people around. In other words, your team's talent should be focused on more creative tasks than a deployment's mechanical activities. *Therefore, a release must not be a big deal*; otherwise, it hints at improvement opportunities (because big deal means big trouble if something goes wrong; or even worse—big risk of something going wrong).

When a release is not a big event, it typically correlates with an ability to deploy your application into the production environment via a button click instantaneously. Also, since it is not a big deal, nobody notices when it starts and completes, so it has no adverse side effects.

What if your deployments are inherently complex, and they never go unnoticed? Interestingly, the preceding breakthrough works the other way around too—first, try treating a release as no big deal, and then watch how people learn to deploy unnoticeably! Such an outcome is a significant achievement and can be a competitive advantage in the following areas:

CHAPTER 7 DEPLOYMENT

- Employees can focus on other essential tasks or go home after work hours instead of working on deployments.
- You can release at any time because of deployment's having minimal to no impact on the system's availability and functioning.
- Since the release is no longer a big deal, you can frequently run this exercise to deliver new capabilities to your customers more quickly and further increase your advantage.

What Is the Catch?

You may think that there is a caveat to such unnoticeable releases, and you are right! Smooth deployments are not straightforward to achieve, but it is possible. In a nutshell, you need to automate and shorten every step of the release, which allows you to run all steps with a single button click at any point in time (importantly, rollback must also be a matter of a button click). Throughout this book, I mentioned multiple techniques that can help you reach this objective, such as the following:

- A group developing an application is responsible for preparing their releases and deploying their systems. Such a culture is an outcome of establishing autonomous teams that form around microservices.
On the other extreme of the spectrum are anti-patterns, such as release management committees, which turn releases into a big deal, so avoid them.
- Each code change is a version that you can quickly deploy to an environment. Ideally, we must perform this action via a script or a simple command execution.

Consequently, the deployment tool can leverage this construct and automatically apply changes to an environment.

- While making modifications, developers need to plan for a release and prepare all components so that CI/CD tools can automatically package and deploy them. Besides code, this advice also applies to the database and any other system changes representing application dependencies.
- Automated tests serve as a verification of a successful deployment. Thus, we need to have the unit, integration, functional, and end-to-end tests available. Deployment tools can execute these constructs to confirm success or detect a failure.
- If automated tests indicate a problem after deployment, we should be able to roll back the changes instantaneously. Again, this is possible if each code change is an independent version that we can uninstall to take the environment to a previous working state.

Is a release worth so much trouble? My answer is a definite Yes! First, we are not talking about a single release but an opportunity for frequent value deliveries to customers. This effect alone is a big win. Furthermore, the low quality of manual and lengthy deployments can decrease customer satisfaction due to human errors. After all, nothing that you do before the release has any benefit if a defective release can instantly kill your reputation.

If we agree, let's move on to the deployment exercise using CI/CD.

CI/CD—Deployment Foundation

If I had to describe the CI/CD with a single sentence, I would only mention this basic yet essential principle: *one build, many deployments*.

In the next two sections, I will briefly recap the topics of continuous integration and continuous deployment, as these two techniques form the foundation of successful releases.

Continuous Integration

Continuous integration (CI, see [Fowler CI]) is responsible for preparing a single artifact that you can deploy into multiple environments. To achieve this objective, the CI needs to carry out the following steps:

- Monitor a codebase for changes and automatically trigger a build-verification process. If any of the subsequent steps fail, CI needs to take course-correcting actions by either informing respective parties or automatically rolling back the changes.
- Compile a new version of the codebase to ensure the feasibility of creating binaries after the last change.
- Perform static code analysis to confirm that the code meets a predefined set of standards.
- Execute automated verification tests to detect possible problems.
- Prepare an electronic artifact representing a package of the application's binaries that needs to go into testing, staging, and production environments.

While the preceding list is short, each item represents a combination of multiple techniques for achieving results. It is crucial that you carefully follow each recommendation's principles, so I suggest that you become familiar with every mentioned topic.

Briefly About Artifacts

Before moving to the next section, I want to draw your attention to something common between both CI and CD—program artifacts.

An artifact is an *immutable package* that you should not attempt to modify during deployments. This expectation is an essential constituent of high-quality deployment processes. Simply put, your goal is to take to production precisely the version of binaries that you verified on the stage and test servers. This effect is only possible if you enforce the immutability of artifacts.

Maybe this precaution sounds evident to you, but I have frequently seen non-immutable artifacts that get patched differently for each target environment. For example, one common scenario is altering configuration file contents or even substituting dependent binaries at the time of deployment. Such tricks pose a risk of there being behavioral differences between environments, which can be the root cause of a well-known quality dilemma.

"But it worked in stage. I don't know what happened in production."

Have you ever heard this puzzling statement indicating that the application misbehaved after releasing it to customers? This situation is what I am trying to prevent. If you ensure that the artifact does not change while it progresses from test to production servers, you have a guarantee of identical behavior. Consequently, you are in control of the quality that your customers receive.

Continuous Deployment

Continuous deployment (CD, see [AA CD]) is responsible for taking a single artifact to all environments, from test to production. While this statement is concise and straightforward, it consists of the following non-trivial steps:

- Take an artifact from CI as input and allow its deployment to each environment.
- Let engineers move an artifact from lower to upper environments, and prevent leaking untested changes into production. There is no shortcut in deploying a package to production without first verifying it on test servers.
- Account for the necessity of both installation and rollback scenarios. Sometimes things go wrong during releases, which is when the CD and respective tools must be handy to undo the unwanted changes quickly.
- Implement accumulative rollouts: very first deployment recreates an entire environment, every subsequent release upgrades version, and every rollback brings back a working snapshot.
- Support automating all deployment tasks instead of necessitating a manual modification of environments.

For each item in the preceding list, there may be multiple options for achieving the objective. You must plan an appropriate CD implementation by thinking through alternatives and picking the right tools.

Before closing this section, I want to stress that the CD exists to *automate* deployments. To honor this guideline, you must avoid any manual intervention with environments. Remember, if you make an ad-hoc tweak to a hosting server, you may be putting yourself on a downhill

to a problem. First, you can accidentally introduce behavioral differences between environments, which diminishes the value of quality checks, as I explained earlier. Besides, if you need to replace or recreate an environment, an automated CD process will not know about your manual tweaks, so you will not be able to precisely accomplish the goal. The crux of the CD is to achieve repeatable mechanical deployments to any environment (production may be an exception if the manual verification of user experience is necessary, but the rest must still be automatic); hence, manual environment alterations go against this basic principle.

Deployment activities described here can only do so much if the engineers do not consider them during the implementation of software systems. Let's look at that topic next.

