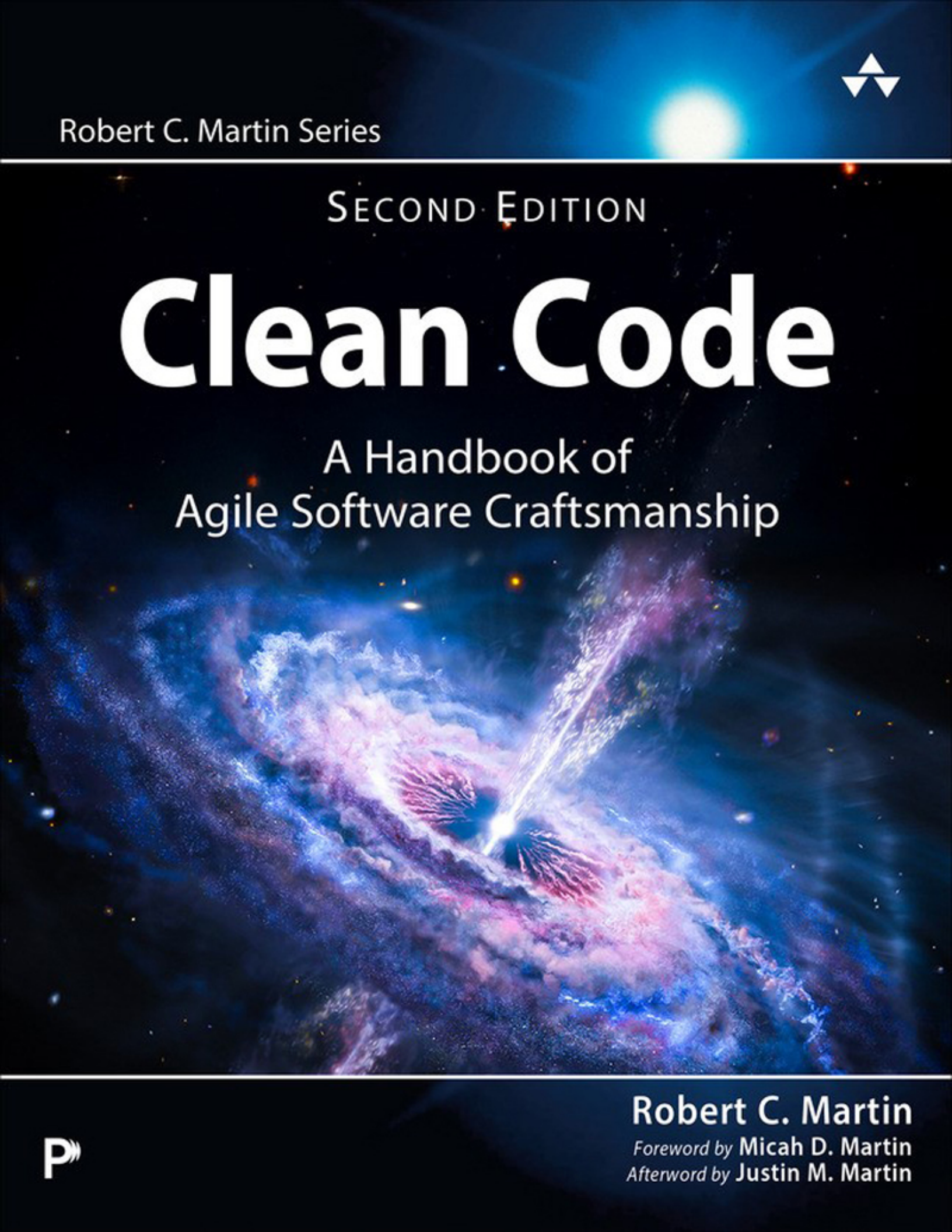
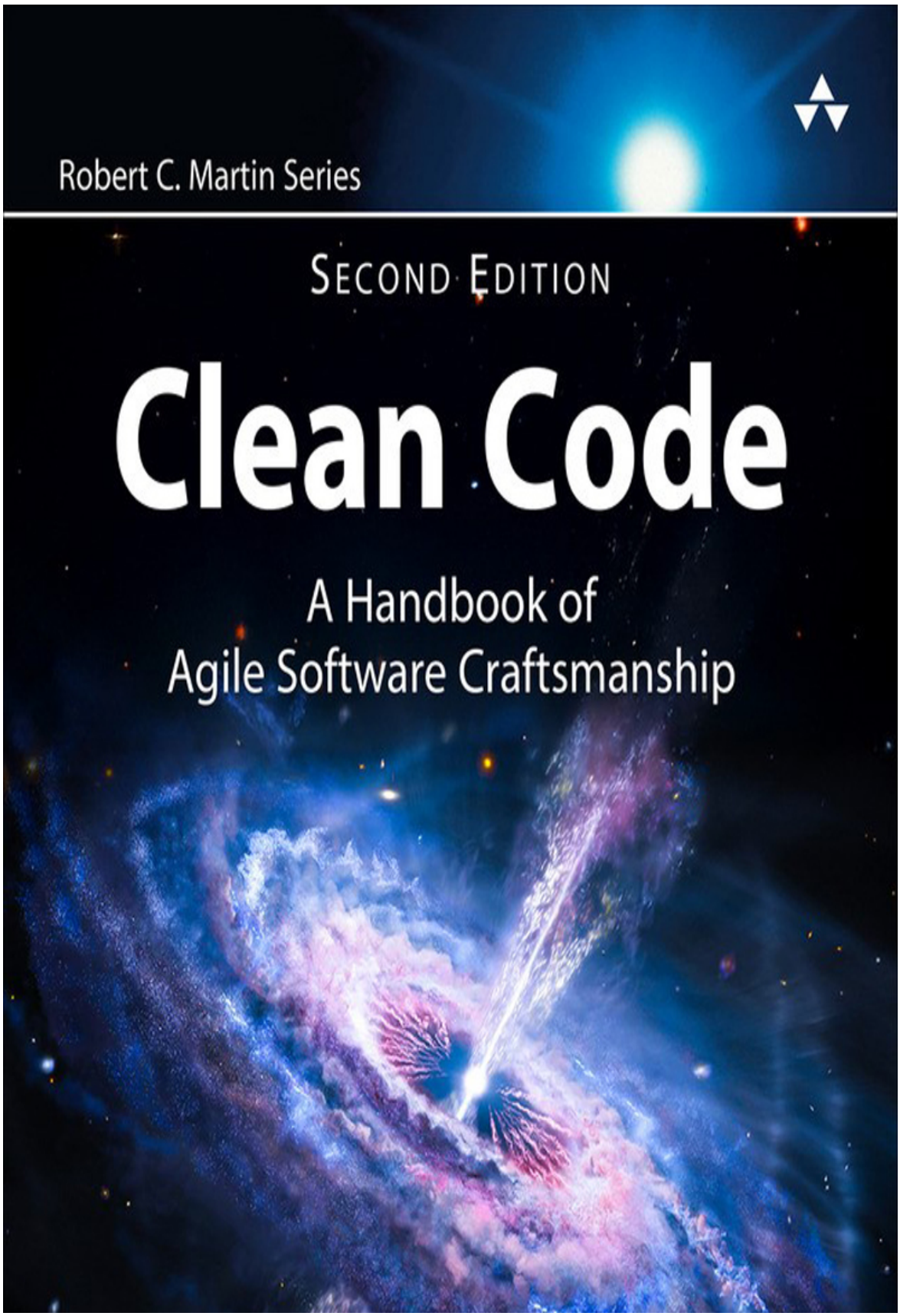

Robert C. Martin Series
SECOND EDITION
Clean Code
A Handbook of
Agile Software Craftsmanship
Robert C. Martin
Foreword by Micah D. Martin
Afterword by Justin M. Martin


Robert C. Martin Series
SECOND EDITION
Clean Code
A Handbook of
Agile Software Craftsmanship


Robert C. Martin
Foreword by Micah D. Martin
Afterword by Justin M. Martin

Clean Code: A Handbook of Agile Software Craftsmanship

Second Edition

Robert C. Martin

A NOTE FOR EARLY RELEASE READERS

With Early Release eBooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this title, please reach out to Pearson at PearsonITAcademics@pearson.com

Contents

Foreword
Introduction
Introduction (from Long Ago)

Chapter 1: Clean Code

Part I: Code

Chapter 2: Clean That Code!
Chapter 3: First Principles
Chapter 4: Meaningful Names
Chapter 5: Comments
Chapter 6: Formatting
Chapter 7: Clean Functions
Chapter 8: Function Heuristics
Chapter 9: The Clean Method
Chapter 10: One Thing
Chapter 11: Be Polite
Chapter 12: Objects and Data Structures
Chapter 13: Clean Classes
Chapter 14: Testing Disciplines

Chapter 15: Clean Tests
Chapter 16: Acceptance Testing
Chapter 17: AIs, LLMs, and God knows What

Part II: Design

Chapter 18: Simple Design
Chapter 19: The SOLID Principles
Chapter 20: Component Principles
Chapter 21: Continuous Design
Chapter 22: Concurrency

Part III: Architecture

Chapter 23: The Two Values of Software
Chapter 24: Independence
Chapter 25: Architectural Boundaries
Chapter 26: Clean Boundaries
Chapter 27: The Clean Architecture

Part IV: Craftsmanship

Chapter 28: Harm
Chapter 29: No Defect in Behavior or Structure
Chapter 30: Repeatable Proof
Chapter 31: Small Cycles
Chapter 32: Relentless Improvement
Chapter 33: Maintain High Productivity
Chapter 34: Work as a Team

Chapter 35: Estimate Honestly and Fairly
Chapter 36: Respect for Fellow Programmers
Chapter 37: Never Stop Learning
Afterword
Appendix

Bibliography

Table of Contents

Foreword
Introduction
A Note About Older Chapters
The Structure of This Book
Introduction (from Long Ago)
Chapter 1. Clean Code
There Will Be Code
Bad Code
The Art of Clean Code?
Putting All This Together
We Read More Than We Write
The Boy Scout Rule

Part I: Code
Chapter 2. Clean That Code!
The Cleaning Process
Conclusion
Postscript: Future Bob Playing with Grok3
Postscript Conclusion
Chapter 3. First Principles
Everything Small, Well Named, Organized, and Ordered

A More Significant Example
A Final Thought
Chapter 4. Meaningful Names
Use Intention-Revealing Names
Final Words
Chapter 5. Comments
Compensating for Failure
Good Comments
Bad Comments
Conclusion
Chapter 6. Formatting
The Purpose of Formatting
Vertical Formatting
Team Rules
Uncle Bob’s Formatting Rules
Chapter 7. Clean Functions
Small!
Reading Code from Top to Bottom: The Stepdown Rule
Switch Statements
Clean Functions: A Deeper Look
Conclusion
Chapter 8. Function Heuristics
Function Arguments
Command Query Separation
Prefer Exceptions to Returning Error Codes
Structured Programming
Conclusion

Chapter 9. The Clean Method
- Make It Right
- Example
- Conclusion

Chapter 10. One Thing
- Extract Method Refactoring
- What Are Large Functions Anyway?
- Conclusion

Chapter 11. Be Polite
- The Newspaper Metaphor
- The Stepdown Rule: Once Again
- The Abstraction Roller Coaster
- This Is How We Write, but Not How We Want to Read

Chapter 12. Objects and Data Structures
- What Is an Object?
- Data Abstraction
- Data/Object Antisymmetry
- The Law of Demeter
- Data Transfer Objects
- The OO/Procedural Trade-off
- But What About Performance?
- Conclusion

Chapter 13. Clean Classes
- Classes and Modules versus Files
- What Should a Class Contain?

Chapter 14. Testing Disciplines
- Discipline 1: Test-Driven Development (TDD)

Discipline 2: Test && Commit || Revert (TCR)
Discipline 3: Small Bundles
Design
Discipline
Keeping Tests Clean
Tests Enable the -ilities
Chapter 15. Clean Tests
Domain-Specific Testing Language
F.I.R.S.T.
Test Design
Conclusion
Chapter 16. Acceptance Testing
The Acceptance Testing Discipline
Conclusion
Chapter 17. AIs, LLMs, and God Knows What
Programming by Prompt
Conclusion

Part II: Design

Chapter 18. Simple Design
YAGNI
Covered by Tests
Maximize Expression
Minimize Duplication
Chapter 19. The SOLID Principles
SRP: The Single Responsibility Principle
OCP: The Open–Closed Principle
LSP: The Liskov Substitution Principle

ISP: The Interface Segregation Principle
DIP: The Dependency Inversion Principle
Chapter 20. Component Principles
Components
A Brief History of Components
Component Cohesion
Component Coupling
Conclusion
Chapter 21. Continuous Design
Continuous Change
Continuous Design
Sailing on the Four Cs of Continuous Design
When Else Do We Design?
Chapter 22. Concurrency
Why Concurrency?
Concurrency Defense Principles
2025 Update and Report from the Field
Conclusion

Part III: Architecture

Chapter 23. The Two Values of Software
Keeping Options Open
Chapter 24. Independence
Use Cases
Operation
Development
Deployment
Leaving Options Open

Chapter 25. Architectural Boundaries
What Lines Do You Draw, and When?
Plug-in Architecture
Case Study: FitNesse
Conclusion
Chapter 26. Clean Boundaries
Third-Party IoT Framework: Lots o’ Boundaries
UI/Application Boundary
Clean Boundaries
Chapter 27. The Clean Architecture
The Dependency Rule
Entities
Conclusion

Part IV: Craftmanship

Chapter 28. Harm
No Harm to Society
Harm to Function
No Harm to Structure
Soft
Tests
Chapter 29. No Defect in Behavior or Structure
Making It Right
Programmers Are Stakeholders
Do Your Best
Chapter 30. Repeatable Proof
Dijkstra
Structured Programming

Functional Decomposition
Test-Driven Development et al.
Chapter 31. Small Cycles
The History of Source Code Control
Continuous Integration
Branches versus Toggles
Continuous Deployment
Continuous Build
Chapter 32. Relentless Improvement
Test Coverage
Mutation Testing
Semantic Stability
Cleaning
Creations
Chapter 33. Maintain High Productivity
Viscosity
Managing Distractions
Time Management
Chapter 34. Work as a Team
Collaborative Programming
Open/Virtual Office
Chapter 35. Estimate Honestly and Fairly
Lies
Honesty, Accuracy, Precision
Lessons from Me
Accuracy and Precision
Aggregation
Honesty

Pressure
Chapter 36. Respect for Fellow Programmers
Chapter 37. Never Stop Learning
Afterword
Appendix. The Clean Code Debate
Introductions
Method Length
Method Length Summary
Comments
Comments Summary
John’s Rewrite of PrimeGenerator
A Tale of Two Programmers
Bob’s Rewrite of PrimeGenerator2
Test-Driven Development
TDD Summary
Closing Remarks
Bibliography

Foreword



Micah Daniel Martin

You hold in your hand the secrets for building high-quality software—secrets that will make you better than you are, secrets that will help you become a master craftsman.

Like Antonio Stradivari (perhaps the best luthier ever), Uncle Bob has spent a lifetime curating and developing the best techniques for crafting code. Unlike Stradivari, who hoarded his secrets so that only he could craft the best violins, Uncle

Bob shares his secrets willingly, openly, hoping that you too will craft masterful code.

This is the Way: the Way of Clean Code. These are my disciplines, my principles, my techniques. I strive to master them all. Why? Because they make me better. And, this is the way my father taught me.

Yes, Uncle Bob is my father. He bestowed upon me the honor of writing this foreword. Furthermore, please allow me to introduce my brother, Justin, also a Clean Coder, who wrote the afterword. How appropriate that two second editions of Uncle Bob get to sandwich his second edition of *Clean Code*. (Yeah . . . that sounded cheesy as I wrote it, but I'm going with it.)

Let me tell you a little about Uncle Bob, or at least the version of him that I know as "Dad."

One of my earliest memories is riding on my dad's shoulders, playing Robot. The rules are simple. My dad was the robot, and he did whatever I told him to do . . . to a T.

Me: "Go to my room."

Dad: "I don't know how to do that."

Me: Thinking . . . "Turn."

Dad: "Which way?"

Me: "Turn right."

Dad: Starts turning clockwise. 90°. . . 180°. . . 360°. . . He doesn't stop turning.

Me: "STOP!"

Dad: Stops. Stands still, facing the absolute wrong direction.

Me: "Turn right . . . STOP!"

Dad: Turns and stops obediently. We are now facing down the hallway, the right direction.

Me: “Start walking.”

Dad: “I don’t know how to walk.”

For the next few minutes, I teach the robot how to “walk” by lifting a leg the appropriate amount and falling forward, then repeating with the other leg. We agree to name this activity “walking.”

Me: “Start walking.”

Dad: Starts walking.

Me: Feeling exhausted by this ignorant robot, but also victorious that I successfully made it do my bidding, I prepare for the final maneuver, a left turn at the end of the hall. “Turn left!”

Dad: “I am still walking” . . . and CRASH we go into the wall. Apparently the robot was single threaded. We bounce off the wall and the robot continues to walk . . . CRASH we go into the wall again . . . and again.

Me: Laughing uncontrollably.

At that tender age of 4 or 5 years old, my dad was teaching me how to program with the Robot game. It was fun. Fast-forward 5 years, and my dad brings home a Commodore 64. “What is it?” I inquired. “A computer,” my dad responded. “What is it for?” I asked, still confused. “Well, with a computer you can blah blah blah blah, blah blah, you can even play games.”

“Games? Show me!” I adopted that Commodore 64, or maybe my dad gave it to me . . . regardless, it lived in my room and I played *lots* of games on it. There was a kid in my neighborhood who knew how to pirate games, and we’d do

a weekly floppy disk exchange. Other than loading games from disks, I didn't really know how to use the computer, and anytime something went wrong, like "PRESS PLAY ON TAPE" . . . I'd shout "DAD! HELP!"

At one point, he sat me down at the Commodore and introduced me to Logo, a programming language with turtle graphics. He showed me how to make the turtle turn, move forward. It was the Robot game! I played with Logo for a while and it was . . . boring. Back to gaming.

Then, I observed my dad working with Logo for what seemed like forever but was probably just a couple days. When he was finally done monopolizing my computer, he showed me his Lunar Lander game that he'd just built. It was a jaw-dropping moment as I realized this is how games are built. Wow! Lunar Lander was inducted into my game rotation, and I proudly showed it off to my friends.

A couple of years later, my dad brought home a brand-new Apple Macintosh. It was AMAZING. "What's that thing?" "This is a mouse." So novel. And it had more games. Better games! The Mac lived in my dad's office (basement), but when he was at work, the Mac was mine. Similar to the Lunar Lander incident, my dad spent weeks monopolizing the Mac to create a game he called "Pharaoh." Pharaoh was much bigger than Lunar Lander. You were the pharaoh, and your goal was to buy enough oxen, plant enough crops, and sustain enough workers every year to build a pyramid before you died. It was a fun game, and I could tell my dad had fun building it.

Programming is so much fun! That's the conclusion I had when I reached college age. For my whole childhood, if I wasn't having fun playing on the computers, my dad was having fun programming them. Why would I choose any other major than computer science? Programming in college

was . . . not fun. Upon graduating, I was offered a job at ThoughtWorks, and, thank goodness, my dad offered me a job working with him at Object Mentor. Of course I accepted my dad's offer, where I got to work with some of the biggest names and best software guys in the industry. Once again, programming was fun. It was at Object Mentor where I learned the way of Clean Code from my dad, building lots of internal tools and open source projects like FitNesse, that you'll see later in this book.

Object Mentor was primarily a training and consulting business. We told people how to build good software. We didn't really write the software for them. I'll admit that after several years as a trainer, I got an itch to build software again, instead of teaching how to build software.

An opportunity arose, and Object Mentor started developing software for a client. It was great, and it turned out to be a solid business model. So, I convinced my dad that Object Mentor should do more software development contracts. He agreed. And then I tried to convince him that I should run that branch of the business. Oops . . . that was one step too far. His response hit me like a truck. "If you want to run things, you should start your own business." Wow! Was that his way of putting me in my place? Or was he actually suggesting I start my own business? I concluded it was the latter, so I told him, "I quit."

I created a company called 8th Light. It was (and still is) a contract software development company. Clients hired us to build software for them. What was unique about 8th Light was that we practiced the Way of Clean Code. We didn't call it that, because the first edition of *Clean Code* hadn't been written yet. But we put into practice everything that my dad had taught me. The Uncle Bob Way of Coding doesn't have the same ring. Moreover, 8th Light adopted apprenticeship, which is something we had dabbled with at Object Mentor.

Everyone who joined 8th Light had to go through a 3-to-12-month apprenticeship where they learned the Way of Clean Code. We were ALL-IN on Clean Code. Did it work? Hell yeah! Demand for Clean Code went through the roof. We couldn't train apprentices fast enough. By the time of my departure from 8th Light in 2015, we had nearly a hundred team members, most of them "Clean Code" Craftsmen working on our clients' projects.

But maybe 8th Light was a fluke. In 2020, my dad and I decided to start Clean Coders Studio. This is a branch off our existing partnership, Clean Coders, where we sell educational software videos (https://cleancoders.com). However, "Studio" is our contract software development branch of the business, and yes, I do run it. Once again, we are ALL-IN on Clean Code, more than ever. Is it working? Hell yeah. Once again, we cannot train apprentices fast enough to meet demand.

The industry wants Clean Code. The industry NEEDS Clean Code. This book is your guide into the world of Clean Code.

I have some advice for you on how to best use this book.

1. **Don't rush it.** There is a lot of ground to cover in these pages. Apprentices at Clean Coders Studio take months to internalize all the material. Don't feel like you need to cram it all in a week.
2. **Use it as a reference.** You might want to stick some tabs on some of the pages. For example, the SOLID principles; you'll likely want to jump back to that section when you're trying to debate a code smell with your peers.
3. **Read the code examples.** When Uncle Bob walks through code changes, like the Video Store, you might be tempted to skip to the end. Don't. Follow along.

Ideally, you'd have Uncle Bob sitting next to you, pairing with you on this code. His walk-throughs are the next best thing.

4. **Practice.** "Calculus is not a spectator sport." I can still hear my professor recite that when he assigned homework. Coding is not a spectator sport either. You need to practice it to internalize it. Especially TDD.

5. **Have fun!** Uncle Bob has fun when he codes. Trust me, he does. So do I. *Clean Code* should help you have more fun writing code too.

—Micah Daniel Martin

Introduction

“Code Quality is a mindset that I’ve seen correlate with wealth and great outcomes over and over again.”

—David Heinemeier Hansson (DHH)

As I type this, it has been over a decade and a half since *Clean Code* was first published. Since then, I have participated in many discussions, podcasts, interviews, blog postings, and social network flame wars inspired by that first edition. Recently it occurred to me that there are quite a few more things I could say; and several more that I would say differently than I did in 2008.

I first proposed the *Clean Code* book in April of 2006. This was before there were iPhones. Mobile internet was mostly GPRS, or maybe EDGE. Nobody was talking about the Internet of Things. YouTube was barely one year old, and hardly anyone had heard of it—fewer took it seriously. Languages like Kotlin, Swift, Go, Dart, Rust, Elm, Elixir, and Clojure had not yet appeared. *Meet the Fockers* was the top-grossing movie up to that point.

Yeah, that was a long time ago—especially in internet time. So, it’s time to take a new look at the old message of *Clean Code*. The essence of that message has not changed; but after a decade and a half our industry has—and remarkably so. After so much change, it would be irresponsible to just

dust off the old manuscript and make a few simple adjustments. So, in light of that decade and a half of progress, learning, and change, I decided to significantly restate and expand the original message.

I have added many new chapters, new ideas, and new disciplines. I have added newer languages, and newer paradigms. I have also revamped the older chapters with a decade and a half of newer insights and knowledge.

I have added chapters on design, architecture, and even ethics. These were missing from the first edition, and I felt it necessary to repair that omission. In some cases, these chapters are heavily edited and abridged excerpts from my more recent books, *Clean Architecture* and *Clean Craftsmanship*; others are entirely new.

In the original edition, I was reluctant to assert my ideas as anything more than one programmer's way of looking at things. Now, many years later, I'm a bit more confident than that. The ideas in that original edition have withstood the test of time and have quite a bit more staying power.

On the other hand, there are other very experienced programmers who have different ideas, and I thought it wise to include some of their contrary viewpoints. After all, I'm not the only one who has ideas about what it means for code to be clean. I like my ideas best, but I think all creditable ideas should be heard and considered.

On yet another hand, in the process of reviewing the newer chapters months after they had been written, I was surprised to find that I had issues with some of the examples I had posed. Rather than erasing that surprise by further "cleaning" the examples, I added some Future Bob comments so that you could see my own reaction.

Why is all this cleanliness important? What benefits do these ideas about clean code bring? Why should we be worried about code cleanliness at all?

Our civilization depends upon us. Nothing happens in our society without computers being intimately involved in virtually every aspect of our lives. Nowadays we cannot microwave a hot dog, watch TV, make a phone call, drive to the store, wash our dishes or our clothes, buy anything, sell anything, or even tell the time of day without software being smack in the middle of it all.

You and I, we programmers, rule the world. Other people think they rule the world, but then they hand those rules to us, and *we* write the rules—the software—that executes in the machines that sit in the middle of the details of billions of people's lives.

And some of the software we wrote has caused two fully loaded passenger jets to slam into the ground at near the speed of sound, cars to accelerate out of control and crash, billions of dollars to be lost in seconds, rockets to explode, spacecraft to crash into Mars, and noxious fumes to be emitted by tens of thousands of cars. Just for starters.

And with such a burden of responsibility and failure sitting upon us, we have to ask ourselves this question: Are we professionals?

A professional is someone who professes a set of standards, disciplines, and ethics. Do we? Do you? If so, can you recite them?

My goal for this book is to provide some standards of code cleanliness, some disciplines of code creation and maintenance, and the beginnings of an ethical framework to help us define a true profession. Because if we don't define and adopt that profession, others who are less able but more

powerful—the politicians of the world—will define it for us and force it down our throats.

I hope there's still time for us to get there before that idea occurs to them.

A Note About Older Chapters

I have taken a great deal of liberty with the older chapters for which I was the sole author. In many cases they have been completely rewritten, or very substantially edited.

Most of the authors of the chapters that I did not write have submitted second editions of their chapters to this effort.

The Structure of This Book

Possibly the greatest change between the first and second editions is the structure of the material. I have divided the book into four parts: Code, Design, Architecture, and Craftsmanship.

These parts are not independent of each other.

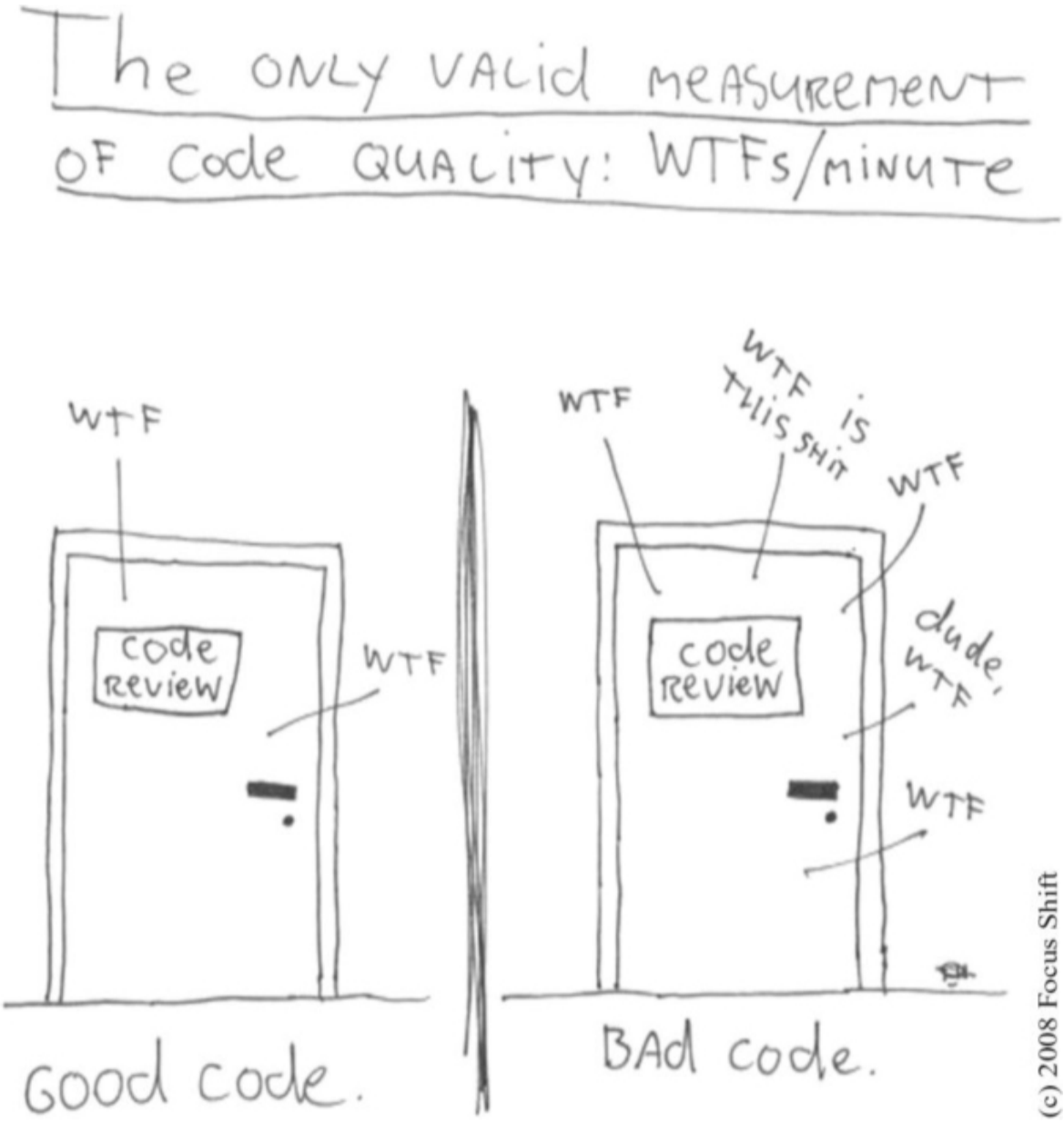
Part I: Code strongly depends upon Part II: Design. As you read Part I, you will find many references to design principles and design strategies that are described in Part II. I encourage you to treat Part II as a reference that you can consult while reading Part I. Then, when you are done with Part I, you should read Part II in its entirety.

Part III: Architecture is a set of deeply edited and abridged excerpts from my book *Clean Architecture*. Again, consider it a reference to assist in your understanding of Part I. However, it also stands alone as a simple introduction to higher-level architectural principles.

Part IV: Craftsmanship is, once again, a set of deeply edited and abridged excerpts from my book *Clean Craftsmanship*—specifically the chapters that have to do with the ethics of software development. It is included as a way to underscore that there is an ethical imperative to writing clean code.

Introduction (from Long Ago)

(This is the Introduction to the first edition. Edited and abridged.)



Reproduced with the kind permission of Thom Holwerda (www.osnews.com/story/19266/WTFs_m).

Look at the drawing above. Which door represents your code? Which door represents your team or your company? Why are we in that room? Is this just a normal code review or have we found a stream of horrible problems shortly after going live?

Are we debugging in a panic, poring over code that we thought worked? Are customers leaving in droves and

managers breathing down our necks? How can we make sure we wind up behind the right door when the going gets tough? The answer is: craftsmanship.

There are two parts to learning craftsmanship: knowledge and work. You must gain the knowledge of principles, patterns, practices, and heuristics that a craftsman knows, and you must also grind that knowledge into your fingers, eyes, and gut by working hard and practicing.

I can teach you the physics of riding a bicycle. Indeed, the classical mathematics is relatively straightforward. Gravity, friction, angular momentum, center of mass, and so forth can be demonstrated with less than a page full of equations. Given those formulae, I could prove to you that bicycle riding is practical and give you all the knowledge you needed to make it work.

And you'd still fall down the first time you climbed on that bike.

Coding is no different. We could write down all the "feel good" principles of clean code and then trust you to do the work (in other words, let you fall down when you get on the bike), but then what kind of teachers would that make us, and what kind of student would that make you?

No. That's not the way this book is going to work.

Learning to write clean code is hard work. It requires more than just the knowledge of principles and patterns. You must *sweat* over it. You must practice it yourself, and watch yourself fail. You must watch others practice it and fail. You must see them stumble and retrace their steps. You must see them agonize over decisions and see the price they pay for making those decisions the wrong way.

Be prepared to work hard while reading this book. This is not a “feel good” book that you can read on an airplane and finish before you land. This book will make you work, *and work hard*. What kind of work will you be doing? You’ll be reading code—lots of code. And you will be challenged to think about what’s right about that code and what’s wrong with it. You’ll be asked to follow along as we (the authors) take modules apart and put them back together again. This will take time and effort; but we think it will be worth it.

Chapter 1. Clean Code



“‘Clean’ isn’t a ding at anyone. Our notion of ‘unclean’ is a way of characterizing code that demands more effort from the average developer to understand or maintain.”

—Jeff Langr

You are reading this book for two reasons. First, you are a programmer. Second, you want to be a better programmer. Good. We need better programmers.

This is a book about good programming. It is filled with code. We are going to look at code from every different direction. We'll look down at it from the top, up at it from the bottom, and through it from the inside out. By the time we are done, we're going to know a lot about code. What's more, we'll be able to tell the difference between good code and bad code. We'll know how to write good code. And we'll know how to transform bad code into good code.

There Will Be Code

One might argue that a book about code is somehow behind the times—that code is no longer the issue; that we should be concerned about AI, large language models (LLMs), and requirements instead. Indeed, some have suggested that we are close to the end of code.[1] That soon, all code will be generated instead of written. That programmers simply won't be needed, because businesspeople will generate programs from specifications—or prompts.

[1] It is hilarious that I wrote that line in 2008, and here it is, 2024. Some things never change.

Nonsense! We will never be rid of code, because code represents the details of the requirements. At some level, those details cannot be ignored or abstracted; they have to be specified. And specifying requirements in such detail that a machine can execute them is programming. Such a specification is code.

I expect that the level of abstraction of our languages will continue to increase. I also expect that the number of domain-specific languages and powerful AIs will continue to

grow. This will be a good thing. But it will not eliminate code. Indeed, all the specifications written in these higher-level and domain-specific languages, and all the prompts written to guide a really good AI, will be code! It will still need to be rigorous, accurate, and so formal and detailed that a machine can understand and execute it.

The folks who think that code will one day disappear are like "mathematicians"[2] who hope one day to discover a mathematics that does not have to be formal. They hope one day to discover a way to create machines that can do what they want rather than what they say. These machines will have to be able to understand us so well that they can translate vaguely specified needs into perfectly executing programs that precisely meet those needs.

[2] In quotes because no real mathematician would hope for this.

This will never happen. Not even humans, with all their intuition and creativity, have been able to create successful systems from the vague feelings of their customers. Indeed, if the discipline of requirements specification has taught us anything, it is that well-specified requirements are as formal as code and can act as executable tests of that code!

Remember that code is really the language in which we ultimately express the requirements. We may create languages that are closer to the requirements. We may create tools that help us parse and assemble those requirements into formal structures. But we will never eliminate the necessary precision and formality—so there will always be code.

Bad Code

Long ago I read the preface to Kent Beck's book *Implementation Patterns.*[3] He said, ". . . this book is based

on a rather fragile premise: that good code matters. . . .” A fragile premise? I disagree! I think that premise is one of the most robust, supported, and overloaded of all the premises in our craft (and I think Kent knows it). We know good code matters because we’ve had to deal for so long with its lack.

[3] [IMP].



I know of one company who, in the late ’80s, wrote a killer app. It was very popular, and lots of professionals bought and used it. But then the release cycles began to stretch. Bugs were not repaired from one release to the next. Load times grew and crashes increased. I remember the day I shut the product down in frustration and never used it again. The company went out of business a short time after that.

Two decades later I met one of the early employees of that company and asked him what had happened. The answer confirmed my fears. They had rushed the product to market and had made a huge mess in the code. As they added more and more features, the code got worse and worse until they simply could not manage it any longer. It was the bad code that brought the company down.

Have you ever been significantly impeded by bad code? If you are a programmer of any experience, you've felt this impediment many times. Indeed, we have a name for it. We call it *wading and slogging*. We wade through bad code. We slog through a morass of tangled brambles and hidden pitfalls. We struggle to find our way, hoping for some hint, some clue, of what is going on; but all we see is more and more senseless code.

Of course you have been impeded by bad code. So then—why did you write it?

Were you trying to go fast? Were you in a rush? Probably so. Perhaps you felt that you didn't have time to do a good job; that your boss would be angry with you if you took the time to clean up your code. Perhaps you were just tired of working on this program and wanted it to be over. Or maybe you looked at the backlog of other stuff that you had promised to get done and realized that you needed to slam this module together so that you could move on to the next. We've all done it.

We've all looked at the mess we just made and chosen to leave it for another day. We've all felt the relief of seeing our messy program work and deciding that a working mess is better than nothing. We've all said we'd go back and clean it up later. Of course, in those days we didn't know—or didn't want to know—that later equals never.[4]

[4] LeBlanc's law.

Attitude

Have you ever waded through a mess so grave that it took weeks to do what should have taken hours? Have you seen what should have been a one-line change, made instead in

hundreds of different modules? These symptoms are all too common.

Why does this happen to code? Why does good code rot so quickly into bad code? We have lots of explanations for it. We complain that the requirements changed in ways that thwart the original design. We bemoan the schedules that were too tight to do things right. We blather about stupid managers and intolerant customers and useless marketing types and telephone sanitizers. But the fault, dear Dilbert, is not in our stars, but in ourselves. We are unprofessional.

This may be a bitter pill to swallow. How could this mess be our fault? What about the requirements? What about the schedule? What about the stupid managers and the useless marketing types? Don't they bear some of the blame?

No. The managers and marketers look to *us* for the information they need to make promises and commitments; and even when they don't look to us, we should not be shy about telling them what we think. The users look to us to validate the way the requirements will fit into the system. The project managers look to us to help work out the schedule. We are deeply complicit in the planning of the project and share a great deal of the responsibility for any failures, especially if those failures have to do with bad code!

"But wait!" you say. "If I don't do what my manager says, I'll be fired." Probably not. Most managers want the truth, even when they don't act like it. Most managers want good code, even when they are obsessing about the schedule. They may defend the schedule and requirements with passion, but that's their job. It's your job to defend the code with equal passion.

To drive this point home, what if you were a doctor and had a patient who demanded that you stop all the silly

handwashing in preparation for surgery because it was taking too much time?[5] Clearly the patient is the boss; and yet the doctor should absolutely refuse to comply. Why? Because the doctor knows more than the patient about the risks of disease and infection. It would be unprofessional (never mind criminal) for the doctor to comply with the patient.

[5] When handwashing was first recommended to physicians by Ignaz Semmelweis in 1847, it was rejected on the basis that doctors were too busy and wouldn't have time to wash their hands between patient visits.

So too it is unprofessional for programmers to bend to the will of managers who don't understand the risks of making messes.

The Primal Conundrum

Programmers face a conundrum of basic values. All developers with more than a few years of experience know that messes slow them down. And yet all developers feel the pressure to make messes in order to meet deadlines. In short, they don't take the time to go fast!

True professionals know that the second part of the conundrum is wrong. You will not make the deadline by making the mess. Indeed, the mess will slow you down. It will also slow down everyone else who must deal with it. And that slowness will build and compound and will force you to miss deadline after deadline.

The only way to make the deadlines—the only way to go fast—is to keep the code clean at all times.

The only way to go fast is to go well.

The Art of Clean Code?

Let’s say you believe that messy code is a significant impediment. Let’s say that you accept that the only way to go fast is to keep your code clean. Then, you must ask yourself: “How do I write clean code?” It’s no good trying to write clean code if you don’t know what it means for code to be clean!

Writing clean code requires the disciplined use of myriad little techniques applied through a painstakingly acquired sense of cleanliness that helps us transform bad code into clean code.

And it is always a transformation. Nobody writes clean code. We all write messy code. Only those of us who take the extra step to clean it will produce clean code. This follows Kent Beck’s law, which you will see over and over in this book:

First, make it work. Then, make it right.

What Is Clean Code?

There are probably as many definitions as there are programmers. So I asked some very well-known and deeply experienced programmers what they thought. I have summarized their responses below.



“Clean code does one thing well.” —Bjarne Stroustrup, inventor of C++ and author of *The C++ Programming Language*

This is an old and well-worn maxim. In the ’70s, we used to say that a module should do one thing, do it well, and do it only. But what does “one thing” mean? Back in the late ’80s, I was the author of a 3,000-line C function named `gi`. The name stood for *graphic interpreter*. If someone had come to me back then and warned me that my long, long function did more than one thing, I would have replied, “No, it interprets graphics.”