Building Deployment-Ready Applications

This section will briefly reiterate recommendations that apply to implementation and coding tasks for smooth deployments. Also, I will explain how outcomes from such practices serve as inputs to successful releases.

Developing Deployable Units

As I stated earlier, implementing each requirement must produce all components that are ready to deploy afterward, such as the following:

- Code changes that meet acceptance criteria
- Scripts or documentation for applicable database changes that need to happen before starting an application

CHAPTER 7 DEPLOYMENT

- Data files or instructions for creating lookup data on which program depends
- Steps for acquiring necessary permissions or configuring user's access

Simply put, you must think through the deployment plan while developing features (this advice goes hand-in-hand with an earlier suggestion to account for test scenarios when writing code). For each of the applicable steps, decide your action—either automate its installation or document instructions that a human can follow. I recommend that you always take the former approach. If you need to rely on documentation, make sure that you minimize the chances of error by creating batch scripts that anybody can quickly execute. A similar approach applies to rollback plans too.

If you have future releases in mind while developing, you are implementing deployment-ready programs out of the box. This mindset simplifies application installation and eliminates the risk of errors. Consequently, your releases will have no negative impact on customer experience and satisfaction.

If you succeed with the mentioned techniques, your deployment will be a matter of clicking buttons or executing batch scripts that take care of releasing all components. If anything goes wrong and a rollback is necessary, you can easily perform that task.

Ensuring Smooth Deployments via CI/CD

In the previous section, I listed several techniques that developers need to employ when implementing deployment-ready applications. Now, let us discuss corresponding preparations that are necessary when constructing CI/CD pipelines. For clarity, I list here outputs from development and CI/CD techniques that support each of those outputs:

- For code changes: CI verifies its correctness and produces an artifact, while CD takes the final package to environments. These capabilities are present in all CI/CD tools out of the box.
- For scripts and batch files: CD needs to run all these commands during deployment. While every CD tool supports this trick nowadays, the only missing piece is determining how to locate scripts and execute them in the right order. One way to implement this technique is via conventions. For example, the CD pipeline can scan a dedicated folder for all scripts within it and execute them in a predefined order, such as alphabetically or ascending. As long as this routine is idempotent, such a simple trick will be sufficient. This construct also tells a developer where to place their script and how to predict its execution order.
- The previous advice applies to data files and acquiring permissions too. Again, my suggestion is to automate as many of the manual steps as possible.

As a last resort, if automation is not an option, a person working on a release will need to follow instructions in the provided deployment document. However, for the sake of high-quality deployments, I strongly recommend avoiding this approach whenever possible.

Dev–Prod Parity

As you remember from earlier topics in this book, dev–prod parity is a concept from 12-Factor principles (see [12factor]) that applies to loosely coupled, distributed application development. In simple words, this pattern advises configuring a development environment similar to

CHAPTER 7 DEPLOYMENT

production setup so that engineers can implement and test features in conditions identical to production. Dev-prod parity increases product quality and decreases the number of bugs caused by environmental differences.

While we are speaking about deployments in this section, I want to explain one way of achieving dev-prod parity in practice.

Since the CD pipeline is responsible for deploying an application into any environment, it could also create a new development environment on demand. Imagine clicking a button and having an application up and running in a couple of minutes. This outcome would qualify as reaching dev-prod parity. When you do not need this sandbox anymore, you can quickly abandon or remove it by executing a rollback procedure.

One essential prerequisite for this entire approach is to have every step of an installation automated. This expectation includes both code and database deployments and underlying host server recreation from scratch. As I just demonstrated, dev-prod parity is one more avenue to accomplish all these objectives.

Effects of Containerization on Deployments

In this chapter, many described techniques relied on automating artifact creation and deployment. As you can imagine, completing both these tasks requires time and talent. If you are short of such resources, you will need to find an alternative that shortens your path to smooth deployments. Here, I will explain how containerization (see [Wiki Containerization]) can help you solve this challenge.

Interestingly, containerization aims to automate both the packaging and the deployment of applications. Furthermore, all cloud hosting providers support container platforms via infrastructure and pipelines similar to those you use when developing. Thus, containerization is a valuable help toward automating packaging and deployments and achieving dev-prod parity.

For instance, if you use Docker, you can easily create images (i.e., packages) and run (i.e., deploy) them on your local machine, an on-premises server, or a public cloud. These steps are reasonably simple due to a rich toolset that comes built into the Docker ecosystem. Since it is the same image that runs everywhere, this approach also accomplishes dev-prod parity.

If you have such a choice, I recommend that you always consider containerizing your solutions. In other cases, containerization is still a useful concept to keep in mind because the CI/CD aims to implement this pattern using alternative tools and techniques.

Summary

How do you feel about your releases now? We touched on topics that should help improve deployments from all angles—culture, mindset, CI/CD, and building applications for smooth releases. With these subjects in mind, you can effectively take quality software solutions to your end users. All that is left is the maintenance of the systems, which we will discuss next.

CHAPTER 8

Maintenance and Support

As you know, this book aims to help you build applications that do not require maintenance and that always work—without production support teams. You can achieve this objective by following the various recommendations that I presented in the previous chapters. So, what is left to discuss in the current chapter?

In the following sections, I will draw a line between implementation techniques and their positive effects on the resulting systems' maintainability. I will also cover a couple of topics that did not fit within the other areas of the book but are still vital for building and delivering maintainable programs.

Maintenance-Free Mindset

Let's start with subjects relevant to mindset and cultural shift. I believe that forming the right attitude and mentality is half of the work, and the rest comes with the concrete techniques that I will cover later.

"Begin. The rest is easy."

—George Jenkins

Organization's Approach to Maintenance

I have seen three different approaches to application maintenance and support activities across various organizations and teams, as follows:

1. *Most* companies have dedicated production support teams responsible for maintaining applications, resolving urgent technical incidents, and delegating troubleshooting to engineering teams only if previous attempts are not successful.
2. *Some* organizations reluctantly give maintenance responsibilities to engineering teams, primarily because they are small or operate under thin margins such that a dedicated team is a luxury that they cannot afford to have.
3. *Only a handful* of organizations consciously decline to assemble production support teams as they hold engineering groups accountable for the development and maintenance of their applications.

The first approach has an unnecessary overhead, assuming that the third option exists. The second scenario often turns into the first one when the budget allows since the engineering teams might not believe that application maintenance is their responsibility.

Therefore, I recommend that you go with the third option: *purposefully create an engineering culture that discourages having dedicated production support teams. Hold developers accountable for running applications that they build without anybody else's help.*

Based on my experience, if you set such expectations, engineers will manage to minimize the maintenance overhead so that they can continue developing new features. I can explain this phenomenon as follows: people naturally find ways to do more of what they like and less of what

they dislike. While every developer loves to build new capabilities, nobody wants to support applications. Due to this preference, engineers are the best candidates for figuring out how to avoid or minimize maintenance problems. When they succeed with this task, you get what I promised—minimal to no spending on production maintenance and support; hence, you achieve a higher ROI.

Support-Oriented Organizations

If you are forming a new engineering organization, the previous section works fine for you as you can make a proactive decision to avoid having production support teams. The direction becomes more delicate when there is an existing group or department with the sole purpose of application maintenance. How do you apply the described advice to such environments? Let me provide a couple of recommendations.

First of all, the shift is not instantaneous since we are talking about shuffling responsibilities and altering engineering teams. Therefore, do not rush into unnecessary layoffs or budget cuts. Instead, employ a planned and gradual approach to avoid frustration and loss of productivity.

Furthermore, existing maintenance groups have valuable knowledge of systems that you can utilize for fulfilling other functions, such as quality assurance or technical documentation writing. Therefore, I encourage engineering leaders to consider the skillset of support groups for different work areas.

Lastly, remember that support-oriented organizations stand on the belief that their existence is necessary. Naturally, this faith can create an unintentional complication to adopting a mission to eliminate such departments. Therefore, be mindful of the existing organizational mindset and structure when proposing a revolution. The time must be right to make significant shifts and breakthroughs.

Award-Winning Support Teams

Earlier in this book, I discouraged awarding testers for finding defects; instead, I suggested to build a team spirit toward avoiding bugs in the first place.

When it comes to production incidents due to broken systems or poor user experiences, the same principle also applies. While the engineering team is responsible for maintaining their applications and resolving incidents, it is not good news when such problems happen. Instead, developers and their leaders need to find ways to improve quality while building and deploying systems so as to avoid such issues altogether. By awarding employees for resolving broken functionality, you effectively encourage developers to allow problems to happen so that somebody can become a hero for fixing them.

To draw an analogy, imagine praising a dentist for fixing your broken tooth, although they damaged the enamel by mistake in the first place. If you decide to compliment the specialist, perhaps you are creating a pattern of behavior—they break things, fix them, and get rewarded. It sounds like a good deal for dentists, but not for your teeth!

Therefore, avoid praising those who resolve production incidents if the root cause came from the same group. Instead, build an engineering culture that treats broken functionality as an unwanted exception and tries to *prevent* such occurrences.

Who Prevents Problems?

This entire book relies on a few vital techniques that lead to building high-quality, working solutions, which I call effective software. One such exercise is a *proactive prevention* of problems instead of solving them reactively. Achieving this objective takes experience, passion, and knowledge.

I have worked at many different places with numerous teams, and I can assure you that individuals able to prevent problems are rare. If you encounter such talent, I recommend keeping them around for as long as you can. Don't get me wrong, it is not easy to solve problems reactively, but it is much harder to prevent them from happening proactively!

One of my favorite quotes nicely summarizes the crux of this section:

"Intellectuals solve problems, geniuses prevent them."

—Albert Einstein

After the innovative thought of being maintenance-free, let me bring you back to reality with the maintenance-aware mindset, which is equally essential to the former topic.

Maintenance-Aware Mindset

The previous section clarified my view of handling application maintenance and support. I believe that these activities are the responsibilities of the engineering team that built the system. While this group tries to minimize or prevent technical problems and incidents, they need to employ various techniques. In this section, I will touch on a couple of topics that will help prevent issues after deployment as long as engineers keep these matters in mind.

Maintaining Applications in Practice

Engineering teams are responsible for maintaining applications they built. At the same time, they need to continue developing new features. For this paradoxical arrangement to work, it is necessary that the group keeps an eye on a ratio of development versus maintenance efforts and takes course-correcting steps to deliver more features with less support.

CHAPTER 8 MAINTENANCE AND SUPPORT

In the early days of an application, this task is easy to achieve since there is hardly any maintenance necessary. When the system grows (or gets a major version upgrade) and the number of support requests spikes, the time available for building features shrinks. When this phenomenon happens, leadership often thinks that it is best to hire an additional dedicated team to handle maintenance activities and free up resources for feature development. This decision only poisons the engineering spirit. Developers lose the need for quality work since maintenance is not their problem anymore, and things worsen over time. Therefore, I suggest taking an alternate route.

As soon as you notice that supporting an application takes an unacceptable amount of time compared to new feature development, you need to *pause and brainstorm* how to stop the trend. Concrete actions depend on the root cause, so I will list a couple of examples to give you some ideas:

- If maintenance is technical, such as performance or scalability improvement, you may need to treat this shortcoming as technical debt and address it soon. Furthermore, develop a habit of systematically implementing technical capabilities besides functional features. Typically, engineering teams often do not pay attention to technical debt, and it grows exponentially and unnoticeably, so you must keep an eye on this.
- Maybe support tickets indicate customers' confusion due to an unclear user experience or ambiguous input validation messages, which you have been treating as a low-priority concern. In this case, you might need to make these improvements sooner than you had planned. By addressing these issues, you are decreasing the number of support requests and optimizing the anticipated versus unplanned work ratio, which

will allow you to deliver more capabilities to your customers in the future. Hence, look at this adjustment as an upfront investment with good returns in the long run.

- Are you receiving the same kind of maintenance request over and over again? Perhaps it is time to think about *automating* resolution so that customers can enjoy self-service instead of waiting for a response from the team.
- If end users complain about broken functionality and bugs, you may need to review your automated test suite and improve coverage via unit, integration, functional, and end-to-end tests. These constructs exist to increase quality, so make sure to build and leverage them at all times.

To summarize the preceding examples, I expect that you will immediately correct course as your development pace slows instead of allowing this effect to turn into a severe problem.

As noted in the preceding discussion, engineering teams that maintain their applications receive feedback directly from end users (often with the help of customer-facing support groups) and can course correct immediately. However, when you place an additional technical support team between the development group and customers' feedback, you add an unnecessary communication channel and overhead to the process. This difference is another reason to avoid forming dedicated technical production support teams.

Fix Root Cause, Not Surface

When an engineering team receives an urgent support request that passes the first-round triage as a “must-fix now,” it is tempting to resolve it in the *simplest way possible* due to time crunch. If an unhappy customer is waiting for a fix, you must undoubtedly go ahead. It is also vital that you perform additional analysis to correct the root cause at a later point because the most straightforward way often is not the *right way*.

In other words, I want to ensure that you did not misinterpret my earlier suggestion of “minimizing maintenance efforts” as scratching a surface when incidents occur. If an issue happened, it is too late to minimize effects; it *already happened!* Instead, your best bet now is to *prevent* the problem from happening again.