The problem with the “one thing” rule is that “one thing” is entirely subjective. But I think I know a way to make it *objective*. We’ll talk about that later in this book. And once I describe it to you, I think you’ll be forced to agree that it is close to, if not entirely, objective. You may not like it; but I wager you’ll find no way to substantially disagree.



“Clean code reads like well-written prose.” —Grady Booch, famed author of many software engineering books

Now that’s one hell of an assertion. When is the last time you encountered code that read like “well-written prose”? And yet, as you read on in this book, I’ll show you a way to achieve, or at least approach, that very goal.

Oh, don’t get me wrong. Your code is not going to read like a Hemmingway novel. But it is entirely possible to construct code that reads so nicely that even nonprogrammers can make some sense[6] of it.

[6] By some definition of the words *some* and *sense*.



“Clean code always looks like it was written by someone who cares.” —Michael Feathers, author of *Working Effectively with Legacy Code*

The words that Michael left out of that wonderful quotation were: “about you.”

The reason that a programmer will go to the effort of cleaning their code is that the programmer cares about the next person to come along. They care that their code will not mislead, misdirect, or befuddle those who must maintain that code. The word that best defines cleanliness is *care*.

The implication of that definition is that code that has not been cleaned was written and released carelessly.



“You know you are working on clean code when each routine you read turns out to be pretty much what you expected.” —Ward Cunningham, inventor of Wiki, inventor of Fit, co-inventor of Extreme Programming; motive force behind Design Patterns; Smalltalk and OO thought leader; the godfather of all those who care about code

Imagine reading code and simply agreeing with it. As you scroll down, you nod in acceptance and affirmation. The act of reading the code is as fluid as the laminar flow of air over an airplane wing. There is no turbulence, no clever tricks, no jarring non sequiturs. You simply read, and agree, from top to bottom.

This may seem like an impossible utopian dream; but certainly it is a dream to strive for. In this book, I will show you ways to approach that dream.



“Clean code fits in your head. A function fits on a single screen, involves no more than a handful of moving parts, and has good names. It’s understandable by peers years after it was written. It minimizes surprises, composes well, and is easy to delete.” —Mark Seeman, author of *Code That Fits in Your Head*, popular blogger on software craftsmanship and functional programming

That kind of says it all, doesn’t it? In some ways, that quotation is a summary of this book.

Putting All This Together

Clean code will read like well-written prose because it has been carefully partitioned into functions and modules the way prose is partitioned into sentences and paragraphs, each dealing with one thing, one topic, one irreducible concept that flows from one to the next to the next to create the expected whole.

Why Should We Be Clean?

Look around you. How many computers are running within 10 feet of you? Do you have a smart phone, a smart watch, smart earbuds, a smart car key? Are you wearing a smart medical device?

Look around your kitchen. How many computers are in the appliances? Do you have a smart microwave oven, a smart regular oven, a smart stove, a smart blender, a smart refrigerator?

Is your TV smart? Is your thermostat smart? Do you have a smart security system or a smart sound system?

Do you own a car? How many lines of code are running in that car? Does it have a GPS? Does your phone? Can you even drive to the store without using them?

Think about this. Think hard about it. Nearly everything we do, in this modern world, is mediated by software. You can't cook a hot dog without software. You can't watch TV. You can't make a phone call. You can't drive to the store. You can't buy anything. You can't sell anything. No law can be passed. No law can be enforced. No insurance claim can be filed. Without software being smack-dab in the middle of it all.

As I told you in the introduction, very little happens in our society without the involvement of software.

You and I rule the world.

Others think they rule the world; but then they hand the rules to us and we write the actual rules that run in the machines that govern everything.

Our civilization depends upon us; and that dependence is critical. Without us, civilization would grind to a halt overnight. No one is prepared to go back to using index cards to run everything.

Are we moral, and professional, and responsible enough to bear civilization upon our shoulders? Or are we on a course to let everyone down? Are you and I going to be responsible for a major catastrophe?

We've seen hints of such catastrophes in the recent past. We've seen software drive two jet airliners into the ground at the speed of sound, killing 346 people. We've seen software errors kill dozens, and injure hundreds, when their cars' computers lost their minds and accelerated out of control. We saw Knight Capital lose $450M in 45 minutes due to a software error. We saw Healthcare.gov stumble and nearly fall over when it was first launched.

One day, possibly one day soon, there will be a software error that kills tens of thousands of people in a single incident. Nowadays, it's not very hard to imagine a scenario like that. And when that occurs, we programmers will be blamed.

Don't think it will be your bosses. Don't think it will be the CEOs. Oh, they'll be in hot water too. But the finger of blame will be pointed directly at us—because, after all, it will have been our fingers on the keyboards.

And then imagine the trials. Imagine the testimony of the expert consultants complaining about thousands of global variables, poorly named arguments, mismanaged and missing semaphores, memory leaks, improperly allocated stacks, heap fragmentation, overdependence on performance over quality, 3,000-line functions, modules with duplicated code of which some, but not all, had been repaired.

We've seen that kind of testimony before—too many times.

Imagine the response of the world's politicians as they try to grapple with the loss of tens of thousands of their citizens.

They would, of course, have to pass new laws.

What kinds of laws would they pass? Among other things, they might tell us what languages we have to use, what platforms and frameworks are allowable, what processes we have to follow, what signatures we have to get, what documents we need to write, what review procedures we have to use, what courses we have to take, and what books we have to read.

I promise you that you and I will not like those new laws. I'm also reasonably certain that those laws will do little to improve the situation.

So, the first answer to why we should clean our code is that our civilization needs us to behave as professionals, and that if we fail in that responsibility, our governments will force inconvenient and ineffective rules upon us.

But there is another reason that we need to clean our code. And this one strikes much closer to home.

Productivity

I've asked you before if you have ever been significantly impeded by bad code. If you are a programmer of any number of years, you most certainly have. Once again, my question is: Why did you write it?

Why do we write the code that we know will significantly impede us? The answer always is: to go fast. I'll leave you to deal with the logical inconsistency of that answer.

Now look, I know all the excuses. They all boil down to: They never give us time. This is false. The only way to go fast is to go well. The best way to meet the schedule, the best way to make the deadline, is to do a good job. If you want to go as

fast as you can, don’t do the things that slow you down! Clean your code!

A messy module slows down everyone who has to deal with it, every time they have to deal with it. The cost of the mess repeats over and over, whereas the cost of cleaning is a one-time thing.

A team who allows messes to persist in their code will gradually slow down to a tiny fraction of their potential productivity. I’ve seen teams who have expanded their estimates out to weeks and months for jobs that ought to take a day or two.

The slower the team goes, the more the pressure builds. The more the pressure builds, the more messes are left uncleaned. Each such mess impedes the team even more; and the slower they all go. It’s a vicious cycle, and if you’ve been a programmer for more than a couple of years, you’ve likely taken a few turns around it.

But it gets worse. Managers, desperate to increase productivity, hire new people. This, of course, has the opposite of the desired effect. The new people, thrown into the mass of messy code, see how things are done around here and—of course—they emulate it. So now there are even more people making messes, which drives the cycle even faster, and drags the productivity even lower.

Managers, astounded that the addition of people did not increase productivity, may try to add even more people into the fracas—because *<sarcasm>* the definition of sanity is to try the same thing over and over and expect different results *</sarcasm>*. In the end, however, the managers have no choice but to consult with the developers to find out why everything is going so slowly.

And the developers have an answer. They've known the answer for months, if not years. And when asked, they are more than willing to volunteer that answer: Redesign the whole system from scratch.

Imagine the shock and horror of the managers. The developers have just told them that all their previous work is worthless and has to be scrapped. The developers are also telling them that this new design won't suffer the same fate—that productivity won't decrease again—that everything, this time, will be great.

Of course the managers don't believe that; and they fight strenuously against the idea, but what choice do they really have? The experts are telling them that the only hope is for the system to be redesigned from scratch. In the end, those poor managers will accede to the demands of the developers. Again, what choice to they have?

And when those managers finally agree, a cheer will go up from the developers. "Hallelujah, we're all going back to the beginning where there's no mess to deal with and life will be good!"

But that's not what actually happens. Instead, the ten best, the Tiger Team, are chosen to redesign the system. These are the people who made the mess in the first place. They are going into a room to work together to take the project in a "new and beautiful" direction. They will chart the course to the future.

Meanwhile, the rest of the team is stuck maintaining the old system. After all, that system is part of the business. Bugs have to be fixed. New features have to be added.

Where does the Tiger Team get their requirements from? Is there a requirements document that anyone seriously trusts? Or have there been so many late-night scrambles

and midnight modifications that the old requirements are currently hash? No matter, because the requirements are reliably recorded in one definite place—the old code.

So the Tiger Team is stuck in their room, poring over the old code, trying to figure out what the hell this system actually does. Meanwhile, the rest of the team is changing that code by fixing bugs and adding new features.

And this sets up the race. The Tiger Team has to catch up to where the maintenance team is. But every time the Tiger Team gets to where the maintenance team was, the maintenance team has moved ahead!

This is Xeno's paradox—the race between the great Achilles and a tortoise with a head start. Every time Achilles reaches the point where the tortoise was when he began, the tortoise will have moved ahead from that spot. Xeno used that argument to prove that all motion was impossible.

It turns out that if you know enough calculus, you can prove that Achilles will, in fact, pass the tortoise. But that math doesn't always work for software projects.

I once worked on a project that was being redesigned from scratch. Ten years later, the "New System" had still not been deployed. The customers had been told, years before, to expect a new and glorious system. But that system never came. Every time the company tried to replace the old systems the customers complained that the new system did not do everything the old system did. Try as they might, the Tiger Team could not catch up to the maintenance team. And since the maintenance team was revenue bearing, they got all the resources they needed, whereas the Tiger Team was always limited.

As the months and years went by, the original members of the Tiger Team were eventually replaced and promoted to

work on other projects. After ten years, none of the originals were left.

In the end, the company, in utter desperation, told their customers that they had no choice but to accept the “new” system. The customers railed and complained; but there was simply no other option. The company could not afford to keep funding two development efforts.

Upon hearing that the new system was going to be deployed, the Tiger Team stood up in horror saying: “You can’t ship this, it’s crap! It’s got to be redesigned!”

OK, I’ve embellished this story a bit. Actually, it’s a composite tale from my own experiences and the stories many of my later clients told me. But I know there are many of you, my gentle readers, who identified with it.

So why should we clean our code? Because cleaning our code is the best way for a team to remain productive. If we allow the mess to build, we lay the groundwork for the disaster I just described. But if we keep the code clean, productivity will remain high, the development team will not demand a redesign, and the managers will not desperately try to increase productivity by adding more and more manpower.

Livability[7]

[7] https://blog.cleancoder.com/uncle-bob/2018/08/13/TooClean.html

It would be easy to interpret all of the above comments as aspiring to some kind of Golden Perfection. That is not our goal! Golden Perfection is paralyzing—it affords no room to move. You cannot live within Golden Perfection.

Clean code is code that you can live with. It is not perfect code. It is more like a clean house as opposed to a show

house. A show house is a house you cannot live in, because the act of living degrades the show-ability of the house. A clean house is a house that you can live in without degrading its livability.

Clean code is code that you can maintain, expand, enhance, and evolve without degrading its livability. Clean code is not sparkling and shiny; it looks lived in but cared for. It may not read like Dickens, but it’s readable. It may not have the perfect design and architecture, but it is carefully designed. Concepts may not be perfectly isolated, but they are isolated enough.

Clean code is comfortable in the way that a clean but lived-in house is comfortable. There might be some crumbs on the floor beneath the breakfast counter, there may be a drop of milk on that counter, there may be some dog hair on the sofa, the pillows on the love seat might be in disarray, but none of that degrades the livability of the house—so long as it doesn’t get out of hand.

Clean code is code that has not gotten out of hand. Those who write and maintain it are not seeking perfection, they don’t spend endless hours polishing it, but they do strive to keep the flotsam and detritus of life from accumulating. Or, to invoke Michael Feathers again, they care.

We Read More Than We Write

Have you ever played back an edit session? In the ’80s and ’90s, we had editors like Emacs that kept track of every keystroke. You could work for an hour and then play back your whole edit session like a high-speed movie. When I did this, the results were fascinating. The vast majority of the playback was scrolling and navigating to other modules!

1. Bob enters the module.

2. He scrolls down to the function needing change.
3. He pauses, considering his options.
4. Oh, he’s scrolling up to the top of the module to check the initialization of a variable.
5. Now he scrolls back down and begins to type.
6. Oops, he’s erasing what he typed!
7. He types it again.
8. He erases it again!
9. He types half of something else but then erases that!
10. He scrolls down to another function that calls the function he’s changing to see how it is called.
11. He scrolls back up and types the same code he just erased.
12. He pauses.
13. He erases that code again!
14. He pops up another window and looks at a subclass. Is that function overridden?

    . . .

You get the drift. Indeed, the ratio of time spent reading versus writing is well over 10:1. We are constantly reading old code as part of the effort to write new code; because the new code has to be kept consistent with all the old code.

Because we read code so much more than we write it, we want the reading of code to be the more efficient operation. And since there’s no way to write code without reading it, making it easy to read actually makes it easier to write.

There is no escape from this logic. You cannot write code if you cannot read the surrounding code. The code you are trying to write today will only be easy to write if the surrounding code is easy to read. So, if you want to go fast, if you want to get done quickly, if you want your code to be easy to write, *make it easy to read*.

The Boy Scout Rule

It’s not enough to write the code well. The code has to be kept clean over time. We’ve all seen code rot and degrade as time passes. So we must take an active role in preventing this degradation.

The Boy Scouts of America have a simple rule that we can apply to our profession.

Leave the campground cleaner than you found it.

If we all checked in our code a little cleaner than when we checked it out, the code simply could not rot.

Check the code in cleaner than you checked it out.

The cleanup doesn’t have to be something big. Just do one small random act of kindness. Change one variable name for the better, break up one function that’s a little too large, eliminate one small bit of duplication, clean up one composite `if` statement.

Can you imagine working on a project where everyone on the team practiced this rule? The code would simply get better as time passed. Do you believe that any other option is professional? Indeed, isn’t continuous improvement an intrinsic part of professionalism—of being a moral human?

Part I

Code

In Part I, we explore the code-level practices of cleanliness. In the first two chapters, we begin with basics and first principles. In subsequent chapters, we look at comments, formatting, and names. Then, we get into the meat of the topic: functions, testing, and partitioning. Throughout, we'll be walking through several worked examples, explained in detail.

And although this part is not specifically about high-level design concepts, they are never far away.

While Java predominates in this part, we'll see many other languages, including Python, Go, Ruby, Clojure, JavaScript, and C. What you should take away from that is that the principles of clean code are independent of language and platform.

Chapter 2. Clean That Code!



Clean code! Is that a noun? Or is it a command? Is the word *clean* an adjective or a verb? In this chapter, and in the book overall, we take the latter view. This is a chapter and a book about cleaning code.

One does not simply write clean code. To create code that is clean one must clean it. Code does not pour from your mind and out through your fingers in a clean state. That's not the way our brains work. This is because cleanliness and functionality are competing orthogonal concerns that we, mere humans, are not particularly good at balancing. Instead, our limited minds first focus upon the concern of functionality as we strive to produce code that *works*. It is only when that goal has been achieved that we can change our focus toward cleaning the mess we just made.

Or again, as Kent Beck so succinctly put it:

First, make it work. Then, make it right.

When we do the first part—make it work—we also usually make a mess. This is because making it work takes all of our cognitive energy, and there is none left over to think about anything else.

Once we have something that works, then we can pull back, look it over, and think of ways to improve it.

You've all had the experience of looking over code that you wrote in the past—perhaps only a few days or hours in the past—and then realized that it could be improved by this or that change.

You've also, very likely, had the same experience with just about anything you've ever done. If you've ever written a paper, or a story, or an article, you likely created rough drafts and then passed over them once, or twice, or thrice, to improve them. If you've ever drawn a picture, you've likely spent a lot of time using your eraser. Indeed, I have

reread the last few paragraphs several times and have made small improvements each time.

This is how we humans work. We produce something fairly ugly at first, and then we massage and manipulate it until it is better.

Perhaps an example will help. I wrote a simple[1] converter from Roman numerals to integers. I did this, with the help of GitHub Copilot, by simply throwing code into a function and fumbling around until I thought it worked. I had some unit tests that Copilot and I likewise threw together, but I was frankly not sure they covered all the cases. Here's the code. I expect that it will make your eyes cross, and cause the blood to drain from your face as you stare in horror.

[1] By some definition of that word.

Listing 2-1

```
package fromRoman;


import java.util.Arrays;



public class FromRoman {
  public static int convert(String roman) {
    if (roman.contains("VIV") ||
      roman.contains("IVI") ||
      roman.contains("IXI") ||
      roman.contains("LXL") ||
      roman.contains("XLX") ||
      roman.contains("XCX") ||
      roman.contains("DCD") ||
      roman.contains("CDC") ||
```

```
  roman.contains("MCM")) {
  throw new InvalidRomanNumeralException(roman);
}
roman = roman.replace("IV", "4");
roman = roman.replace("IX", "9");
roman = roman.replace("XL", "F");
roman = roman.replace("XC", "N");
roman = roman.replace("CD", "G");
roman = roman.replace("CM", "O");
if (roman.contains("IIII") ||
  roman.contains("VV") ||
  roman.contains("XXXX") ||
  roman.contains("LL") ||
  roman.contains("CCCC") ||
  roman.contains("DD") ||
  roman.contains("MMMM")) {
  throw new InvalidRomanNumeralException(roman);

}

int[] numbers = new int[roman.length()];
int i = 0;
for (char digit : roman.toCharArray()) {
  switch (digit) {
    case 'I' -> numbers[i] = 1;
    case 'V' -> numbers[i] = 5;
    case 'X' -> numbers[i] = 10;
    case 'L' -> numbers[i] = 50;
    case 'C' -> numbers[i] = 100;
```

```
        case 'D' -> numbers[i] = 500;
        case 'M' -> numbers[i] = 1000;
        case '4' -> numbers[i] = 4;
        case '9' -> numbers[i] = 9;
        case 'F' -> numbers[i] = 40;
        case 'N' -> numbers[i] = 90;
        case 'G' -> numbers[i] = 400;
        case 'O' -> numbers[i] = 900;
        default -> throw new InvalidRomanNumeralException(roman);
      }
      i++;
    }
    int lastDigit = 1000;
    for (int number : numbers) {
      if (number > lastDigit) {
        throw new InvalidRomanNumeralException(roman);
      }

      lastDigit = number;
    }
    return Arrays.stream(numbers).sum();
  }

  public static
  class InvalidRomanNumeralException extends RuntimeException {
    public InvalidRomanNumeralException(String roman) {
    }
  }
}
```

> ***Future Bob:*** *I'm reading this code two months later, and I'm laughing hysterically at just how awful some of it is (like* `49FNGO`*). . . . And yet how insightful parts of it are.*

Here are the tests I used. I had my doubts about whether they were complete.

```
package fromRoman;


import fromRoman.FromRoman.InvalidRomanNumeralException;
import org.junit.jupiter.api.Test;


import static fromRoman.FromRoman.convert;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.is;
import static org.junit.jupiter.api.Assertions.assertThrows;


public class FromRomanTest {
  @Test
  public void valid() throws Exception {
    assertThat(convert(""), is(0));
    assertThat(convert("I"), is(1));
    assertThat(convert("II"), is(2));
    assertThat(convert("III"), is(3));
    assertThat(convert("IV"), is(4));
    assertThat(convert("V"), is(5));
    assertThat(convert("VI"), is(6));
    assertThat(convert("VII"), is(7));
```

```
  assertThat(convert("VIII"), is(8));
  assertThat(convert("IX"), is(9));
  assertThat(convert("X"), is(10));
  assertThat(convert("XI"), is(11));
  assertThat(convert("XII"), is(12));
  assertThat(convert("CXC"), is(190));
}

@Test
public void invalid() throws Exception {
  assertInvalid("IIII");
  assertInvalid("VV");
  assertInvalid("XXXX");
  assertInvalid("LL");
  assertInvalid("CCCC");
  assertInvalid("DD");
  assertInvalid("MMMM");
  assertInvalid("IIV");
  assertInvalid("IVI");
  assertInvalid("IXI");
  assertInvalid("VIV");
  assertInvalid("XVX");
  assertInvalid("XIIII");
  assertInvalid("XVV");
  assertInvalid("XIVI");
  assertInvalid("XIXI");
  assertInvalid("XVIV");
  assertInvalid("LXL");
```

```
    assertInvalid("XLX");
    assertInvalid("XCX");
    assertInvalid("CDC");
    assertInvalid("DCD");
    assertInvalid("MCMC");
    assertInvalid("MCDM");
  }

  private void assertInvalid(String r) {
    assertThrows(InvalidRomanNumeralException.class, () -> convert
  }
}
```

I hope you agree with me that this needs some cleanup. It feels ragged and jumbled. Later in this chapter, and in the chapters to follow, I'll walk through the cleaning procedure. But for now, I just want to show you the result.

Before I present the result, I want to say that the time it took to clean the code was roughly equal to the time it took to write the code in the first place. Copilot was moderately helpful during the initial writing but significantly less helpful during the cleaning process.

During the cleaning process, I found one or two small bugs simply through the effort of organizing my thoughts and the structure of the code and tests. It's remarkable what a second look can find[2]—and cleaning is the ultimate second look.

[2] After I conduct the close-up preflight inspection on my airplane, I stand back, walk around the plane, and take a second look from a farther perspective. It's remarkable what I've found with that approach. There are things that are obvious from a distance that are nearly invisible during closer inspection.

```
package fromRoman;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

public class FromRoman {
  private String roman;
  private List<Integer> numbers = new ArrayList<>();
  private int charIx;
  private char nextChar;
  private Integer nextValue;
  private Integer value;
  private int nchars;
  Map<Character, Integer> values = Map.of(
    'I', 1,
    'V', 5,
    'X', 10,
    'L', 50,
    'C', 100,
    'D', 500,
    'M', 1000);

  public FromRoman(String roman) {
    this.roman = roman;
  }

  public static int convert(String roman) {
```

```
  return new FromRoman(roman).doConversion();
}

private int doConversion() {
  checkInitialSyntax();
  convertLettersToNumbers();
  checkNumbersInDecreasingOrder();
  return numbers.stream().reduce(0, Integer::sum);
}

private void checkInitialSyntax() {
  checkForIllegalPrefixCombinations();
  checkForImproperRepetitions();
}

private void checkForIllegalPrefixCombinations() {
  checkForIllegalPatterns(
    new String[]{"VIV", "IVI", "IXI", "IXV", "LXL", "XLX",
                 "XCX", "XCL", "DCD", "CDC", "CMC", "CMD"});
}

private void checkForImproperRepetitions() {
  checkForIllegalPatterns(
    new String[]{"IIII", "VV", "XXXX", "LL", "CCCC", "DD", "MMMM
}

private void checkForIllegalPatterns(String[] patterns) {
  for (String badString : patterns)
```

```
    if (roman.contains(badString))
      throw new InvalidRomanNumeralException(roman);
}

private void convertLettersToNumbers() {
  char[] chars = roman.toCharArray();
  nchars = chars.length;
  for (charIx = 0; charIx < nchars; charIx++) {
    nextChar = isLastChar() ? 0 : chars[charIx + 1];
    nextValue = values.get(nextChar);
    char thisChar = chars[charIx];
    value = values.get(thisChar);
    switch (thisChar) {
      case 'I' -> addValueConsideringPrefix('V', 'X');
      case 'X' -> addValueConsideringPrefix('L', 'C');
      case 'C' -> addValueConsideringPrefix('D', 'M');
      case 'V', 'L', 'D', 'M' -> numbers.add(value);
      default -> throw new InvalidRomanNumeralException(roman);
    }
  }
}

private boolean isLastChar() {
  return charIx + 1 == nchars;
}

private void addValueConsideringPrefix(char p1, char p2) {
  if (nextChar == p1 || nextChar == p2) {
```

```
      numbers.add(nextValue - value);
      charIx++;
    } else numbers.add(value);
  }

  private void checkNumbersInDecreasingOrder() {
    for (int i = 0; i < numbers.size() - 1; i++)
      if (numbers.get(i) < numbers.get(i + 1))
        throw new InvalidRomanNumeralException(roman);
  }

  public static
  class InvalidRomanNumeralException extends RuntimeException {
    public InvalidRomanNumeralException(String roman) {
      super("Invalid Roman numeral: " + roman);
    }
  }
}
```

I think this is much cleaner. What makes it cleaner, in my opinion, are the short, well-named functions, the well-named instance variables, and the fact that the functions are listed in the order that they are called. You can read the code from the top to the bottom and it reads rather like a story.

> Future Bob*:* Two months later I'm torn. The first version, ugly as it was, was not as chopped up as this one. It's true that the names and the ordering of the extracted functions read like a story and are

a big help in understanding the intent; but there were several times that I had to scroll back up to the top to assure myself about the types of instance variables. I found the choppiness, and the scrolling, to be annoying. However, and this is critical, I am reading this cleaned code after having first read the ugly version and having gone through the work of understanding it. So now, as I read this version, I am annoyed because I already understand it and find the chopped-up functions and the instance variables redundant.

Don't get me wrong, I still think the cleaner version is better. I just wasn't expecting the annoyance. When I first cleaned it, I thought it was going to be annoyance free.

I suppose the question you should ask yourself is which of these two pieces of code you would rather have read first. Which tells you more about the intent? Which obscures the intent? Certainly the latter is better in that regard.

This annoyance is an issue that John Ousterhout[3] and I have debated.[4] When you understand an algorithm, the artifacts intended to help you understand it become annoying. Worse, if you understand an algorithm, the names or comments you write to help others will be biased by that understanding and may not help the reader as much as you think they will. A good example of that, in this code, is the `addValueConsideringPrefix` function. That name made perfect sense to me when I understood the

algorithm. But it was a bit jarring two months later. Perhaps not as jarring as `49FNG0`, but still not quite as obvious as I had hoped when I wrote it. It might have been better written as `numbers.add(decrementValueIfThisCharIsaPrefix);`, since that would be symmetrical with the `numbers.add(value);` in the nonprefixed case.

[3] [APOD].

[4] See the appendix to read that debate.

> The bottom line is that your own understanding of what you are cleaning will work against your ability to communicate with the next person to come along. And this will be true whether you are extracting well-named methods, or adding descriptive comments. Therefore, take special care when choosing names and writing comments; and don't be surprised if others are annoyed by your choices. Lastly, a look after a few months can be both humbling and profitable.

The `convertLettersToNumbers` function is a bit longer than I like; but it hangs together pretty well, and I think that decomposing it into smaller functions would be more obscuring than helpful.

Some of you may be looking at this "cleaner" code with a more skeptical eye. Perhaps you have some questions or complaints.

You might complain that it's longer. That's true; but most of that extra length is due to the nicely named functions that explain what's going on.

You might complain that the creation of an object is overkill and expensive. It's true that allocating memory, and then

garbage-collecting it later, takes extra time. There is also time spent in the calling of all those small methods. If you are writing embedded real-time code with very tight timing constraints, or if you are writing code for a high-performance flight simulator or first-person shooter game, then this decision could seem insane, not to mention unclean. But for my purposes, the extra time is the least of my concerns. I don't need the speed. For my situation, I think trading speed for clarity, within reason, is a good trade to make.

You might be a functional programmer horrified that the functions are not "pure." But, in fact, the static `convert` function is as pure as a function can be. The others are just little helpers that operate within a single invocation of that overarching pure function. Those instance variables are very convenient for allowing the individual methods to communicate without having to resort to passing arguments. This shows that one good use for an object is to allow the helper functions that operate within the execution of a pure function to easily communicate through the instance variables of the object.

So I think this is pretty clean. Am I right? I think I am; but you may disagree. That's OK. We are allowed to disagree. There is no single standard for what cleanliness is. This is mine, and mine may not be yours. All I'll ask of you is that you consider mine and develop your own standard. The most important thing is to have a standard that you apply with discipline.

If you are a member of a team, then the team should have a single standard[5] of cleanliness that is negotiated, and then followed and supported, by all members of that team.

[5] This might tempt you to create a standards document. I recommend otherwise. The document that represents the standard should be the code itself. The only reason to have a standards document outside of the code is if the code does not currently represent the standard.

By the way, the tests I initially wrote *were* inadequate. As I worked through the cleaning process, I found a number of situations that needed further testing. For the sake of completeness, I've included the final tests below.

```
package fromRoman;


import fromRoman.FromRoman.InvalidRomanNumeralException;
import org.junit.jupiter.api.Test;


import static fromRoman.FromRoman.convert;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.is;
import static org.junit.jupiter.api.Assertions.assertThrows;


public class FromRomanTest {
  @Test
  public void valid() throws Exception {
    assertThat(convert(""), is(0));
    assertThat(convert("I"), is(1));
    assertThat(convert("II"), is(2));
    assertThat(convert("III"), is(3));
    assertThat(convert("IV"), is(4));
    assertThat(convert("V"), is(5));
    assertThat(convert("VI"), is(6));
    assertThat(convert("VII"), is(7));
    assertThat(convert("VIII"), is(8));
    assertThat(convert("IX"), is(9));
    assertThat(convert("X"), is(10));
    assertThat(convert("XI"), is(11));
```

```
  assertThat(convert("XII"), is(12));
  assertThat(convert("XIII"), is(13));
  assertThat(convert("XIV"), is(14));
  assertThat(convert("XV"), is(15));
  assertThat(convert("XVI"), is(16));
  assertThat(convert("XIX"), is(19));
  assertThat(convert("XX"), is(20));
  assertThat(convert("XXX"), is(30));
  assertThat(convert("XL"), is(40));
  assertThat(convert("L"), is(50));
  assertThat(convert("LX"), is(60));
  assertThat(convert("LXXIV"), is(74));
  assertThat(convert("XC"), is(90));
  assertThat(convert("C"), is(100));
  assertThat(convert("CXIV"), is(114));
  assertThat(convert("CXC"), is(190));
  assertThat(convert("CD"), is(400));
  assertThat(convert("D"), is(500));
  assertThat(convert("CDXLIV"), is(444));
  assertThat(convert("DCXCIV"), is(694));
  assertThat(convert("CM"), is(900));
  assertThat(convert("M"), is(1000));
  assertThat(convert("MCM"), is(1900));
  assertThat(convert("MCMXCIX"), is(1999));
  assertThat(convert("MMXXIV"), is(2024));
}

@Test
```

```
public void invalid() throws Exception {
  assertInvalid("IIII");
  assertInvalid("VV");
  assertInvalid("XXXX");
  assertInvalid("LL");
  assertInvalid("CCCC");
  assertInvalid("DD");
  assertInvalid("MMMM");
  assertInvalid("XIIII");
  assertInvalid("LXXXX");
  assertInvalid("DCCCC");
  assertInvalid("VIIII");
  assertInvalid("MCCCC");
  assertInvalid("VX");
  assertInvalid("IIV");
  assertInvalid("IVI");
  assertInvalid("IXI");
  assertInvalid("IXV");
  assertInvalid("VIV");
  assertInvalid("XVX");
  assertInvalid("XVV");
  assertInvalid("XIVI");
  assertInvalid("XIXI");
  assertInvalid("XVIV");
  assertInvalid("LXL");
  assertInvalid("XLX");
  assertInvalid("XCX");
  assertInvalid("XCL");
```

```
    assertInvalid("CDC");
    assertInvalid("DCD");
    assertInvalid("CMC");
    assertInvalid("CMD");
    assertInvalid("MCMC");
    assertInvalid("MCDM");
  }

  private void assertInvalid(String r) {
    assertThrows(InvalidRomanNumeralException.class,
                 () -> convert(r));
  }
}
```

The sheer number of tests is a bit daunting. There is probably some redundancy in the `valid` tests. I'm reasonably sure that there is no redundancy in the `invalid` tests.

> Future Bob: I am slightly annoyed that I did not find a more abstract method of determining and testing invalid conditions. I'm sure one exists.

The Cleaning Process

The process I used to first write and then clean that code was relatively simple. I'm going to walk you through each "major" decision; but overall it was just a matter of making smallish improvements while keeping all the tests passing.

The ugly code in Listing 2-1 was not my first attempt. Instead, I fumbled around with a few bad ideas.[6] One was based on a state machine strategy that I cooked up in the

shower. I actually wrote the following code and then threw it away before it got too far.

[6] Linus Pauling was once asked how he had so many good ideas. His reply was something to the effect of he had lots of ideas and threw away the bad ones.

```
package fromRoman;


public class FromRoman {
  enum state {
    I, V, X, L, C, D, M
  }


  public static int convert(String roman) {
    int currentSum = 0;
    for (char c : roman.toCharArray()) {
      switch (c) {
        case 'I' -> {
          if (currentSum == 3) {
            throw new InvalidRomanNumeralException(roman);
          }
          currentSum += 1;
        }
        case 'V' -> currentSum += 5;
        case 'X' -> currentSum += 10;
        case 'L' -> currentSum += 50;
        case 'C' -> currentSum += 100;
        case 'D' -> currentSum += 500;
        case 'M' -> currentSum += 1000;
      }
```

```
    }
    return currentSum;
  }

  public static
  class InvalidRomanNumeralException extends RuntimeException {
    public InvalidRomanNumeralException(String roman) {
    }
  }
}
```

I finally settled on the strategy in Listing 2-1, and then I massaged it into working, with all the tests passing.

To clean this I looked for some low-hanging fruit. The first I found was the ugly code that checked for illegal prefix combinations. So I extracted that code out into the `checkForIllegalPrefixCombinations` function. Then, I noticed that there was a similar stretch of code, further down, that checked for improper repetitions of letters. So I extracted that code out as the `checkForImproperRepetitions` functions.

In a similar fashion, I extracted out the replacement of the digraphs, the conversions of the letters to numbers, and the check for the numbers being in descending order.

This left me with the following code.

```
package fromRoman;


import java.util.Arrays;


public class FromRoman {
```

```
public static int convert(String roman) {
  checkForIllegalPrefixCombinations(roman);
  checkForImproperRepetitions(roman);
  roman = replaceDigraphs(roman);
  int[] numbers = convertLettersToNumbers(roman);
  checkNumbersAreInOrder(numbers, roman);
  return Arrays.stream(numbers).sum();
}

private static
void checkNumbersAreInOrder(int[] numbers, String roman) {
  int lastDigit = 1000;
  for (int number : numbers) {
    if (number > lastDigit) {
      throw new InvalidRomanNumeralException(roman);
    }
    lastDigit = number;
  }
}

private static int[] convertLettersToNumbers(String roman) {
  int[] numbers = new int[roman.length()];
  int i = 0;
  for (char digit : roman.toCharArray()) {
    switch (digit) {
      case 'I' -> numbers[i] = 1;
      case 'V' -> numbers[i] = 5;
      case 'X' -> numbers[i] = 10;
```

```
      case 'L' -> numbers[i] = 50;
      case 'C' -> numbers[i] = 100;
      case 'D' -> numbers[i] = 500;
      case 'M' -> numbers[i] = 1000;
      case '4' -> numbers[i] = 4;
      case '9' -> numbers[i] = 9;
      case 'F' -> numbers[i] = 40;
      case 'N' -> numbers[i] = 90;
      case 'G' -> numbers[i] = 400;
      case 'O' -> numbers[i] = 900;
      default -> throw new InvalidRomanNumeralException(roman);
    }
    i++;
  }
  return numbers;
}

private static String replaceDigraphs(String roman) {
  roman = roman.replace("IV", "4");
  roman = roman.replace("IX", "9");
  roman = roman.replace("XL", "F");
  roman = roman.replace("XC", "N");
  roman = roman.replace("CD", "G");
  roman = roman.replace("CM", "O");
  return roman;
}

private static void checkForImproperRepetitions(String roman) {
```

```
  if (roman.contains("IIII") ||
    roman.contains("VV") ||
    roman.contains("XXXX") ||
    roman.contains("LL") ||
    roman.contains("CCCC") ||
    roman.contains("DD") ||
    roman.contains("MMMM")) {
    throw new InvalidRomanNumeralException(roman);
  }
}

private static void checkForIllegalPrefixCombinations(String rom
  if (roman.contains("VIV") ||
    roman.contains("IVI") ||
    roman.contains("IXI") ||
    roman.contains("LXL") ||
    roman.contains("XLX") ||
    roman.contains("XCX") ||
    roman.contains("DCD") ||
    roman.contains("CDC") ||
    roman.contains("MCM")) {
    throw new InvalidRomanNumeralException(roman);
  }
}

public static
class InvalidRomanNumeralException extends RuntimeException {
  public InvalidRomanNumeralException(String roman) {
```

```
    }
  }
}
```

I liked the way this made the `convert` function read. However, I didn't like passing all those arguments around to the helper functions. So I created the internal object with instance variables so that those helper functions could communicate without the overhead of passing arguments.

I really didn't like the ad hoc "49FNGO" character replacement for the digraphs. That was just too clever.[7] So I decided to merge the digraph processing and the conversions of letters to numbers into the same function. I did this one digraph at a time, keeping all the tests passing. This left things as follows.

[7] "Rule of thumb: if you think something is clever and sophisticated, beware—it is probably self-indulgence." [DOET].

```
package fromRoman;


import java.util.ArrayList;
import java.util.List;
import java.util.Map;


public class FromRoman {
  private String roman;
  List<Integer> numbers = new ArrayList<>();
  Map<Character, Integer> values = Map.of(
    'I', 1,
    'V', 5,
```

```
  'X', 10,
  'L', 50,
  'C', 100,
  'D', 500,
  'M', 1000);

public FromRoman(String roman) {
  this.roman = roman;
}

public static int convert(String roman) {
  return new FromRoman(roman).doConversion();
}

private int doConversion() {
  checkForIllegalPrefixCombinations();
  checkForImproperRepetitions();
  convertToNumbers();
  checkNumbersAreInOrder();
  return numbers.stream().mapToInt(Integer::intValue).sum();
}

private void checkForIllegalPrefixCombinations() {
  if (roman.contains("VIV") ||
    roman.contains("IVI") ||
    roman.contains("IXI") ||
    roman.contains("LXL") ||
    roman.contains("XLX") ||
```

```
    roman.contains("XCX") ||
    roman.contains("DCD") ||
    roman.contains("CDC") ||
    roman.contains("MCM")) {
    throw new InvalidRomanNumeralException(roman);
  }
}

private void checkForImproperRepetitions() {
  for (String badRep : new String[]

                       {"IIII", "VV", "XXXX", "LL",
                        "CCCC", "DD", "MMMM"}) {
    if (roman.contains(badRep)) {
      throw new InvalidRomanNumeralException(roman);
    }
  }
}

private void convertToNumbers() {
  char[] chars = roman.toCharArray();
  int l = chars.length;
  for (int i = 0; i < l; i++) {
    char nextChar = i + 1 < l ? chars[i + 1] : 0;
    int nextValue = values.get(nextChar);
    switch (chars[i]) {
      case 'I' -> {
        if (nextChar == 'V' || nextChar == 'X') {
```

```
            numbers.add(nextValue - 1);
            i++;
          } else numbers.add(1);
        }
        case 'V' -> numbers.add(5);
        case 'X' -> {
          if (nextChar == 'L' || nextChar == 'C') {
            numbers.add(nextValue - 10);
            i++;
          } else numbers.add(10);
        }
        case 'L' -> numbers.add(50);
        case 'C' -> {
          if (nextChar == 'D' || nextChar == 'M') {
            numbers.add(nextValue - 100);
            i++;
          } else numbers.add(100);
        }
        case 'D' -> numbers.add(500);
        case 'M' -> numbers.add(100);
        default -> throw new InvalidRomanNumeralException(roman);
      }
    }
  }

  private void checkNumbersAreInOrder() {
    int lastDigit = 1000;
    for (int number : numbers) {
```

```
      if (number > lastDigit) {
        throw new InvalidRomanNumeralException(roman);
      }
      lastDigit = number;
    }
  }

  public static
  class InvalidRomanNumeralException extends RuntimeException {
    public InvalidRomanNumeralException(String roman) {
      super("Invalid Roman numeral: " + roman);
    }
  }
}
```

After that, it was just a matter of reducing all the duplication in the cases and removing as many of the magic numbers as possible.

Then, I turned to the tests. They were pretty ad hoc. So I looked for patterns in the test data and reorganized them. As part of this process, I found some missing invalid combinations and added them.

That's pretty much the whole story. I just made one simple change after the next, while keeping all the tests passing. This is pretty much the procedure that Martin Fowler recommends in his wonderful book, *Refactoring*.[8]

[8] [Refactoring].

Conclusion

First, I made it work. Then, I made it right—or at least better.

The following chapters in this part of the book are going to expand upon the iterative and incremental techniques I just described. We'll see many other examples, in several other languages. We'll look at the details of low-level coding; details like good naming, comments, and formatting. We'll study higher-level techniques like TDD, Collaborative Programming, and Extract Till You Drop. Finally, we'll look at slightly higher-level concepts such as module structure, objects, data structures, classes, and error handling.

Postscript: Future Bob Playing with Grok3

All the reviews have come back, and I am now preparing the final manuscript for publication. It is February of 2025, and Grok3 just came out. So I fed my "clean" version of the above code into Grok3 and asked it to make improvements. After a bit of thought, it returned the following code, which compiled and which passed all my tests.

```
package fromRoman;


import java.util.Map;


public class FromRoman {
    private String roman;
    private static final String REGEX =
      "^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$";
    private static final Map<Character, Integer> VALUES = Map.of(
        'I', 1,
        'V', 5,
```

```
    'X', 10,
    'L', 50,
    'C', 100,
    'D', 500,
    'M', 1000
);

/** Constructs a FromRoman instance,
    validating the input Roman numeral. */
public FromRoman(String roman) {
    if (!roman.matches(REGEX)) {
        throw new InvalidRomanNumeralException(roman);
    }
    this.roman = roman;
}

/** Converts the Roman numeral to an integer. */
public int toInt() {
    int sum = 0;
    for (int i = 0; i < roman.length(); i++) {
        int currentValue = VALUES.get(roman.charAt(i));
        if (i < roman.length() - 1) {
            int nextValue = VALUES.get(roman.charAt(i + 1));
            if (currentValue < nextValue) {
                sum -= currentValue;
            } else {
                sum += currentValue;
            }
```

```
            } else {
                sum += currentValue;
            }
        }
        return sum;
    }

    /** Static method to convert a Roman numeral string to an inte
    public static int convert(String roman) {
        return new FromRoman(roman).toInt();
    }

    public static
    class InvalidRomanNumeralException extends RuntimeException {
        public InvalidRomanNumeralException(String roman) {
            super("Invalid Roman numeral: " + roman);
        }
    }
}
```

I’m quite pleased with the improvement to the algorithm. It really reduces things to the bare essence. I’m not a big fan of using regular expressions, but this one is not too hard to understand. I don’t find any of the comments helpful, so I’d likely delete them.

Then, I asked it to improve my tests. Here’s what it came up with:

```
package fromRoman;
```

```
import org.junit.jupiter.api.Test;

import static fromRoman.FromRoman.convert;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.is;

import static org.junit.jupiter.api.Assertions.assertThrows;

public class FromRomanTest {
    @Test
    public void valid() {
        // Edge case
        assertThat(convert(""), is(0));
        // Single characters
        assertThat(convert("I"), is(1));
        assertThat(convert("V"), is(5));
        assertThat(convert("X"), is(10));
        assertThat(convert("L"), is(50));
        assertThat(convert("C"), is(100));
        assertThat(convert("D"), is(500));
        assertThat(convert("M"), is(1000));
        // Subtractive notation
        assertThat(convert("IV"), is(4));
        assertThat(convert("IX"), is(9));
        assertThat(convert("XL"), is(40));
        assertThat(convert("XC"), is(90));
        assertThat(convert("CD"), is(400));
        assertThat(convert("CM"), is(900));
        // Complex cases
```

```
        assertThat(convert("XIV"), is(14));
        assertThat(convert("CDXLIV"), is(444));
        assertThat(convert("MCMXCIX"), is(1999));
    }


    @Test
    public void invalid() {
        // Too many repeats
        assertInvalid("IIII");
        assertInvalid("VV");
        assertInvalid("MMMM");
        // Invalid subtractive notation
        assertInvalid("VX");  // V can't precede X
        assertInvalid("IIV"); // Incorrect order
        assertInvalid("IXI"); // Invalid sequence
        // Improper structure
        assertInvalid("CMC"); // Multiple subtractive errors
        assertInvalid("XCL"); // Incorrect order
    }

    private void assertInvalid(String r) {
        assertThrows(FromRoman.InvalidRomanNumeralException.class,
                     () -> convert(r));
    }
}
```

I like that it organized them so nicely. I remember struggling to find that organization and eventually just deciding to flood the zone. So, kudos to Grok3 for that. On the other

hand, the tests are strongly relying on the regular expression and so does, what I consider to be, an inadequate job of testing invalid inputs. For example, it does not check that `"IM"` is invalid. So I'd likely add a few more tests for things like that.

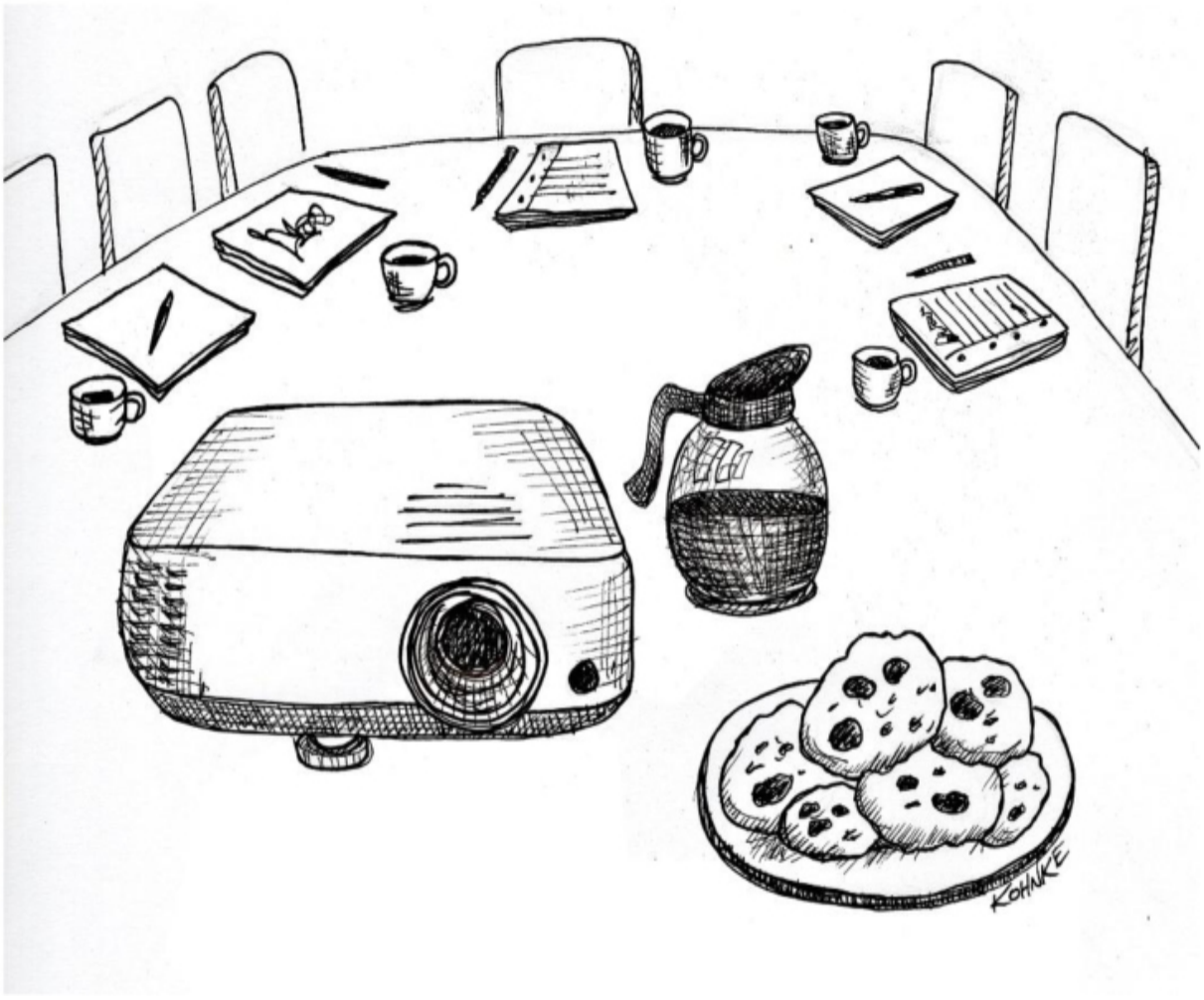
Postscript Conclusion

Grok3 did a nice job. One might therefore question whether my effort of cleaning and testing was worth it. On the other hand, I, and not Grok3, am the final arbiter of the code that gets produced. The buck stops with me. I'm pleased that Grok3 was able to help, and I will be happy of that help when I ask for it. But in order to know that Grok3, or any LLM/AI, is giving me good code, it remains incumbent upon me to do the work to ensure that I can evaluate what it produces. Doing that work will usually involve writing and cleaning the code that I ask it to improve so that I can properly judge the outcome.

The trade-off is very similar to using an autopilot in an airplane. The autopilot can vastly reduce the workload. But under no circumstances should it be implicitly trusted. Rather, the pilot must continuously oversee its operation. By the same token, no pilot should ever fly without possessing the necessary flying skills that they continuously maintain and refine.

So it is with us. We must stay in practice, and keep our skills high, because we must be able to competently oversee the automation that reduces our workload.

Chapter 3. First Principles



If you read this chapter and grasp all the principles within, then you'll be 90% of the way toward cleaning code. The ideas presented here are old and well established. They

have stood the test of time and should be part of every programmer's regimen of discipline.

Keep in mind that these are not strict laws. They are not rules that must be followed. They are (at the risk of using a trite expression) "more what you'd call guidelines."[1] Each should be applied in the context of the problem at hand, and every programmer should feel free to take whatever license is necessary to solve that problem.

[1] Hector Barbossa, *Pirates of the Caribbean*.

But, when all else is equal, and all the variables are averaged out, these are the principles that I believe emanate from well-cleaned code.

If, as you read what follows, you encounter concepts that are foreign or novel, be assured that all will be explained in subsequent chapters. Indeed, when you have finished reading this book, you may wish to complete the circle by rereading this chapter.

Everything Small, Well Named, Organized, and Ordered

If you stop now and read nothing more than that heading, you'll have much of the essence of what it takes to clean code.

Keep everything small. Choose names for those small elements that communicate to other programmers. Define and maintain a structure and organization that present the software in such a way that others can easily consume it. Keep the elements within that organization well-ordered so that one concept follows another in a rational and sensible way.

Remember that your own understanding of the code will likely bias your attempts at providing good names and structures; so work hard to put yourself in the headspace of those who do not understand the code.

Functions

Functions should be small. Most should be just a handful of lines. This allows them to have nice, descriptive names and promotes the separation of concerns. It ensures that each function does one, and only one, thing.

Let's start with the name. If you see a snippet of lines that does something that can be reasonably named, then that snippet should be moved into a function named for what that snippet does. The name should be a verb that says what the function does and that allows the calling sequence to read "like well-written prose."[2]

[2] Grady Booch, private email.

As an example, consider this function, adapted from the `JCommon` library. It finds the date of a specified weekday before a given date. The weekday codes are integers 1 (Sunday) through 7 (Saturday).

```
public static Date getPreviousDayOfWeek(int weekday, Date from) {
  // check arguments…
  if (!isValidWeekdayCode(weekday)) {
    throw new IllegalArgumentException(
      "Invalid day-of-the-week code."
    );
  }

  // find the date…
```

```
  final int adjust;
  final int baseDOW = from.getDayOfWeek();
  if (baseDOW > weekday) {
    adjust = Math.min(0, weekday - baseDOW);
  } else {
    adjust = -7 + Math.max(0, weekday - baseDOW);
  }

  return Date.addDays(adjust, from);
}
```

There are at least two snippets of code that we could extract into smaller and better-named functions. And that extraction will lead to some simplifications.

```
public static Date getPreviousDayOfWeek(int weekday, Date from) {
  checkWeekdayArgument(weekday);
  return addDays(-daysBefore(weekday, from), from);
}

private static void checkWeekdayArgument(int weekday) {
  if (!isValidWeekdayCode(weekday))
    throw new IllegalArgumentException("Invalid day-of-the-week co
}

private static int daysBefore(int targetWeekday, Date from) {
  int fromWeekday = from.getDayOfWeek();
  int diff = fromWeekday - targetWeekday;
  return diff + (diff > 0 ? 0 : 7);
}
```

Notice how the simplified `getPreviousDayOfWeek` function reads. It's short, and it's obvious. If you need to know more, all you have to do is look down to see the extracted functions. However, most often you would not need to know more. You'd simply look at that function, agree with it, and move on. It may not read like the prose of a Crichton novel, but it reads pretty well.

> **Future Bob:** I don't like the leading minus sign. It would have been better to create a `subtractDays` function in order to get rid of it. That's a small thing, but small things matter.

A More Significant Example

Here I present the Uncle Bob Conference Room system. Scan through it. It's not very complicated—probably.

```
——Statement.java——

package ubConferenceCenter;


import java.util.ArrayList;

import java.util.List;


public class Statement {


  public enum CatalogItem {SMALL_ROOM, LARGE_ROOM,

                           PROJECTOR, COFFEE, COOKIES}


  public record RentalItem(CatalogItem type,

                           int days,
```

```
                          int unitPrice,
                          int price,
                          int tax) {
}

public record Totals(int subtotal, int tax) {
}

private String customerName;
private int subtotal = 0;
private int tax = 0;
private List<RentalItem> items = new ArrayList<>();

public Statement(String customerName) {
  this.customerName = customerName;
}

public void rent(CatalogItem item, int days) {
  int unitPrice = switch (item) {
    case SMALL_ROOM -> 100;
    case LARGE_ROOM -> 150;
    case PROJECTOR -> 50;
    case COFFEE -> 10;
    case COOKIES -> 15;
  };

  boolean eligibleForDiscount = switch (item) {
    case SMALL_ROOM, LARGE_ROOM -> days == 5;
```

```
    case PROJECTOR, COFFEE, COOKIES-> false;
  };

  int price = unitPrice * days;

  if (eligibleForDiscount) price = (int) Math.round(price * .9);

  subtotal += price;
  int thisTax = switch (item) {
    case SMALL_ROOM, LARGE_ROOM, PROJECTOR ->
         (int) Math.round(price * .05);
    case COFFEE,COOKIES -> 0;
  };
  tax += thisTax;
  items.add(new RentalItem(item, days, unitPrice, price, thisTax
}

public RentalItem[] getItems() {
  List<RentalItem> items = new ArrayList<>(this.items);
  boolean largeRoomFiveDays = items.stream().anyMatch(
    item -> item.type() == CatalogItem.LARGE_ROOM && item.days()
  boolean coffeeFiveDays = items.stream().anyMatch(
    item -> item.type() == CatalogItem.COFFEE && item.days() ==
  if (largeRoomFiveDays && coffeeFiveDays)
    items.add(new RentalItem(CatalogItem.COOKIES, 5, 0, 0, 0));
  return items.toArray(new RentalItem[0]);
}
```

```
  public String getCustomerName() {
    return customerName;
  }

  public Totals getTotals() {
    return new Totals(subtotal, tax);
  }
}
```

I’m sure you figured it out. Users can rent small rooms, large rooms, coffee, projectors, and cookies. (How do you rent cookies?) Small rooms rent for $100 per day. Large rooms are $150. You get a 10% discount if you rent a room for the whole week. There’s a 5% (rounded up) tax on all nonfood items. Coffee is $10 per day. Cookies are $15 per day. Projectors are $50 per day. And if you rent a large room and coffee for a whole week, you get one day of free cookies.

The `Statement` class represents a sales statement that includes each item rented, the price of the item, the number of days rented, the subtotal, and the tax. The `Statement` also contains the subtotals of all the items and all the tax.

Easy peasy.

This is not too hard to understand, but you can tell that it was slapped together. It’s a bit unkempt, but not terrible. We could probably clean it up a bit, but would that be worth the effort?

But let me ask you. Have you ever seen a system that started out simple and then got much, much more complicated? If you’ve been in this business for more than a year or so, I *know* you have. Indeed, if you’ve been in the business for five years or more, you’ve likely seen a system

that began a bit unkempt but grew into a horrible mess, becoming a living hell for the developers and the organization.

So let’s make this example a bit more real. You don’t suppose that this program started out in its current form, do you? No, at first it just rented one small room. Then, many small rooms. Then, large rooms with a different unit price. Then, coffee, with no sales tax. Then, the discount for a full week was added. And then cookies, and the bonus for renting a large room for a week.

In other words, this program grew.

Let’s suppose that we’ve just gotten back from an all-hands meeting at which the CEO gave us a glowing report about how the company is expanding. There are new markets, new states, new laws to contend with. There will be new discounts and new promotions, and new tax regulations. It’s going to be great for the business; but you look at this little `rent` function and you think . . . it’s going to continue to grow, isn’t it? And as it grows, it will become ever more tangled and confusing. It will degrade, like a piece of rotting meat.

So let’s stop that from happening. Let’s figure out a way to get ahead of the coming business growth and let this function grow without rotting from the inside out. Let’s tidy it up first.[3]

[3] [Tidy].

Look back at the `rent` function and ask yourself what you don’t like about it. Personally, I think it’s a bit large and disorganized. It does several things that I can pull apart into functions that do one thing.

So first, let’s just use the extract method refactoring to create functions that do one thing, and gather together the

things that are related and separate the things that are different.[4]

[4] The Single Responsibility Principle.

——Statement.java——

```
…
  public void rent(CatalogItem item, int days) {
    int unitPrice = getUnitPrice(item);
    int price = calculatePrice(item, days, unitPrice);
    int thisTax = getTax(item, price);
    items.add(new RentalItem(item, days, unitPrice, price, thisTax
    subtotal += price;
    tax += thisTax;
  }

  private int getUnitPrice(CatalogItem item) {
    return switch (item) {
      case SMALL_ROOM -> 100;
      case LARGE_ROOM -> 150;
      case PROJECTOR -> 50;
      case COFFEE -> 10;
      case COOKIES -> 15;
    };
  }

  private int calculatePrice(CatalogItem item, int days, int unitP
    boolean eligibleForDiscount = isEligibleForDiscount(item, days
    int price = unitPrice * days;
    if (eligibleForDiscount) price = (int) Math.round(price * .9);
```

```
    return price;
  }

  private boolean isEligibleForDiscount(CatalogItem item, int days
    return switch (item) {
      case SMALL_ROOM, LARGE_ROOM -> days == 5;
      case PROJECTOR, COFFEE, COOKIES-> false;
    };
  }

  private int getTax(CatalogItem item, int price) {
    return switch (item) {
      case SMALL_ROOM, LARGE_ROOM, PROJECTOR ->
           (int) Math.round(price * .05);
      case COFFEE,COOKIES -> 0;
    };
  }
```

> Future Bob: Having used Grok3 on the `fromRoman` module in the last chapter, I thought I'd do the same here. So I told Grok3 to improve my initial implementation. It was not particularly surprising that its solution was nearly identical to mine.

Perhaps you look at this and think that it's more code. But it's not more executable code; it's just more names and more structure. That extra structure makes room for growth. For example, if the tax rules become more complicated, it is not the `rent` function that will grow; instead, in all likelihood, it will be the `getTax` function that grows.

This is an example of the Single Responsibility Principle (SRP)[5] in action. The stakeholders of our system who are most concerned about taxation will most probably be the only ones asking for changes to the `getTax` function, whereas the stakeholders who are interested in discounts will most likely be the only ones affecting the `calculatePrice` and `isEligibleForDiscount` functions.

[5] See Chapter 19, "The SOLID Principles."

This is helpful to us because, after this change, we are not very likely to break the discounts when the taxes change, or break the taxes when the price calculation changes. The farther we can keep these different stakeholders' responsibilities apart and isolated, the safer our code will be when changes are made.

If we ignore the SRP, then we risk the symptom of fragility. A fragile system breaks in unexpected ways; for example, when a stakeholder asks us to modify taxes and we inadvertently break discounts. Such occurrences are terrifying to our stakeholders, managers, and users because it gives them a glimpse of what's under the hood; and they don't like what they see.

Our stakeholders have the right to expect that a change to taxes will not break discounts. When we violate that expectation, the only conclusion they can draw is that we've lost control of the system and don't really know what the hell we are doing.

Now look at that new code. What don't you like about it? One thing we might focus on are those `switch` statements.

`Switch` statements aren't intrinsically bad. However, if the number of cases is likely to grow, then they violate the Open–Closed Principle (OCP).

The OCP suggests that when we add a new feature, we should add that new feature in one place, not in many places. Right now, if we were to add a new `CatalogItem`, like `NotePads`, we'd have to add those changes to as many as five different places in the code. We can reduce this down to two by turning the `CatalogItem`s into data structures and replacing the `switch` statements with accesses of those data structures.

```
——Statement.java——
…
public enum CatalogItem {
  SMALL_ROOM(100, .05),
  LARGE_ROOM(150, .05),
  PROJECTOR(50, .05),
  COFFEE(10, 0),
  COOKIES(15, 0);

  private double taxRate;
  private int unitPrice;

  CatalogItem(int unitPrice, double taxRate) {
    this.unitPrice = unitPrice;
    this.taxRate = taxRate;
  }
}
…
public void rent(CatalogItem item, int days) {
  int unitPrice = item.unitPrice;
  int price = calculatePrice(item, days, unitPrice);
  int thisTax = (int) Math.round(price * item.taxRate);
```

```
  items.add(new RentalItem(item, days, unitPrice, price, thisTax))
  subtotal += price;
  tax += thisTax;
}
…
```

This is better—I have to say that I like that Java's `enum`s are actually classes, and that each enumerator is an instance. Now when we add `NotePads`, we'll only have to change the `enum`, and, perhaps, the `getItems` function. However, the OCP tells us to keep new changes out of old modules. So it would be better, from an OCP point of view, to get that `enum` out of the `Statement` module altogether so that when we add `NotePads`, we can leave the `Statement` module untouched.

```
———CatalogItem.java———
package ubConferenceCenter;

public enum CatalogItem {
  SMALL_ROOM(100, .05),
  LARGE_ROOM(150, .05),
  PROJECTOR(50, .05),
  COFFEE(10, 0),
  COOKIES(15, 0);

  public double taxRate;
  public int unitPrice;

  CatalogItem(int unitPrice, double taxRate) {
    this.unitPrice = unitPrice;
```

```
    this.taxRate = taxRate;
  }
}
```

We're doing pretty well here. We've isolated the `CatalogItem` `enum`s, and we've gotten rid of the `switch` statements. The SRP and the OCP are pretty happy. But there's another principle that is sounding an alarm.

What modules ought to be recompiled if we add `NotePads` to `CatalogItem`? Answer: `CatalogItem.java` (of course) and probably[6] `Statement.java` because it depends upon `CatalogItem.java`. Now ask yourself whether `Statement.java` *should have* to be recompiled.

[6] The Java compiler is often smart enough to determine whether or not upstream modules need to be recompiled based on downstream changes. The safe bet, however, is to assume that those modules will be recompiled.

I contend that at least the `rent` function of `Statement.java` should not have to be recompiled when we add `NotePads`, because nothing within that function needs to know anything about `NotePads`.

This is a violation of the Dependency Inversion Principle (DIP). This principle is violated when high-level policies directly depend on low-level details. The addition of `NotePads` is a low-level detail that the `rent` function should not depend upon.

You may not think this is a big issue. Recompiling a module is quick and easy, and why should you care if you compile more modules than is required? For small projects, you could be right to make that decision. But for larger projects, the recompile and redeployment burden can get pretty large. This is especially true for JavaScript applications where redeployments are downloaded into browsers.

There’s also the cognitive burden of trying to remember which modules depend on which others. So let’s assume that this project is becoming large enough for those issues to be a concern. How can we invert the dependencies to reduce the recompilation, redeployment, and cognitive burdens?

The SRP will help us here. The `Statement.java` module is doing two major things. The `rent` method adds `CatalogItem`s, and the `getItems` method totals them and computes the cookie bonus. Hmmm, `getItems` isn’t the best name, is it? We’ll take care of that momentarily. First, we should separate those two major behaviors.

The first step is to pull out the `RentalItem` record into its own module.

```
——RentalItem.java___
package ubConferenceCenter;


public record RentalItem(CatalogItem type,
                         int days,
                         int unitPrice,
                         int price,
                         int tax) {
}
```

Next, we’ll create a new module for the `ItemList`.

```
——ItemList.java——
package ubConferenceCenter;


import java.util.ArrayList;
import java.util.List;
```

```
public class ItemList {
  private List<RentalItem> items = new ArrayList<>();

  public void add(CatalogItem item, int days, int unitPrice,
                  int price, int thisTax) {
    items.add(new RentalItem(item, days, unitPrice, price, thisTax
  }

  public RentalItem[] getItems() {
    boolean largeRoomFiveDays = items.stream().anyMatch(
      item -> item.type() == CatalogItem.LARGE_ROOM && item.days()
    boolean coffeeFiveDays = items.stream().anyMatch(
      item -> item.type() == CatalogItem.COFFEE && item.days() ==
    if (largeRoomFiveDays && coffeeFiveDays)
      items.add(new RentalItem(CatalogItem.COOKIES, 5, 0, 0, 0));
    return items.toArray(new RentalItem[0]);
  }
}
```

We'll have to come back to this module because that `getItems` method gives me the willies.[7] It smells of an SRP violation. But for now, all that remains is to make a few adjustments to the `Statement.java` module.

[7] An archaic term that means nervous, uneasy, or creeped out. (According to Grok.)

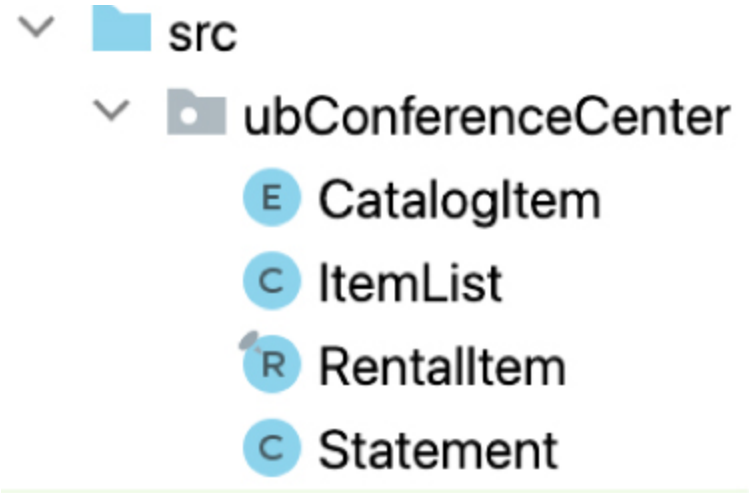
```
…
public class Statement {

  …
```

```
  private ItemList items = new ItemList();
  …

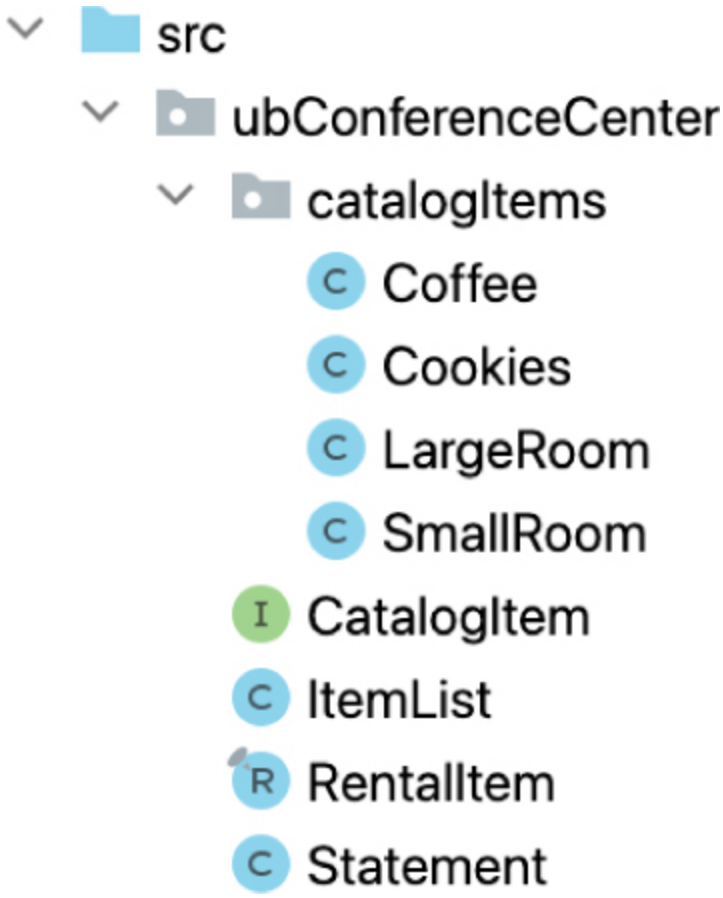
  public RentalItem[] getItems() {
    return items.getItems();
  }
  …
}
```

Again, you may be concerned that we're chopping this up into too many pieces, and that all those pieces will make the code harder to understand. But our assumption has been that this code is going to grow, and that we're trying to make room for that growth. In any case, if you look at the directory structure, you'll see that there's a nice road map that will help anyone understand the organization.



Looking at this, you can see that the names may not be perfect, but we'll come back to that when we understand more about where this reorganization is going. In the meantime, we still have to reduce the recompile and redeployment burden.

To do this, we'll change the `CatalogItem` from an `enum` to an interface, with each enumeration becoming a derived class. This creates several new classes and makes the directory look like this.



———CatalogItem.java———

```
package ubConferenceCenter;


public interface CatalogItem {
  boolean isEligibleForDiscount(int days);
  int getUnitPrice();
  double getTaxRate();
  String getName();
}
```

This represents a significant change in the way the application works. Now instead of using `switch` and `if` statements to check for enumerators, we defer to the polymorphic methods of the `CatalogItem` interface. The `getTaxRate` and `getUnitPrice` methods are familiar, but the other two are new. You can see how they are used in the `Statement.java` and `ItemList.java` modules.

——Statement.java——

```
package ubConferenceCenter;


public class Statement {
  …
  public void rent(CatalogItem item, int days) {
    int unitPrice = item.getUnitPrice();
    int price = calculatePrice(item, days, unitPrice);
    int thisTax = (int) Math.round(price * item.getTaxRate());
    items.add(item, days, unitPrice, price, thisTax);
    subtotal += price;
    tax += thisTax;
  }


  private int calculatePrice(CatalogItem item, int days, int unitP
    boolean eligibleForDiscount = item.isEligibleForDiscount(days)
    int price = unitPrice * days;
    if (eligibleForDiscount) price = (int) Math.round(price * .9);
    return price;
  }
 …
}
```

```
——ItemList.java——
package ubConferenceCenter;

import java.util.ArrayList;
import java.util.List;

public class ItemList {
  private List<RentalItem> items = new ArrayList<>();

  public void add(CatalogItem item, int days, int unitPrice,
                  int price, int thisTax) {
    items.add(new RentalItem(item.getName(), days,
              unitPrice, price, thisTax));
  }

  public RentalItem[] getItems() {
    boolean largeRoomFiveDays = items.stream().anyMatch(
      item -> item.type().equals("LARGE_ROOM") && item.days() == 5
    boolean coffeeFiveDays = items.stream().anyMatch(
      item -> item.type().equals("COFFEE") && item.days() == 5);
    if (largeRoomFiveDays && coffeeFiveDays)
      items.add(new RentalItem("COOKIES", 5, 0, 0, 0));
    return items.toArray(new RentalItem[0]);
  }
}
```

Notice that the `type` field of the `RentalItem` record has been changed from the `enum` to a `String`. This was necessary because, when we abandoned the `enum`, we needed some kind of token to represent the `CatalogItem` type; and a `String` seemed the most obvious choice. However, using that string means that we've had to sacrifice a bit of static type safety. The compiler cannot check that those names are correct. This minor loss of static type safety always accompanies the effort to reduce the recompile and redeployment burden, and the isolation of low-level details from high-level policy.

```
——RentalItem.java——

package ubConferenceCenter;


public record RentalItem(String type,
                         int days,
                         int unitPrice,
                         int price,
                         int tax) {
}
```

With the exception of the `getItems` method of the `ItemList` class, the last few listings above represent the high-level policy of the application. We're going to have to do something about that `getItems` method. For now, let's look at how the low-level details have been isolated in the derivatives of `CatalogItem`.

```
——SmallRoom.java——

package ubConferenceCenter.catalogItems;


import ubConferenceCenter.CatalogItem;
```

```
public class SmallRoom implements CatalogItem {
  public String getName() {
    return "SMALL_ROOM";
  }

  public boolean isEligibleForDiscount(int days) {
    return days == 5;
  }

  public int getUnitPrice() {
    return 100;
  }

  public double getTaxRate() {
    return 0.05;
  }
}

———LargeRoom.java———
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class LargeRoom implements CatalogItem {
  public boolean isEligibleForDiscount(int days) {
    return days == 5;
  }

  public int getUnitPrice() {
```

```
    return 150;
  }

  public double getTaxRate() {
    return 0.05;
  }

  public String getName() {

    return "LARGE_ROOM";
  }
}

———Coffee.java———
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class Coffee implements CatalogItem {
  public boolean isEligibleForDiscount(int days) {
    return false;
  }

  public int getUnitPrice() {
    return 10;
  }

  public double getTaxRate() {
    return 0;
```

```
  }

  public String getName() {
    return "COFFEE";
  }
}

———Cookies.java———

package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class Cookies implements CatalogItem {
  public boolean isEligibleForDiscount(int days) {
    return false;
  }

  public int getUnitPrice() {
    return 15;
  }

  public double getTaxRate() {
    return 0;
  }

  public String getName() {
    return "COOKIES";
  }
```

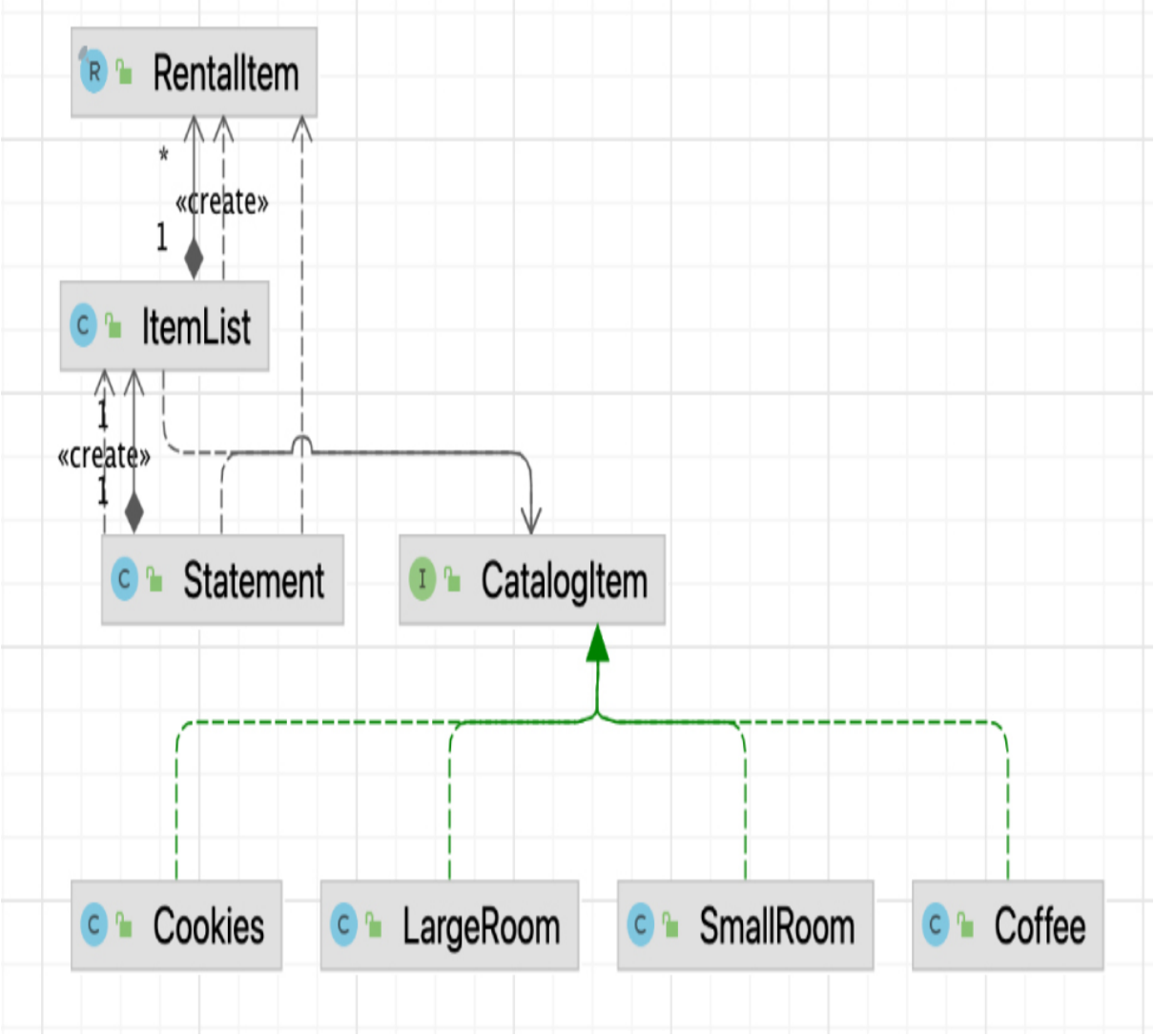
```
}
```

One of the people who commented on the first edition of this book looked at code like this and called it “just dreadful.” You might be feeling that now too. But remember, we are expecting business growth, and we have decided that it is necessary to make room in this code for that growth.

Some folks might charge us with violating YAGNI. They think it stands for You Aren’t Going to Need It. But the original intent of YAGNI was to ask yourself: “What if you aren’t going to need it?” It was a way of asking us to count the cost before we invested in making room in our code for things we might not need.

But given the enthusiasm of the CEO and the kinds of markets he is courting, we have decided we need to make this room. So, in our case, we’ve decided to answer YAGNI’s question with: “Yes, we are going to need it.”

Why is this code better? Let’s look at the dependency diagram generated by my IDE.



The four classes that represent our high-level policy—`RentalItem`, `ItemList`, `Statement`, and `CatalogItem`—are tied together with a rat's nest of dependencies. We'll deal with that later. But look at how well isolated the low-level details are. The previous dependencies have been inverted. Nothing within the high-level policy depends upon the low-level details. The low-level details depend only on the `CatalogItem`. Just say this to yourself, over and over: High-level policy should not depend on low-level details.

This means that if any of the low-level details are changed, nothing in the high-level policy ought necessarily to be recompiled or redeployed. Nor will changes to the high-level

policy force recompilation and redeployment of the low-level details.

Let me say that differently. If the unit price of `Cookies` changes, or if the tax rate of `Coffee` changes, or if the discount for `SmallRoom` changes, none of the classes in the high-level policy will need to be recompiled or redeployed. Changes to those low-level details are completely isolated from the high-level policy.

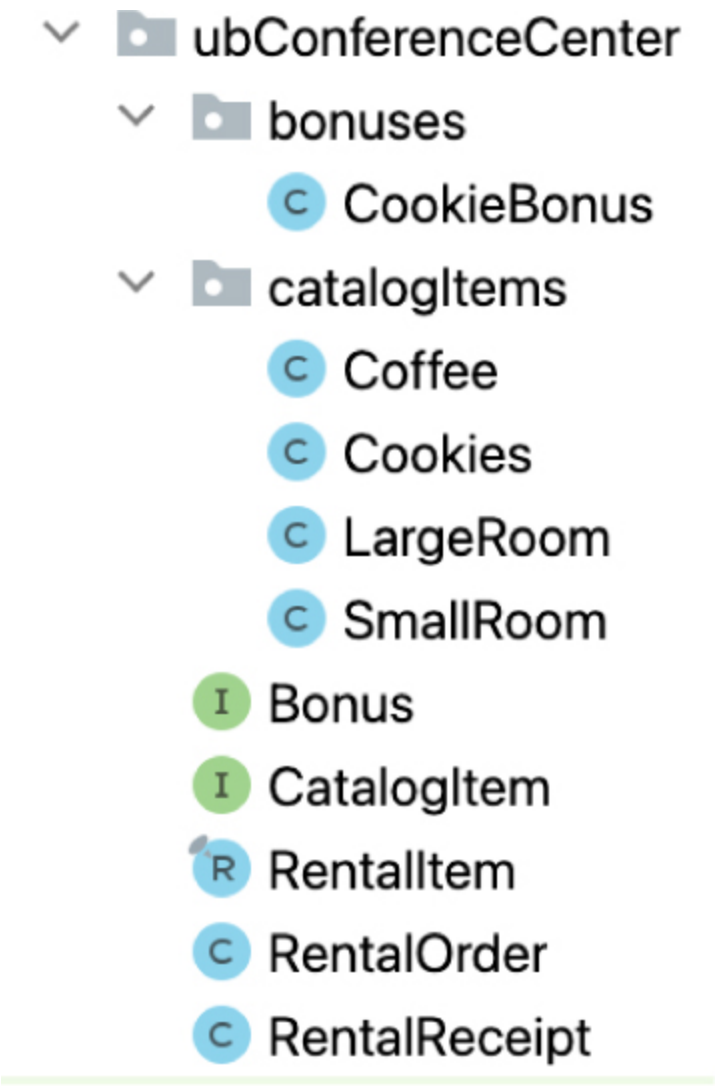
Indeed, the low-level details have become a plug-in to the high-level policy. Or, at least they are in a position to become such a plug-in.

But now we need to do something about the rat's nest that's caused by the `ItemList`.