One example of a preventive approach toward a newly found bug is fixing it via TDD (Test Driven Development, see [Tengiz BFVTDD]). In simple words, TDD ensures that a unit test will fail if the bug occurs again in the future. However, a unit test that you wrote after the code modification without TDD qualifies as a scratch of the surface: this approach might not catch a recurring bug or could introduce dead code, causing more problems in the future.

Building Blocks of Maintainable Systems

Software systems must utilize proper architecture and tools to become highly maintainable, allowing teams to optimize support and troubleshooting activities.

While there are endless techniques that you can employ for increasing maintainability, I will list a few common examples here:

- Logging informational, warning, and error messages ensures that you can troubleshoot problems or verify a system’s functionality. A typical trap in this area is doing too much or too little logging, or having log

messages that do not help. The best way to find a golden medium is to think backward: brainstorm possible maintenance scenarios and decide what to log to support each of these use cases.

- Analytics and health monitoring tools are a good starting point if you need to analyze complex scenarios involving multiple systems.
- Do not hesitate to adjust the application's architecture for maintainability (this advice is for architects or individuals performing this responsibility). For example, if you expect that configuration values need to change more often than upon deployment, consider keeping such data in a separate service instead of placing it in static files. Subsequently, you will have the option to modify configured values without redeploying systems.
- Developing tools or specialized administrative screens for repeating maintenance or housekeeping tasks can help you complete support requests faster via such predefined functionality. These constructs qualify as Admin Processes from 12-factor principles (see [12factor], which should help you understand the idea behind my recommendation.

To conclude, maintenance and support must be in your mind while you build and deploy applications. This mental exercise ensures that you effectively eliminate dedicated production support teams while maintaining or accelerating the pace of development.

Summary

We have concluded the maintenance topic after discussing the shift of mindset in both directions—maintenance aware and maintenance free. I am confident these techniques will positively impact your application maintenance practices. After such a long journey, it is time to wrap up everything we've learned as a high-level theme underpinning the book.

Afterword: Wrap-up

You have reached the end of the book. We talked about many topics, and perhaps it seems unrealistic to remember all of them at all times. Such a consequence is normal, and that is why I am providing a wrap-up—to reiterate the essential aspects of this publication.

If you asked me what this text's single guiding principle is, it would be a top-down flow of structure and information.

Everything starts at an enterprise level by examining business problems that the organization wants to solve. Once you distill this area, you can drill down into other aspects of the software engineering organization to align technology under the domain space.

Solutions to cross-cutting concerns are the supporting mechanisms to other SDLC (Software Development Life Cycle) phases, so you must look into culture, professionalism, talent, and requirements management as your next step.

After you get all the mentioned topics right, look at the phases of the SDLC. Each area seems straightforward at first glance until you accept the fact that the typical approaches prevalent today might not be the most efficient and cost effective. What is the underlying principle of those topics in the book? It is to refuse the status quo! Instead, be persistent and guide decisions by the structure that the earlier steps defined. For example, architecture defines microservices under the business domain, while the code implements the architecture.

That is the crux of this publication in short. If you ever need additional information about any of the specific topics that I mentioned here, you can always go back and review the corresponding sections of the book.

AFTERWORD: WRAP-UP

I hope you enjoyed the reading. Now, go and build software systems that can withstand time and serve successful businesses to help them gain a competitive advantage due to increased return on investment (ROI).

You can do it!

References

[12factor]

<https://12factor.net/>

The 12-factor app website

[AA BDD]

<https://www.agilealliance.org/glossary/bdd/>

Explanation of Behavior-Driven Development (BDD) by Agile Alliance

[AA CD]

<https://www.agilealliance.org/glossary/continuous-deployment/>

Explanation of Continuous Deployment by Agile Alliance

[AA INVEST]

<https://www.agilealliance.org/glossary/invest/>

Explanation of INVEST criteria by Agile Alliance

[AA TDD]

<https://www.agilealliance.org/glossary/tdd/>

Explanation of TDD (Test-Driven Development) by Agile Alliance

[APIE AIAPI]

<https://apievangelist.com/2012/01/12/the-secret-to-amazons-success-internal-apis/>

"The Secret to Amazons Success Internal APIs" - an article published on the API EVANGELIST website

[ATSN DTOTS]

<https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing>

"The different types of software testing" - an article by Sten Pittet, published on Atlassian website

REFERENCES

[Bob PSD]

<https://www.youtube.com/watch?v=zwtg7lIMJaQ&t=4s>

"Robert C. Martin – Professional Software Development" – a YouTube video published by "gnbitcom" channel

[Brandolini ES]

<https://www.eventstorming.com/book/>

Introducing EventStorming – book by Alberto Brandolini

[C4Model]

<https://c4model.com/>

"C4 Model" – a lean graphical notation technique for modeling the architecture of software systems

[CG OA]

https://www.codeguru.com/csharp/csharp/cs_misc/designtechniques/understanding-onion-architecture.html

"Understanding Onion Architecture" – an article posted by Tapas Pal on Codeguru website

[Docker]

<https://www.docker.com/>

Docker – a containerization platform

[Evans MEW]

<http://domainlanguage.com/ddd/whirlpool/>

"Whirlpool Process of Model Exploration" – an article published on Domain Language web page that depicts a basic idea behind a term coined by Eric Evans – "Model Exploration Whirlpool"

[Evans DDD]

Domain Driven Design – a book by Eric Evans.

[Evans/Fowler SPEC]

<https://martinfowler.com/apsupp/spec.pdf>

"Specifications" – a document by Eric Evans and Martin Fowler that describes specification pattern

[Foote BBOM]

<http://www.laputan.org/mud/>

"Big Ball of Mud" – an article by Brian Foote and Joseph Yoder

[Fowler ADM]

<https://martinfowler.com/bliki/AnemicDomainModel.html>

"AnemicDomainModel" – an article by Martin Fowler

[Fowler AR]

<https://www.martinfowler.com/eaaCatalog/activeRecord.html>

"Active Record" – an article by Martin Fowler

[Fowler CI]

<https://martinfowler.com/articles/continuousIntegration.html>

"Continuous Integration" – an article by Martin Fowler

[Fowler CQRS]

<https://martinfowler.com/bliki/CQRS.html>

"CQRS" – an article by Martin Fowler

[Fowler DE]

<https://www.martinfowler.com/eaaDev/DomainEvent.html>

"Domain Event" – an article by Martin Fowler

[Fowler DI]

<https://martinfowler.com/articles/injection.html>

"Inversion of Control Containers and the Dependency Injection

Pattern" – an article by Martin Fowler

[Fowler FI]

<https://martinfowler.com/bliki/FluentInterface.html>

"FluentInterface" – an article by Martin Fowler

[Fowler IDD]

<https://martinfowler.com/articles/designDead.html>

"Is Design Dead?" – an article by Martin Fowler

[Fowler Microservices]

<https://martinfowler.com/articles/microservices.html>

"Microservices" – an article by Martin Fowler

REFERENCES

[Fowler Plugin]

<https://www.martinfowler.com/eaaCatalog/plugin.html>

"Plugin" – an article by Martin Fowler

[Fowler PO]

<https://martinfowler.com/bliki/PageObject.html>

"PageObject" – an article by Martin Fowler

[Fowler POEAA]

Patterns of Enterprise Application Architecture – a book by

Martin Fowler.