We called this class `ItemList` because initially it contained our list of items, and we needed a place to put that bonus code. Now I want to take that bonus code out of there and do to it what we did to the `CatalogItems`. So I'll create a `Bonus` interface and have the `CookieBonus` implement it. The `ItemList` can contain a list of `Bonus` instances and simply iterate through all of them to add all the bonuses. That means that the `ItemList` *finalizes* all the bonuses. And that suggests that the `ItemList` should be renamed something like `RentalReceipt`, since it represents the final state of the order.

That name change suggests that `Statement` should be renamed something like `RentalOrder`. These name changes are important. As we pick apart the design and start making room for growth, we gain insights into what's really going on. When the problem is small, names can afford to be inconsistent or vague because there aren't that many concepts to keep track of. But as the problem grows, names become much more important because they help us keep the concepts straight in our heads.

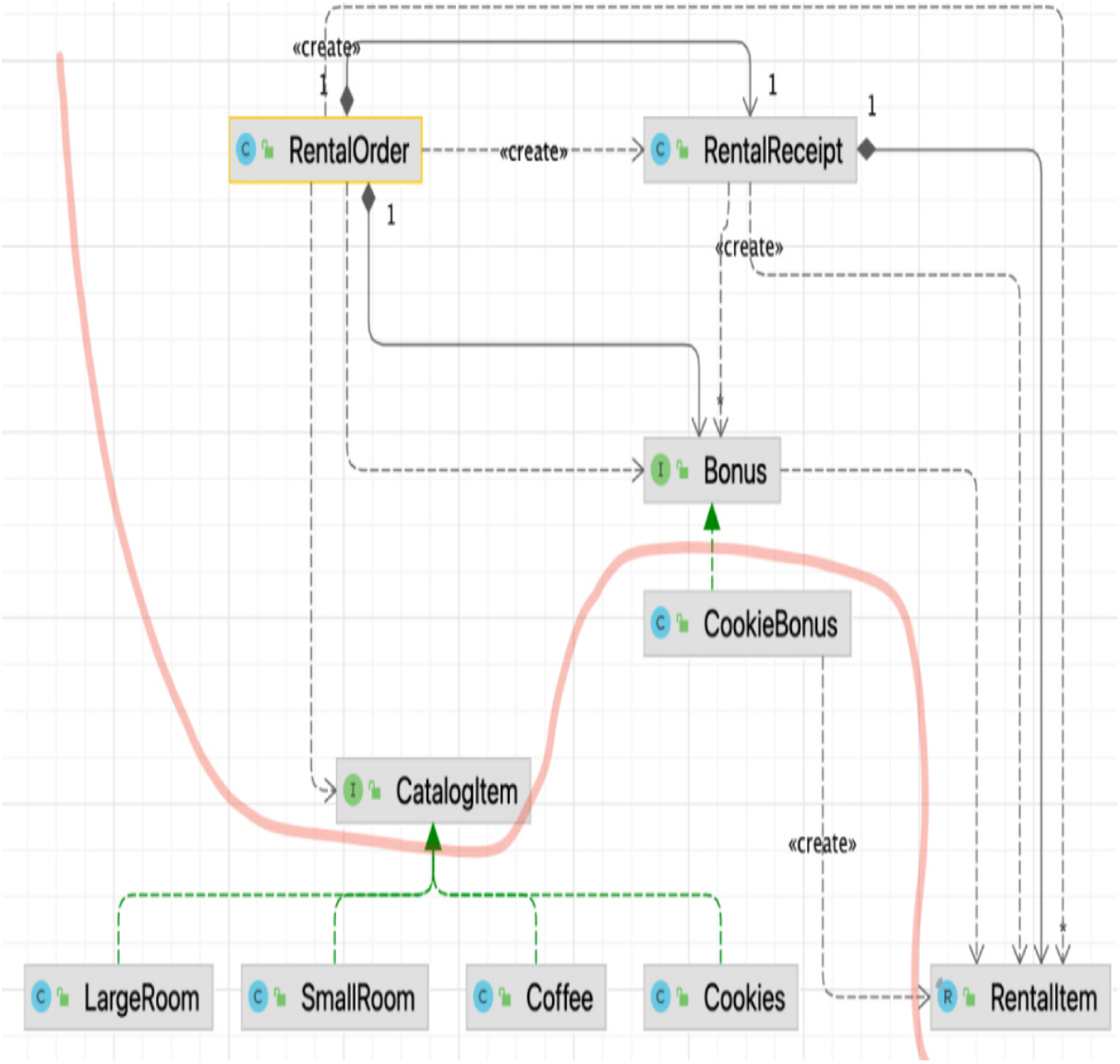
So with those changes in place, here's the directory structure:



The diagram looks like this. The `RentalOrder` depends on the `RentalReceipt`, but not vice versa. Both depend upon the `Bonus`, but the `CookieBonus` is an independent low-level detail.

The lowest-level detail is the `RentalItem` record. That's a bit of a concern because it is so concrete. If it gets changed in any way, then pretty much everything has to recompile and redeploy. There are ways to resolve that—but I think that's a battle for another day.

Notice the curvy border line. That line divides the project into two components. One contains the high-level policy, and the other contains the low-level details. All dependencies cross that line in one direction, toward the higher-level component. That line is an architectural boundary that separates those two components—and the rule for architectural boundaries is that dependencies cross only toward the higher-level side.



First, let’s look at the code within the high-level component.

——RentalOrder.java——

```
package ubConferenceCenter;

import java.util.ArrayList;
import java.util.List;


public class RentalOrder {
  public record Totals(int subtotal, int tax) {
  }

  private String customerName;
  private int subtotal = 0;
  private int tax = 0;
  private RentalReceipt receipt = new RentalReceipt();
  private List<Bonus> bonuses = new ArrayList<>();

  public RentalOrder(String customerName) {
    this.customerName = customerName;
  }

  public void addBonus(Bonus bonus) {
    bonuses.add(bonus);
  }

  public void rent(CatalogItem item, int days) {
    int unitPrice = item.getUnitPrice();
    int price = item.getDiscountedPrice(days);
    int thisTax = (int) Math.round(price * item.getTaxRate());
    receipt.add(new RentalItem(item.getName(), days,
```

```
            unitPrice, price, thisTax));
    subtotal += price;
    tax += thisTax;
  }


  public RentalItem[] getReceipt() {
    return receipt.finalize(bonuses);
  }

  public String getCustomerName() {
    return customerName;
  }

  public Totals getTotals() {
    return new Totals(subtotal, tax);
  }
}
```

Notice that virtually all of the business rules have fled this module. The only ones that remain are the tax and total calculations. But the discount and bonus calculations have been moved into the appropriate derivatives of `CatalogItem` and `Bonus`.

```
———RentalReceipt.java———
package ubConferenceCenter;


import java.util.ArrayList;
import java.util.List;
```

```
public class RentalReceipt {
  private List<RentalItem> items = new ArrayList<>();

  public void add(RentalItem item) {
    items.add(item);
  }

  public RentalItem[] finalize(List<Bonus> bonuses) {
    List<RentalItem> finalItems = new ArrayList<>(this.items);

    finalItems.addAll(addBonuses(bonuses));
    return finalItems.toArray(new RentalItem[0]);
  }

  private List<RentalItem> addBonuses(List<Bonus> bonuses) {
    List<RentalItem> bonusItems = new ArrayList<>();
    for (Bonus bonus : bonuses)
      bonus.checkAndAdd(items, bonusItems);
    return bonusItems;
  }
}
```

The `RentalReceipt` class walks through and applies the bonuses; but it does not know any of the details about those bonuses. It creates a finalized receipt with the bonuses added to all the items.

Next come the two interfaces that inverted the dependencies between the high-level policy and the low-level detail.

```
———CatalogItem.java———
package ubConferenceCenter;

public interface CatalogItem {
  int getDiscountedPrice(int days);
  int getUnitPrice();
  double getTaxRate();
  String getName();
}

———Bonus.java———
package ubConferenceCenter;

import java.util.List;

public interface Bonus {
  void checkAndAdd(List<RentalItem> items, List<RentalItem> bonusI
}
```

The last module in the high-level component is the `RentalItem`. This is a simple concrete data structure.

```
———RentalItem.java———
package ubConferenceCenter;

public record RentalItem(String type,
                         int days,
                         int unitPrice,
                         int price,
```

```
                        int tax) {
}
```

As previously stated, this module is somewhat problematic. Any changes made to this data structure will force recompilation and redeployment of both components. And changes to this data structure are not at all unlikely.

There are ways of dealing with this. For example, we could turn the `RentalItem` into a hash map. But that's probably going well beyond our current goal of making room for growth. We'll keep that idea in our back pocket for now.

Now let's look at the low-level component.

```
——SmallRoom.java——
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class SmallRoom implements CatalogItem {
  public String getName() {
    return "SMALL_ROOM";
  }

  public int getDiscountedPrice(int days) {
    double discountRate = (days == 5) ? 0.9 : 1.0;
    return (int) Math.round(getUnitPrice() * days * discountRate);
  }

  public int getUnitPrice() {
    return 100;
```

```
  }

  public double getTaxRate() {
    return 0.05;
  }
}

———LargeRoom.java___
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class LargeRoom implements CatalogItem {
  public int getDiscountedPrice(int days) {
    double discountRate = (days == 5) ? 0.9 : 1.0;
    return (int) Math.round(getUnitPrice() * days * discountRate);
  }

  public int getUnitPrice() {
    return 150;
  }

  public double getTaxRate() {
    return 0.05;
  }

  public String getName() {
    return "LARGE_ROOM";
```

```
  }
}

———Coffee.java———
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class Coffee implements CatalogItem {
  public int getDiscountedPrice(int days) {
    return getUnitPrice() * days;
  }

  public int getUnitPrice() {
    return 10;
  }

  public double getTaxRate() {
    return 0;
  }

  public String getName() {
    return "COFFEE";
  }
}

———Cookies.java———
package ubConferenceCenter.catalogItems;
```

```
import ubConferenceCenter.CatalogItem;

public class Cookies implements CatalogItem {
  public int getDiscountedPrice(int days) {
    return getUnitPrice() * days;
  }

  public int getUnitPrice() {
    return 15;
  }

  public double getTaxRate() {
    return 0;
  }

  public String getName() {
    return "COOKIES";
  }
}

———CookieBonus.java———
package ubConferenceCenter.bonuses;

import ubConferenceCenter.Bonus;
import ubConferenceCenter.RentalItem;

import java.util.List;
```

```
public class CookieBonus implements Bonus {
  public void checkAndAdd(List<RentalItem> items,
                          List<RentalItem> bonusItems) {
    boolean largeRoomFiveDays = items.stream().anyMatch(
      item -> item.type().equals("LARGE_ROOM") && item.days() == 5
    boolean coffeeFiveDays = items.stream().anyMatch(
      item -> item.type().equals("COFFEE") && item.days() == 5);
    if (largeRoomFiveDays && coffeeFiveDays)
      bonusItems.add(new RentalItem("COOKIES", 5, 0, 0, 0));
  }
}
```

As you can see, all the business rules have been tucked away into their corresponding derivatives. If a new discount is needed for coffee, we can place it in the `Coffee` class. If a new bonus is needed for small rooms, we can create a `Bonus` derivative for that. If we want to add `NotePads` or `Projector`s, we can do that by adding those classes to the low-level component. None of those changes will cause the high-level component to be recompiled or redeployed. We have achieved that isolation by conforming to the DIP.

You might be concerned about one thing, though. Perhaps you are wondering where all those derivatives are getting instantiated. I never showed you my tests, did I. That's where all that work was done. Here they are.

```
——RentalOrderTest.java——
package ubConferenceCenter;


import org.junit.jupiter.api.BeforeEach;
```

```
import org.junit.jupiter.api.Test;
import ubConferenceCenter.RentalOrder.Totals;
import ubConferenceCenter.bonuses.CookieBonus;
import ubConferenceCenter.catalogItems.Coffee;
import ubConferenceCenter.catalogItems.Cookies;
import ubConferenceCenter.catalogItems.LargeRoom;
import ubConferenceCenter.catalogItems.SmallRoom;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class RentalOrderTest {
  private final SmallRoom SMALL_ROOM = new SmallRoom();
  private final LargeRoom LARGE_ROOM = new LargeRoom();
  private final Coffee COFFEE = new Coffee();
  private final Cookies COOKIES = new Cookies();
  private RentalOrder order;

  @BeforeEach
  void setUp() {
    order = new RentalOrder("Customer Name");
    order.addBonus(new CookieBonus());
  }

  private void assertTotalAndTax(int subtotal, int tax) {
    Totals totals = order.getTotals();
    assertEquals(subtotal, totals.subtotal());
    assertEquals(tax, totals.tax());
  }
```

```
private void assertTypeDaysUnitTotalTax(RentalItem item, String t
                                        int days, int unitPrice,
                                        int price, int tax) {
  assertEquals(type, item.type());
  assertEquals(days, item.days());
  assertEquals(unitPrice, item.unitPrice());
  assertEquals(price, item.price());
  assertEquals(tax, item.tax());
}

@Test
public void oneSmallRoomForOneDay() throws Exception {
  assertEquals("Customer Name", order.getCustomerName());
  order.rent(SMALL_ROOM, 1);
  RentalItem[] receipt = order.getReciept();
  assertEquals(1, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 1, 100, 1
  assertTotalAndTax(100, 5);
}

@Test
public void oneSmallRoomForTwoDays() throws Exception {
  order.rent(SMALL_ROOM, 2);
  RentalItem[] receipt = order.getReciept();
  assertEquals(1, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 2, 100, 2
  assertTotalAndTax(200, 10);
```

```
}

@Test
public void oneLargeRoomForThreeDays() throws Exception {
  order.rent(LARGE_ROOM, 3);
  RentalItem[] receipt = order.getReciept();
  assertEquals(1, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "LARGE_ROOM", 3, 150, 4
  assertTotalAndTax(450, 23);
}

@Test
public void oneSmallRoomForOneWeek() throws Exception {
  order.rent(SMALL_ROOM, 5);
  RentalItem[] receipt = order.getReciept();
  assertEquals(1, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 5, 100, 4
  assertTotalAndTax(450, 23); /* 10% discount */
}

@Test
public void twoSmallRoomsForOneDay() throws Exception {
  order.rent(SMALL_ROOM, 1);
  order.rent(SMALL_ROOM, 1);
  RentalItem[] receipt = order.getReciept();
  assertEquals(2, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 1, 100, 1
  assertTypeDaysUnitTotalTax(receipt[1], "SMALL_ROOM", 1, 100, 1
```

```
  assertTotalAndTax(200, 10);
}

@Test
public void oneSmallRoomAndCoffeeForOneDay() throws Exception {
  order.rent(SMALL_ROOM, 1);
  order.rent(COFFEE, 1);
  RentalItem[] receipt = order.getReciept();
  assertEquals(2, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 1, 100, 1
  assertTypeDaysUnitTotalTax(receipt[1], "COFFEE", 1, 10, 10, 0)
  assertTotalAndTax(110, 5); /* No tax on coffee */
}

@Test
public void oneSmallRoomAndCookiesForOneDay() throws Exception {
  order.rent(SMALL_ROOM, 1);
  order.rent(COOKIES, 1);
  RentalItem[] receipt = order.getReciept();
  assertEquals(2, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 1, 100, 1
  assertTypeDaysUnitTotalTax(receipt[1], "COOKIES", 1, 15, 15, 0
  assertTotalAndTax(115, 5); /* No tax on cookies */
}

@Test
public void oneSmallRoomCookiesCoffeeForFiveDays() throws Except
  order.rent(SMALL_ROOM, 5);
```

```
    order.rent(COFFEE, 5);
    order.rent(COOKIES, 5);
    RentalItem[] receipt = order.getReciept();
    assertEquals(3, receipt.length);
    assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 5, 100, 4
    assertTypeDaysUnitTotalTax(receipt[1], "COFFEE", 5, 10, 50, 0)
    assertTypeDaysUnitTotalTax(receipt[2], "COOKIES", 5, 15, 75, 0
    /* 10% discount on room but not on coffee */
    assertTotalAndTax(575, 23);
  }

  @Test
  public void oneLargeRoomAndCoffeeForWeekGetsCookies() throws Exc
    order.rent(LARGE_ROOM, 5);
    order.rent(COFFEE, 5);
    RentalItem[] receipt = order.getReciept();
    assertEquals(3, receipt.length);
    assertTypeDaysUnitTotalTax(receipt[0], "LARGE_ROOM", 5, 150, 6
    assertTypeDaysUnitTotalTax(receipt[1], "COFFEE", 5, 10, 50, 0)
    assertTypeDaysUnitTotalTax(receipt[2], "COOKIES", 5, 0, 0, 0);
  }
}
```

These tests were written before any of the refactoring in this chapter had begun. At each step of that refactoring, I kept these tests passing. Some of the changes made to the production code caused changes to the tests, but I was able to minimize them by keeping the tests and production code decoupled and by isolating the details of the production

code from the details of the tests. For example, the final fields named for the `CatalogItems` replaced the old enumerations without too much fuss.

Independent Deployability

Our design now has a strong architectural boundary that separates it into two components. Those two components can be independently deployed. They can be compiled into two separate `jar` files. When one component changes, only that `jar` changes, and the other does not need to be redeployed.

Imagine that this code was written in JavaScript as opposed to Java. Imagine that it will be sent to a browser to run there. The fact that we have divided it into two components has a very specific advantage. If one of those two components changes, and if the browser maintains a cache of those components, then only the component that changed needs to be reloaded into the browser. On slow network connections, that could be a big advantage.

A Final Thought

This major cleaning will greatly facilitate the growth of the project. The time it took will be paid back many times over simply because there’s a clear and obvious place to put the needed changes; and those changes aren’t likely to interfere with each other or with the overall flow of the system.

However, the new structure is not cost free. Those who complain that it is harder to read because of all the different modules and all the indirection are not without a point. Yes, adding structure adds complexity—and that complexity is not free. There’s a cognitive price to pay. But we knew that the cognitive load was going to grow, and so our goal was to

minimize that growth by creating an organization that would accept it gracefully.

What about performance? Is the new structure slower than the old one? Probably so. There are a few more references that need to be followed, and `switch` statements are easier for compilers to optimize than polymorphic dispatch. But the difference in performance is probably so tiny that you could not easily measure it. For most applications, that tiny decrease in performance would be of no concern at all.

> *Future Bob: Grok3, good as it is, did not drive the design of the original code to the level of depth we just saw. Grok3 did not invert the dependencies, nor did it attempt to separate high-level policy from low-level detail. Large language models (LLMs)/AIs have their uses; but they are not human and are not adept at understanding higher architectural goals.*

This chapter, and the chapter before, were a whirlwind tour through the low-hanging fruit of clean code. But now that tour is over, and it’s time to get down to the basics. And, of course, we begin with names.

Chapter 4. Meaningful Names

by Tim Ottinger and Uncle Bob



Beginning in the late ’90s I ran a company named Object Mentor Inc. Our website sported a number of technical documents that we offered to the software community. The most downloaded of all those documents was Tim Ottinger’s rules for names. What you are about to

read is the descendant and natural evolution of Tim's classic document.

—Uncle Bob

Names are everywhere in software. We name our variables, our functions, our arguments, classes, and packages. We name our source files and the directories that contain them. We name our `jar` files and `dll` files and `gem` files. We name and name and name; and because we do so much of it, we'd better do it well.

It might seem that names are so subjective that defining the principles of "goodness" is a fool's errand. However, despite their subjectivity, we know of good restaurants, good movies, good paintings, and good books.

Subjectivity is tempered by the realization that any work is good if it is accepted by its audience. This means that we aren't dealing with individual preferences and prejudices, but rather, with the desires of a collective audience.

The audience for this book is the set of programmers who are interested enough in improving their code craft to purchase, borrow, or in some other way acquire a copy of this book. If the writing reaches you and helps you, then we've written good-enough words in a good-enough order.

As an author of code, your audience is the set of people who will work in your codebase. You are not only an author, but also a member of the codebase community. The audience is larger than you are, and they collectively will decide what is welcome and what is unwelcome in the codebase.

Write for your audience, not for yourself. You must remember that your experience of writing the code is less important and more transient than their experience of reading that code.

That said, we find there is a fairly consistent band of preferences across many audiences, and this generalized advice is intended to help you find that "sweet spot" with your current software audience.

Choosing good names takes time, but saves more than it takes. So take care with your names and change them when you find better

ones. Everyone who reads your code (including you) will be happier if you do.

To help you in that effort, here are some of the more useful recommendations we have found.

Use Intention-Revealing Names

The name of a variable, function, or class should answer all the big questions. It should tell its audience why it exists, what it does, and how it is used.

If a name requires a comment to communicate to its audience, then the name does not reveal its intent.

```
int d; // elapsed time in days
```

The name `d` reveals very little. It does not evoke a sense of elapsed time; specifically, of days. We should choose a name that specifies what is being measured and the unit of that measurement. One of these might be better: `elapsedDays`, `daysSinceCreation`, `daysSinceModification`, or `fileAgeInDays`.

These examples are longer than the original name (`d`), but that doesn't necessarily mean that long names are better than short names, only that clear names are better than unclear ones. For example:

```
int d = getElapsedTimeInDays();
```

Here the name is perfectly clear since it is defined by the context and would be appropriate if the scope that contains the name is relatively small. (See "Use Names of Appropriate Length.")

Build a System of Names

Choosing names that reveal intent can make it much easier to understand and change code. What follows is an example that breaks this rule. Can you determine the purpose of this code?

```
  public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
```

```
    for (int[] x : theList)
      if (x[0] == 4)
        list1.add(x);
    return list1;
  }
```

Why is it hard to tell what this code is doing? There are no complex expressions. Spacing and indentation are reasonable. There are only three variables and two constants mentioned. There aren't even any fancy classes or polymorphic methods, just a list of arrays (or so it seems).

The problem isn't the simplicity of the code. The problem is that the code provides no context. The code implicitly requires that we know the answers to questions such as

- What kinds of things are in `theList`?
- What does the zeroth subscript signify?
- What is the meaning of the value 4?
- How would I use the list being returned?

The answers to these questions are not present in the code sample. However, a good system of names would have made them clear.

Let's say that code came from the old *Minesweeper* game.[1] The *Minesweeper* game board is a list of cells called `theList`. Let's rename that to `gameBoard`.

[1] A game from the '90s that shipped with Windows 3.11.

Each cell on the board is represented by a primitive array. The zeroth subscript is the location of a status value, and a status value of 4 means "flagged." Just by giving these concepts names we can improve the code considerably:

```
  public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard)
      if (cell[STATUS_VALUE] == FLAGGED)
        flaggedCells.add(cell);
```

```
    return flaggedCells;
  }
```

Notice that the simplicity of the code has not changed. Only the clarity has improved.

It still has exactly the same number of operators and constants, with exactly the same number of nesting levels. But the code has become much more explicit through naming.

Note also that if this was the very first bit of code you came across, you would already know that this was a game with a board of cells that had statuses, one of which was “flagged.” A good system of names communicates a great deal about the application as a whole.

We can go further and write a simple class for cells instead of using an array of `int`s. It can include an intention-revealing function (call it `isFlagged`) to hide the magic numbers. It results in a new version of the function:

```
  public List<Cell> getFlaggedCells() {
    List<Cell> flaggedCells = new ArrayList<Cell>();
    for (Cell cell : gameBoard)
      if (cell.isFlagged())
        flaggedCells.add(cell);
    return flaggedCells;
  }
```

With these simple name changes, it’s not difficult to understand what’s going on. This is the power of choosing good names.

Avoid Disinformation

Programmers must avoid leaving false clues that obscure the meaning of code. For example, do not refer to a grouping of accounts as an `accountList` unless it’s actually a `List`.

The word *list* means something specific to programmers. If the container holding the accounts is an array or a JSON document, it may lead to false conclusions.[2] So `accountGroup` or `bunchOfAccounts` or just

plain `accounts` may be shorter (and thereby easier to take in at a glance) and more honest.

[2] As we'll see later on, even if the container is a `List`, it's probably better not to encode the container type into the name.

Beware of using names that vary in small ways. How long does it take to spot the subtle difference between an `XYZControllerForEfficientHandlingOfStrings` in one module and, somewhere a little more distant, an `XYZControllerForEfficientStorageOfStrings`? The words have frightfully similar shapes. Would you realize that they were different if they weren't right next to each other?

Those names are doubly difficult because they are long and have similar beginnings. Can you spot the optical illusion below?

```
XYZControllerForEfficientHandlingOfStrings

XXZControllerForEfficientHandlingOfStrings
```

Spelling similar concepts similarly is *information*. Using inconsistent spellings is *disinformation*. With modern environments, we enjoy automatic code completion. We write a few characters of a name and are quickly rewarded with a list of possible completions for that name. Inconsistent spellings will confound that list.

Modern environments also help by using different fonts, styles, and colors for different syntactic elements. Still, it is wise to be careful of some of the optical illusions inherent in our fonts. For example, be careful with the use of lowercase *L* and uppercase *O*. In certain fonts, they can be indistinguishable from the constants 1 and 0, respectively.

```
int a = l;

if ( O == l )

  a = O1;

else

  l = 01;
```

Combining these letters with numbers and `x` can make it worse. Consider `0xll` as a name.

Make Meaningful Distinctions

Programmers create problems for themselves when they write code solely to satisfy a compiler or interpreter. For example, when you want to use the same name to refer to two different things in the same scope, you might be tempted to change one name in an arbitrary way to keep them distinct. Sometimes this is done by misspelling one, leading to the surprising situation where correcting spelling errors leads to an inability to compile.[3]

[3] Consider, for example, the truly hideous practice of creating a variable named `klass` just because the name `class` was used for something else.

It is not sufficient to add number series or noise words, even though the compiler is satisfied. If names must be different, then they should also mean something different.

Number-series naming (`a1`, `a2`, ... `aN`) is the opposite of intentional naming. Such names are not dis-informative, they are uninformative; they provide no clue to the author's intention. Consider:

```
public static void copyChars(char a1[], char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
}
```

This function reads much better when `source` and `destination` are used for the argument names.

In general, when you see variables named `X1`, `X2`, `X3`, the code is practically begging you for either an array or a data structure. Consider what makes them so similar that they have the same name and only a numeric difference, and look for some meaning that will inspire either different names or different structures entirely.

In the code above, it was just lazy naming. Changing those names to `source` and `destination` makes a better solution than trying to create a structure, class, or container. Not all problems are deep.

Noise words are another meaningless distinction. Imagine that you have a `Product` class. If you have another called `ProductInfo` or `ProductData`, you have made the names different without making them

mean anything different. `Info` and `Data` are indistinct noise words. They add no new information, because every class contains “data” and every class contains “info.”

Noise words are redundant. The word *variable* should never appear in a variable name. The word *table* should never appear in a table name. How is `NameString` better than `Name`? Would a `Name` ever be a floating-point number? If so, it likely breaks an earlier rule about disinformation. Imagine finding one class named `Customer` and another named `CustomerObject`. What should you understand as the distinction? Which one will represent the best path to a customer’s payment history?

Here are some disturbingly meaningless function names we found in one particular application.

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

How are the programmers in this project supposed to know which of these functions to call? How many active accounts are there? Which of them are returned by the first function? And what is the difference between the second and third functions? In any universe, this “system” of names is a nightmare.

Use Pronounceable Names

Humans are good at words. A significant part of our brains is dedicated to the concept of words. And words are, by definition, pronounceable. It would be a shame not to take advantage of that huge portion of our brains that has evolved to deal with spoken language. So make your names pronounceable.

If you can’t pronounce it, you can’t conveniently discuss it. “Well, over here on the bee cee arr three cee enn tee we have a pee ess zee kyew int, see?” Convenience of discussion matters because programming is a social activity.

One company had the name `genymdhms` (generation date, year, month, day, hour, minute, and second), so they walked around saying “gen why emm dee aich emm ess.” This in inconvenient, so the

pronunciation gradually evolved into “gen-yah-mudda-hims.” This might sound silly, but they were all in on the joke, so it was fun. Fun or not, they were tolerating poor naming. New developers had to have the variables explained to them, and then they spoke about it in silly, made-up words instead of using proper English terms.

Compare:

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
```

to:

```
class Customer {
  private Date generated;
  private Date modified;
  private final String recordId = "102";
```

Intelligent conversation is now possible: “Hey, Mikey, take a look at this record! The generation timestamp is set to tomorrow’s date! How can that be?”

Use Searchable Names

Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text. One might easily grep for `MAX_CLASSES_PER_STUDENT`, but the number 7 could be more troublesome. Searches may turn up the digit as part of filenames, as other constant definitions, and in various expressions where the value is used with different intent. It is even worse when a constant is a long number and someone might have transposed digits, thereby creating a bug while simultaneously evading the programmer’s search.

Likewise, the name `e` is not a searchable name. It is the most common letter in the English language and likely to show up in every passage of text in every program. In general, short names are less searchable than long names, which is one more reason that when it

comes to being searchable, longer names can trump shorter names, and any searchable name trumps a constant in code.

The name `e` is often used to denote the specific error that was raised in a `try`/`catch` block, especially in Python and Go. In this case, though, the name is familiar in its context to many programmers, and it occurs in such a small scope as to make searches unnecessary. Short names in limited and well-known circumstances generally don't confound searches .

Our preference is that very short, or single-letter, names ought *only* to be used as local variables inside very short scopes. If a variable or constant might be seen or used in multiple places in a body of code, it is best to give it a search-friendly name. Once again, compare:

```
for (int j=0; j<34; j++) {
   s += (t[j]*4)/5;
}
```

to:

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
   int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
   int realTaskWeeks = (realTaskDays / WORK_DAYS_PER_WEEK);
   sum += realTaskWeeks;
}
```

Note that `sum`, above, is not a particularly useful name but at least is searchable. The intentionally named code makes for a longer function, but consider how much easier it will be to find `WORK_DAYS_PER_WEEK` than to find all the places where `5` was used and filter the list down to just the instances with the intended meaning.

Use Names of Appropriate Length

Our rules for the length of names may surprise you, but they have withstood the test of time.

Variable Names

We have a simple rule for the names of variables. The length of a variable's name should be proportional to the length of the scope that contains it. For example:

```
for (int n=0; n<20; n++)
  System.out.println("" + n + " " + n*n);
```

The name `n` is appropriately short because its scope is two lines long. We would find it annoying if that variable were named `theNumberToBeSquared`.

On the other hand, in a longer scope, such small names can be opaque. We would usually find an instance variable named `n` annoying; especially if `m`, `p`, `i`, and `j` were also instance variables.

As the size of the scope increases, so the length of the variable names within that scope should increase. Arguments and local variables are scoped to a single function and so should have relatively short names; perhaps a word or two. Instance variables are scoped to a whole class. Their names should be relatively longer. Global variable names should be longer still.

Function Names

Our rule for functions is exactly the opposite. The length of their names should be inversely proportional to the length of their scope. Thus, global functions will have short names, while private methods within a class will have relatively long names. Tests, which have the shortest scope of all (effectively zero), have the longest names of all.

We justify this in two ways. First, the larger the scope of a function, the more frequently it will be used. So, for the sake of convenience, we'd like the name to be short. We would find it quite annoying if the function that opens a file was named `openFileAndThrowExceptionIfNotFound` instead of `open`.

Second, the larger the scope of a function, the more general it is. Again, `open` is a very general function. Methods within a class have a much narrower context and so will likely need more words to explain their intent. Private methods within a class have very small scopes and are called from very few places, or even just one. They are not

general at all and may therefore require many words to describe properly. Their names should be correspondingly longer.

Class Names

Our rule for classes is the same as our rule for functions. The longer the scope, the shorter the name. Classes at the global scope should have relatively short names (a word or two), while derived classes, inner classes, and private classes should have longer names.

Property and Attribute Names

Some languages, like C++ and C#, have elements that look like variables but are actually functions. They look like variables because they can be put on the left of assignment statements. Therefore, they should follow the variable-naming rule. The length of their names should be proportional to the scope that contains them.

Namespace Names

Most languages now allow us to put our classes, functions, and variables into namespaces. Namespace names should be descriptive. Thus, they are likely to be longish and are often part of a hierarchy of namespaces. Consider, for example, the namespace from a space war game[4] Uncle Bob wrote a few years back: `spacewar.ui.tactical-scan`. Such long and qualified names are generally inconvenient, which is why most languages that have namespaces have some way to create namespace aliases that can be used within smaller scopes.

[4] https://github.com/unclebob/spacewar

Thus, while the length of namespace names is not a function of the scope that contains them, the length of their aliases is. In a very short module, we might use the alias `ts`, whereas in a longer module, we might use `tac-scan`, and in a very long module (which ought to be rare), we might use `tactical-scan`.

Avoid Encodings

We have enough encodings to deal with without adding more to our burden. Encoding type or scope information into names simply adds an extra burden of deciphering. It hardly seems reasonable to require each new employee to learn yet another encoding “language” in

addition to learning the (usually considerable) body of code that they'll be working in. On top of that, encoded names are seldom pronounceable and are easy to mistype.

In days of old, when we worked in name length–challenged languages, we violated this rule out of necessity and with regret. Fortran forced encodings by making the first letter a code for the type. Early versions of BASIC allowed only a letter plus one digit. Hungarian notation[5] (HN) took this to a whole new level.

[5] So named because it was invented by Charles Simonyi, a Hungarian-born architect at Microsoft during the DOS era.

HN was considered to be pretty important back in the Windows C API, when everything was an integer handle or a long pointer or a void pointer, or one of several implementations of "string" (with different uses and attributes). The C compiler did not check types in those days, so the programmers needed a crutch to help them remember the types.

In modern languages, we have much richer type systems, and the compilers remember and enforce the types. What's more, there is a trend toward smaller classes and shorter functions so that people can usually see the point of declaration of each variable they're using.

Today's programmers get enough help from their compilers and IDEs that they don't usually need to encode types within their variable names. So nowadays, HN and other forms of type encoding are simply impediments. They make it harder to change the name or type of a variable, function, or class. They make it harder to read the code. And they create the possibility that the encoding system will mislead the reader.

```
PhoneNumber phoneString;

// name not changed when type changed!
```

Member Prefixes

Prefixing member variables with something like `m_` is another form of type encoding that is no longer necessary. Your classes and functions should be small enough that you don't need such a crutch. You should be using an editing environment that highlights or colorizes members to make them distinct.

```
public class Part {
  private String m_dsc; // The textual description
  void setName(String name) {
    m_dsc = name;
  }
}
```

```
public class Part {
  String description;
  void setDescription(String description) {
    this.description = description;
  }
}
```

Interfaces and Implementations

We do not recommend prefixing interfaces with `I`, because that's just another kind of type encoding. For example, say you are building an Abstract Factory[6] for the creation of `Shape`s. This factory will be an interface and will be implemented by a concrete class. What should we name these classes? `IShapeFactory` and `ShapeFactory`? We prefer to leave interfaces unadorned. The preceding `I` is a distraction at best and too much information at worst. After all, the whole point of an interface is that users don't need to know it's an interface. We just want them to know that it's a `ShapeFactory`. So, if we must encode either the interface or the implementation, we prefer to add a suffix to the implementation; for example, `ShapeFactoryImp`.

[6] [GOF95].

Use Appropriate Parts of Speech

Some elements of our code are best represented by nouns, and others by verbs.

Class Names

Classes and objects should generally have noun or noun phrase names like `Customer`, `WikiPage`, `Account`, and `AddressParser`. Generally the noun should be singular and not plural. A class that represents customers should be named `Customer` and not `Customers`. On the other hand, a class that holds many `Customer` instances probably *should* be called `Customers`.

Avoid noise words like `Manager`, `Processor`, `Data`, or `Info` in the name of a class. We already know that classes manage and process data and info—you don't need to tell us.

Method Names

Methods should generally have verb or verb phrase names like `postPayment`, `deletePage`, or `save`.

Depending on your language, accessors, mutators, and predicates should be named according to the local convention. In Java, they are named for their value and prefixed with `get`, `set`, and `is` according to the JavaBeans standard.[7]

[7] http://java.sun.com/products/javabeans/docs/spec.html

There is a de facto convention in some languages to name a function after the value it returns, that is, as a noun. For example, the function that returns the name of a customer might be called `name`. We do not prefer this convention, but we will follow it when working with teams that use it.

In most cases, it's best if a mutator is named for the mutation it causes, such as `add`, `remove`, `sort`, and so on. These tend to be verb names, and so the standard advice is to "name methods with verb names."

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())…
```

C++, Java, and C# have constructors that take the name of the class. This means you cannot give the constructor a meaningful name. When such constructors are overloaded, it is better to use static factory methods with names that describe the arguments. For example:

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

is generally better than

```
Complex fulcrumPoint = new Complex(23.0);.
```

Consider enforcing their use by making the corresponding constructors private.

Consider Keyword Parameters

If your language allows keyword or hash map parameters, such as Python, C#, Ruby, and Clojure do, you may find it best to limit the positional parameters of a function and use explicit keyword-only parameters for most variations.

```
def sort(iterable, *, key=None, reverse=False ) :
  … # body elided
pets_sorted = sort(pets)
pets_reversed = sort(pets, reverse=True)
pets_by_age = sort(pets, key=lambda x: x.birthdate() )
```

Don’t Be Cute

If names are too clever, they will be memorable only to people who share the author’s sense of humor, and only as long as these people remember the joke. Will they know what the function named `HolyHandGrenade` is supposed to do? Sure it’s cute, but maybe in this case `DeleteItems` might be a better name. Choose clarity over entertainment value.

Cuteness in code often appears in the form of colloquialisms or slang. For example, don’t use the name `whack()` to mean `kill()`. Don’t tell little culture-dependent jokes like `eatMyShorts()` to mean `abort()`.

Say what you mean. Mean what you say.

Pick One Word per Concept

Pick one word for one concept, and stick with it. For example, don't use `amount` in one module, `fee` in another, `price` in yet another, and `cost` in still another. Pick a name for each concept and use that name throughout. Make sure that everyone on the team knows those names.

We will often audit the names in our systems for consistency. This is a small task, and often an IDE will have the option to provide an "outline view" or "object view," or else "fold/collapse all functions" so that you can look at method and variable names without scrolling. We highly recommend harmonizing all the names in a given class, module, or test suite. You will find it satisfying, we have no doubt.

A consistent lexicon (A Ubiquitous Language[8]) is a great boon to the programmers who must use your code.

[8] [DDD].

Use Solution Domain Names

Remember that the people who read your code will be programmers. They will be familiar with solution domain terms.

When you are underneath the domain level and into pure implementation, go ahead and use computer science (CS) terms, algorithm names, pattern names, math terms, and so forth. It is not wise to draw every name from the problem domain, because we don't want our coworkers to have to run back and forth to the customer asking what every name means when they already know the concept by a different name.

The name `AccountVisitor` means a great deal to a programmer who is familiar with the Visitor[9] pattern. What programmer would not know what a `JobQueue` or `FIFO` is?

[9] [GOF95].

There are lots of very technical things that programmers have to do. Choosing technical names for doing technical things is usually the most appropriate course.

Programs are often best described using programming. You don't have to make everything look like either English prose or

mathematical formulae. Use the level of abstraction that works best for the algorithm and the audience.

Use Problem Domain Names

Use names from the problem domain. They are likely to be familiar and help set the context for developers on your team. If not, then at least the programmer who maintains your code can ask a domain expert what it means.

Separating solution and problem domain concepts is part of the job of a good programmer and designer. The code that has more to do with problem domain concepts should have names drawn from the problem domain. We don't force solution domain names on the problem, nor problem domain names on the solutions.

Add Meaningful Context

There are a few names that are meaningful in and of themselves—most are not. Instead, you need to place names in context for your reader by enclosing them in well-named classes, functions, or namespaces.

Imagine that you have variables named `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state`, and `postcode`. Taken together, it's pretty clear that they form an address. But what if you just saw the `state` variable being used alone in a method? Would you automatically infer that it was part of an address, or would you infer that you are in a finite state machine and tracking which state that machine was currently in?

You can remove this ambiguity by creating a class or data structure named `Address`. Then, even the compiler will know that the variables belong to a bigger concept.

Programmers sometimes fall into the code smell of Primitive Obsession,[10] avoiding the creation of classes, structures, and module for too long. It is much better, for example, to work with `GlobalLocations` than to deal with primitive variables for `latitude` and `longitude` in hours, minutes, and seconds. It is far easier to work with two "locations" than with 12 floating-point integers, all similarly named.

[10] One of the code smells in [Refactoring].

As another example, consider the following Python code. Do the variables need a more meaningful context? The function name provides only part of the context; the algorithm provides the rest.

```
private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
      number = "no";
      verb = "are";
      pluralModifier = "s";
    } else if (count == 1) {
      number = "1";
      verb = "is";
      pluralModifier = "";
    } else {
      number = Integer.toString(count);
      verb = "are";
      pluralModifier = "s";
    }
    String guessMessage = String.format(
      "There %s %s %s%s", verb, number, candidate, pluralModifier
    );
    print(guessMessage);
  }
```

Once you read through the function, you see that the three variables—`number`, `verb`, and `pluralModifier`—are part of the “guess statistics” message. Unfortunately, the context must be inferred. When you first look at the method, the meanings of the variables are opaque.

The function is a bit too long and the variables are used throughout. To split the function into smaller pieces we need to create a `GuessStatisticsMessage` class and make the three variables fields of this

class. This provides a clear context for the three variables. They are definitively part of the `GuessStatisticsMessage`. The improvement of context also allows the algorithm to be made much cleaner by breaking it into many smaller functions.

```
public class GuessStatisticsMessage {
  private String number;
  private String verb;
  private String pluralModifier;
  public String make(char candidate, int count) {
    createPluralDependentMessageParts(count);
    return String.format(
      "There %s %s %s%s",
       verb, number, candidate, pluralModifier );
  }
  private void createPluralDependentMessageParts(int count) {
    if (count == 0) {
      thereAreNoLetters();
    } else if (count == 1) {
      thereIsOneLetter();
    } else {
      thereAreManyLetters(count);
    }
  }
  private void thereAreManyLetters(int count) {
    number = Integer.toString(count);
    verb = "are";
    pluralModifier = "s";
  }
  private void thereIsOneLetter() {
    number = "1";
    verb = "is";
    pluralModifier = "";
  }
```

```
  private void thereAreNoLetters() {
    number = "no";
    verb = "are";
    pluralModifier = "s";
  }
}
```

Don’t Add Gratuitous Context

In an imaginary application called Gas Station Deluxe, it is a bad idea to prefix every class with `GSD`. Frankly, you are working against your tools. You type `G` and press the completion key and are rewarded with a mile-long list of every class in the system. Is that wise? Why make it hard for the IDE to help you?

Likewise, say you invented a `MailingAddress` class in `GSD`’s `accounting` module, and you named it `GSDAccountAddress`. Later, you need a mailing address for your customer contact application. Do you use `GSDAccountAddress`? Does it sound like the right name? Ten of 17 characters are redundant or irrelevant.

The names `accountAddress` and `customerAddress` are fine names for instances of the class `Address` but could be poor names for classes. `Address` is a fine name for a class. If we need to differentiate between MAC addresses, port addresses, and Web addresses, we might consider `PostalAddress`, `MAC`, and `URI`. The resulting names are more precise, which is the point of all naming.

Final Words

The hardest thing about choosing good names is that it requires good descriptive skills and a shared cultural background. It’s not a self-pleasing, universal, learn-one-time-and-run-on-autopilot kind of skill.

Each time we change codebases we have to start the process over. Who are the people who comprise our audience now? What does “everyone know” in this space? What is familiar and what seems “weird” or “alien” here?

There is a degree of subjectivity here, as you write for an audience. What you like personally may not be what they need and understand. As a result, many programmers don't learn to do it very well. It requires contact with your user base, just like any good software development requires contact with real users.

Since it's hard to get it right (even with these fine guidelines), we often have to iterate on our software names. Sometimes other developers are surprised that we, after having written so many times and so many words on naming, will often call a variable by an intentionally bad name temporarily, until a better name comes to mind.

We rename things. As Josh Kerievsky reminds us, *all good writing is based on revision*.

People may defer renaming things for fear that some other developers will object. We do not share that fear and find that we are actually grateful when names change (for the better).

Most of the time we don't memorize the names of classes and methods and demand that they remain the same. Memorization is a waste of our time. We use modern tools to deal with details like that. We focus more on whether the code reads like paragraphs and sentences, or at least like equations, tables, and data structures.

You may surprise someone with a name change, just like you might with any other code improvement. Don't let it stop you in your tracks.

Try these guidelines to see if they help in your situation.

So far, they seem to help most people most of the time. We're confident that you will also find help here.

Chapter 5. Comments



“Don’t comment bad code—rewrite it.”

—Brian W. Kernighan and P. J. Plaugher[1]

[1] [KP78], p. 144.

“Comments are the distracting footnotes of code.”

—Jeff Langr

Nothing can be quite so helpful as a well-placed comment. Nothing can clutter up a module more than frivolous dogmatic comments. Nothing can be quite so damaging as an old crufty comment that propagates lies and misinformation.

Compensating for Failure

Comments are, at best, a necessary evil. If our programming languages were expressive enough, or if you and I had the talent to subtly wield those languages to express our intent, we would not need comments very much—perhaps not at all.

The proper use of comments is to compensate for our failure to express ourselves in code. Note that I used the word *failure*. I meant it. Comments are always failures of either our languages or our abilities.

This doesn’t mean that we never write comments. We must sometimes write comments because we cannot always figure out how to express ourselves without them. However, their use is not a cause for celebration.

When you find yourself in a position where you need to write a comment, think it through and see whether there isn’t some way to turn the tables and express yourself in code. Every time you express yourself in code you should pat yourself on the back. Every time you write a comment you should grimace and feel the failure of our languages and of our ability to express ourselves in those languages.

The comments you write may be necessary; but that necessity is also unfortunate.

It has been said that advice like this gives license to younger programmers to avoid writing comments. Shame on any programmers, young or old, who use these recommendations as an excuse for their laziness and/or unwillingness to add proper explanations and context. In the end, I cannot concern myself with how others might abuse my advice. My sole purpose here is help those who will not.

Hidden or Obscured Comments

Does your IDE hide the first comment in every source file? Mine does. That's because we seldom want to read that comment. We expect that it's some kind of boilerplate license/legalese that offers no insight into the actual intent of the module.

But collapsing the first comment is not the only way that our IDEs hide or obscure comments from us.

What color does your IDE use to display comments? For some of you, it will be a nice green—like the grass. And so, like the grass, you can ignore them. They fade into the background like the neighborhood lawns.

For others of you, it will be a nice soft gray; not invisible, but strongly de-emphasized.

Why? Because we don't want them in the way. We don't really want to see them. We like the fact that they are there; but we don't want them in our face.

I have my IDE paint comments in bright fire-engine red. I do this because I reckon that if someone wrote a comment, I should read it. And then, if that comment does not help me, I delete it on the spot. Or if that comment needs adjusting, I adjust it on the spot.

Comments, when they exist, should be visible!

Lying Comments

Why am I so down on comments? I'm really not. I'm only down on bad comments. I wholly support good comments. The problem, however, is that too many of the comments I come across fall into the former category.

First of all, they often lie. The lies aren't intentional, but they are also not infrequent. The older a comment is, and the farther away it is from the code it describes, the more likely it is to be just plain wrong. The reason is simple. Programmers can't realistically maintain them.

Code will often change and evolve. Chunks of it will move from here to there. Those chunks bifurcate and reproduce and come together again to form hybrids and chimeras. Unfortunately, the comments don't, and can't, always follow them. All too often the comments get separated from the code they describe and become orphaned blurbs of ever-decreasing accuracy. For example, look what has happened to this comment and the line it was intended to describe:

```
MockRequest request;
private final String HTTP_DATE_REGEXP =
  "[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s"+
  "[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT";
private Response response;
private FitNesseContext context;
private FileResponder responder;
private Locale saveLocale;
// Example: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Other instance variables that were probably added later were interposed between the `HTTP_DATE_REGEXP` constant and it's explanatory comment.

You might complain that programmers should be disciplined enough to keep the comments in a high state of repair, relevance, and accuracy. However, I would rather that discipline and energy go toward making the code so clear and expressive that it does not need the comments in the first place.

Inaccurate comments are far worse than no comments at all. They delude and mislead. They set expectations that will never be fulfilled. They lay down old rules that need not, or should not, be followed any longer.

Truth can only be certain in one place: the code. Only the code can be trusted to tell you what it does. It is the most reliable source of accurate information. Therefore, though comments are sometimes necessary, we ought to expend significant energy to minimize them.

Comments That Are Too Intimate

A significant problem with comments is that they are most often written by programmers who have an intimate understanding of the code, and are therefore written from the point of view of that intimacy. Unfortunately, the target reader of the comments does not share that intimacy, and therefore does not have the context to understand the comments.

It has happened to me, far more than once, that I could not understand a comment so helpfully provided by the author until I understood that author's code. The words in the comment simply didn't make sense to me, nor help me in any way. Once I read through the code and understood it, then the intent of the comment became clear—but not before.

Comments Do Not Make Up for Bad Code

One of the more common motivations for writing comments is bad code. We write a module and we know it is confusing and disorganized. We know it's a mess. So we say to ourselves, "Ooh, I'd better comment that!"

No! You'd better *clean* it!

Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments. Rather than spend your time writing the comments that explain the mess you've made, spend it cleaning that mess.

Explain Your Intent in Code

There are certainly times when code makes a poor vehicle for explanation. Unfortunately, many programmers have taken this to mean that code is seldom, if ever, a good means for explanation. This is patently false. Which would you rather see? This:

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```

or this?

```
if (employee.isEligibleForFullBenefits())
```

It takes only a few seconds of thought to explain most of your intent in code. In many cases, it’s simply a matter of creating a function that says the same thing as the comment you want to write.

Good Comments

Some comments are necessary or beneficial. What follows are a few examples of comments that are worthy of the bits they consume—or are at least not worth deleting.

Legal Comments

Sometimes our corporate coding standards force us to write certain comments for legal reasons. For example, copyright and authorship statements are necessary and reasonable things to put into a comment at the start of each source file.

Here, for example, is the standard comment header that we put at the beginning of every source file in FitNesse. I am happy to say that our IDE hides this comment from acting as clutter by automatically collapsing it.

```
// Copyright (C) 2003-2005 by Object Mentor, Inc. All rights reserved.
// Released under the GNU General Public License version 2 or later.
```

Comments like this should not be contracts or legal tomes. Where possible, refer to a standard license or other external document rather than putting all the terms and conditions into the comment.

Informative Comments

It is sometimes useful to provide basic information with a comment. For example, consider this comment that explains the return value of an abstract method:

```
// Returns an instance of the Responder being tested.
protected abstract Responder responderInstance();
```

Why did the author resort to a comment instead of giving the method a better name like `responderBeingTested`? And why would I allow this comment rather than changing the name of the method? That's because the author was using the well-known Singleton[2] design pattern. When you use a design pattern like that, you should follow the canonical form specified by that pattern. In the Singleton pattern, the name of the accessor function always ends in the word `Instance`. So the author decided to fall back on a comment, rather than abandon the canonical form of the pattern.

[2] [GOF95] p. 127.

Here's another case where the comment adds helpful information.

```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
   "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

I would not delete this comment, because regular expressions are always a nightmare to understand. This one is no different, and it was polite of the author to give us a hint with a comment that lets us know that the regular expression is intended to match a timestamp. Any Java programmer would easily infer this from the `kk:mm:ss EEE, MMM dd, yyyy` string that uses the codes from the `SimpleDataFormat` class.

However, the comment is a lie—or at least significantly misinformative. That's because the regular expression would also match the following string: `"::  ,  , "`. And that is definitely not a timestamp. So I would fix the regular expression and let the comment remain.

Explanation of Intent

Sometimes a comment goes beyond just useful information about the implementation and provides the intent behind a decision. In

the following case, we see an interesting decision documented by a comment. When comparing two objects, the author decided that he wanted to sort objects of the containing class higher than objects of any other.

```
public int compareTo(Object o)
{
  if(o instanceof WikiPagePath)
  {
    WikiPagePath p = (WikiPagePath) o;
    String compressedName = StringUtil.join(names, "");
    String compressedArgumentName = StringUtil.join(p.names, "");
    return compressedName.compareTo(compressedArgumentName);
  }
  return 1; // we are greater because we are the right type.
  }
```

Here’s an even better example. You might not agree with the programmer’s solution to the problem, but at least you know what he was trying to do.

```
public void testConcurrentAddWidgets() throws Exception {
  WidgetBuilder widgetBuilder = . . .

    //This is our best attempt to get a race condition
    //by creating large number of threads.
    for (int i = 0; i < 25000; i++) {
      WidgetBuilderThread widgetBuilderThread =
        new WidgetBuilderThread(widgetBuilder, text, parent, failFlag)
      Thread thread = new Thread(widgetBuilderThread);
      thread.start();
    }
    assertEquals(false, failFlag.get());
  }
```

I would not delete that comment as part of a code review. However, I'd be tempted to get rid of it by extracting that `for` loop out into a function named `attemptRaceCondition`.

> Future Bob**:** As an aside, this is a terrible way to force a race condition. When you start up a bunch of threads in hopes that they will race, what they will do instead is line up single file and execute without racing. If you want to force a race condition, you either have to set the race up with semaphores, or introduce random delays and hope that the threads will slip past one another and accidentally enter a race configuration.

Clarification

Sometimes it is just helpful to translate the meaning of some obscure argument or return value into something that's readable. In general, it is better to find a way to make that argument or return value clear in its own right; but when it's part of the standard library, or in code that you cannot alter, a helpful clarifying comment can be useful.

I wrote the following comments as part of a feature in FitNesse. I was concerned that the long list of `assertTrue` statements would make my readers' eyes cross. Optical impediments and illusions like this are particularly common in code. So I added the comments to be polite.

```
public void testCompareTo() throws Exception
{
  WikiPagePath a = PathParser.parse("PageA");
  WikiPagePath ab = PathParser.parse("PageA.PageB");
  WikiPagePath b = PathParser.parse("PageB");
  WikiPagePath aa = PathParser.parse("PageA.PageA");
  WikiPagePath bb = PathParser.parse("PageB.PageB");
  WikiPagePath ba = PathParser.parse("PageB.PageA");
```

```
  assertTrue(a.compareTo(a) == 0);    // a == a
  assertTrue(a.compareTo(b) != 0);    // a != b
  assertTrue(ab.compareTo(ab) == 0);  // ab == ab
  assertTrue(a.compareTo(b) == -1);   // a < b
  assertTrue(aa.compareTo(ab) == -1); // aa < ab
  assertTrue(ba.compareTo(bb) == -1); // ba < bb
  assertTrue(b.compareTo(a) == 1);    // b < a
  assertTrue(ab.compareTo(aa) == 1);  // ab > aa
  assertTrue(bb.compareTo(ba) == 1);  // bb > ba
}
```

There is a substantial risk, however, that a clarifying comment is incorrect. In fact, one of the comments in the preceding code is wrong. Go through the previous example and see if you can find it; and note how difficult it is to verify them. That difficulty explains why the clarification was necessary and also why it can be risky.

Warning of Consequences

Sometimes it is useful to warn other programmers about certain consequences. For example, here is a comment that explains why a particular test case is turned off:

```
// Don't run unless you have some time to kill.
public void _testWithReallyBigFile()
{
  writeLinesToFile(10000000);


  response.setBody(testFile);
  response.readyToSend(this);
  String responseString = output.toString();
  assertSubString("Content-Length: 1000000000", responseString);
  assertTrue(bytesSent > 1000000000);
}
```

```
}
```

Nowadays, of course, we'd turn off the test case by using the `@Ignore` attribute with an appropriate explanatory string: `@Ignore("Takes too long to run")`. But back in the days before JUnit 4, putting an underscore in front of the method name was a common convention. The comment, while flippant, makes the point pretty well.

Here's another, more poignant example:

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
  //SimpleDateFormat is not thread safe,
  //so we need to create each instance independently.
  SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM  yyyy HH:mm:
  df.setTimeZone(TimeZone.getTimeZone("GMT"));
  return df;
}
```

You might complain that there are better ways to solve this problem. I might agree with you. But the comment, as given here, is perfectly reasonable.[3] It will prevent some overly eager programmer from using a static initializer in the name of efficiency.

[3] You've got to wonder how, in the language of the internet, an environment that is intrinsically asynchronous, this function has been allowed to remain thread unsafe for so many decades.

Amplification

A comment may be used to amplify the importance of something that may otherwise seem inconsequential.

A good example of that is the call to `trim()` below. The `trim` function is called so frequently and so frivolously that it's easy to ignore. The comment makes clear that you should not ignore this one.

```
String listItemContent = match.group(3).trim();
// the trim is real important.  It removes the starting
// spaces that could cause the item to be recognized
// as another list.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

Javadocs (and Their Ilk) in Public APIs

Nowadays, most language systems have some kind of documentation system that scrapes special document comments from your source files and arranges them in a nicely formatted document. In Java these are called *Javadocs*.

There is nothing quite so helpful and satisfying as a well-described public API. The Javadocs for the standard Java library are a case in point. It would be difficult, at best, to write Java programs without them.

If you are writing a public API, then you should certainly write good document comments for it. But keep in mind the rest of the advice in this chapter. Document comments can be just as misleading, nonlocal, and dishonest as any other kind of comment.

Bad Comments

Bad comments are usually crutches or excuses for poor code or justifications for insufficient decisions, amounting to little more than the programmer talking to themselves.

When I see comments in this category, I delete them. What follows are some specific examples of such comments.

Mumbling

Plopping in a comment just because you feel you should or because the process requires it is a hack. If you decide to write a

comment, then spend the time necessary to make sure it is the best comment you can write.

Here, for example, is a case I found in FitNesse. It appears to me that the author was in a hurry, or just not paying much attention. His mumbling left behind an enigma:

```
public void loadProperties()
{
  try
  {
   String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
   FileInputStream propertiesStream =
                  new FileInputStream(propertiesPath);
   loadedProperties.load(propertiesStream);
  }
  catch(IOException e)
  {
    // No properties files means all defaults are loaded
  }
}
```

What does that comment in the catch block mean? Clearly it meant something to the author, but the meaning does not come through all that well. Apparently, if we get an `IOException`, it means that there was no “properties” file; and in that case, all the defaults are loaded. But who loads all the defaults? Were they loaded before the call to `loadProperties.load`? Or did `loadProperties.load` catch the exception, load the defaults, and then pass the exception on for us to ignore? Or did `loadProperties.load` load all the defaults before attempting to load the file? Was the author trying to comfort himself about the fact that he was leaving the catch block empty? Or—and this is the scary possibility—was the author trying to tell himself to come back here later and write the code that would load the defaults?

Our only recourse is to examine the code in other parts of the system to find out what's going on here. Any comment that forces you to look in another module for the meaning of that comment has failed to communicate to you and is not worth the bits it consumes.

Wouldn't it have been better if the programmer had written the following?

```
catch(IOException e)
{
  LoadedProperties.loadDefaults();
}
```

Redundant Comments

The following code shows a simple function with a header comment that tells us nothing that the code does not. The comment probably takes longer to read than the code itself.

```
// Utility method that returns when this.closed is true.
// Throws an exception if the timeout is reached.

public synchronized void
waitForClose(long timeoutMillis) throws Exception
{
   if(!closed)
   {
      wait(timeoutMillis);
      if(!closed)
        throw new Exception("MockResponseSender could not be closed");
   }
}
```

What purpose does this comment serve? It's certainly not more informative than the code. It does not justify the code, nor provide

intent or rationale. It is not easier to read than the code. Indeed, it is less precise than the code and entices the reader to accept that lack of precision in lieu of true understanding. It is rather like a glad-handing used-car salesman assuring you that you don’t need to look under the hood.

Misleading Comments

Sometimes, with all the best intentions, a programmer makes a statement in their comments that allows misinterpretation. Consider, for another moment, the badly redundant but also subtly misleading comment we saw in the last example.

Did you discover how the comment was misleading? The method does not return when `this.closed` becomes `true`. It returns if `this.closed` is `true`; otherwise, it waits for a blind timeout and then throws an exception *if* `this.closed` is still not `true`.

This subtle bit of misinformation, couched in a comment that is harder to read than the body of the code, could cause another programmer to blithely call this function in the expectation that it will return as soon as `this.closed` becomes `true`. That poor programmer would then find themselves in a debugging session trying to figure out why their code executed so slowly.

Redundancy and Imprecision

Now consider the legion of useless and redundant Javadocs in this code taken long ago from Tomcat.

```
public abstract class ContainerBase
  implements Container, Lifecycle, Pipeline,
  MBeanRegistration, Serializable {


  /**
   * The processor delay for this component.
   */
  protected int backgroundProcessorDelay = -1;
```

```
/**
 * The lifecycle event support for this component.
 */
protected LifecycleSupport lifecycle =
  new LifecycleSupport(this);


/**
 * The container event listeners for this Container.
 */
protected ArrayList listeners = new ArrayList();


/**
 * The Loader implementation with which this Container is
 * associated.
 */
protected Loader loader = null;


/**
 * The Logger implementation with which this Container is
 * associated.
 */
protected Log logger = null;


/**
 * Associated logger name.
```

```
 */
protected String logName = null;

/**
  * The Manager implementation with which this Container is
  * associated.
  */
protected Manager manager = null;



/**
 * The cluster with which this Container is associated.
 */
protected Cluster cluster = null;



/**
 * The human-readable name of this Container.
 */
protected String name = null;



/**
 * The parent Container to which this Container is a child.
 */
protected Container parent = null;



/**
 * The parent class loader to be configured when we install a
 * Loader.
```

```
     */
    protected ClassLoader parentClassLoader = null;



    /**
     * The Pipeline object with which this Container is
     * associated.
     */
    protected Pipeline pipeline = new StandardPipeline(this);



    /**
     * The Realm with which this Container is associated.
     */
    protected Realm realm = null;



    /**
     * The resources DirContext object with which this Container
     * is associated.
     */
    protected DirContext resources = null;
```

These comments serve only to clutter and obscure the code. They serve no documentary purpose at all. Look at the first one. The variable is named `backgroundProcessorDelay`. The comment is `the processor delay for this component.`.

The redundancy is painful. The variable name says more than the comment. And what in the world did the author mean by the word *component*?

Look further down. The same redundancy glares at us from every one of those Javadocs. On top of that, the author seems to use the words *component* and *container* interchangeably. Is that

intentional? Is there a difference between `component` and `container`? And why are they sometimes capitalized and sometimes not? And look at `parentClassLoader`. Is the lack of `component` or `container` significant?

Perhaps you think I'm being picky. I probably am. But code is precise, and imprecision in the comments can be both confusing and misleading. It also makes me wonder if that imprecision carries over into the code.

Mandated Comments

It is just plain silly to have a rule that says that every function must have a Javadoc, or every variable must have a comment. Comments like this just clutter up the code, propagate misinformation, and lend to general confusion and disorganization.

For example, required Javadocs for every function lead to abominations like this:

```
/**
 * @param title The title of the CD
 * @param author The author of the CD
 * @param tracks The number of tracks on the CD
 * @param durationInMinutes The duration of the CD in minutes
 */
public void addCD(String title, String author,
                  int tracks, int durationInMinutes) {
  CD cd = new CD();
  cd.title = title;
  cd.author = author;
  cd.tracks = tracks;
  cd.duration = duration;
  cdList.add(cd);
}
```

If you ran the Javadoc tool over this code, and then deleted the comments and ran the tool again, the two HTML files created would be nearly identical. So the clutter of this Javadoc adds nothing and serves only to obfuscate the code and create the potential for misdirection.

Journal Comments

Sometimes people add a comment to the start of a module every time they edit it. These comments accumulate as a kind of journal, or log, of every change that has ever been made. I have seen some modules with dozens of pages of these run-on journal entries.

```
* Changes (from 11-Oct-2001)
* --------------------------
* 11-Oct-2001 : Re-organised the class and moved it to new
*               package com.jrefinery.date (DG);
* 05-Nov-2001 : Added a getDescription() method, and
*               eliminated NotableDate class (DG);
* 12-Nov-2001 : IBD requires setDescription() method, now
*               that NotableDate class is gone (DG);
*               Changed getPreviousDayOfWeek(),
               getFollowingDayOfWeek() and
               getNearestDayOfWeek() to correct
*               bugs (DG);
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
* 29-May-2002 : Moved the month constants into a separate
*                interface (MonthConstants) (DG);
*
 27-Aug-2002 : Fixed bug in addMonths() method, thanks to
               N???levka Petr (DG);
* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
* 13-Mar-2003 : Implemented Serializable (DG);
* 29-May-2003 : Fixed bug in addMonths method (DG);
```

```
* 0Sep-2003 : Implemented Comparable.  Updated the isInRange
*             javadocs (DG);
* 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
```

Long ago there was a good reason to create and maintain these log entries at the start of every module. We didn't have source code control systems that did it for us. Nowadays, however, these long journals are just more clutter to obfuscate the module. They should be completely removed. Let the source code control system keep the journal for you.

Noise Comments

Sometimes you see comments that are nothing but noise. They restate the obvious and provide no new information.

```
/**
 * Default constructor.
 */
protected AnnualDateRule() {
}
```

No, really? Or how about this:

```
    /** The day of the month. */
    private int dayOfMonth;
```

And then there's this paragon of redundancy:

```
/**
 * Returns the day of the month.
 *
 * @return the day of the month.
 */
public int getDayOfMonth() {
```

```
  return dayOfMonth;
}
```

These comments are so noisy that we learn to ignore them. As we read through the code, our eyes simply skip over them. Eventually, as the code changes, we fail to notice that the comments have begun to lie.

The first comment below seems appropriate.[4] It explains why the `catch` block is being ignored. But the second comment is pure noise. Apparently the programmer was just so frustrated with writing `try`/`catch` blocks in this function that he needed to vent.

[4] The current trend for IDEs to check spelling in comments will be a balm for those of us who read a lot of code.

```
  private void startSending()
  {
    try
    {
      doSending();
    }
    catch(SocketException e)
    {
      // normal. someone stopped the request.
    }
    catch(Exception e)
    {
      try
      {
        response.add(ErrorResponder.makeExceptionString(e));
        response.closeAll();
      }
      catch(Exception e1)
      {
```

```
      //Give me a break!
    }
  }
}
```

Rather than venting in a worthless and noisy comment, the programmer should have recognized that his frustration could be resolved by improving the structure of his code. He should have redirected his energy to extracting that last `try`/`catch` block into a separate function, like this:

```
private void startSending()
{
  try
  {
    doSending();
  }
  catch(SocketException e)
  {
    // normal. someone stopped the request.
  }
  catch(Exception e)
  {
    addExceptionAndCloseResponse(e);
  }
}

private void addExceptionAndCloseResponse(Exception e)
{
  try
  {
    response.add(ErrorResponder.makeExceptionString(e));
    response.closeAll();
```

```
    }
    catch(Exception e1)
    {
    }
  }
```

Replace the temptation to create noise with the determination to clean your code. You'll find it makes you a better and happier programmer.

Scary Noise

Javadocs can also be noisy. What purpose do the following Javadocs (from a well-known open source library) serve? Answer: nothing. They are just redundant noisy comments written out of some misplaced desire to provide documentation.

```
/** The name. */
private String name;

/** The version. */
private String version;

/** The licenceName. */
private String licenceName;

/** The version. */
private String info;
```

Read these comments again more carefully. Do you see the cut-paste error? If authors aren't paying attention when comments are written (or pasted), why should readers be expected to profit from them?

TODO Comments

In the first edition of this book, I was in favor of leaving `TODO` comments in the code as reminders of things that need to be taken care of. I was enamored of the features in my IDE that allowed me to quickly see all the `TODO` comments in my projects.

```
//TODO-MdM these are not needed
// We expect this to go away when we do the checkout model
protected VersionInfo makeVersion() throws Exception
{
  return null;
}
```

I have since changed my mind about this. After a few years of reading code written by others and by myself, I came to understand that `TODO` means *Don't Do*. So I changed my rule.

I still use `TODO` comments, but I will not check them in. I will either do the thing that needs to be done, or eliminate the reason that it needs to be done, or put it into the backlog of tasks that need to be performed.

Use a Function or a Variable Instead of a Comment

Consider the following stretch of code:

```
// does the module from the global list <mod> depend on the
// subsystem we are part of?
if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))
```

What does this comment mean? What is `<mod>`, and why is it in angle brackets? I put this code into my IDE and extracted out a few explanatory variables. Here is the result.

```
ArrayList moduleDependants = smodule.getDependSubsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependants.contains(ourSubSystem))
```

It seems to me that this refactoring helps the final `if` statement read like well-written prose and is easier to understand than the comment (which I still don't understand).

Position Markers

Sometimes programmers like to mark a particular position in a source file. For example, I recently found this in a program I was looking through:

```
// Actions ////////////////////////////////
```

There are rare times when it makes sense to gather certain functions together beneath a banner like this. But in general, they are clutter that should be avoided—especially the noisy train of slashes at the end.

Think of it this way. A banner is startling and obvious, especially if you don't see banners very often. So use them very sparingly, and only when the benefit is significant. If you overuse banners, they'll fall into the background noise and be ignored—like the little boy who cried "Wolf!"

Attributions and Bylines

```
/* Added by Rick */
```

This is just graffiti. Please don't spray-paint your name all over the source code. Source code control systems are very good at remembering who added what, when. There is no need to pollute the code with little bylines. You might think that such comments would be useful in order to help others know who to talk to about the code. But the reality is that they tend to stay around for years and years, getting less and less accurate and relevant. The same can be said for PR numbers and JIRA tags. Put those in commit comments, not in the source code.

Commented-Out Code

Few practices are as odious as commenting-out code. Commented-out code is an abomination before nature and nature’s God. Don’t check this stuff in!

```
    InputStreamResponse response = new InputStreamResponse();
    response.setBody(formatter.getResultStream(),
                     formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.setContent(reader.read(formatter.getByteCount()));
```

Others who see that commented-out code will be unlikely to have the courage to delete it. They’ll think it is there for a reason and is too important to delete. So commented-out code gathers like dregs at the bottom of a bad bottle of wine. Consider this from Apache Commons:

```
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
writeResolution();
//dataPos = bytePos;
if (writeImageData()) {
    writeEnd();
    this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
}

else {
     this.pngBytes = null;
}
return this.pngBytes;
```

Why are those two lines of code commented? Are they important? Were they left as reminders for some imminent change? Or are

they just cruft that someone commented out years ago and has simply not bothered to clean up?

There was a time, back in the sixties, when commenting-out code might have been useful. But we've had good source code control systems for a very long time now. Those systems will remember the code for us. We don't have to comment it out anymore.

I still comment out code while debugging from time to time. It can be a useful technique. But my rule is to never check in the commented-out code.

HTML Comments

HTML in source code comments can be awful, as you can tell by reading the code below. It makes the comments hard to read in the one place where they should be easiest to read: the code. If comments are going to be extracted by some tool (like Javadoc) to appear in a nicely formatted document, then it should be the responsibility of that tool, and not the programmer, to adorn the comments with appropriate formatting.

```
/**
 * Task to run fit tests.
 * This task runs fitnesse tests and publishes the results.
 * <p/>
 * <pre>
 * Usage:
 * &lt;taskdef name=&quot;execute-fitnesse-tests&quot;
 *     classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quot;
 *     classpathref=&quot;classpath&quot; /&gt;
 * OR
 * &lt;taskdef classpathref=&quot;classpath&quot;
 *              resource=&quot;tasks.properties&quot; /&gt;
 * <p/>
 * &lt;execute-fitnesse-tests
```

```
*     suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot;
*     fitnesseport=&quot;8082&quot;
*     resultsdir=&quot;${results.dir}&quot;
*     resultshtmlpage=&quot;fit-results.html&quot;
*     classpathref=&quot;classpath&quot; /&gt;
* </pre>
*/
```

Nonlocal Information

If you must write a comment, then make sure it describes the code it appears near. Don’t offer systemwide information in the context of a local comment. Consider, for example, the Javadoc comment below. Aside from the fact that it is horribly redundant, it also offers information about the default port. And yet the function has absolutely no control over what that default is. The comment is not describing the function, but some other, far-distant part of the system. Of course, there is no guarantee that this comment will be changed when the code containing the default is changed.

```
/**
 * Port on which fitnesse would run. Defaults to 8082.
 *
 * @param fitnessePort
 */
public void setFitnessePort(int fitnessePort)
{
  this.fitnessePort = fitnessePort;
}
```

Too Much Information

Don’t put interesting historical discussions or irrelevant descriptions of details into your comments. The comment below

was extracted from a module designed to test that a function could encode and decode base64. Other than the RFC number, someone reading this code has no need for the arcane information contained in the comment.

```
  /*
      RFC 2045 - Multipurpose Internet Mail Extensions (MIME)
      Part One: Format of Internet Message Bodies
      section 6.8.  Base64 Content-Transfer-Encoding
      The encoding process represents 2bit groups of input bits as outpu
      strings of 4 encoded characters. Proceeding from left to right, a
      2bit input group is formed by concatenating 3 8-bit input groups.
      These 24 bits are then treated as 4 concatenated 6-bit groups, eac
      of which is translated into a single digit in the base64 alphabet.
      When encoding a bit stream via the base64 encoding, the bit stream
      must be presumed to be ordered with the most-significant-bit first
      That is, the first bit in the stream will be the high-order bit in
      the first 8-bit byte, and the eighth bit will be the low-order bit
      in the first 8-bit byte, and so on.
    */
```

Unobvious Connection

The connection between a comment and the code it describes should be obvious. If you are going to the trouble to write a comment, then at least you'd like the reader to be able to look at the comment and the code and understand what the comment is talking about. Consider, for example, this comment drawn from Apache Commons:

```
 /*
  * start with an array that is big enough to hold all the pixels
  * (plus filter bytes), and an extra 200 bytes for header info
  */
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200]
```

```
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

What is a filter byte? Did the author mean filler byte? Do those bytes relate to the +1? Or to the *3? Both? Is a pixel a byte? Why 200? The purpose of a comment is to explain code that does not explain itself. It is a pity when a comment needs its own explanation.

Function Headers

Short functions don't need much description. A well-chosen name and signature for a small function that does one thing is *usually* better than a comment header. For example, I would prefer:

```
public static int[] generateNPrimes(int n) {
```

over:

```
    /**
     * Computes the first prime numbers; the return value contains the
     * computed primes, in increasing order of size.
     * @param n
     *      How many prime numbers to compute.
     */
    public static int[] generate(int n) {
```

Javadocs in Nonpublic Code

As useful as Javadocs are for public APIs, they are more of an impediment than a benefit to code that is maintained by a small team and is not intended for public consumption. The team already knows the code and is not likely to run the Javadoc tool to generate Javadoc pages, so the extra formality of the Javadoc comments amounts to little more than cruft and distraction.

Example

I wrote the following module for the first XP Immersion. It was intended to be an example of bad coding and commenting style. Kent Beck then refactored this code into a much more pleasant form in front of several dozen enthusiastic students. Later I adapted the example for my book *Agile Software Development, Principles, Patterns, and Practices* and the first of my Craftsman articles published in *Software Development* magazine.

What I find fascinating about this module is that there was a time when many of us would have considered it “well documented.” Now we see it as a small mess. See how many different comment problems you can find.

```
/**
 * This class Generates prime numbers up to a user specified
 * maximum.  The algorithm used is the Sieve of Eratosthenes.
 * <p>
 * Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --
 * d. c. 194, Alexandria.  The first man to calculate the
 * circumference of the Earth.  Also known for working on
 * calendars with leap years and ran the library at Alexandria.
 * <p>
 * The algorithm is quite simple.  Given an array of integers
 * starting at 2.  Cross out all multiples of 2.  Find the next
 * uncrossed integer, and cross out all of its multiples.
 * Repeat until you have passed the square root of the maximum
 * value.
 *
 * @author Alphonse
 * @version 13 Feb 2002 atp
 */
import java.util.*;
public class GeneratePrimes
{
```

```
/**
 * @param maxValue is the generation limit.
 */
public static int[] generatePrimes(int maxValue)
{
  if (maxValue >= 2) // the only valid case
  {
    // declarations
    int s = maxValue + 1; // size of array
    boolean[] f = new boolean[s];
    int i;
    // initialize array to true.
    for (i = 0; i < s; i++)
      f[i] = true;
    // get rid of known non-primes
    f[0] = f[1] = false;
    // sieve
    int j;
    for (i = 2; i < Math.sqrt(s) + 1; i++)
    {
      if (f[i]) // if i is uncrossed, cross its multiples.
      {
        for (j = 2 * i; j < s; j += i)
          f[j] = false; // multiple is not prime
      }
    }
    // how many primes are there?
    int count = 0;
    for (i = 0; i < s; i++)
    {
      if (f[i])
```

```
          count++; // bump count.
      }
      int[] primes = new int[count];
      // move the primes into the result
      for (i = 0, j = 0; i < s; i++)
      {
        if (f[i])             // if prime
          primes[j++] = i;
      }
      return primes;  // return the primes
    }
    else // maxValue < 2
      return new int[0]; // return null array if bad input.
  }
}
```

In the following, you can see a refactored version of the same module. Note that the use of comments is significantly restrained. There are just two comments in the whole module. Both comments are explanatory in nature.

```
/**
 * This class Generates prime numbers up to a user specified
 * maximum.  The algorithm used is the Sieve of Eratosthenes.
 * Given an array of integers starting at 2:
 * Find the first uncrossed integer, and cross out all its
 * multiples.  Repeat until there are no more multiples
 * in the array.
 */
public class PrimeGenerator
{
  private static boolean[] crossedOut;
  private static int[] result;
```

```
public static int[] generatePrimes(int maxValue)
{
  if (maxValue < 2)
    return new int[0];
  else
  {
    uncrossIntegersUpTo(maxValue);
    crossOutMultiples();
    putUncrossedIntegersIntoResult();
    return result;
  }
}

private static void uncrossIntegersUpTo(int maxValue)
{
  crossedOut = new boolean[maxValue + 1];
  for (int i = 2; i < crossedOut.length; i++)
    crossedOut[i] = false;
}

private static void crossOutMultiples()
{
  int limit = determineIterationLimit();
  for (int i = 2; i <= limit; i++)
    if (notCrossed(i))
      crossOutMultiplesOf(i);
}

private static int determineIterationLimit()
{
```

```
  // Every multiple in the array has a prime factor that
  // is less than or equal to the root of the array size,
  // so we don't have to cross out multiples of numbers
  // larger than that root.
  double iterationLimit = Math.sqrt(crossedOut.length);
  return (int) iterationLimit;
}

private static void crossOutMultiplesOf(int i)
{
  for (int multiple = 2*i;
       multiple < crossedOut.length;
       multiple += i)
    crossedOut[multiple] = true;
}

private static boolean notCrossed(int i)
{
  return crossedOut[i] == false;
}

private static void putUncrossedIntegersIntoResult()
{
  result = new int[numberOfUncrossedIntegers()];
  for (int j = 0, i = 2; i < crossedOut.length; i++)
    if (notCrossed(i))
      result[j++] = i;
}

private static int numberOfUncrossedIntegers()
{
```

```
    int count = 0;
    for (int i = 2; i < crossedOut.length; i++)
      if (notCrossed(i))
        count++;
    return count;
  }
}
```

It is easy to argue that the first comment is redundant because it reads very much like the `generatePrimes` function itself. Still, I think the comment serves to ease the reader into the algorithm, so I'm inclined to leave it.

The second comment is almost certainly helpful. It explains the rationale behind the use of the square root as the loop limit. I could find no simple variable name, nor any different coding structure that made this point clear.

Conclusion

Comments are not like Schindler's list—they are not "pure good." They are sometimes utterly essential and other times dangerous and damaging. The only language of "truth" at our disposal is our code. Unfortunately, that language is not always adequate to explain the intention behind our decisions. In those cases, the risk of using an ambiguous, amorphous, and contradictory language like English (or whatever your native language is) can sometimes be worth it. But tread carefully—there dwell dragons.

Chapter 6. Formatting



I wrote this nearly two decades ago. The sample projects I used in my statistical analysis are well into their third, and approaching their fourth, decade. And yet there is little I

needed to change in this chapter. I was tempted to add statistical studies for some other languages, but I decided that would not improve the message of this chapter. Other languages may result in slightly different numbers, but you, gentle reader, are smart enough to account for those minor differences.

—Uncle Bob

When people look under the hood, we want them to be impressed with the neatness, consistency, and attention to detail that they perceive. We want them to be struck by the orderliness. We want their eyebrows to rise as they scroll through the modules. We want them to perceive that professionals have been at work. If instead they see a scrambled mass of code that looks like it was written by a bevy of drunken sailors, then they are likely to conclude that the same inattention to detail pervades every other aspect of the project.

You should take care that your code is nicely formatted. You should choose a set of simple rules that govern the format of your code, and then you should consistently apply those rules. If you are working on a team, then the team should agree to a single set of formatting rules and all members should comply. It helps to have an automated tool that can apply those formatting rules for you.