[Fowler RTAAM]

<https://martinfowler.com/articles/refactoring-adaptive-model.html>

"Refactoring to an Adaptive Model" – an article by Martin Fowler

[Fowler UT]

<https://www.martinfowler.com/bliki/UnitTest.html>

"UnitTest" – an article by Martin Fowler

[Gamma GOF]

Design Patterns: Elements of Reusable Object-Oriented Software – a book by Erich Gamma and others

[Gartner EA]

<https://www.gartner.com/en/information-technology/glossary/enterprise-architecture-ea>

"Enterprise Architecture (EA)" – definition of term on Gartner website

[GFG TOST]

<https://www.geeksforgeeks.org/types-software-testing/>

"Types of Software Testing" – an article published on

GeeksforGeeks website

[Gherkin]

<https://cucumber.io/docs/gherkin/reference/>

"Gherkin Reference" – a language reference published on

Cucumber website

[Google ESI]

<https://www.google.com/search?q=event+storming&tbo=isch>

A Google image search for "event storming" to demonstrate how event storming looks visually

[Google WBMII]

<https://www.google.com/search?q=class-diagram&tbo=isch>

A Google image search for "class-diagram" to demonstrate how class diagram looks visually

[Hodgson DOO]

<https://martinfowler.com/articles/domain-oriented-observability.html>

"Domain-Oriented Observability" – an article by Pete Hodgson, published on martinfowler.com website

[Hohpe EIP]

Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions - a book by Gregor Hohpe and others

[Inc GSP]

<https://www.inc.com/adam-robinson/google-employees-dedicate-20-percent-of-their-time-to-side-projects-heres-how-it-works.html>

"Want to Boost Your Bottom Line? Encourage Your Employees to Work on Side Projects" – an article by Adam Robinson published on "Inc." website that explains ideology behind Google's side projects for employees

[Infoq MSA]

<https://www.infoq.com/news/2015/12/microservices-amazon>

"Microservices and Teams at Amazon" – an article by Jan Stenberg published on InfoQ website that explains how Amazon organizes teams around microservices

[Kim Phoenix]

The Phoenix Project – a book by Gene Kim.

REFERENCES

[MS MUI]]

"Creating Composite UI Based on Microservices" – an article that explains monolithic UI patterns on Microsoft documents website

[msio]

<https://microservices.io/>

A website that focuses on microservice architecture

[msio CSUIC]

<https://microservices.io/patterns/ui/client-side-ui-composition.html>

"Pattern: Client-side UI composition" – explanation of a pattern on microservices.io website

[msio SSPFC]

<https://microservices.io/patterns/ui/server-side-page-fragment-composition.html>

"Pattern: Server-side page fragment composition" – explanation of a pattern on microservices.io website

[RAPINET Versioning]

<https://restfulapi.net/versioning/>

"REST API Versioning" – an article on a "REST API Tutorial" website

[Redgate SC]

<https://www.red-gate.com/products/sql-development/sql-compare/>

SQL Compare – a software product by Redgate (red-gate.com)

[RG CCS]

https://www.researchgate.net/publication/273846854_The_Root_Cause_of_Failure_in_Complex_IT_Projects_Complexity_Itself

"The Root Cause of Failure in Complex IT Projects: Complexity Itself" – a research paper by Kaitlynn Marie Castelle and Charles Daniels

[Ries LS]

The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses – a book by Eric Ries

REFERENCES

[SAFe]

<https://www.scaledagileframework.com/>

Scaled Agile Framework's website that describes the large scale Agile process called SAFe

[Selenium]

<https://www.selenium.dev/>

A Selenium website that provides tools, patterns, and guidance for automating web tests

[Tengiz ADPDJ]

<https://www.tutisani.com/software-architecture/postponed-decision.html>

“Architectural Debates: Postponed Decision” – an article by Tengiz Tutisani

[Tengiz AJSAI]

<https://www.tutisani.com/software-architecture/advanced-js-app-architecture.html>

“Advanced JavaScript Application Architecture” – an article by Tengiz Tutisani

[Tengiz CVSI]

<https://www.tutisani.com/software-architecture/composition-vs-inheritance.html>

“Composition vs. Inheritance” – an article by Tengiz Tutisani

[Tengiz BFVTDD]

<https://www.tutisani.com/software-architecture/bug-fix-via-tdd.html>

“Bug Fix via TDD” – an article by Tengiz Tutisani

[Tengiz BPVSIP]

<https://www.tutisani.com/software-architecture/bulk-processing-vs-single-item-processing.html>

“Bulk Processing vs Single Item Processing” – an article by Tengiz Tutisani

REFERENCES

[Tengiz DDGOB]

<https://www.tutisani.com/software-architecture/data-driven-good-or-bad.html>

"Data Driven - Good or Bad?" – an article by Tengiz Tutisani

[Tengiz GTLAA]

<https://www.tutisani.com/software-architecture/global-three-layer-architecture.html>

"Global Three Layer Application Architecture" – an article by Tengiz Tutisani

[Tengiz HTDBO]

<https://www.tutisani.com/software-architecture/how-to-design-business-objects.html>

"How to Design Business Objects" – an article by Tengiz Tutisani

[Tengiz IOM]

<https://www.tutisani.com/software-architecture/intuitive-object-models.html>

"Intuitive Object Models" – an article by Tengiz Tutisani

[Tengiz LSA]

<https://www.tutisani.com/software-architecture/layering-software-architecture.html>

"Layering Software Architecture" – an article by Tengiz Tutisani

[Tengiz Modeling]

<https://www.tutisani.com/software-architecture/software-modeling.html>

"Modeling - Ultimate Way of Architecting Software" – an article by Tengiz Tutisani

[Tengiz MSA]

<https://www.tutisani.com/software-architecture/modular-software-architecture.html>

"Modular Software Architecture" – an article by Tengiz Tutisani

REFERENCES

[Tengiz MVDB]

<https://www.tutisani.com/software-architecture/merging-versioned-databases.html>