The Purpose of Formatting

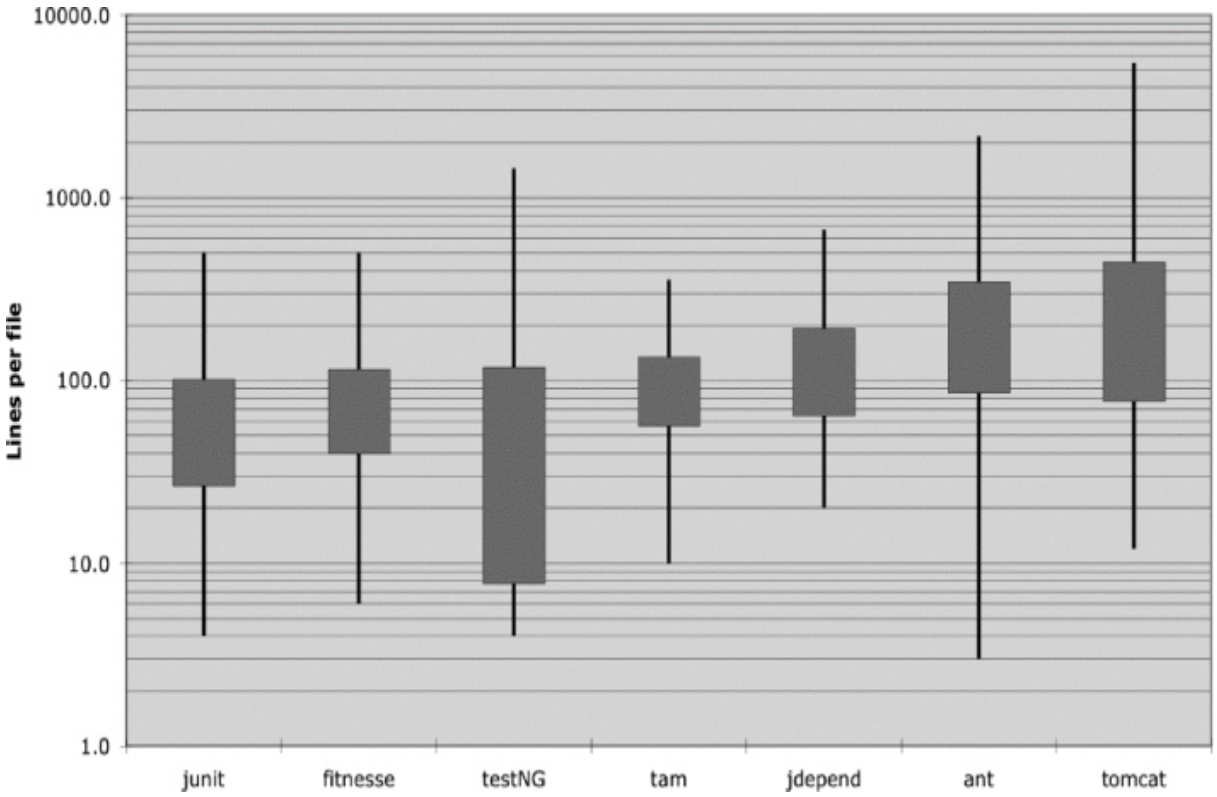
Code formatting is *important*. It is too important to ignore and it is too important to treat religiously. Code formatting is about communication, and communication is the professional developer’s first order of business.

Perhaps you thought that “getting it working” was the first order of business for a professional developer. I hope that this book has begun to disabuse you of that idea. The functionality that you create today has a good chance of changing in the next release, but the readability of your code will have a profound effect on all the changes that will ever be made. The coding style and readability set precedents that continue to affect maintainability and extensibility long after the original code has been changed beyond recognition. Your style and discipline survive, even though your code does not.

So, what are the formatting issues that help us communicate best?

Vertical Formatting

Let’s start with vertical size. How big should a source file be? It turns out that there is a huge range of sizes and some remarkable differences in style. The following shows some of those differences.



Seven different projects are depicted: Junit, FitNesse, TestNG, Time and Money, JDepend, Ant, and Tomcat. The lines through the boxes show the minimum and maximum file lengths in each project. The box shows approximately one-third (one standard deviation[1]) of the files. The middle of the box is the mean. So the average file size in the FitNesse project is about 65 lines, and about one-third of the files are between 40 and 100+ lines. The largest file in *FitNesse* is about 400 lines and the smallest is 6 lines. Note that this is a log scale, so the small difference in vertical position implies a very large difference in absolute size.

[1] The box shows sigma/2 above and below the mean. Yes, I know that the file length distribution is not normal, and so the standard deviation is not mathematically precise. But we're not trying for precision here. We're just trying to get a feel.

Junit, FitNesse, and Time and Money are composed of relatively small files. None are over 500 lines and most of those files are less than 200 lines. *Tomcat* and *Ant*, on the

other hand, have some files that are several thousand lines long, and close to half are over 200 lines.

What does that mean to us? It appears to be possible to build significant systems (at the time of this analysis, FitNesse was close to 50,000 lines) out of files that are typically 200 lines long, with an upper limit of 500. Although this should not be a hard-and-fast rule, it should be considered very desirable. Small files are usually easier to understand and maintain than large files.

Vertical Openness between Concepts

Nearly all code is read left to right and top to bottom. Each line represents an expression or a clause, and each group of lines represents a complete thought. Those thoughts should be separated from each other with blank lines.

Consider the following code. There are blank lines that separate the package declaration, the import(s), and each of the functions. This extremely simple rule has a profound effect on the visual layout of the code. Each blank line is a visual cue that identifies a new and separate concept. As you scan down the listing, your eye is drawn to the first line that follows a blank line.

```
package fitnesse.wikitext.widgets;


import java.util.regex.*;


public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)
    Pattern.MULTILINE + Pattern.DOTALL
```

```
  );

  public BoldWidget(ParentWidget parent, String text) throws Excep
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }

  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Taking those blank lines out has a remarkably obscuring effect on the readability of the code.

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)
    Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws Excep
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
```

```
    addChildWidgets(match.group(1));}
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

This effect is even more pronounced when you unfocus your eyes. In the first example, the different groupings of lines pop out at you, whereas the second example looks like a muddle. The difference between these two examples is just a bit of vertical openness.

Vertical Density

If openness separates concepts, then vertical density implies close association. So lines of code that are tightly related should appear vertically dense. Notice how the useless comments in the code below break the close association of the two instance variables.

```
public class ReporterConfig {
  /**
   * The class name of the reporter listener
   */
  private String m_className;


  /**
   * The properties of the reporter listener
   */
  private List<Property> m_properties = new ArrayList<Property>();
```

```
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
}
```

The code below is much easier to read. It fits in an “eye-full,” or at least it does for me. I can look at it and see that this is a class with two variables and a method, without having to move my head or eyes much. The previous listing forces me to use much more eye and head motion, not to mention the extra processing of the distracting comments, to achieve the same level of comprehension.

```
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);


  }
}
```

Vertical Distance

Have you ever chased your tail through a class, hopping from one function to the next, scrolling up and down the source file, trying to divine how the functions relate and operate, only to get lost in a rat’s nest of confusion? Have

you ever hunted up the chain of inheritance for the definition of a variable or function? This is frustrating because you are trying to understand what the system does, but you are spending your time and mental energy on trying to locate and remember where the pieces are.

Concepts that are closely related should be kept vertically close to each other. Clearly this rule doesn't work for concepts that belong in separate files. But then closely related concepts should not be separated into different files unless you have a very good reason. Indeed, this is one of the reasons that `protected` variables should be avoided.

For those concepts that are so closely related that they belong in the same source file, their vertical separation should be a measure of how important each is to the understandability of the other. We want to avoid forcing our readers to hop around through our source files and classes.

Variable Declarations

Variables should be declared as close to their usage as possible. Because our functions are very short, local variables should appear at the top of each function, as in this longish function from JUnit 4.3.1.

```
private static void readPreferences() {
  InputStream is= null;


  try {
    is= new FileInputStream(getPreferencesFile());
    setPreferences(new Properties(getPreferences()));
    getPreferences().load(is);
  } catch (IOException e) {
    try {
```

```
        if (is != null)
          is.close();
      } catch (IOException e1) {
      }
    }
  }
```

Control variables for loops should usually be declared within the loop statement, as in this cute little function from the same source.

```
  public int countTestCases() {
    int count= 0;
    for (Test each : tests)
      count += each.countTestCases();
    return count;
  }
```

In rare cases, a variable might be declared at the top of a block or just before a loop in a long-ish function. You can see such a variable in this snippet from the midst of a very long function in TestNG.

```
…
for (XmlTest test : m_suite.getTests()) {
      TestRunner tr = m_runnerFactory.newTestRunner(this, test);
      tr.addListener(m_textReporter);
      m_testRunners.add(tr);

      invoker = tr.getInvoker();

      for (ITestNGMethod m : tr.getBeforeSuiteMethods()) {
```

```
      for (ITestNGMethod m : tr.getBeforeSuiteMethods()) {
        beforeSuiteMethods.put(m.getMethod(), m);
      }

      for (ITestNGMethod m : tr.getAfterSuiteMethods()) {
        afterSuiteMethods.put(m.getMethod(), m);
      }
    }
…
```

Instance variables, on the other hand, should be declared at either the top or the bottom of a class. This should not increase the vertical distance of these variables, because in a well-designed class, they are used by many, if not all, of the methods of the class.

There have been many debates over where instance variables should go. In C++, we commonly practiced the so-called *scissors rule*, which put all the instance variables at the bottom. The rationale for this rule is that public items should go at the top and private items should go at the bottom.

This makes a lot more sense than the Java/C# convention of putting private variables at the top. But since this convention is so pervasive in these languages, it is likely better to follow it than to confuse everyone by appealing to logic and common sense.

The important thing is for the instance variables to be declared in one well-known place. Everybody should know where to go to see the declarations.

Consider, for example, the strange case of the `TestSuite` class in JUnit 4.3.1. I have greatly attenuated this class to make the point. If you look about halfway down the listing, you

will see two instance variables declared there. It would be hard to hide them in a better place. Someone reading this code would have to stumble across the declarations by accident (as I did).

```
public class TestSuite implements Test {
  static public Test createTest(Class<? extends TestCase> theCl
                                String name)  {
    …
  }

  public static Constructor<? extends TestCase>
  getTestConstructor(Class<? extends TestCase> theClass)
  throws NoSuchMethodException {
    …
  }

  public static Test warning(final String message) {
    …
  }

  private static String exceptionToString(Throwable t) {
    …
  }

  private String fName;

  private Vector<Test> fTests= new Vector<Test>(10);

  public TestSuite() {
```

```
    }

     public TestSuite(final Class<? extends TestCase> theClass) {
       …
    }

    public TestSuite(Class<? extends TestCase>  theClass, String
      …
    }
    … … … … …
  }
```

Dependent Functions

If one function calls another, they should be vertically close, and the caller should be above[2] the callee, if at all possible. This gives the program a natural flow. If the convention is followed reliably, readers will be able to trust that function definitions will follow shortly after their use. Consider, for example, the snippet from FitNesse below. Notice how the topmost function calls (in bold) those below it. Notice also that the called functions appear in the order that they were called. This makes it easy to find the called functions and greatly enhances the readability of the whole module.

[2] Or below in inverted languages like Clojure.

```
public class WikiPageResponder implements SecureResponder {
  protected WikiPage page;
  protected PageData pageData;
  protected String pageTitle;
```

```
protected Request request;
protected PageCrawler crawler;

public Response makeResponse(FitNesseContext context, Request re
  throws Exception {
  String pageName = getPageNameOrDefault(request, "FrontPage");
  loadPage(pageName, context);
  if (page == null)
    return notFoundResponse(context, request);
  else
    return makePageResponse(context);
}

private String
getPageNameOrDefault(Request request, String defaultPageName)
{
  String pageName = request.getResource();
  if (StringUtil.isBlank(pageName))
    pageName = defaultPageName;
  return pageName;
}

protected void loadPage(String resource, FitNesseContext context
  throws Exception {
  WikiPagePath path = PathParser.parse(resource);
  crawler = context.root.getPageCrawler();
  crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
  page = crawler.getPage(context.root, path);
```

```
    if (page != null)
      pageData = page.getData();
  }

  private Response
  notFoundResponse(FitNesseContext context, Request request)
    throws Exception {
    return new NotFoundResponder().makeResponse(context, request);
  }

  private SimpleResponse makePageResponse(FitNesseContext context)
    throws Exception {
    pageTitle = PathParser.render(crawler.getFullPath(page));
    String html = makeHtml(context);
    SimpleResponse response = new SimpleResponse();
    response.setMaxAge(0);
    response.setContent(html);
    return response;
  }
...
```

As an aside, this snippet provides a nice example of keeping constants at the appropriate level. The `"FrontPage"` constant could have been buried in the `getPageNameOrDefault` function, but that would have hidden a well-known and expected constant in an inappropriately low-level function. It was better to pass that constant down from where knowing it makes sense.

Conceptual Affinity

Certain bits of code want to be near other bits. They have a certain conceptual affinity. The stronger that affinity is, the less vertical distance there should be between them.

As we have seen, this affinity might be based on a direct dependence, such as one function calling another, or a function using a variable. But there are other possible causes of affinity. Affinity might be caused because a group of functions perform a similar operation. Consider this snippet of code from JUnit 4.3.1:

```
public class Assert {
  static public void assertTrue(String message, boolean conditi
    if (!condition)
      fail(message);
  }

  static public void assertTrue(boolean condition) {
    assertTrue(null, condition);
  }

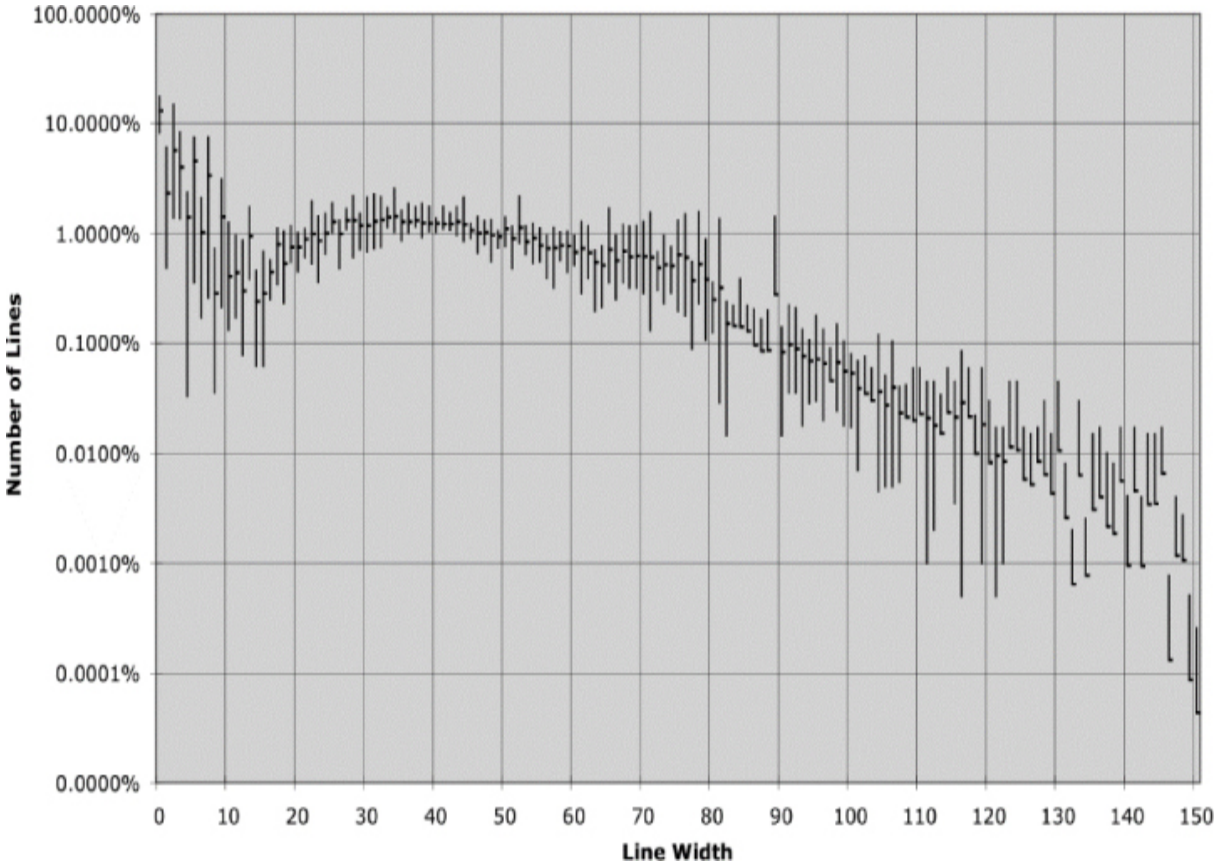
  static public void assertFalse(String message, boolean condit
    assertTrue(message, !condition);
  }

  static public void assertFalse(boolean condition) {
    assertFalse(null, condition);
  }
…
```

These functions have a strong conceptual affinity because they share a common naming scheme and perform variations of the same basic task. The fact that they call each other is secondary. Even if they didn’t, they would still want to be close together.

Horizontal Formatting

How wide should a line be? To answer that, let’s look at how wide lines are in typical programs. Again, we examine the seven different projects. The following image shows the distribution of line lengths of all seven projects. The regularity is impressive, especially right around 45 characters. Indeed, every size from 20 to 60 represents about 1% of the total number of lines. That’s 40%! Perhaps another 30% are less than ten characters wide. Remember, this is a log scale, so the linear appearance of the drop-off above 80 characters is really very significant. Programmers clearly prefer short lines.



This suggests that we should strive to keep our lines short. The old Hollerith limit of 80 is a bit arbitrary, and I'm not opposed to lines edging out to 100 or even 120. But beyond that is probably just careless.

I follow the rule that you should never force your readers to scroll to the right. Given the size and resolution of modern displays I personally set my horizontal limit at 120. I have my IDE draw a soft gray line at that column, and I won't go beyond it.

Horizontal Openness and Density

We use horizontal white space to associate things that are strongly related and disassociate things that are more weakly related. Consider the following function:

```
  private void measureLine(String line) {
```

```
    lineCount++;
    int lineSize = line.length();
    totalChars += lineSize;
    lineWidthHistogram.addLine(lineSize, lineCount);

    recordWidestLine(lineSize);
  }
```

I surrounded the assignment operators with white space to accentuate them. Assignment statements have two distinct and major elements: the left side and the right side. The spaces make that separation obvious.

On the other hand, I didn’t put spaces between the function names and the opening parenthesis. This is because the function and its arguments are closely related. Separating them makes them appear disjoined instead of conjoined. I separate arguments within the function call parenthesis to accentuate the comma and show that the arguments are separate.

Another use for white space is to accentuate the precedence of operators.

```
public class Quadratic {
  public static double root1(double a, double b, double c) {
    double determinant = determinant(a, b, c);
    return (-b + Math.sqrt(determinant)) / (2*a);
  }


  public static double root2(int a, int b, int c) {
    double determinant = determinant(a, b, c);
    return (-b - Math.sqrt(determinant)) / (2*a);
```

```
  }

  private static double determinant(double a, double b, double c)
    return b*b - 4*a*c;
  }
}
```

Notice how nicely the equations read. The factors have no white space between them, because they are high precedence. The terms are separated by white space because addition and subtraction are lower precedence.

Unfortunately, most tools for reformatting code are blind to the precedence of operators and impose the same spacing throughout. So subtle spacings like those shown above tend to get lost after you reformat the code.

Horizontal Alignment

When I was an assembly language programmer,[3] I used horizontal alignment to accentuate certain structures. When I started coding in C, C++, Java, and eventually Clojure, I continued to try to line up all the variable names in a set of declarations, or all the r-values in a set of assignment statements. My code might have looked like this:

[3] Who am I kidding? I still am an assembly language programmer. You can take the boy away from the metal, but you can't take the metal out of the boy!

```
public class FitNesseExpediter implements ResponseSender
{
  private   Socket           socket;
  private   InputStream     input;
  private   OutputStream     output;
```

```
private   Request         request;
private   Response         response;
private   FitNesseContext context;
protected long            requestParsingTimeLimit;
private   long            requestProgress;
private   long            requestParsingDeadline;
private   boolean          hasError;

public FitNesseExpediter(Socket          s,
                         FitNesseContext context) throws Exce
{
  this.context =            context;
  socket =                  s;
  input =                   s.getInputStream();
  output =                  s.getOutputStream();
  requestParsingTimeLimit = 10000;
}
```

I have found, however, that this kind of alignment is not useful. The alignment seems to emphasize the wrong things and leads my eye away from the true intent. For example, in the list of declarations above, you are tempted to read down the list of variable names without looking at their types. Likewise, in the list of assignment statements, you are tempted to look down the list of r-values without ever seeing the assignment operator.

So, in the end, I don’t do this kind of thing anymore. Nowadays, I usually prefer unaligned declarations and assignments, as shown below, because they point out an important deficiency. If I have long lists that need to be

aligned, the problem is the length of the lists, not the lack of alignment. The length of the list of declarations in `FitNesseExpediter` below suggests that this class should be split up.

```
public class FitNesseExpediter implements ResponseSender
{
  private Socket socket;
  private InputStream input;
  private OutputStream output;
  private Request request;

  private Response response;
  private FitNesseContext context;
  protected long requestParsingTimeLimit;
  private long requestProgress;
  private long requestParsingDeadline;
  private boolean hasError;

  public FitNesseExpediter
  (Socket s, FitNesseContext context) throws Exception
  {
    this.context = context;
    socket = s;
    input = s.getInputStream();
    output = s.getOutputStream();
    requestParsingTimeLimit = 10000;
  }
```

Indentation

A source file is a hierarchy rather like an outline. There is information that pertains to the file as a whole, to the individual classes within the file, to the methods within the classes, to the blocks within the methods, and recursively to the blocks within the blocks. Each level of this hierarchy is a scope into which names can be declared and in which declarations and executable statements are interpreted.

To make this hierarchy of scopes visible, we indent the lines of source code in proportion to their position in the hierarchy. Statements at the level of the file, such as most class declarations, are not indented at all. Methods within a class are indented one level to the right of the class. Implementations of those methods are implemented one level to the right of the method declaration. Block implementations are implemented one level to the right of their containing block, and so on.

Programmers rely heavily on this indentation scheme. They visually line up the left edge to see what scope each line appears in. This allows them to quickly hop over scopes, such as implementations of `if` or `while` statements, that are not relevant to their current situation. They scan the left for new method declarations, new variables, and even new classes. Without indentation, programs would be virtually unreadable by humans.

Consider the following programs that are syntactically and semantically identical:

```
public class FitNesseServer implements SocketServer
{ private FitNesseContext context;


public FitNesseServer(FitNesseContext context)
```

```
{ this.context = context; } public void

serve(Socket s) { serve(s, 10000); }
public void serve(Socket s, long requestTimeout) { try

{ FitNesseExpediter sender = new FitNesseExpediter(s, context); se

setRequestParsingTimeLimit(requestTimeout);
sender.start(); } catch(Exception e)

{ e.printStackTrace(); } } }

-----

public class FitNesseServer implements SocketServer {
  private FitNesseContext context;

  public FitNesseServer(FitNesseContext context) {
    this.context = context;
  }

  public void serve(Socket s) {
    serve(s, 10000);
  }

  public void serve(Socket s, long requestTimeout) {
    try {
      FitNesseExpediter sender = new FitNesseExpediter(s, context)
```

```
      sender.setRequestParsingTimeLimit(requestTimeout);
      sender.start();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Your eye can rapidly discern the structure of the indented file. You can almost instantly spot the variables, constructors, accessors, and methods. It takes just a few seconds to realize that this is some kind of simple front end to a socket, with a timeout. The unindented version, however, is virtually impenetrable without intense study.

Breaking Indentation

It is sometimes tempting to break the indentation rule for short `if` statements, short `while` loops, or short functions. Whenever I have succumbed to this temptation, I have almost always gone back and put the indentation back in. So I usually avoid collapsing scopes down to one line, like this:

```
public class CommentWidget extends TextWidget {
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r


  public CommentWidget(ParentWidget parent, String text)
  {super(parent, text);}
```

```
  public String render() throws Exception {return ""; }
}
```

I prefer to expand and indent the scopes instead, like this:

```
public class CommentWidget extends TextWidget {
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r

  public CommentWidget(ParentWidget parent, String text) {
    super(parent, text);
  }

  public String render() throws Exception {
    return "";
  }
}
```

There are times, however, when a simple one-liner is more visually appealing than dangling braces.

```
public int getCount() {return count;}
```

Team Rules

The title of this section is a play on words. Every programmer has their own favorite formatting rules, but if they work in a team, then the team rules.

A team of developers should agree upon a single formatting style, and then every member of that team should use that style. We want the software to have a consistent style. We

don’t want it to appear to have been written by a bunch of disagreeing individuals.

When I started the FitNesse project back in 2002, I sat down with the team to work out a coding style. This took about ten minutes. We decided where we’d put our braces, what our indent size would be, how we would name classes, variables, and methods, and so forth. Then, we encoded those rules into the code formatter of our IDE and have stuck with them ever since. These were not the rules that I prefer; they were rules decided by the team. As a member of that team, I followed them when writing code in the FitNesse project.

Remember, the source code of a good software system is a set of documents that read nicely. They need to have a consistent and smooth style. The reader needs to be able to trust that the formatting gestures they have seen in one source file will mean the same thing in others. The last thing we want to do is add more complexity to the source code by writing it in a jumble of different individual styles.

Uncle Bob’s Formatting Rules

The rules I use personally are very simple and are illustrated by the code below. Consider this an example of how code makes the best coding standard document.

```
public class CodeAnalyzer implements JavaFileAnalysis {
  private int lineCount;
  private int maxLineWidth;
  private int widestLineNumber;
  private LineWidthHistogram lineWidthHistogram;
  private int totalChars;
```

```
public CodeAnalyzer() {
  lineWidthHistogram = new LineWidthHistogram();
}

public static List<File> findJavaFiles(File parentDirectory) {
  List<File> files = new ArrayList<File>();
  findJavaFiles(parentDirectory, files);
  return files;
}

private static void
findJavaFiles(File parentDirectory, List<File> files) {
  for (File file : parentDirectory.listFiles()) {
    if (file.getName().endsWith(".java"))
      files.add(file);
    else if (file.isDirectory())
      findJavaFiles(file, files);
  }
}

public void analyzeFile(File javaFile) throws Exception {
  BufferedReader br = new BufferedReader(new FileReader(javaFile
  String line;
  while ((line = br.readLine()) != null)
    measureLine(line);
}

private void measureLine(String line) {
```

```
  lineCount++;
  int lineSize = line.length();
  totalChars += lineSize;
  lineWidthHistogram.addLine(lineSize, lineCount);
  recordWidestLine(lineSize);
}

private void recordWidestLine(int lineSize) {
  if (lineSize > maxLineWidth) {
    maxLineWidth = lineSize;
    widestLineNumber = lineCount;
  }
}

public int getLineCount() {
  return lineCount;
}

public int getMaxLineWidth() {
  return maxLineWidth;
}

public int getWidestLineNumber() {
  return widestLineNumber;
}

public LineWidthHistogram getLineWidthHistogram() {
  return lineWidthHistogram;
```

```
  }

  public double getMeanLineWidth() {
    return (double)totalChars/lineCount;
  }

  public int getMedianLineWidth() {
    Integer[] sortedWidths = getSortedWidths();
    int cumulativeLineCount = 0;
    for (int width : sortedWidths) {
      cumulativeLineCount += lineCountForWidth(width);
      if (cumulativeLineCount > lineCount/2)
        return width;
    }
    throw new Error("Cannot get here");
  }

  private int lineCountForWidth(int width) {
    return lineWidthHistogram.getLinesForWidth(width).size();
  }

  private Integer[] getSortedWidths() {
    Set<Integer> widths = lineWidthHistogram.getWidths();
    Integer[] sortedWidths = (widths.toArray(new Integer[0]));
    Arrays.sort(sortedWidths);
    return sortedWidths;
  }
}
```

Chapter 7. Clean Functions



In the earliest days of computing, programmers composed their systems of a sequence of routines that performed

individual tasks. They quickly discovered that some of those routines were generally useful, and they wrote them in their notebooks and called them *subroutines*. When needed, they would transcribe those subroutines from their notebooks directly into the program they were creating. There were no calls—those subroutines were simply encoded inline within the program.

Later, in the early ’50s, instructions were added to the machines that would allow subroutines to be called. At first, programmers were reluctant to use such call instructions because they were expensive in time. But as machines grew in capability, the call instructions became less expensive, and programmers began to use them more and more frequently.

Then, in the era of Fortran and PL/1, we started composing our systems primarily of subroutines, and we introduced the special case of *functions* that take arguments and return values. Nowadays, only the function survives. Functions are the first line of organization in any modern program. Writing them well is the topic of this chapter.

Small!

Over the last six decades, I have written functions of all different sizes. In my early years, I wrote several nasty 3,000-line abominations. Later I wrote scads of functions in the 100-to-300-line range. As I matured, I wrote functions that were 20 to 30 lines long. And in the last two decades, I’ve shortened that to a dozen lines or less. What those decades have taught me, through long trial and error, is that functions should be very small.

Of course I am not the first person to say this. The idea of keeping functions small is an old one. In the eighties, it was common to hear that a function should be no bigger than a

screenful. Of course, this was said at a time when VT100 screens were 24 lines by 80 columns, and our editors used 4 lines for administrative purposes. This effectively put the limit at around twenty lines.

In 1999, I went to visit Kent Beck at his home in Oregon. We sat down and did some programming together so that he could show me the TDD[1] discipline. I was a skeptic about TDD and wanted to see it firsthand.

[1] Back in those days, it was called *test-first programming*.

One failing test at a time, we cobbled together a cute little Java/Swing program that he called *Sparkle*. It produced a visual effect on the screen very similar to the magic wand of the fairy godmother in the movie *Cinderella*. As you moved the mouse, the sparkles would drip from the cursor with a satisfying scintillation, falling to the bottom of the window through a simulated gravitational field. As we wrote the code together, making one test pass after another, I was struck by how small the functions were. I was accustomed to functions in Swing programs that took up miles of vertical space. Every function in *this* program was just two, or three, or four lines long. Each had a descriptive name. Each was transparently obvious. Each told a story. And each led you to the next in a compelling order. It was that experience that drove home to me the importance of keeping my functions very small.

Well-Written Prose

Keeping functions very small implies that the blocks within `if` statements, `else` statements, `while` statements, and so on will be only one or two lines long. Often that block will contain a call to another function. Not only does this keep the enclosing function small, but it also adds documentary value because the functions called within the block will have nicely descriptive names. Such `if`/`else`/`while` statements read nicely

because the function names flow well with the keyword. They will read like well-written prose.

```
if(employee.shouldBePaidToday())


  employee.pay();
```

This also implies that small functions will not be large enough to hold deeply nested structures. The indent level of a function will seldom be greater than one or two. This, of course, makes the functions easier to read and understand.

This is not a hard-and-fast rule, and it can be overdone. There are certain functions that read better if they are not decomposed into smaller four-line functions. But these are exceptional cases. In general, keeping functions small is the best strategy.

One Level of Abstraction per Function

In order to make sure our functions are doing “one thing,” we need to make sure that the statements within our functions are all at the same level of abstraction. Look at the following bit of Golang code.[2]

[2] This is from the Video Store example that we’ll be studying much more thoroughly in Chapter 10, “One Thing.”

```
func (customer *Customer)

statement() string {

  totalAmount := 0.0

  frequentRenterPoints := 0

  result := "Rental Record for " + customer.name + "\n"

  for _, rental := range customer.rentals {

    thisAmount := 0.0
```

```
    switch rental.movie.movieType {
    case NewRelease:
      thisAmount += float64(rental.daysRented * 3)
    case Regular:
      thisAmount += 2
      if rental.daysRented > 2 {
        thisAmount += float64(rental.daysRented-2) * 1.5
      }
    case Childrens:
      thisAmount += 1.5
      if rental.daysRented > 3 {
        thisAmount += float64(rental.daysRented-3) * 1.5
      }
    }
    frequentRenterPoints++
    if rental.movie.movieType == NewRelease && rental.daysRented >
      frequentRenterPoints++
    }
    result += fmt.Sprintf("\t%s\t%.1f\n", rental.movie.title, thisA
    totalAmount += thisAmount
  }
  return result + fmt.Sprintf("Amount owed is %.1f\n" +
                              "You earned %d frequent renter points
                              totalAmount, frequentRenterPoints)
}
```

It is easy to see how this violates the rule. There are concepts in there that are at a very high level of abstraction, such as looping over `customer.rentals`. There are other concepts that

are at an intermediate level of abstraction, such as the `switch` statement that selects the movie type. And there are still other concepts that are remarkably low level, such as `frequentRenterPoints++` and the code that calculates prices for each movie type.

Mixing levels of abstraction within a function is usually confusing. Readers may not be able to tell whether a particular expression is an essential concept or a detail. Worse, like broken windows, once details are mixed with essential concepts, more and more details tend to accrete within the function.

Reading Code from Top to Bottom: The Stepdown Rule

We want the code to read like a top-down narrative.[3] We want every function to be followed by those at the next level of abstraction so that we can read the program from top to bottom, descending one level of abstraction at a time. I call this *The Stepdown Rule*. We can see this in the following refactoring of the previous example.

[3] [KP78], p. 37.

```
func (statement *RentalStatement)
makeStatement() string {
  statement.clearTotals()
  return statement.makeHeader() +
         statement.makeDetails() +
         statement.makeFooter()
}


func (statement *RentalStatement)
```

```
clearTotals() {
  statement.totalAmount = 0.0
  statement.frequentRenterPoints = 0
}

func (statement *RentalStatement)
makeHeader() string {
  return "Rental Record for " + statement.name + "\n"
}

func (statement *RentalStatement)
makeDetails() string {
  rentalDetails := ""
  for _, rental := range statement.rentals {
    rentalDetails += statement.makeDetail(rental)
  }
  return rentalDetails
}

func (statement *RentalStatement)
makeDetail(rental *Rental) string {
  amount := rental.determineAmount()
  statement.frequentRenterPoints += rental.determinePoints()
  statement.totalAmount += amount
  return statement.formatDetail(rental, amount)
}

func (statement *RentalStatement)
formatDetail(rental *Rental, amount float64) string {
```

```
  return fmt.Sprintf("\t%s\t%.1f\n", rental.getTitle(), amount)
}


func (statement *RentalStatement)
makeFooter() string {
  return fmt.Sprintf("Amount owed is %.1f\n"+
    "You earned %d frequent renter points",
    statement.totalAmount, statement.frequentRenterPoints)
}
```

To read this we start at the top with the `makeStatement` function. To make a statement we first clear the totals, then make the header, the details, and the footer, and concatenate them all together. All those statements are at the same level of abstraction, one level below the name of the function.

To clear the totals we set `totalAmount` and `frequentRenterPoints` to zero. Again, each line is at the same level of abstraction, one level below the name of the function.

To make the header we format the name from the rental statement with header text. The single line in that function is one level of abstraction below the name of the function.

In the ’70s and ’80s, we called this *functional decomposition*. It was an important part of the *structured programming* discipline. High-level functions are decomposed into lower-level functions, creating a hierarchy of function calls.

When you follow this discipline, your functions will generally be very small because each function contains just one level of abstraction and defers to the next level down through another nicely named function call.

The discipline of refactoring is a critical element of this approach. When we initially write the code for a function, we

tend to do a depth-first search through the levels of abstraction. The code that results from that depth-first search looks like the first version above. But once we get the function working, it is very easy to refactor it by extracting out the different levels of abstraction into nicely named functions.

So, first, make it work. Then, make it right.

Entanglement

One objection to this approach, offered by John Ousterhout, is that when you decompose larger functions into smaller ones, you can sometimes create small functions that are tangled together. Two functions are entangled when, in order to understand one, you have to understand the other.

John is right about this, but he and I differ on the relative cost. I can tolerate a bit of entanglement so long as the functions step down one level of abstraction, and the lower-level functions are positioned after the higher-level functions. John is much more averse to entanglement and would rather two entangled functions be merged into a single function.

Switch Statements

It's hard to make a small `switch` statement.[4] It's also hard to make a `switch` statement that does one thing. By their nature, `switch` statements always do *N* things. Unfortunately, we can't always avoid `switch` statements, but we can make sure that each `switch` statement is buried in a low-level module and is never repeated. We do this, of course, with polymorphism.

[4] And, of course, I include `if/else` chains in this.

Consider, for example, this `switch` statement extracted from the example above.

```
func (rental Rental) determineAmount() float64 {
  amount := 0.0
  switch rental.movie.movieType {
  case NewRelease:
    amount += float64(rental.daysRented * 3)
  case Regular:
    amount += 2
    if rental.daysRented > 2 {
      amount += float64(rental.daysRented-2) * 1.5
  }
  case Childrens:
    amount += 1.5
    if rental.daysRented > 3 {
      amount += float64(rental.daysRented-3) * 1.5
    }
  }
  return amount
}
```

There are several problems with this function.

1. It’s larger than I’d like, and it will grow when new movie types are added.
2. It very clearly does more than one thing.
3. It violates the Single Responsibility Principle[5] (SRP) because there is more than one reason for it to change.

[5] See Chapter 19, “The SOLID Principles.”

4. It violates the Open–Closed Principle[6] (OCP) because it must change whenever new types are added.

[6] Ibid.

But possibly the worst problem with this function is that there are an unlimited number of other functions that will have the same structure. For example, we already have `determinePoints`, and we might later have `applyCoupon` or `getAgeRestriction`. It's not hard to think of many others. All of which would contain a related `switch` statement.

One good solution to this problem is to bury the `switch` statement in the basement of an Abstract Factory,[7] and never let anyone see it. The factory uses the `switch` statement to create appropriate instances of the derivatives of `RentalType`.

[7] [GOF95].

```
type RentalTypeFactory interface {
  make(typeName string) RentalType
}


type RentalTypeFactoryImpl struct{}


func (r RentalTypeFactoryImpl) make(typeName string) RentalType {
  switch typeName {
  case "childrens":
    return ChildrensRental{}
  case "new release":
    return NewReleaseRental{}
  case "regular":
    return RegularRental{}
  }
```

```
  return nil
}
```

My general rule for `switch` statements is that they can be tolerated if they appear only once, are positioned in a concrete module like `main`, are used to create polymorphic objects, and are hidden behind an interface so that the rest of the system can’t see them.

Clean Functions: A Deeper Look

A clean function should have five attributes. It should be nameable, insulated, homogenous, contextual, and pure. If you reorder that, it spells PINCH—but that’s the wrong order in which to present them.

Contextual

The concept and discipline of modularity was not immediately obvious to the earliest of programmers. It slowly grew out of sheer necessity. In the late 1940s, Grace Hopper wrote down common sequences of code in her notebooks to help her program the Harvard Mark 1. She called these sequences *subroutines*. They were transcribed into programs, which were then painstakingly punched onto the long paper tapes that drove the execution of the machine.

Those subroutines were the first modules. From there, the concept of modularity grew into callable subroutines, and finally functions.

By the 1960s, programmers were adopting a discipline called *modular programming*, which stated that the best programs were decomposed into many different modules. By the late ’60s, Dahl and Nygaard advanced the concept of modularity by inventing object-oriented programming.[8] In the early ’70s,

David Parnas continued the advancement by promoting the benefits of information hiding.

[8] Though they did not give it that name. *Object-oriented* was coined by Alan Kay a few years later.

Nowadays, we understand that every function lives within a context. A context is a cooperative grouping of functions and data structures. Every context has an external interface and an internal implementation.

In OO languages, contexts are most commonly defined with classes. In Java, for example, the public methods of a class represent the external interface, while the private data and methods comprise the internal implementation.

In C, we use *.h* header files for the external interface and *.c* files for the internal implementation.

Some languages provide a rich syntax for defining a context. In other languages, the programmer must define a context by informal conventions such as filenames, directories, and comments.

But irrespective of the language and the syntax, every function lives within a context—and it is the programmer's job to identify, create, and manage those contexts.

Some of the functions within a context will be public, meaning that they are known and used outside of that context. They are part of the external interface of the context. Other functions within the context will be private, meaning that they are known and used only within the context and are not used outside of it.

Many OO languages provide a syntax to declare functions either public or private. Some have other categories, like `protected` or `package`, as ways to create partial restrictions. Other languages depend on the discipline of the

programmers and various programmer conventions to achieve the same result.

A context creates a boundary between the things that are known external to the context and those things that remain private within the context. This boundary, specified by the programmer, describes the programmer's intent regarding the things that are higher level and likely won't change very often and those things that are lower level and likely will experience frequent change.

Nameable

There are at least two things to consider when naming a function. The name must be appropriately descriptive within its context, and it must be appropriately convenient.

Descriptive

Every clean function should have a name that describes what that function does. Since the name describes behavior, the name should be a verb, a verb phrase, or an implicit verb, like `Math.sign(x)`.

The description should be slightly more abstract than the code within the function. In other words, the name should hide the implementation.

So, as an absurd counter-example:

```
public static double addAtoBandDivideBy2(double a, double b) {
  return (a+b)/2.0;
}
```

That name certainly tells us what the code does; but it is much too concrete. Not only does it tell us what the function

does, but it tells us how it does it. It exposes the implementation. We should abstract it just a bit.

```
  public static double average(double a, double b) {
    return (a+b)/2.0;
  }
```

That's nice. The name is an implicit verb (to average) that hides the implementation. That's good because any programmer who sees the name will know what to expect; but any programmer who wants to change the implementation will not have to change the name. For example, expanding the function to use a variable number of arguments would not affect the name of the function at all.

```
public static double average(double... ns ) {
  return Arrays.stream(ns).sum()/ns.length;
}
```

A one-word name, like `average`, may work for some functions, but others will need more words in order to be appropriately descriptive. For example:

```
  public void addSalesReceipt(SalesReceipt salesReceipt) {
    salesReceipts.add(salesReceipt);
  }
```

The function appears simple enough, but the name is three words long. Why? Perhaps we should look at the context.

```
public class CommissionedEmployee extends Employee {
  private List<SalesReceipt> salesReceipts = new ArrayList<>();
  //…
  public void addSalesReceipt(SalesReceipt salesReceipt) {
```

```
    salesReceipts.add(salesReceipt);
  }
}
```

This function is a method of a derived class. Derived classes are more detailed than their base classes. Thus, the `addSalesReceipt` function lives in a detailed context. Detailed contexts require more words to describe. Thus, you should expect that, in more detailed contexts, longer names will be necessary in order to be appropriately descriptive.