"Merging Versioned Databases" – an article by Tengiz Tutisani

[Tengiz SACISD]

<https://www.tutisani.com/software-architecture/simplicity-and-complexity-in-software-development.html>

"Simplicity and Complexity in Software Development" – an article by Tengiz Tutisani

[Tengiz SARIAP]

<https://www.tutisani.com/software-architecture/software-architect-in-agile.html>

"Software Architect's Role in Agile Processes" – an article by Tengiz Tutisani

[Tengiz STVDB]

<https://www.tutisani.com/software-architecture/switching-to-versioned-databases.html>

"Switching to Versioned Databases" – an article by Tengiz Tutisani

[Tengiz VDB]

<https://www.tutisani.com/software-architecture/versioned-databases.html>

"Versioned Databases" – an article by Tengiz Tutisani

[Tengiz VSDB]

<https://www.tutisani.com/software-architecture/versioning-shared-database.html>

"Versioning Shared Database" – an article by Tengiz Tutisani

[Tengiz WHATISA]

<https://www.tutisani.com/software-architecture/what-is-software-architecture.html>

"What is Software Architecture?" – an article by Tengiz Tutisani

REFERENCES

[Tengiz WHOISA]

<https://www.tutisani.com/software-architecture/who-is-software-architect.html>

“Who Is Software Architect?” – an article by Tengiz Tutisani

[Tengiz WTDDIFTUT]

<https://www.tutisani.com/software-architecture/why-tdd-is-faster-than-unit-testing.html>

“Why TDD Is Faster than Unit Testing” – an article by Tengiz Tutisani

[Tengiz WWWSD]

<https://www.tutisani.com/software-architecture/whats-wrong-with-software-development.html>

“What’s Wrong with Software Development” – an article by Tengiz Tutisani

[Tengiz XGQ]

<https://github.com/tutisani/Xunit.Gherkin.Quick>

Xunit.Gherkin.Quick – a lightweight BDD framework by Tengiz Tutisani, available on GitHub and Nuget for download

[TOGAF]

<https://www.opengroup.org/togaf>

TOGAF – Technology Open Group Architecture Framework. An Enterprise Architecture framework by The Open Group

[TTWI FT]

<https://whatis.techtarget.com/definition/flaky-test>

“flaky test” – a definition of term available on TechTarget website

[Tutisani web]

<https://www.tutisani.com/>

tutisani.com website

[TW SL]

<https://www.thoughtworks.com/radar/techniques/structured-logging>

“Structured logging” – trend overview by ThoughtWorks Technology Radar

REFERENCES

[Tzu TAOW]

The Art of War – a book by Sun Tzu

[Vocke TPTP]

<https://martinfowler.com/articles/practical-test-pyramid.html>

“The Practical Test Pyramid” – an article by Ham Vocke, published on martinifowler.com website

[WhiteStarUML]

<http://whitestaruml.sourceforge.net/>

WhiteStarUML – a UML tool

[Wiki AWD]

https://en.wikipedia.org/wiki/Adaptive_web_design

“Adaptive web design” – a Wikipedia page that explains the term

[Wiki CAP]

https://en.wikipedia.org/wiki/CAP_theorem

“CAP theorem” – a Wikipedia page that explains the term

[Wiki Containerization]

<https://en.wikipedia.org/wiki/Containerization>

“Containerization” – a Wikipedia page that explains the term

[Wiki EC]

https://en.wikipedia.org/wiki/Eventual_consistency

“Eventual consistency” – a Wikipedia page that explains the term

[Wiki EDA]

https://en.wikipedia.org/wiki/Event-driven_architecture

“Event-driven architecture” – a Wikipedia page that explains the term

[Wiki HA]

[https://en.wikipedia.org/wiki/Hexagonal_architecture_\(software\)](https://en.wikipedia.org/wiki/Hexagonal_architecture_(software))

“Hexagonal architecture (software)” – a Wikipedia page that explains the term

REFERENCES

[Wiki IM]

https://en.wikipedia.org/wiki/Object-relational impedance_mismatch

“Object-relational impedance mismatch” – a Wikipedia page that explains the term

[Wiki Kanban]

[https://en.wikipedia.org/wiki/Kanban_\(development\)](https://en.wikipedia.org/wiki/Kanban_(development))

“Kanban (development)” – a Wikipedia page that explains the term

[Wiki LOD]

https://en.wikipedia.org/wiki/Law_of_Demeter

“Law of Demeter” – a Wikipedia page that explains the term

[Wiki MTA]

https://en.wikipedia.org/wiki/Multitier_architecture

“Multitier architecture” – a Wikipedia page that explains the term

[Wiki MVC]

<https://en.wikipedia.org/wiki/Model%20view%20controller>

“Model-view-controller” – a Wikipedia page that explains the term

[Wiki MVP]

<https://en.wikipedia.org/wiki/Model%20view%20presenter>

“Model-view-presenter” – a Wikipedia page that explains the term

[Wiki MVVM]

<https://en.wikipedia.org/wiki/Model%20view%20viewmodel>

“Model-view-viewmodel” – a Wikipedia page that explains the term

[Wiki OOP]

https://en.wikipedia.org/wiki/Object-oriented_programming

“Object-oriented programming” – a Wikipedia page that explains the term

REFERENCES

[Wiki PT]

https://en.wikipedia.org/wiki/Pomodoro_Technique

"Pomodoro Technique" – a Wikipedia page that explains the term

[Wiki RWD]

https://en.wikipedia.org/wiki/Responsive_web_design

"Responsive web design" – a Wikipedia page that explains the term

[Wiki SA]

https://en.wikipedia.org/wiki/Software_architecture

"Software architecture" – a Wikipedia page that explains the term

[Wiki SaaS]

https://en.wikipedia.org/wiki/Software_as_a_service

"Software as a service" – a Wikipedia page that explains the term

[Wiki SDP]

https://en.wikipedia.org/wiki/Software_design_pattern

"Software design pattern" – a Wikipedia page that explains the term

[Wiki SOA]

https://en.wikipedia.org/wiki/Service-oriented_architecture

"Service-oriented architecture" – a Wikipedia page that explains the term

[Wiki SOLID]

<https://en.wikipedia.org/wiki/SOLID>

"SOLID" – a Wikipedia page that explains the term

[Wiki SPA]

https://en.wikipedia.org/wiki/Single-page_application

"Single-page application" – a Wikipedia page that explains the term

Index

A

Acceptance test-driven development (ATDD), 231, 232
Agile, 15, 16, 34, 35, 39, 96, 109
Amazon's lambda C# project, 156, 157
Amazon Web Services (AWS), 132, 156
Anemic domain model (ADM), 137, 191
Application database, 163, 164
Architecture landscape, 122
software code, 164
Architecture as a service, 111–113
Architecture team, 112–114
Artifact, 2, 3, 117, 281, 286
Automate deployments, 282
Automated scripts, 239–241, 264, 269
Automation scripts, 243, 264–266, 268