It is hard to overestimate the value of good names. Remember Ward Cunningham's principle: "You know you are working on clean code when each routine turns out to be pretty much what you expected." Half the battle to achieving that principle is choosing good names for small functions that do one thing.

Don't be afraid to make a name long. A long descriptive name is better than a short enigmatic name. A long descriptive name is often better than a long descriptive comment. Use a naming convention like `CamelCase`, or `snake_case`, or `kebob-case`. This allows multiple words in a function's name to be easily read. Those words should describe what the function does.

Don't be afraid to spend time choosing a name. Indeed, you should try several different names and read the code with each in place. Modern IDEs make it trivial to change names, so experiment with different names until you find one that is as descriptive as you can make it.

Choosing descriptive names will clarify the design of the module in your mind and help you improve it. It is not at all uncommon that hunting for a good name results in a favorable restructuring of the code.

Convenient

In order to be convenient, the name of a function must be short and memorable. But short and memorable names are not always very descriptive. So there is a trade-off to make. The value of convenience depends on how frequently the function is called. A function that is called a lot should have a short and memorable name. A function that is called just once can afford a longer and much more descriptive name.

The two conflicting priorities of descriptiveness and convenience conspire to create the simple heuristic we discussed in Chapter 4.

The length of the name of a function should be inversely proportional to the size of the containing scope.

A scope is the span of code in which a name is known and used. For example, the scope of the **average** function is effectively global. Whereas the scope of `addSalesReceipt` is restricted to the users of the `CommissionedEmployee` class. Thus, the more detailed the context, the smaller the scope. The more general the context, the larger the scope.

Functions in a small scope do not require much convenience. So their names can be longer. This is good because functions in a detailed context usually require longer names to be descriptive. Functions in a large scope require more convenience. This is good because those contexts are usually more general and so do not usually require longer names.

For example, the function `File.open(String filename)` is short and memorable because it lives at the global scope and is very general. We would find it inconvenient if this function were named `openFileAndThrowExceptionIfNotFound`. By the same token, we would find it somewhat disconcerting if the `addSalesReceipt` function were named simply `add`.

So, the bigger the scope, the more general the function and the smaller the name. The smaller the scope, the more detailed the function and the longer the name.

From this, we would predict that global functions will usually have short, one-word names. Public functions within a context will have slightly longer names. Private functions called by the public functions will have even longer names. And private functions called by other private functions may have very long names. The deeper down you go in the function hierarchy, the longer the names will likely be. This is true of Java, C, Clojure, Python, Ruby, or any other language that allows long names.

Insulated

In days long past, we called the notion of insulation *low coupling* or sometimes *encapsulation*. But neither of these terms seem to me to be quite sufficient.

The word *complex* means more than one strand. One of the factors that make functions complex is the number of inputs and outputs they must deal with. Each of those inputs and outputs is a coupling, and each coupling is another strand weaving its way through that function. Therefore, to manage the complexity of our functions, we seek to limit the number of such couplings.

The more arguments a function has, the more it is coupled to its callers. A function with five arguments is likely more coupled than a function with two. Thus, we strive to limit the number of function arguments to some manageable limit. I prefer a limit of three where feasible. I'll talk more about that in the next chapter.

The fewer the arguments, the less coupled the function is and the simpler that function is to call. A function with no

arguments is very easy to call. However, such functions aren't particularly useful unless they live within a context that has been prepared for them by other functions.

For example, you might have a context that represents a finite state machine. That context will contain variables that represent the current state of the machine, as well as the network of state transitions.

Let's use the following state machine. It represents a simple subway turnstile. There are two states: `Locked` and `Unlocked`. There are two events: `Coin` and `Pass`, which represent the user dropping a coin in the slot, and passing through the gate, respectively. Finally, there are five actions: `alarmOff`, `alarmOn`, `lock`, `unlock`, and `thankyou`. The state transition table looks like this.

```
Locked    Coin Unlocked {alarmOff unlock}
Locked   Pass Locked   alarmOn
Unlocked Coin Unlocked thankyou
Unlocked Pass Locked   lock
```

The logic is very simple.

- When the machine is in the `Locked` state and it gets a `Coin` event, it will transition to the `Unlocked` state and invoke the `alarmOff` and `unlock` actions.
- When the machine is in the `Locked` state and it gets a `Pass` event, it will remain in the `Locked` state and invoke the `alarmOn` action.
- When the machine is in the `Unlocked` state and it gets a `Coin` event, it will remain in the `Unlocked` state and invoke the `thankyou`[9] action.

[9] Perhaps it lights up a little "Thank You" light, or emits the words "Thank you" from a speaker.

- When the machine is in the `Unlocked` state and it gets a `Pass` event, it will transition to the `Locked` state and invoke the `lock` action.

From this, it should be obvious that when the system is running, the external software will need to send the `Coin` and `Pass` events. This can be accomplished by calling functions that take no arguments named `coin()` and `pass()`. We can capture this in an interface named `TurnstileEvents`.

```
public interface TurnstileEvents {
  void coin();
  void pass();
}
```

The five actions, which also take no arguments, can be captured in an interface named `TurnstileActions`.

```
public interface TurnstileActions {
  void lock();
  void thankyou();
  void alarmOn();
  void unlock();
  void alarmOff();
}
```

Then, the logic of the state machine can be captured in an abstract class named `TurnstileFSM`.

```
public abstract
class TurnstileFSM implements TurnstileActions, TurnstileEvents {
  private enum State {Locked, Unlocked}
  private enum Event {Coin, Pass}
```

```
private State state = State.Locked;

private void setState(State s) {
  state = s;
}

public void coin() {
  handleEvent(Event.Coin);
}
public void pass() {
  handleEvent(Event.Pass);
}

private void handleEvent(Event event) {
  switch (state) {
    case Locked -> {
      switch (event) {
        case Coin -> {
          setState(State.Unlocked);
          alarmOff();
          unlock();
        }
        case Pass -> {
          setState(State.Locked);
          alarmOn();
        }
      }
    }
    case Unlocked -> {
```

```
        switch (event) {
          case Coin -> {
            setState(State.Unlocked);
            thankyou();
          }
          case Pass -> {
            setState(State.Locked);
            lock();
          }
        }
      }
    }
  }
}
```

The running system that detects and sends events need know only about `TurnstileEvents`. It is completely decoupled from the logic and the actions. The logic of the state machine lives behind a very strong insulation barrier. Nothing outside of `TurnstileFSM` knows about the states or the logic.

Finally, the `TurnstileFSM` class is abstract in order to allow the action functions to be implemented in a derived class, thereby decoupling the logic of the machine from the implementations of the actions.

From this example, you can see the contexts that encapsulate the state machine logic, the events, and the actions. You can see how limiting the number of arguments in the functions that cross the boundaries between the contexts helps keep the coupling very low.

Of course, there will be many times when you will need functions that take one, two, or even three or more

arguments. But once you pass more than three, you have to ask yourself why you aren't passing in an object or data structure that encapsulates them. If you truly need that many arguments passed into a function, it's worth considering whether the data could be separated into separate contexts, and whether the behavior of the function could be split up to live within those contexts.

Homogenous

The lines of a function should all be at the same level of abstraction, which is one level below the name. In the previous section, I called this *The Stepdown Rule*.

The operation of a function takes place at one particular level of abstraction described by the name of that function. The name of the function represents the abstraction that the lines within the function implement.

But what if the lines of a function exist at more than one level of abstraction? For example, consider the following function:

```
public static int orientation(Point a, Point b, Point c) {
  double val = (b.x() - a.x()) * (c.y() - a.y()) -
               (b.y() - a.y()) * (c.x() - a.x());
  if (val == 0) {
    return 0;
  }
  return (val > 0) ? +1 : -1;
}
```

Given three points (a, b, c) on a plane, this function determines whether traveling from a to b to c is a turn to the right (−1), a turn to the left (1), or no turn at all (0).

The first part determines the cross product of the two vectors a→b and b→c. The sign of this product determines the answer.

The determination of the cross product is appropriately one level of abstraction below the name. But the other four lines are lower level than the first. So let’s put those four lines into a function of their own.

```
public static int orientation(Point a, Point b, Point c) {
  double val = (b.x() - a.x()) * (c.y() - a.y()) -
               (b.y() - a.y()) * (c.x() - a.x());
  return sign(val);
}


private static int sign(double n) {
  if (n == 0) {
    return 0;
  }
  return (n > 0) ? +1 : -1;
}
```

Now both lines of the original function are at the same level of abstraction. This makes the implementation of the `orientation` function homogenous.

But we can do better, because the names are not particularly communicative, and the return values are unnamed and easy to get wrong.

```
public enum Chirality {left, right, none};


public static Chirality getChirality(Point a, Point b, Point c) {
  return chirality(crossProduct(a, b, c));
```

```
}

private static double crossProduct(Point a, Point b, Point c) {
  double abx = b.x() - a.x();
  double aby = b.y() - a.y();
  double acx = c.x() - a.x();
  double acy = c.y() - a.y();
  return abx * acy - aby * acx;
}

private static Chirality chirality(double n) {
  if (n == 0) {
    return none;
  }
  return (n > 0) ? left : right;
}
```

You might be bothered that this is twice as long as the original; but it seems to me to be much more readable—especially for someone seeing it for the first time. You might also be bothered that this takes a few more CPU cycles than the original. If you have some desperate need to save 50ns, then by all means trim those cycles back. Just keep in mind that someone else is going to have to maintain this after you, and their time has value—possibly a lot more than 50ns will cost you.

Pure

The behavior of a pure function is dependent upon nothing other than its arguments; and its execution does not cause

the behavior of any other function to change. Following are some of the advantages of pure functions.

- A pure function is a true mapping of arguments to return values—and can be replaced with such a mapping in order to trade execution speed for memory.
- Pure functions are much less susceptible[10] to race conditions and problems of concurrent update because they do not change the state of the underlying system.

[10] They are not perfectly immune, because any stateful system can be simulated with pure functions. See [Functional Design].

- Pure functions are easy to combine into networks of functionality. This is because they have no external dependencies and so are free to be called at any time by any other function.
- Pure functions are easy to test. This is because the test setup does not require the coercion of the underlying system into any special state and the test does not need to assert that the underlying state of the system has been changed.
- Pure functions are easy to distribute between multiple processors. Since the output of a pure function depends only on its arguments, it doesn't matter where, or in which processor, the function executes.

OK, so they're great. How do you create a pure function? Simple. Don't change the value of any variables; or to paraphrase the famous line from *Mommy Dearest*: "No Assignment Statements Ever!" Or to say that in yet another way: Pure functions are immutable.[11]

[11] [Functional Design], Chapter 1.

But how realistic is a rule like that? That depends upon your point of view. For example:

```
public static double sigma(double... ns) {
  var mu = mean(ns);
  var deviations =
    Arrays.stream(ns).map(x->(x-mu)*(x-mu)).boxed().mapToDouble(x->
  double variance = deviations.sum() / ns.length;
  return Math.sqrt(variance);
}
```

The `sigma` function is pure. The assignment statements within it are initializations, not assignments. They do not alter the state of any existing variable.

But what if we turn them into true assignments by splitting the initializations into declarations and assignments?

```
public static double sigma(double... ns) {
  double mu;
  mu = mean(ns);
  DoubleStream deviations;
  deviations =
    Arrays.stream(ns).map(x->(x-mu)*(x-mu)).boxed().mapToDouble(x->
  double variance;
  variance = deviations.sum() / ns.length;
  return Math.sqrt(variance);
}
```

This version of `sigma` is still pure because even though the values of the variables change, those changes are not seen outside of the function. How far can we push this? Let's use a much more traditional[12] implementation:

[12] Of all the `sigma` implementations shown in this chapter, this is my favorite. No goofy syntax; no strange concatenation of dots, parentheses, and arrows; no annoying conversion functions. Just nice, straightforward code.

```
public static double sigma(double... ns) {
  double mu = mean(ns);
  double variance = 0;
  for (double n : ns) {
    var deviation = n - mu;
    variance += deviation * deviation;
  }
  variance /= ns.length;
  return Math.sqrt(variance);
}
```

This version of `sigma` is modifying variables all over the place but it is still pure because all those modifications are hidden from any outside observer. The internals are not pure at all, but to the outside observer, the illusion of purity has been preserved.

We can even go so far as to create a mutable object and still keep the `sigma` function pure.

```
public static double sigma(double... ns) {
  return new SigmaCalculator(ns).invoke();
}


private static class SigmaCalculator {
  private double[] ns;
  private double mu;
  private double variance = 0;
  private double deviation;
```

```
  public SigmaCalculator(double... ns) {
    this.ns = ns;
  }

  public double invoke() {
    mu = mean(ns);
    for (double n : ns) {
      deviation = n - mu;
      variance += deviation * deviation;
    }
    variance /= ns.length;
    return Math.sqrt(variance);
  }
}
```

Now we've got instance variables and all manner of variable manipulations. And yet, the `sigma` function is pure. None of those impure operations are visible outside the `sigma` function.

The bottom line is that purity is an external characteristic of a function, not an internal characteristic. It does not matter how impure the internals of a function are—that function will be pure so long as all the impurity is hidden from all external observers (including other threads).

And, by the way, that is how all functional programming languages claim purity. They just hide the impurity of the underlying implementation.

Partial Purity

As we just saw, purity is an observed quality, not an intrinsic quality. Therefore, whether a function is pure or not depends upon what is being observed.

Consider the C library function `fopen(char* name, char* mode)`. This function has the side effect of leaving the named file open. This means that `fopen` is not pure, because there are functions, such as `fgetc` and `fputc`, that will not behave properly unless `fopen` is called first. Indeed, if you call `fopen` with the same arguments twice in a row without an intervening call to `fclose`, the second invocation will likely fail because the file is already open. So `fopen` is not pure, because of the side effect that leaves the system in an altered state.

However, we can make the `fopen` function appear to be pure, at least to certain observers, by ensconcing it in another function that hides the side effect.

```
void openAndDo(char* fileName, void (*doTo)(FILE*)) {
  FILE* f = fopen(fileName, "r+");
  doTo(f);
  fclose(f);
}
```

This function opens the file, executes the function pointed at by `doTo`, and then closes the file. This puts the system back into the original state, hiding the impurity of `fopen`. For all intents and purposes, this use of `openAndDo` is pure. We might use it as follows:

```
void show(FILE* f) {
  int c;
  while ((c=fgetc(f)) != EOF)
    putchar(c);
}
```

```
int main(int ac, char** av) {
  openAndDo("x.x", show);
}
```

This program prints the `"x.x"` file on standard out. Let’s make this a bit more interesting by modifying the contents of the file. We’ll add a line end, if the file does not end in one already.

```
void appendLineEnd(FILE* f) {
  int c;
  int lastC;
  while ((c = fgetc(f)) != EOF) {
    lastC = c;
  }
  if (lastC != '\n')
    fputc('\n', f);
  }


int main(int ac, char** av) {
  openAndDo("x.x", appendLineEnd);
  openAndDo("x.x", show);
}
```

Now the program adds the line end and then displays the file. This program does leave the system in an altered state, because the contents of the `"x.x"` file have been changed. However, if we do not care to observe the contents of the `"x.x"` file, then the `openAndDo` function is pure as far as we are concerned.

Conclusion

Clean functions are small, do one thing, and read like well-written prose. Each fits into its place in a hierarchy of functions that lead, one step at a time, from high-level abstractions to low-level details. Clean functions exist within a well-defined context, are isolated from each other, are well and conveniently named, and are pure to all external observers.

Chapter 8. Function Heuristics



What follows are some heuristics that I have found helpful in keeping my code clean. I don’t follow them as hard-and-fast rules, but when all else is equal, these are my biases.

Function Arguments

Every argument you add to a function signature makes that function harder to understand. The easiest argument list to understand is the empty argument list.

In the first edition of this book, this statement confused a number of people. Their complaint was that a function with an empty argument list

can do little other than return a constant. So, apparently, some further explanation is needed.

In an object-oriented program, every method of an object has at least one argument. In Python and Go, it is explicitly declared and named `self` by convention. In Java, C#, and C++, it is the implicit `this`. This argument, whether explicit or implicit, represents the enclosing scope.

Do functions in C have an implied first argument? Of course they do; it is the enclosing global scope. A C function always has access to that scope, so it plays the same role as `this` or `self`. Thus, every function ever written has an implied argument that represents the enclosing scope.

Therefore, when I say that the easiest argument list to understand is the empty argument list, I am not counting the implied enclosing scope. The easiest argument list to understand is the argument list with no declared arguments.

Do such methods or functions exist? Of course they do. `Stack.pop()`, `file.close()`, `semaphore.release()`, `buffer.flush()`: They are very common and very easy to read.

The next-easiest argument list to understand is the list with one declared argument. The call `f(x)` is pretty easy to understand. There's no chance you'll get the arguments out of order.

Two arguments aren't too hard. Keeping track of which comes first is pretty easy—I mean, there's only one way to get it wrong. You can make this easier by making the two arguments commutative.

Three arguments starts to get difficult. There are five ways to get them into the wrong order. Yes, I know that the IDEs have gotten quite helpful with this; but still, why tempt fate?

Arguments are couplings and are therefore hard enough as it is without having to worry about their order in the calling sequence. What's more, they are harder to read than they are to write. The IDE will give you a lot of help while writing, but much less while reading. So three is usually my limit.

Yes, a limit of three is arbitrary. You might be comfortable with more. You do you, I'll do me.

If I need to pass more than three arguments into a function, I try to find a way to group them. I might be able to put them into an object—after all, if four things are so cohesive that they can be passed, as a unit, into a function, then why aren't they already an object? Or, I might put them into a hash map with nice names to identify them.

One of the simplest ways to reduce argument lists is to create fields in the current class to hold the values that we would otherwise pass as arguments. Indeed, one of the reasons that OO is a powerful tool is that it allows us to create contexts that allow functions to communicate without passing a lot of arguments.

Variadic Arguments

Sometimes we want to write functions that take an unlimited number of arguments. In Java, the `String.format` function is like that.

```
public String format(String format, Object... args)
```

I think it's safe to assume that this function really just takes two arguments: the `format`, and the array of `Object`s named `args`.

More Than Three?

Sometimes there is a rationale behind the ordering. For example, it is not hard to keep the arguments of `distance(x1,y1,x2,y2)` in order. On the other hand, if there is no rationale, then there are 23 ways to get the order wrong. Even with the help of the IDE, that's a fairly large risk. It's an even bigger risk while reading the code, since you are not likely to notice when the arguments are out of order. So usually, if there are more than three arguments, I find some other means.

Keyword Arguments

Some languages allow you to name the arguments in your function calls. For example, in C# you can prefix your arguments with the name of the argument:

```
createRental(movie: m, daysRented: 3);
```

When using a language feature like that, I don't worry as much about the order of the arguments and might, on occasion, pass more than three.

In some languages, hash maps are so easy to define that they provide a nice way to pass named arguments. For example, in Clojure I can pass a hash map as follows:

```
(create-rental {:movie m, :daysRented 3})
```

Ruby provides a similar bit of syntactic sugar:

```
createRental(movie: m, daysRented: 3)
```

In Ruby, if the `createRental` function is declared to take a hash map, then the arguments will automatically be assumed to be keyword/value pairs.

Flag Arguments

Flag arguments are ugly. Passing a `boolean` into a function is generally careless. It immediately complicates the signature of the function, and loudly proclaims that the function does more than one thing. It does one thing if the flag is `true` and another if the flag is `false`. They are also difficult to read. For example:

```
String monthNames[] = getMonths(false);
```

What does this mean? Are we getting the months or not?

The definition of the function shows us that the `boolean` argument is used to select between long and short month names. But there's no way to know that by looking at the call. The only way to know it is to read the implementation of the function, or at least the ugly Javadoc.

```
/**
 * Returns an array of month names.
 *
 * @param shortened a flag indicating that shortened month names should
 *                  be returned.
 * @return an array of month names.
 */
public static String[] getMonths(final boolean shortened) {

  if (shortened) {
    return DATE_FORMAT_SYMBOLS.getShortMonths();
  } else {
    return DATE_FORMAT_SYMBOLS.getMonths();
  }
}
```

Boolean flags can be pretty confusing if you don't know what they mean. That glaring `true` or `false` is indicating some kind of option; but if you don't

know what that option is, you'll have to click through to the function and read the implementation—and who wants to do that?

The solution, where possible, is to split the function into two functions: one that does the `true` case and one that does the `false` case.

```
String[] getLongMonthNames() {…}

String[] getShortMonthNames() {…}
```

There are, of course, exceptions to this rule. For example, some objects have binary flags within them that are set with a `setFlag(boolean)` method. I'm not going to complain about them.

Output Arguments

Arguments are most naturally interpreted as inputs to a function. But sometimes people will use arguments to catch *output* values. This is often because they want to return more than one value from a function. For example, in C# you can use `out` arguments, in C you can use pointers, and in C++ you can pass references as follows.

```
void stats(double* ns, int n, double& mean, double& sum) {
  sum=0;
  for (int i=0; i<n; i++)
    sum += ns[i];
  mean=sum/n;
}
```

I try to avoid output arguments like this. They violate the principle of least surprise.[1] People generally don't expect an argument to catch an output, so if they see the following call, they may wonder why we are passing the sum and mean into the function.

[1] The Law of Least Astonishment*, PL/1 Bulletin*, 1967.

```
stats(list, listSize, listMean, listSum);
```

Reading a line of code like that can cause the reader to do a double take or to stop and wonder what's going on. That's because they expect data to go into functions through the arguments and out from functions through the return value.

In Go, Python, or Ruby,[2] we could return multiple values from a function.

[2] Ruby returns a mutable array. Caveat emptor.

```
  mean, sum = stats(list);
```

In other languages, we could return an object with two elements.

Or, in all cases, we could create two functions: one for `sum` and the other for `mean`. All else being equal, that's probably the best solution.

In the days before object-oriented programming, it was sometimes necessary to have output arguments. However, much of the need for output arguments disappears in OO languages because the object itself can act as the output argument of its methods. In other words, it might be better for `stats.calculate` to be invoked as follows:

```
stats.calculate(list);
double mean = stats.getMean();
double sum = stats.getSum();
```

Error Codes

Go programmers often use the ability to return multiple values from a function to return error codes along with values:

```
f, err := os.Open("someFile")
if err != nil {
    log.Fatal(err)
}
```

This strategy can also be used in languages with `out` arguments.

```
File f = os.open("someFile", out error);
```

So long as this convention is applied narrowly and consistently, I think it's a reasonable convention; especially in a language like Go that does not have exceptions.

Command Query Separation

CQS is a convention that many programmers adopt when and where appropriate. I've used it, and it is often very helpful. It should be considered where practical.

Under the CQS convention, functions either do something or answer something, but not both. Either a function is a command that creates a side effect (i.e., it changes the state of the system), or it is a query that

returns some information. Doing both can lead to confusion. Consider, for example, the following function:

```
public boolean set(String attribute, String value);
```

This function sets the value of a named attribute and returns `true` if it is successful and `false` if no such attribute exists. This leads to odd statements like this:

```
if (set("username", "unclebob"))...
```

Imagine this from the point of view of the reader. What does it mean? Is it asking whether the "`username`" attribute was previously set to "`unclebob`"? Or is it asking whether the "`username`" attribute was successfully set to "`unclebob`"? It's hard to infer the meaning from the call because it's not clear whether the word *set* is a verb or an adjective.

The author intended `set` to be a verb, but in the context of the `if` statement, it *feels* like an adjective. We could try to resolve this by renaming the `set` function to `setAndCheckIfExists`, but that doesn't much help the readability of the `if` statement. The real solution is to separate the command from the query so that the ambiguity cannot occur.

```
if (attributeExists("username")) {
    setAttribute("username", "unclebob");
    ...
}
```

But, as I said, there are situations where conforming to CQS is impractical or just inconvenient. For example, consider the following:

```
int top = stack.pop();
```

That simple line of code violates CQS. It queries the top of the stack and also changes the state of the stack.

Is there a cost to violating CQS in convenient situations like this? Yes: concurrency. If the `pop` function is not made atomic with the proper use of semaphores or locks, then it could be possible for a race condition to cause the top of the stack to be returned to two different threads before that object was properly removed from the stack.

Those of you who are C++ programmers will recognize that the C++ standard template library does not allow you to query and alter a container

in a single statement. In that library the above `pop` operation requires two statements that conform to CQS:

```
Thing top = myStack.top();
myStack.pop();
```

In C++, the reason for this, aside from the concurrency argument, is that the `Thing` class may override the assignment operator,[3] and it might therefore throw an exception. If the pop operation both changed the stack and returned the top, then when an exception was thrown by the assignment operator it could leave the stack in an unknown state.

[3] That's right, folks, in C++ you can override the = operator for the assignment of objects.

So, although I cannot claim to use CQS in every circumstance, I take the advice seriously and violate it only after considering the possible outcomes.

Prefer Exceptions to Returning Error Codes

Exceptions can be a tricky matter. In most of our modern languages, they are a reliable way to transmit error information up the calling stack. In older languages, most notably C++, that reliability is not guaranteed. And in some languages, like Go, exceptions just don't exist.

Caveat Emptor

In general, I prefer exceptions to error codes where possible. Returning error codes from command functions is a subtle violation of command query separation. It promotes commands being used as expressions in the predicates of `if` statements.

```
if (deletePage(page) == E_OK)
```

This does not suffer from verb/adjective confusion but does lead to deeply nested structures. When you return an error code, you create the problem that the caller must deal with the error immediately.

```
  if (deletePage(page) == E_OK) {
    if (registry.deleteReference(page.name) == E_OK) {
      if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
        logger.log("page deleted");
        return E_OK;
      } else {
```

```
      logger.log("configKey not deleted");
      return E_ERROR;
    }
  } else {
    logger.log("deleteReference from registry failed");
    return E_ERROR;
  }
} else {
  logger.log("delete failed");
  return E_ERROR;
}
```

Programmers sometimes use guards to pretend that the structure is not deeply nested. But the lack of those `else` clauses doesn't mean they aren't actually there.

```
if (deletePage(page) != E_OK) {
  logger.log("delete failed");
  return E_ERROR;
}

if (registry.deleteReference(page.name) != E_OK) {
  logger.log("deleteReference from registry failed");
  return E_ERROR;
}

if (configKeys.deleteKey(page.name.makeKey()) != E_OK){
  logger.log("configKey not deleted");
  return E_ERROR;
}

logger.log("page deleted");
return E_OK;
```

On the other hand, if you use exceptions instead of returned error codes, then the error processing code can be separated from the happy path code

and can be simplified:

```
try {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
  logger.log(e.getMessage());
}
```

Extract `Try/Catch` Blocks

`Try/catch` blocks are ugly in their own right. They confuse the structure of the code and mix error processing with normal processing. So it is better to extract the bodies of the `try` and `catch` blocks out into functions of their own.

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }

}

private void deletePageAndAllReferences(Page page) throws Exception {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
  logger.log(e.getMessage());
}
```

In the above, the `delete` function is all about error processing. It is easy to understand and then ignore. The `deletePageAndAllReferences` function is all about the processes of fully deleting a `page`. Error handling can be ignored. This provides a nice separation that makes the code easier to understand and modify.

Error Handling Is One Thing

As we have learned, functions should do one thing. Error handling is one thing. Thus, a function that handles errors should do nothing else. This implies (as in the example above) that if the keyword `try` exists in a function, it should be the very first word in the function (after, perhaps, some variable declarations) and that there should be nothing after the `catch`/`finally` blocks.

The Dependency Magnet of Error Codes

If you must return error codes, be careful not to turn them into a dependency magnet.

In C, we used to create the header file *error.h*. It would look like this (though it would usually be much longer):

```
#define E_OK 0
#define E_INVALID 1
#define E_NO_SUCH 2
#define E_LOCKED 3
#define E_OUT_OF_RESOURCES 4
```

Every module in the system would `#include` *error.h*. Anytime a new error code was added to *error.h*, every module in the system had to be recompiled—because the build system used the dates of the source files to determine which were out of date and needed recompilation.

This put a negative pressure on the error codes. Programmers didn’t want to add new errors, because then they had to rebuild and redeploy everything, and in those days a rebuild could take hours. So they reused old error codes in “creative” ways instead of adding new ones. And, of course, that led to confusion, disillusion, and heartache.

Modern build environments can be smarter than that. For example, in Java you might use enums in a file named *Error.java*:

```
public enum Error {
   OK,
   INVALID,
   NO_SUCH,
   LOCKED,
   OUT_OF_RESOURCES;
}
```

A file like this is still a dependency magnet because many other classes will import and use it. However, when programmers add a new error code to *Error.java*, only those files that are directly impacted by the change will need to be recompiled and redeployed. Files that simply use the previously existing error codes will not.

So it's a good idea to know how your build system works, and whether dependency magnets will cause thrashing of the build and require redeployment.

In languages like Go, `Error` is an interface, and any error codes that the programmers want to add implement that interface. That means they can be kept in separate source files and run no risk of becoming a dependency magnet.

Of course, this approach can be used in any language that supports interfaces, or an equivalent kind of dynamic polymorphism. For example, error codes can be independent duck-type classes in Python.

DRY: Don't Repeat Yourself

We have long felt that duplication of code should be avoided. We've heard this advice many times over the years. The authors of *The Pragmatic Programmer*[4] told us this way back in 2000. But they were certainly not the first. Most of Codd's normal forms for organizing databases are techniques for reducing duplication in the data. Bertrand Meyer told us this way back in 1988 when he wrote about the Single Choice Principle.[5] Suffice it to say that the advice is old.

[4] [Prag].

[5] [OOSC].

Simple Repeated Code

So, what does it mean to duplicate or repeat code? There's the obvious case where the same *L* lines of code are repeated over and over in *P*

different places. This is often the result of a programmer being a bit overanxious with the mouse. If both *L* and *P* are small, then very little harm is done by the duplication, and the effort to eliminate that duplication may do more harm than good.

For example, if you were to see these two lines repeating twice, it would probably not make much sense to turn them into a function just for the purpose of removing the duplication.

```
count=0
list=[]
```

You might extract them into the `initializeList` function as a convenience to your readers. But that's a different matter. The duplication itself is not very harmful.

What if *L* is likely to grow? For example, what if you need to add a new variable to that initialization?

```
count=0
odds=0
list=[]
```

If *P* is 2, then you have to make that change in two places. That might be a little inconvenient, but it's a minor issue.

On the other hand, if *P* is large, then you'd have to find all the duplicated instances and add the new initialization. If, on the other hand, you had extracted the initialization out into the `initializeList` function, you could make that change in one place.

So, if *P* is large, even if *L* is small, it is probably best to extract the duplication into a single, nicely named function.

What are the odds that a snippet of *L* lines will need modification later? That's probably a function of *L*. The larger *L* is, the larger the odds that the snippet will need to be changed.

Thus, when *L* is large, even if *P* is small, it probably pays to extract the duplication.

We can describe this with this table.

	L is small.	*L* is large.
P is small.	Maybe extract.	Extract
P is large.	Extract	Extract

Similar Code

Often we will find repeated snippets of code that are not perfectly duplicated. They may differ by one or two constants or variables. Such snippets often tend to have a larger *L*, and so should be extracted.

Here's an example from the first edition of this book—and one that was in the FitNesse[6] project circa 2001. The snippets in bold are similar and could be extracted.

[6] http://fitnesse.org; https://github.com/unclebob/fitnesse

```
public class TestableHtml {
  public static String testableHtml(
    PageData pageData,
    boolean includeSuiteSetup) throws Exception {
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute("Test")) {
      if (includeSuiteSetup) {
        WikiPage suiteSetup =
          PageCrawlerImpl.getInheritedPage(
            SuiteResponder.SUITE_SETUP_NAME, wikiPage);
        if (suiteSetup != null) {
          WikiPagePath pagePath =
            suiteSetup.getPageCrawler().getFullPath(suiteSetup);
          String pagePathName = PathParser.render(pagePath);
          buffer.append("!include -setup .")
            .append(pagePathName)
            .append("\n");
        }
      `
```

```
  }
  WikiPage setup =
    PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
  if (setup != null) {
    WikiPagePath setupPath =
      setup.getPageCrawler().getFullPath(setup);
    String setupPathName = PathParser.render(setupPath);
    buffer.append("!include - setup.")
      .append(setupPathName)
      .append("\n");
  }
}
buffer.append(pageData.getContent());
if (pageData.hasAttribute("Test")) {
  WikiPage teardown =
    PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
  if (teardown != null) {
    WikiPagePath tearDownPath =
      teardown.getPageCrawler().getFullPath(teardown);
    String tearDownPathName = PathParser.render(tearDownPath);
    buffer.append("\n")
      .append("!include - teardown.")
      .append(tearDownPathName)

      .append("\n");
  }
  if (includeSuiteSetup) {
    WikiPage suiteTeardown =
      PageCrawlerImpl.getInheritedPage(
        SuiteResponder.SUITE_TEARDOWN_NAME,
        wikiPage
      );
    if (suiteTeardown != null) {
      WikiPagePath pagePath =
        suiteTeardown.getPageCrawler().getFullPath(suiteTeardown);
      String pagePathName = PathParser.render(pagePath);
      buffer.append("!include - teardown.")
```

```
          buffer.append("!include -teardown .")
            .append(pagePathName)
            .append("\n");
        }
      }
    }
    pageData.setContent(buffer.toString());
    return pageData.getHtml();
  }
}
```

Usually the extraction of similar snippets like this simply involves adding an argument or two to the extracted function. Let’s try this.

```
public class TestableHtml {
  public static String testableHtml(
    PageData pageData,
    boolean includeSuiteSetup) throws Exception {
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute("Test")) {
      if (includeSuiteSetup) {
        includeInheritedPage(SuiteResponder.SUITE_SETUP_NAME,
          wikiPage, buffer, "!include -setup .");
      }
      includeInheritedPage("SetUp", wikiPage, buffer,
                           "!include -setup .");
    }
    buffer.append(pageData.getContent());
    if (pageData.hasAttribute("Test")) {
      includeInheritedPage("TearDown", wikiPage, buffer,
                           "\n!include -teardown .");
      if (includeSuiteSetup) {
        includeInheritedPage(SuiteResponder.SUITE_TEARDOWN_NAME,
          wikiPage, buffer, "!include -teardown .");
      }
    }
```

```
    pageData.setContent(buffer.toString());
    return pageData.getHtml();
  }

  private static void
  includeInheritedPage(String pageName, WikiPage wikiPage,
                       StringBuffer buffer,
                       String includeString) {
    WikiPage inheritedPage =
      PageCrawlerImpl.getInheritedPage(pageName, wikiPage);
    if (inheritedPage != null) {
      WikiPagePath pagePath =
        inheritedPage.getPageCrawler().getFullPath(inheritedPage);
      String pagePathName = PathParser.render(pagePath);
      buffer.append(includeString)
        .append(pagePathName)
        .append("\n");
    }
  }
}
```

This is clearly better. It vastly improves the flow of the `testableHtml` function and it makes sure that any modifications to `includeInheritedPage` will be in one place instead of four. However, the number of arguments to our new function is larger than I like. So let’s move some of those variables into fields.

```
public class TestableHtml {

  private static StringBuffer buffer;
  private static WikiPage wikiPage;

  public static String testableHtml(
    PageData pageData,
    boolean includeSuiteSetup) throws Exception {
    wikiPage = pageData.getWikiPage();
    buffer = new StringBuffer();
```

```
    if (pageData.hasAttribute("Test")) {
      if (includeSuiteSetup) {
        includeInheritedPage(SuiteResponder.SUITE_SETUP_NAME,
          "!include -setup .");
      }
      includeInheritedPage("SetUp", "!include -setup .");
    }
    buffer.append(pageData.getContent());
    if (pageData.hasAttribute("Test")) {
      includeInheritedPage("TearDown", "\n!include -teardown .");
      if (includeSuiteSetup) {
        includeInheritedPage(SuiteResponder.SUITE_TEARDOWN_NAME,
          "!include -teardown .");
      }
    }
    pageData.setContent(buffer.toString());
    return pageData.getHtml();
  }

  private static void
  includeInheritedPage(String pageName, String includeString) {
    WikiPage inheritedPage =
      PageCrawlerImpl.getInheritedPage(pageName, wikiPage);
    if (inheritedPage != null) {
      WikiPagePath pagePath =
        inheritedPage.getPageCrawler().getFullPath(inheritedPage);
      String pagePathName = PathParser.render(pagePath);
      buffer.append(includeString)
        .append(pagePathName)
        .append("\n");
    }
  }
}
```

This is nicer still. If you look closely at the result, you ought to be able to detect more duplication. I'll leave the elimination of that to you.

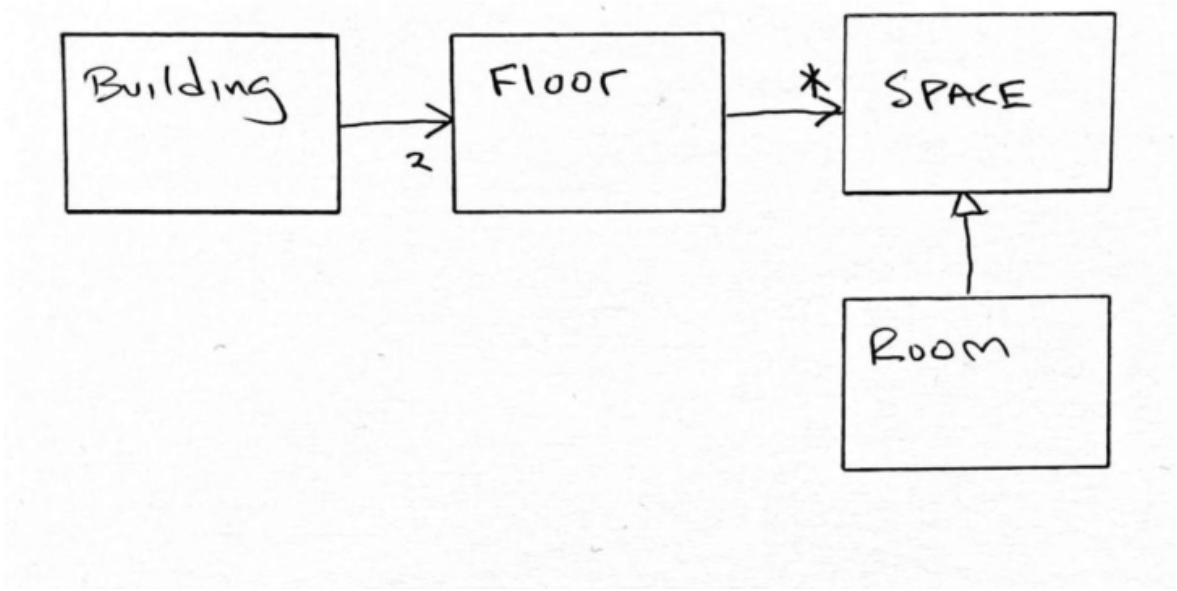
Loop Duplication

Often we find that there are data structures within our application that we must traverse in many different places. The loop that does the traversal is always the same, but the body of that loop is always different.

Nowadays, the popular way to eliminate that duplication is to use a lambda. Older, and sometimes better options are to use the Command pattern, the Strategy pattern, or the TemplateMethod pattern.[7]

[7] [GOF95].

As an example of these techniques, consider a problem my team and I faced way back in the late ’90s. We had a data structure that described the internals of a building. Below is a vastly simplified diagram of that data structure.



Our task was to determine several dozen different scorable features of the building by inspecting each of the floors and spaces (and doors and windows and hallways, but never mind that). The scores for these features were determined by a set of C++ classes, each of which had to walk the data structure in order to perform the calculation.

Below is some Ruby code that shows the duplication that could have resulted from all those features.

```
class SpaceFeature
  def score(building)
    spaces = 0
    building.floors.each do |floor|
```

```
      floor.spaces.each do |space|
        spaces += 1
      end
    end
    spaces
  end
end

class AreaFeature
  def score(building)
    area = 0
    building.floors.each do |floor|
      floor.spaces.each do |space|
        area += space.width * space.length
      end
    end
    area
  end
end
```

You can see the problem. The nested loops are identical, but the body of the loop is different in each case. Now imagine this repeated several dozen times with a much more complicated data structure and traversal loop.

We can address this in modern languages by using a lambda (sometimes called a *block*) as follows.