B

Behavior-driven development (BDD), 227, 230, 231
Big Ball of Mud (BBOM), 109, 137, 138
Buy vs. Build, 123–124

C

CI/CD techniques, 284–285
Client-side UI composition (CSUIC), 148, 149
C4 modeling, 189
Code-level testing developers, 227
TDD/ADD/BDD, 230, 231
unit tests, 227–229
Coding
CI continuous refactoring, 181
definition, 181
WIP, 181–183
CI/CD, 232, 233
declarative design, 211, 213–215
designing

INDEX

- Coding (*cont.*)
 design patterns, 193–195
 implementation, 187, 188
 OOP, 189–191, 193
 techniques, 189
planning deployment, 233, 234
planning maintenance, 234
professionalism, 178
 quality *vs.* quantity, 184, 185
reviews, 185, 186
- Command Query Responsibility Segregation (CQRS), 118, 151–152, 205
- Containerization, 155, 156, 171, 173, 276, 286, 287
- Content delivery network (CDN), 218, 219
- Context switching, 9, 17, 37, 38, 99, 238, 267
- Continuous deployment (CD), 272–273, 282, 283
- Continuous improvement (CI), 166, 180, 181
- Continuous integration (CI)
 artifact, 280, 281
 steps, 280
- Continuous integration and continuous deployment (CI/CD), 232, 280
- Continuous refactoring, 181
- Cross-cutting concerns, 106–111
- Culture
 candid feedback, 46, 47
 change of mind, 47, 48
- complexity as job safety, 50, 51
- delegation, responsibilities, 43
- identify talent, 44, 45
- keep it fun, 53
- professionalism, software development, 40, 41
- relaxed team atmosphere, 45
- releases
 smooth deployments, 278
 software, 276, 277
 unimportance, 277, 278
- social aspect,
 engineering, 48, 49
- team spirit, 51, 52
- trust, 42
- work-life balance
 techniques, 46
- Customers' needs
 agile processes,
 planning work
 backlog, 85
 deadlines, 101
 DoD, 98, 99
 estimates *vs.* velocity, 100
 feasibility, 86, 87
 managing
 dependencies, 88–93
 predictability, 101–103
 technical stories, 96, 97
 valuable stories, 94, 95
 end users, 60
 interview, 63
 knowledge exploration
 exercises

- approaches, 65
 - free-formed sketch elements, 67
 - SMEs, 64
 - organization's response
 - customer interviews, 68
 - hierarchical context map, 69–71
 - problem domain, 72
 - subdomain cost, 74, 75
 - partnership/customer focus, 61, 62
 - paying clients, 61
 - software development team, 60
- D**
- Databases
 - application database *vs.* data warehouse, 163, 164
 - identity value generation, 162
 - versioned databases, 164
 - Database technology, 161–162
 - Data-driven models, 137
 - Data transfer objects (DTOs), 196
 - Data warehouse, 163, 164
 - Defect-free software, 9
 - Definition of Done (DoD), 98–99, 101
 - Dependency injection (DI), 168, 211
 - Deployable systems
 - architecture of applications, 169
 - containerization, 171
 - versioning, 169, 170
 - Deployment, 37, 47, 106, 170, 232, 269, 275
 - Deployment-ready applications
 - containerization, 286, 287
 - deployment unit, 283, 284
 - Dev-Prod parity, 285, 286
 - smooth deployments via CI/CD, 284, 285
 - Developers, 236, 238, 239, 241–243, 265, 267
 - Development managers, 32, 179, 237
 - Dev-prod parity, 157, 158, 285, 286
 - Docker, 155, 287
 - Document Object Model (DOM), 251, 253
 - Domain-driven designs (DDD), 91, 113, 137, 165, 181, 195
 - Domain experts, 49, 61–66, 76–78, 80–82, 93, 94, 113, 114, 116, 119, 120, 154
 - Domain modeling
 - ADM, 137
 - anti-patterns, 138
 - BBOM, 137
 - rich domain model, 136
- E**
- E-Commerce, 89
 - e-commerce website, 150
 - Effective software, 7
 - accelerate development pace, 11
 - defects, 9, 10

INDEX

Effective software (*cont.*)
highly paid experts, 5
horizontal scalability, 10
learning curve, 3
naive hopes, 5, 6
quality over quantity, 3, 4
ROI per Developer, 11
team and software, 11
technical production support
team, 11
users' expectations, 8

End-to-end test, 231,
260, 263–265

Engineering organizations, 14, 22,
54, 55, 291

Engineering teams, 16, 97, 111, 114,
116, 165, 172, 175, 291–296

Event-driven architecture (EDA),
118, 125, 153, 154, 205

Event Storming (ES), 65–67,
118, 154

Eventual consistency (EC), 147,
153, 154

Evolving design, 96, 108, 165–166

Exception handling
catch exceptions, 223
direct dependencies, 224, 225
throw, 224
unhandled exceptions, 225

Execution, leadership and
management
ad-hoc tasks *vs.* process, 17
Agile transformation, 15, 16
hands-on leader, 18

importance, software
development
transformation, 14
software development
transformation, 15
support, 19
technology
transformation, 15, 16

F

Feature teams, 26–28
Flaky tests, 270, 271
Frenemy tests, 270, 271
Front-end application
architecture, 159–160
Front-end development, coding
form factors, 220, 221
hosting static assets, 218, 219
JavaScript programming,
215, 216
limits, 222
repositories/services, 216–218
Functional test, 245, 262

G

Gap analysis, 116–117
Globally unique identity (GUID), 162
Good architecture
decisions, 127
and ideal architecture, 124
practice, 125, 127
technical debt, 124, 125

H

- Hexagonal Architecture (HA),
141, 142
High-quality applications, 235, 236
Horizontal organizational silos, 31

I

- Impedance mismatch (IM),
162, 199
Inefficient Monoliths, history, 1–2
Infrastructure teams, 26–29, 150
Integrated development
environment (IDE), 133
Integration test, 244, 245,
260–262, 271
Is Design Dead (IDD), 110
Iterative improvement, 180

J

- JavaScript programming, 215

K

- Knowledge exploration, 64–66, 118,
154, 276

L

- Layered architecture
application frameworks, 140
guardrails, 140
HA, 142

- layers, 141
OA, 141
Layering patterns, 140
Legacy systems, 125, 126, 173

M

- Maintainable systems
common maintenance task,
173, 174
fixing problems, 174, 175
mindset shift, 172
simple *vs.* complex systems, 175
working product, 172
working systems, 172, 173
Maintenance-aware mindset
engineering teams, 293–295
fix root cause, 296
software systems, 296
techniques, 296, 297
Maintenance-free mindset
award-winning support
teams, 292
organizations approach, 290, 291
prevent problems, 293
support-oriented
organizations, 291
Micro-management strategies, 42
Microservices, 24
autonomy/scale, 144, 145
ecosystem
CSUIC, 149
MUI, 148
SSPFC, 150, 151

INDEX

Microservices (*cont.*)

- EDA, 154
 - implementation, 143
 - integration, 144
 - reusability *vs.* duplication, 146, 147
 - scale out, 145
 - SOA, 148
 - and teams, 114
 - terminology, 142
- Microsoft Test Management (MTM), 250
- Model Exploration Whirlpool (MEW), 65
- Model, view, and controller (MVC), 140
- Monolithic UI (MUI), 148

N

NoSQL databases, 34

O

- Object-Oriented Programming (OOP), 189–193
- Onion Architecture (OA), 141
- Organizational growth, 115
- Organizational structure
 - autonomy *vs.* reuse, 33, 34
 - bounded context, 20
 - bounded contexts within subdomains, 23
 - microservice, 20

organizational silos

- horizontal, 31
 - vertical silos, 31, 33
- programs, 29, 30
- projects, 29, 30
- requirements, 29, 30
- subdomains, 20, 21
- team
 - bounded contexts, 25
 - platform/infrastructure, 27
- technology organizations, 20

Organizational structure

- microservice, 24
 - subdomains, 21, 22
 - subsystem, 22, 23
- team
 - bounded contexts, 25, 26
 - feature, 26, 27
 - platform/infrastructure, 27–29

Organizations, 1, 5

P

- Professionalism, 40–42, 178–179
- Phoenix Project, 124
- Platform teams, 27, 28

Q

- QA engineers, 99, 231, 232, 241, 242, 244
- QA manager, 237
- Quality assurance (QA), 31, 167, 169, 231, 235–237, 240, 241

R

Recruitment

- classification, well-performance engineers, 56, 57
- corrective *vs.* preventive, 57
- hire best talent, 55
- role, 54

Refactoring to deeper insight, 181

- Return on investment (ROI), 108, 110, 119, 120, 129, 136, 137, 139, 153, 157, 177, 236, 240, 242, 243, 259, 266

Rich domain model, 136, 138, 152, 153

Rule of thumb, 52, 134, 141, 223

S

SDLC, 299

Security, 160, 161

Servers, 2, 281

Server-side page fragment composition (SSPFC), 148, 150–151

Simple Notification Service (SNS), 133

Single-page apps (SPA), 179

Software architect, 107, 108, 113, 119, 120, 124, 130, 155, 165, 167, 172, 174

Software Architect's Role in Agile Processes (SARIAP), 110

Software architecture (SA), 106
brainstorm, coding, 110

definition, 107

implementation

- development tools, 167
- domain layer's code, 166
- evolving design, 165
- languages, 167
- tactical DDD, 165

performance

- vs.* clean design, 158, 159
- designing components, 159

quality, 168

reuse knowledge, 111

technical solution, caveat, 118, 120, 121

Software as a Service (SaaS), 129

Software development

- perfectionism, 6–7

Software development teams, 2, 25, 75, 92, 236

Software engineering, 14, 19, 30, 41, 108, 112, 160, 299

Software engineers, 44, 77, 109, 119, 130

Software systems, 296, 300

codebase, 130

deployment, 169

quality, 168

Stakeholders, 17, 47, 73, 95

"Startup syndrome", 3, 4

Story writing, customer needs

executable specifications, 81

Gherkin, 84

Halfway, into Gherkin, 82, 83

INDEX

Story writing, customer

needs (*cont.*)

technical language, 78

ubiquitous language, 76,

77, 79, 80

Support-oriented

organizations, 291

T

Tactical DDD patterns

active record entities, 196–199

command, 208, 209, 211

enforcing modular

architecture, 201

entities, 196

factory, 206

GoF factory, 206, 207

module, 199–201

non-relational database, 204

relational database, 202, 203

reports, 205

repository, 202

Team's responsibility, 236, 237

Technical staff, 94, 125

Technologists, 135

Technology

abstractions, 134

consolidate, 135, 136

definition, 128

designing systems, 133

domain, 130, 131

rewriting program, 129

rule of thumb, 134

software cost, 128

technologists, 135

vendor lock-in/vendor

lockdown, 132, 133

Technology organizations, 20

Testable systems

testable application, 168, 169

testable code, 168

Test and automate everything

assurance, 239, 240

automation, 238

developer, 238

low quality standards, 238

manual testing, 238

program, 238

real-time indicators, 239

Test automation process

best practices, 243

blur line, 243, 244

delayed automation, 265, 266

deployments

CD, 272

compensating

actions, 269–271

fast feedback, 269–271

design patterns

benefits, 248

Gherkin test case pattern,

248, 249

repository pattern, 248

scenario object pattern,

250, 251

developers, 241

early automation, 266–269

- engineers, 241
layered test architecture,
 257, 258
observations, 240
page object pattern
 designing, 255
 GetProducts() method, 255
problems, 253, 254
product names, 252
ProductsPage class, 252, 254
Selenium's objects, 251
test code, 253
test methods, 254
 web page functionality, 251
patterns, 257
test object pattern, 255–257
time, 242
ubiquitous language, 258, 259
Test cases, 242, 246–248, 250, 251,
 259, 265–267
Test data management, 259
 end-to-end tests, 263–265
 functional tests, 262
 integration tests, 260, 261
 unit tests, 259, 260
Test-driven development (TDD),
 178, 227, 230, 231
Testers, 236, 237, 239, 242, 243, 245,
 246, 265, 266
Test types
 functional test, 245
 integration test, 244, 245
Transformations, 13–16, 39, 40
Transparent teams, 36–37
Trust, 42
Twelve-factor app, 155, 157
- U**
- Unit tests, 40, 101, 174, 178, 184,
 227–230, 259, 260
Upfront design, 116
 comprehensive analysis, 117
 domain experts, 116
 implementation, 116
 requirements/technical
 challenges, 116
 techniques/prerequisites,
 117, 118
- V**
- Vendor lock-in, 132–133, 153,
 156, 161
Versioned databases, 164
Vertical scalability, 10
Vertical silos, 31–33
- W, X, Y, Z**
- Waterfall, 34, 35, 38, 85, 100, 109
Whiteboard Modeling, 65–67,
 117, 118
Work-life balance techniques, 46