```
class Feature
  def do_score(building, block)
    value = 0
    building.floors.each do |floor|
      floor.spaces.each do |space|
        value = block.call(value, floor, space)
      end
    end
    value
  end
End
```

```
class SpaceFeature < Feature
  def score(building)
    do_score(building, ->(value, floor, space) {return value+1})
  end
end

class AreaFeature < Feature
  def score(building)
    do_score(building, ->(value, floor, space)
                       {return value + (space.width * space.length)})
  end
end
```

Now the loop that traverses the data structure exists only in the `Feature` class. If any changes were made to that data structure, the odds are that only that class would need to be changed. The specific features, like `SpaceFeature` and `AreaFeature`, would likely remain unaffected by such changes.

Another technique would have been to use the TemplateMethod pattern.

```
class Feature
  def score(building)
    value = 0
    building.floors.each do |floor|
      floor.spaces.each do |space|
        value = score_space(value, floor, space)
      end
    end
    value
  end

  def score_space(value, floor, space)
    raise "Not Implemented"
  end
end
```

```
class SpaceFeature < Feature
  def score_space(value, floor, space)
    value+1
  end
end

class AreaFeature < Feature
  def score_space(value, floor, space)
    value + (space.width * space.length)
  end
end
```

I actually like this better than the lambda solution—and it's the one we used way back in the '90s in C++. Back then we called it "Write a Loop Once" because the *Design Patterns* book had not yet been published.

I'll leave you to check out the Strategy and Command patterns for yourself.

Accidental versus Essential Duplication

So far we have been talking about essential duplication. But sometimes two stretches of code appear to be duplicates, when in fact they are not. Those are accidental duplicates.

Essential duplicates change together. When one changes, they all change. They evolve together.

Accidental duplicates do not change together. They evolve apart from each other. So it is very important not to combine them into a common function.

You can tell an accidental duplication from an essential duplication by considering the Single Responsibility Principle (SRP).[8] If two similar stretches of code are in two modules that are responsible to different actors, then they are likely to evolve separately as those actors ask for different changes. If you extract them into a common function, then you are likely to break the system for one actor while you try to satisfy the other.

[8] See Chapter 19, "The SOLID Principles."

On the other hand, if two similar stretches of code are in modules responsible to the same actor, then they are likely to evolve together. If you

forget to change one of those stretches, then you will break the system for that actor.

Side Effects

In the years since the first edition of this book was written, and with the rise of functional programming, the definition of the term *side effect* has gone through a transition. Twenty years ago most programmers thought that a function with a side effect had a primary purpose but then also did something else on the side. The example I used in the first edition was a function that checked the validity of a login password but then also initialized the current session if the password was found to be valid.

Since then, the definition has been generalized. Nowadays, we define a side effect to be any state change that outlives the function, even if that state change is the primary intent of the function. In other words, a function with a side effect is impure.

So, for example, if `stack.pop()` removes the first item on the top of the `stack`, then it leaves that `stack` in an altered state; the alteration of that state is a side effect.

Side effects create complications for software systems because they introduce an insidious form of coupling called *temporal coupling*—coupling in time. A statement with a side effect separates the code above it from the code below it with a change of state. That change of state means that the order of those lines of code must be preserved.

So you can't close a file until you've opened it, and you can't open it if it's already open. You can't release a semaphore until you've seized it. You can't free a block before you've allocated it. You can't release a graphics context until you've taken possession of it. Et cetera.

Side effect functions often come in pairs, like `open/close`, `seize/release`, or `malloc/free`. One creates the side effect and the other undoes it. So side effects are like The Sith: "always two there are."

We Are Bad at This

The proof that we are bad at managing side effects is the fact that almost all of our major languages now feature *garbage collection*. Garbage collection has been added to our languages to cure the fact that we are just so bloody awful at balancing `malloc` with `free`. We've screwed this up so often, and relegated so many systems to be rebooted every midnight, that we finally created this horrible hack of a feature as a desperate measure to relieve us from the burden of cleaning up memory side effects.

How many languages do you know that have gone to great lengths to help programmers manage the ownership of memory blocks? How many have used reference counting, or RAII,[9] or God knows what? And why? Because we poor, feeble programmers cannot seem to figure out how to return the books we've borrowed from the library.

[9] Resource Acquisition Is Initialization—the key insight behind the old `auto_ptr` (now `unique_ptr` and `shared_ptr`) facilities in C++.

The problem is that facilities like garbage collection only address one of the many Sith-like creatures that we must deal with. We don't have garbage collection for file descriptors, or graphics contexts, or semaphores, or. . . Well, I think you get the point. We are bad at this.

When order matters, temporal couplings can leak out into other parts of the code, creating situations that are as difficult to debug as a memory leak. I'm sure you've had the experience of debugging a misbehaving system only to find that if you swap the order of two adjacent function calls, the system suddenly works—and you have no idea why.

Temporal couplings created by side effects are at the root of all concurrent update problems, all reentrancy problems, all race conditions, and nearly all initialization and finalization problems. Temporal couplings are big trouble. So how do we avoid them?

Functional Languages

You can avoid a lot of side effects by using a functional language. This is not a cure-all. It is still possible to create temporal couplings with functional languages—but most functional languages have facilities that make it very obvious when you are creating them.

The primary mechanism of functional languages that suppresses side effects is the lack of an assignment statement. In a functional language, you are not allowed to change the state of a variable. In fact, there are no variables in functional languages. Named values cannot be varied.

Some of you may have used a functional language before, and may now be ripping this page into little pieces out of frustration. Please relax. Let me explain.

Functional languages allow names to be assigned to values. The source code may look a lot like an assignment statement. However, in a strictly functional language, while you may be able to reassign the name, you cannot overwrite the value.

For example, I found the following in a Clojure program I wrote a while ago.

```
(defn update-clouds-age [ms world]
  (let [clouds (:clouds world)
        decay (Math/pow glc/cloud-decay-rate ms)
        clouds (map #(update % :concentration * decay) clouds)
        clouds (filter #(> (:concentration %) 1) clouds)
        clouds (doall clouds)]
    (assoc world :clouds clouds)))
```

The *name* `clouds` has been assigned and reassigned several times. However, the previous values have not been overwritten—they still exist. The name `clouds` does not represent a slot in memory that has been overwritten by the reassignments. Rather, the four versions of `clouds` refer to four different blocks of memory that retain their values. The name `clouds` does not represent a variable, because nothing has varied.

Now, go get some tape and repair this page.

Why would functional languages prevent overwriting values? Because overwriting is a side effect. It separates the lines above it and the lines below it by a change of state. It creates a temporal coupling. The order matters.

Of course, you can still create code with side effects when using functional languages. Most of the time this is because the language has special facilities for doing so. For example, Clojure has a special data type called an `atom` that represents a *variable* who's value can be changed using a set of special functions.

And, of course, there are still operating system facilities that have side effects. Files still need to be opened and closed, for example.

More generally, it is possible to simulate any system with a functional language—including a system that supports side effects. For example, I could write a Lua execution environment in even the strictest functional language, and then write programs in Lua that had blatant side effects.

Now, this is not a book about functional programming, and so I'm not going to continue this at any length. Suffice it to say that using a functional language can help you manage side effects; but it cannot, and will not, eliminate them.

If you want a much more in-depth discussion of these matters, I suggest you read my book *Functional Design*.[10]

[10] [Functional Design].

Object-Oriented Languages

One of the big complaints that has lately been levied against object-oriented programming languages (OOPLs) is that they promote side effects. It’s easy to see why this complaint has gotten traction. Objects are containers of state, and the methods of objects often change their state, leading to side effects and temporal couplings.

However, what is not often argued is the fact that OOPLs have very good facilities to suppress and hide side effects. For instance, consider this simplistic and contrived example in C++.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class LineCounter {
  public:
   static int count(const char* fileName);
  private:
   static int nLines;
   static ifstream file;

   static void initialize(const char* fileName);
   static void countTheLines();
   static int finalize();
};

int LineCounter::nLines;
ifstream LineCounter::file;

void LineCounter::initialize(const char* fileName) {
  nLines = 0;
  file.open(fileName);
}

void LineCounter::countTheLines() {
```

```
  string line;
  while (getline (file, line))
    nLines++;
}

int LineCounter::finalize() {
  file.close();
  return nLines;
}

int LineCounter::count(const char* fileName) {
  initialize(fileName);
  countTheLines();
  return finalize();
}
```

What I want you to notice in this example is that every one of the private methods in the `LineCounter` class has a side effect; however, the sole public method, which calls them, does not. This means that none of the users of `LineCounter` are exposed to a temporal coupling. Assuming this is a single-threaded application, the side effects are entirely hidden.

In this example, we have used the `LineCounter` as a protected namespace. There is no `LineCounter` object. The class is just a place that allows us to hide the variables and functions that might cause temporal couplings.

Notice that I added the single-threaded constraint. In a multithreaded environment, those static variables could be exposed to concurrent update issues. We could solve this by surrounding the `count` function with a semaphore. Or we could solve it by turning the `LineCounter` class into a real object by getting rid of the static variables and functions and forcing users to create an instance before calling `count`.

The point is that the facilities of most OOPLs provide a handy way to hide side effects behind encapsulation boundaries, thus preventing temporal couplings from leaking out into the application at large.

Structured Programming

In 1968, Edsger Dijkstra wrote a paper that stirred up the programming community for the better part of a decade. Back in those days the ideas he

expressed in that paper were deeply controversial and hotly debated. Some felt that Dijkstra was a fool. Others thought him a god.

But by the time the ’80s rolled around, Dijkstra’s simple idea of Structured Programming was well accepted. By the ’90s, it was hard to find a language that allowed any other style.

The paper, of course, was “Go To Statement Considered Harmful.” It was published as a letter to the editor in the March 1968 issue of the *Communications of the ACM*. The title was given to it by the editor at the time: Niklaus Wirth.[11] And that paper changed the world—of programming.

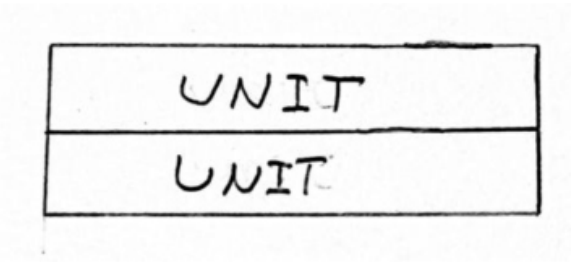
[11] The inventor of the Pascal language.

The idea behind the paper is very simple. All programs should be written using only three control structures: sequence, selection, and iteration.

This was revolutionary at the time because most programming languages used jumps (`GOTO`) as the primary, or even the sole, control structure, and programmers were used to jumping hither and yon within their programs. Such programs often became rats’ nests of control flow.

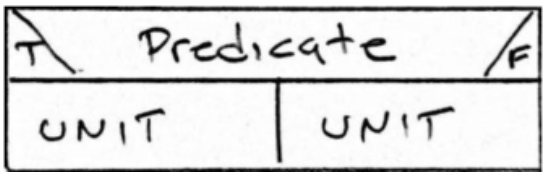
Dijkstra said that all programs should be ordered as recursively applied units with a single entry and a single exit. The units came in three varieties.

Sequence



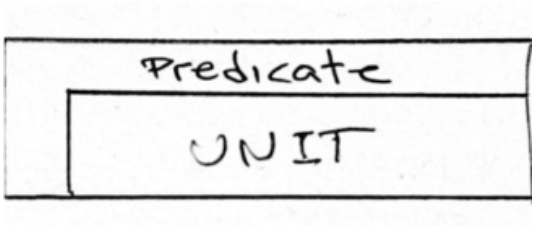
A sequence is two (or more) units that are executed one after the other. Each such unit has a single entry at the top and a single exit at the bottom. A unit might be as simple as a single statement, like `i=0;`, or it might be a function call, or its own sequence of units.

Selection



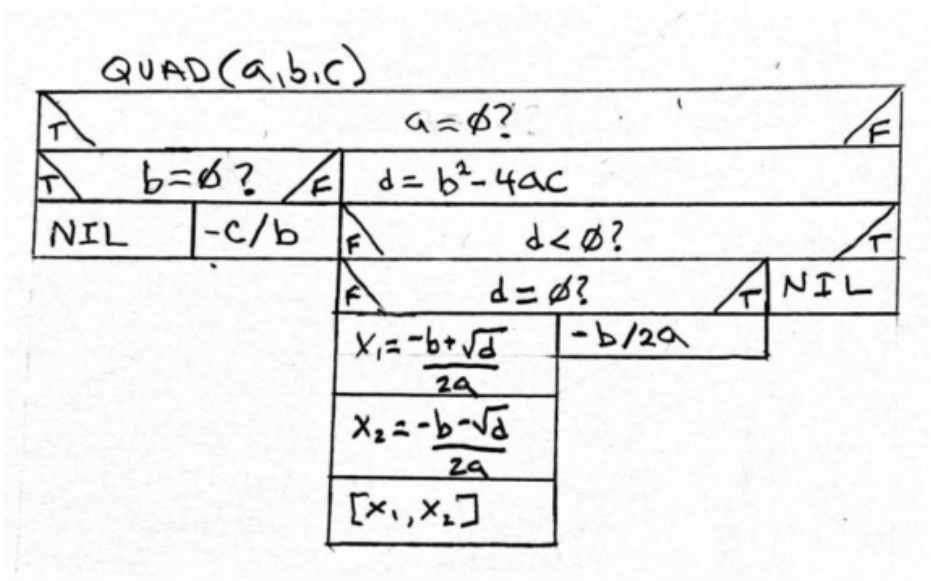
A selection is a boolean predicate that selects one of two units below it. The predicate is any boolean expression, such as `i<0`.

Iteration



An iteration is a predicate that executes the unit below it until the predicate is false.

These units can be recursively applied into a program as follows.



This style of diagram is called a *Nassi–Schneiderman flowchart*. The diagram itself depicts the flow of control through a quadratic equation solver.

Dijkstra's rationale for restricting control flow to these structures was based on the argument that such structures are relatively easy to reason about. You can follow each path and evaluate the outcomes. However, if `goto` statements are allowed to transfer control from any point in the

program to any other point in the program, then following the paths, and therefore reasoning about the problem, becomes impractical.

Nowadays, our languages restrict control flow for us. Most of us don't use languages that have a `goto` statement. Those of us who do, tend not to use it.

When writing clean functions, it is helpful to keep this recursive application of the three structures in mind.

This Is Too Much to Constantly Keep in Mind

Writing software is like any other kind of writing. When you write a paper or an article, you get your thoughts down first, and then you massage it until it reads well. The first draft might be clumsy and disorganized, so you wordsmith it and restructure it and refine it until it reads the way you want it to read.

When I write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists. The names are arbitrary, and there is duplicated code, and temporal couplings. But I also have a suite of unit tests that cover every one of those clumsy lines of code.

So then I massage and refine that code, splitting out functions, changing names, eliminating duplication, and making my functions purer. I shrink the methods and reorder them. Sometimes I break out whole classes, all the while keeping the tests passing.

In the end, I wind up with functions that follow the attributes and heuristics I've laid down in the last two chapters. I don't write them that way to start. I don't think anyone could.

First, make it work. Then, make it right.

Conclusion

Every system is built from a domain-specific language designed by the programmers to describe that system. Functions are the verbs of that language, and contexts are the nouns. It is a very old truth that the art of programming is, and has always been, the art of language design.

Master programmers think of systems as stories to be told rather than programs to be written. They use the facilities of their chosen programming language to construct a much richer and more expressive language that can be used to tell that story. Part of that domain-specific

language is the hierarchy of functions that describe all the actions that take place within that system. In an artful act of recursion, those actions are written to use the very domain-specific language they define to tell their own small part of the story.

The last two chapters have been about the mechanics of writing functions well. If you follow the ideas herein, your functions will be short, well named, and nicely organized. But never forget that your real goal is to tell the story of the system, and that the functions you write need to fit cleanly together into a clear and precise language to help you with that telling.

Chapter 9. The Clean Method



How do we write software? Nowadays, of course, most of us sit at our laptop and type into an IDE. We operate in a loop.

1. We write a small bit of code (with the possible help of an AI).
2. We compile and test it.
3. We go back to step 1.

That loop might take anywhere from five seconds to an hour depending upon the discipline you follow.

Of course, that’s not how it used to be. In the earliest days of programming, the loop was more like this.

1. We wrote a large bit of code by hand.
2. We desk-checked it by hand.
3. We submitted it to be encoded—by hand—into a machine-readable form.
4. We submitted it for compilation and test.

That loop often took between one and three days. The difference is stark. In the past, the investment per line of code was enormous. Nowadays, that cost is trivial.

When costs are high, we spend a lot of time planning up front in order to avoid waste. When costs are low, it makes more sense to explore by failing. In both cases, the idea of cleaning the code is generally ignored.

Make It Right

In the early days, cleaning code was inconvenient because the code was written by hand on sheets of paper. Erasing, cutting, and pasting sheets of paper is very inefficient, and just not fun—especially when small changes can spread through the body of the code in very inconvenient ways.

Nowadays, cleaning the code has been made much more convenient by our IDEs and refactoring tools. But we often bypass cleaning the code because that tight loop allows us to make progress so quickly that pausing to clean the code seems like a misuse of time. We drive hard to make the code work—and then we check the code in and move on to the next problem.

Over two decades ago, Kent Beck stated the simple rule that guides this book.

First, make it work.

Then, make it right.

There’s more to this rule that we’ll talk about later. For now, just look at those two lines. Most of us do the first part because we consider it to be the reason we were hired. We were hired to make code work. But many of us bypass the second part because it seems like a waste of time. After all, if we are good programmers, we should be writing right code all the time. We shouldn’t need to have to stop and make it right.

Perhaps you are that good. Perhaps the code that comes out of your fingers is right the first time and every time. If so, you can put this book down.

However, most of us are not particularly adept at keeping two goals in our head at the same time. The primary goal of making the code work pushes aside the secondary goal of making the code right. Most of us find that the mental effort required to get the code to work is so great that we have no mental effort left to simultaneously make the code right.

Thus, by the time we get the code to work, the code is also a mess. And that’s when the second line of Beck’s rule comes into play. Once we have the code working, then we will have the mental capacity to clean it—to make it right.

Of course, we’d better have a way to ensure that in making the code right we don’t break the fact that it works. It is not our goal to create clean code that doesn’t work. How are we to get that assurance? How will we make sure that the act of cleaning does not break the code?

We will talk about testing disciplines in a later chapter. Suffice it to say that if we don't have a test suite that we can run quickly and that we implicitly trust to show that the code works, then we are not going to do a lot of cleaning.

Clean code depends on fast, convenient, and comprehensive tests.

So how do we make the code right? What's the process? It's another tight loop similar to writing the code.

1. We clean one little thing.
2. We run the tests.
3. If the tests fail, we revert.
4. Go back to step 1.

The timing of this loop is typically between five seconds and two minutes. The name of this loop is *refactoring*, and each cleaning step is *a refactoring*.

To quote Martin Fowler, refactoring is "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior."[1]

[1] [Refactoring], p. 45.

How often should we transition between the writing loop and the refactoring loop? As often as possible. This implies an outer loop that includes both the loop to make it work and the loop to make it right.

1. Get something small to work.
    a. We write a small bit of code along with tests.
    b. We compile and test it.
    c. If it's not too messy, we go back to step 1a.
    Otherwise. . .

2. Clean it up.
    a. We clean one little thing.
    b. We run the tests.
    c. If the tests fail, we revert and go to step 2a.
    d. If there is more to clean, we go back to step 2a. Otherwise. . .
3. Go back to step 1.

The timing of this loop depends on your testing discipline and is on the order of a few minutes to an hour or so.

The bottom line is that, unless you are a multitalented mega being, writing clean code means that you are going to have to sit in that outer loop, patiently cleaning up after yourself, and never allowing the mess to get too big to clean.

Be like the sushi chef who never stops making sushi and who never stops cleaning the implements and the environment.

Example

Perhaps the best way to understand how to write clean code is to watch someone do it. So, in this chapter, we will walk through the procedure for writing a clean module. The technique will involve

- The testing discipline of TDD
- Refactoring
- Design principles
- Design patterns

We will begin with an imaginary Python application: the Tax Calculator for the mythical state of Bobolia. It's been a long time since I've written any Python, so bear with me.

Here's the first test.

```
———bobolia_taxes_test.py———
import unittest
import bobolia_taxes


class bobolia_tax_tests(unittest.TestCase):
    def test_simplest_case(self):
        tax_return = {"income": {"salary": 50000}}
        tax_calculator = bobolia_taxes.tax_calculator(tax_return)
        self.assertEqual(7500, tax_calculator.get_tax())



if __name__ == '__main__':
    unittest.main()
```

In the Great State of Bobolia, the base tax rate is 15%. So someone earning 50,000 bobbles must pay 7,500 bobbles in tax. Note that the tax return is a simple map with an `income` key that contains yet another map with a `salary` key.

We can get this to pass with a simple degenerate module.

```
———bobolia_taxes.py———
class tax_calculator:
    def __init__(self, tax_return):
        self.tax_return = tax_return
```

```
    def get_tax(self):
        return 7500
```

Actually, I made it fail first by returning `0`, and then I made it pass by returning `7500`. This is the TDD way. First we see the test fail, and then we make it pass. If that sounds like a waste of time, let me just say that this simple practice has saved me a lot of debugging time.

But let's move on to the next tax law. Citizens of Bobolia who make no more than 30,000 bobbles per year do not have to pay any tax at all.

```
——bobolia_taxes_test.py——
class BoboliaTaxTests(unittest.TestCase):
    def setUp(self):
        self.tax_calculator = bobolia_taxes.TaxCalculator()


    def test_simplest_case(self):
        tax_return = {"income": {"salary": 50000}}
        self.assertEqual(7500, self.tax_calculator.get_tax(tax_ret


    def test_30K_limit(self):
        tax_return = {"income": {"salary": 30000}}
        self.assertEqual(0, self.tax_calculator.get_tax(tax_return
```

Here you can see I've already begun to make some design changes. Rather than construct the `TaxCalculator` with the `tax_return`, I decided to pass the `tax_return` into the `get_tax` function. I also changed the names of the classes to comport with the PEP 8 standard.

Making these tests pass was pretty simple.

———bobolia_taxes.py———

```
class TaxCalculator:
    def __init__(self):

        pass


    def get_tax(self, tax_return):
        total_income = tax_return["income"]["salary"]
        if total_income > 30000:
            return total_income * 0.15
        return 0
```

Now, clearly this isn’t fair. You can’t have a citizen who earns 30,001 bobbles pay 4,500 in tax. Therefore, no citizen’s after-tax income should be less than 30,000.

———bobolia_taxes_test.py———

```
class BoboliaTaxTests(unittest.TestCase):
    def setUp(self):
        self.tax_calculator = bobolia_taxes.TaxCalculator()


    def test_15pct_tax_rate(self):
        tax_return = {"income": {"salary": 50000}}
        self.assertEqual(7500, self.tax_calculator.get_tax(tax_ret


    def test_no_tax_for_30K_and_below(self):
        tax_return = {"income": {"salary": 30000}}
        self.assertEqual(0, self.tax_calculator.get_tax(tax_return


    def test_no_after_tax_income_less_than_30K(self):
```

```
        tax_return = {"income": {"salary": 30001}}
        self.assertEqual(1, self.tax_calculator.get_tax(tax_return
        tax_return = {"income": {"salary": 35000}}
        self.assertEqual(5000, self.tax_calculator.get_tax(tax_ret
```

No surprise here. I improved the names of the tests. Name improvement is something that often takes place once you know more about the application you are writing. Getting these tests to pass was pretty easy, though it forced me to move most of the older code around a bit.

```
———bobolia_taxes.py———
def get_tax(self, tax_return):
    total_income = tax_return["income"]["salary"]
    tax = total_income * 0.15
    after_tax_income = total_income - tax
    if after_tax_income < 30000:
        return total_income - 30000
    else:
        return tax
```

Now then, let’s talk about the social demerit system. Bobolia is a free state. We believe that what our neighbors do is none of our darn business. However, there are behaviors that are detrimental to the state of Bobolia, and we believe citizens who engage in these behavior should compensate society for the damage they’ve done. So all citizens earn badbob points through the course of a year, and they must report these badbobs on their tax returns. (Don’t cheat! We have ways. . .)

Badbob points are used in calculating taxes. So we need to add a `badbobs` key to the tax return. Oh, but that means we

have to go back and change all the tests.

Anytime you have to go back and change a bunch of tests, it means there's a flaw in the design of your tests. Let's fix that flaw before we proceed.

```
——bobolia_taxes_test.py——
def make_return(args):
    tax_return = {"income": {"salary": 0},
                  "badbobs": 0}
    tax_return.update(args)
    return tax_return



class BoboliaTaxTests(unittest.TestCase):
    def setUp(self):
        self.tax_calculator = bobolia_taxes.TaxCalculator()

    def test_15pct_tax_rate(self):
        tax_return = make_return({"income": {"salary": 50000}})
        self.assertEqual(7500, self.tax_calculator.get_tax(tax_ret

    def test_no_tax_for_30K_and_below(self):
        tax_return = make_return({"income": {"salary": 30000}})
        self.assertEqual(0, self.tax_calculator.get_tax(tax_return

    def test_no_after_tax_income_less_than_30K(self):
        tax_return = make_return({"income": {"salary": 30001}})
        self.assertEqual(1, self.tax_calculator.get_tax(tax_return
```

```
        tax_return = make_return({"income": {"salary": 35000}})
        self.assertEqual(5000, self.tax_calculator.get_tax(tax_ret
```

This simple change has decoupled the tests from the expectations of the production code. The `make_return` function will ensure that any omitted fields are given a reasonable default value. Thus, when new fields are added to the tax return, the existing tests are not likely to require changes.

Now, on with the badbob processing.

The purchase of alcoholic beverages is detrimental to society. So, if you spend 100 bobbles on whiskey, you should earn 100 badbobs. Alcohol badbobs are of class 1 and will be added to your tax at a rate of 50%. So that 100-bobble bottle of whiskey will cost you another 50 bobbles at tax time.

```
———bobolia_taxes_test.py———
def make_return(args):
    tax_return = {"income": {"salary": 0},
                  "badbobs": {"class1": ()}}
    tax_return.update(args)
    return tax_return



class BoboliaTaxTests(unittest.TestCase):
    …
  def test_class_1_badbobs(self):
        tax_return = make_return({"income": {"salary": 50000},
                                  "badbobs": {"class1": (100,)}}
```

```
                                )

        self.assertEqual(7550, self.tax_calculator.get_tax(tax_ret
```

While writing this test, I realized that the structure of the `badbobs` that I had placed in `make_return` was insufficient. So I changed that to add the `class1` key and the empty tuple. The test shows the extra `50` bobbles in tax for the 100 class 1 badbobs. Making this pass was easy, but a little unsettling.

```
———bobolia_taxes.py———
def get_tax(self, tax_return):
    total_income = tax_return["income"]["salary"]
    tax = total_income * 0.15
    after_tax_income = total_income - tax
    if after_tax_income < 30000:
        tax = total_income - 30000
    tax += sum(tax_return["badbobs"]["class1"]) / 2
    return tax
```

This function is getting messy. We could at least break it up into a couple of nicer functions.

```
———bobolia_taxes.py———
def get_tax(self, tax_return):
    total_income = tax_return["income"]["salary"]
    tax = self.determine_base_tax(total_income)
    tax += self.determine_badbob_adjustment(tax_return)
    return tax


def determine_badbob_adjustment(self, tax_return):
    return sum(tax_return["badbobs"]["class1"]) / 2
```

```
def determine_base_tax(self, total_income):
    tax = total_income * 0.15
    after_tax_income = total_income - tax
    if after_tax_income < 30000:
        tax = total_income - 30000
    return tax
```

That's a bit better. I'm not sure why I put these functions into a class. The class is just forcing me to add `self` everywhere. Maybe I should get rid of that class—but I've got a nagging feeling that the class is going to be important, even if for nothing more than the name space.

Perhaps that class will come in handy for allowing the methods to communicate through instance variables rather than arguments. Let's try that.

```
———bobolia_taxes.py———
class TaxCalculator:

    def __init__(self):
        self.tax_return = None


    def get_tax(self, tax_return):
        self.tax_return = tax_return
        tax = self.determine_base_tax()
        tax += self.determine_badbob_adjustment()
        return tax


    def determine_badbob_adjustment(self):
        return sum(self.tax_return["badbobs"]["class1"]) / 2
```

```
    def determine_base_tax(self):
        total_income = self.tax_return["income"]["salary"]
        tax = total_income * 0.15
        after_tax_income = total_income - tax
        if after_tax_income < 30000:
            tax = total_income - 30000
        return tax
```

I think that's a little better. I'm going to keep the class.

The Great State of Bobolia considers the purchase of gasoline-powered vehicles to be very detrimental to society. Such purchases earn class 2 badbobs at a rate of one badbob per bobble. Class 2 badbobs increase taxes according to the following table:

Class 2 badbobs	% of total income
1001–10,000	5%
10,001–50,000	10%
>50,000	15%

```
———bobolia_taxes_test.py———
def make_return(args):
    tax_return = {"income": {"salary": 0},
                  "badbobs": {"class1": (),
                              "class2": ()}}
```

```
    if "income" in args:
        tax_return["income"].update(args["income"])
    if "badbobs" in args:
        tax_return["badbobs"].update(args["badbobs"])
    return tax_return


class BoboliaTaxTests(unittest.TestCase):
    …
    def test_class_2_badbob_purchases_under_1001(self):
        tax_return = make_return({"income": {"salary": 50000},
                                  "badbobs": {"class2": (400, 600)
                                 )
        self.assertEqual(7500, self.tax_calculator.get_tax(tax_ret

    def test_class_2_badbob_purchases_under_10001(self):
        tax_return = make_return({"income": {"salary": 50000},
                                  "badbobs": {"class2": (10000,)}}
                                 )
        self.assertEqual(7500 + 2500,
                         self.tax_calculator.get_tax(tax_return))

    def test_class_2_badbob_purchases_under_50001(self):
        tax_return = make_return({"income": {"salary": 50000},
                                  "badbobs": {"class2": (10000,400
                                 )
        self.assertEqual(7500 + 5000,
                         self.tax_calculator.get_tax(tax_return))
```

```
    def test_class_2_badbob_purchases_over_50000(self):
        tax_return = make_return(
          {"income": {"salary": 50000},
           "badbobs": {"class2": (10000,40000,1000)}}
          )
        self.assertEqual(7500 + 7500,
                         self.tax_calculator.get_tax(tax_return))
```

These tests show the four table entries pretty clearly. I made these tests pass one at a time and ended up with the following production code.

```
———bobolia_taxes.py———
class TaxCalculator:
    def __init__(self):
        self.tax_return = None
        self.total_income = None

    def get_tax(self, tax_return):
        self.tax_return = tax_return
        tax = self.determine_base_tax()
        tax += self.determine_badbob_adjustment()
        return tax

    def determine_badbob_adjustment(self):
        class1 = sum(self.tax_return["badbobs"]["class1"])
        class1_adjustment = class1 /2
        class2 = sum(self.tax_return["badbobs"]["class2"])
        if class2<1001:
```

```
        if class2<10001:
            class2_adjustment = 0
        elif class2<10001:
            class2_adjustment = self.total_income * 0.05
        elif class2<50001:
            class2_adjustment = self.total_income * 0.1
        else:
            class2_adjustment = self.total_income * 0.15
        return class1_adjustment + class2_adjustment


    def determine_base_tax(self):
        self.total_income = self.tax_return["income"]["salary"]
        tax = self.total_income * 0.15
        after_tax_income = self.total_income - tax
        if after_tax_income < 30000:
            tax = self.total_income - 30000
        return tax
```

This is pretty awful. I expect all those numbers to change over time. I even expect the number of rows in the table to change. Maintaining this is going to be labor intensive and error prone. We can probably do better.

```
——bobolia_taxes.py——
class TaxCalculator:
    …


    def get_tax(self, tax_return):
        self.tax_return = tax_return
        tax = self.determine_base_tax()
```

```
        tax += BadBobAdjuster(self).get_adjustment()
        return tax

    def determine_base_tax(self):
        self.total_income = self.tax_return["income"]["salary"]
        tax = self.total_income * 0.15
        after_tax_income = self.total_income - tax
        if after_tax_income < 30000:
            tax = self.total_income - 30000
        return tax


class BadBobAdjuster:
    def __init__(self, tax_calculator):
        self.tax_calculator = tax_calculator

    def get_adjustment(self):
        class1_adjustment = self.class1_adjustment()
        class2_adjustment = self.get_class2_badbob_adjustment()
        return class1_adjustment + class2_adjustment

    def get_class2_badbob_adjustment(self):
        class2 = sum(self.tax_calculator.tax_return["badbobs"]["cl
        class2_rate = self.get_class2_rate(class2)

        return class2_rate * self.tax_calculator.total_income

    def get_class2_rate(self, class2):
```

```
        class2_rate = 0
        class2_table = ((1001, 0),
                        (10001, 0.05),
                        (50001, 0.1),
                        ("max", 0.15))
        for row in class2_table:
            if row[0] == "max" or class2 < row[0]:
                class2_rate = row[1]
                break
        return class2_rate


    def class1_adjustment(self):
        class1 = sum(self.tax_calculator.tax_return["badbobs"]["cl
        class1_adjustment = class1 / 2
        return class1_adjustment
```

So I created the table and the loop to extract the rate. Then, I thought that the whole badbob idea should be moved into its own class, just to have a namespace for it.

Notice what's happening here. The production code is evolving and morphing to become more generic. Parts of it are being peeled off into their own classes and separated from each other. Meanwhile, the tests are simply growing in a linear fashion. With each new test case the test class becomes more specific.

As the tests get more specific, the code gets more generic.

This strategy keeps the tests from knowing too much about the production code. When changes to the production code are made, fewer changes need to be made to the tests.

But now we have a problem. If a citizen saves up to buy an expensive car, they might end up with a tax rate that reduces their after-tax income to significantly less than 30,000 bobbles. In one sense, this is fair. Citizens who harm society ought to compensate for that harm. On the other hand, forcing a citizen to live on less than, say, 20,000 bobbles a year would be cruel. So let's limit the effect of the badbobs so that no badbob adjustment will reduce after-tax income to below 20,000 bobbles.

```
——bobolia_taxes_test.py——

…

def test_badbobs_do_not_reduce_after_tax_income_below_20000(self):
    tax_return = make_return({"income": {"salary": 20000},
                              "badbobs": {"class2": (10000,)}}
                             )
    self.assertEqual(0, self.tax_calculator.get_tax(tax_return))
```

This failed in an unexpected way. It returned a tax of −9,000 bobbles. Negative tax! I bet the citizens would love that. On the other hand, the BRS (Bobolia Revenue Service) would not be pleased. This was a bug. I fixed it by putting in a guard.

```
——bobolia_taxes.py——

…

def determine_base_tax(self):
    self.total_income = self.tax_return["income"]["salary"]
    if self.total_income <= 30000:
        return 0

…
```

Yes, TDD is not a magic bullet. You will still write some bugs. Fortunately, as shown in this case, other tests are likely to uncover those bugs. With that fixed, I could get the test to pass pretty easily.

```
——bobolia_taxes.py——

…
def get_tax(self, tax_return):
    self.tax_return = tax_return
    base_tax = self.determine_base_tax()
  badbob_adujustment = BadBobAdjuster(self).get_adjustment()
    total_tax = base_tax + badbob_adujustment
    after_tax_income = self.total_income - total_tax
    if self.total_income < 20000:
        return 0
    elif after_tax_income < 20000:
        total_tax = self.total_income - 20000
    return total_tax
```

Those two guards that I just added are a bit ugly. I think I can just use the `max` function instead.

```
——bobolia_taxes.py——

…
    def get_tax(self, tax_return):
        self.tax_return = tax_return
        base_tax = self.determine_base_tax()
        badbob_adujustment = BadBobAdjuster(self).get_adjustment()
        total_tax = base_tax + badbob_adujustment
        after_tax_income = self.total_income - total_tax
        if after_tax_income < 20000:
```

```
            total_tax = max(0, self.total_income - 20000)
        return total_tax


    def determine_base_tax(self):
        self.total_income = self.tax_return["income"]["salary"]
        tax = self.total_income * 0.15
        after_tax_income = self.total_income - tax
        if after_tax_income < 30000:
            tax = max (0, self.total_income - 30000)
        return tax
```

That’s better.

Now then. . . A year has gone by, and everyone is paying their taxes. But after some riots and really mean tweets, we have decided that here in the Great State of Bobolia income inequality is a social harm. Low earners were paying too much, and high earners were paying too little. Those who earn too many bobbles should simply pay a higher tax rate than those who earn less. Not just more taxes, but a higher rate. We’ve determined that the following table would represent “their fair share”:

Income	Tax rate
0–30,000	0%
30,001–100,000	15% of the amount over 30,000
100,001–250,000	10,500 + 20% of the amount over 100,000
250,001–500,000	40,500 + 30% of the amount over 250,000
>500,000	115,500 + 40% of the amount over 500,000

Of course, this changes the way taxes are calculated, so many of our previous tests are going to need deleting or changing. Since most of our tests use the salary of 50,000, let's create a constant for the tax on 50,000, named `tax_50K`, and then we'll use that in all the tests. That way, if the tax calculation changes again, we only have to change the constant instead of all the tests.

```
——bobolia_taxes_test.py——

…
tax_50K = 3000



class BoboliaTaxTests(unittest.TestCase):
    def setUp(self):
        self.tax_calculator = bobolia_taxes.TaxCalculator()
        self.tax_return = None
```

```
def make_return(self, args):
    tax_return = {"income": {"salary": 0},
                  "badbobs": {"class1": (),
                              "class2": ()}}
    if "income" in args:
        tax_return["income"].update(args["income"])
    if "badbobs" in args:
        tax_return["badbobs"].update(args["badbobs"])
    self.tax_return = tax_return

def make_simple_return(self, salary):
    self.make_return({"income": {"salary": salary}})

def get_tax(self):
    return self.tax_calculator.get_tax(self.tax_return)

def test_class_1_badbobs(self):
    self.make_return({"income": {"salary": 50000},
                      "badbobs": {"class1": (100,)}}
                     )
    self.assertEqual(tax_50K + 50, self.get_tax())

def test_class_2_badbob_purchases_under_1001(self):
    self.make_return({"income": {"salary": 50000},
                      "badbobs": {"class2": (400, 600)}}
                     )
    self.assertEqual(tax_50K, self.get_tax())
```

```
def test_class_2_badbob_purchases_under_10001(self):
    self.make_return({"income": {"salary": 50000},
                      "badbobs": {"class2": (10000,)}}
                     )
    self.assertEqual(tax_50K + 2500, self.get_tax())

def test_class_2_badbob_purchases_under_50001(self):
    self.make_return({"income": {"salary": 50000},
                      "badbobs": {"class2": (10000, 40000)}}
                     )
    self.assertEqual(tax_50K + 5000, self.get_tax())

def test_class_2_badbob_purchases_over_50000(self):
    self.make_return({"income": {"salary": 50000},
                      "badbobs": {"class2": (10000, 40000, 100
                     )
    self.assertEqual(tax_50K + 7500, self.get_tax())

def test_badbobs_do_not_reduce_after_tax_income_below_20000(se
    self.make_return({"income": {"salary": 20000},
                      "badbobs": {"class2": (10000,)}}
                     )
    self.assertEqual(0, self.get_tax())

def test_0_30_tax_bracket(self):
    self.make_simple_return(15000)
    self.assertEqual(0, self.get_tax())
```

```
        self.make_simple_return(30000)
        self.assertEqual(0, self.get_tax())

    def test_30_100_tax_bracket(self):
        self.make_simple_return(30001)
        self.assertEqual(0, self.get_tax())
        self.make_simple_return(50000)
        self.assertEqual(tax_50K, self.get_tax())
        self.make_simple_return(100000)
        self.assertEqual(10500, self.get_tax())

    def test_100_250_tax_bracket(self):
        self.make_simple_return(100001)
        self.assertEqual(10500, self.get_tax())
        self.make_simple_return(200000)
        self.assertEqual(30500, self.get_tax())
        self.make_simple_return(250000)
        self.assertEqual(40500, self.get_tax())

    def test_250_500_tax_bracket(self):
        self.make_simple_return(250001)
        self.assertEqual(40500, self.get_tax())
        self.make_simple_return(500000)
        self.assertEqual(115500, self.get_tax())

    def test_500_up_tax_bracket(self):
        self.make_simple_return(500001)
        self.assertEqual(115500, self.get_tax())
```

```
        self.make_simple_return(1000000)
        self.assertEqual(315500, self.get_tax())
```

I made quite a few changes to the tests to try to keep the noise down. I moved the `make_return` method into the test class, and I added `make_simple_return` and `get_tax`. I also made the `tax_return` into an instance variable of the test class so that I didn't have to keep assigning and passing it.

The production code is a relatively simple change to the `determine_base_tax` method.

```
———bobolia_taxes.py———
…
def determine_base_tax(self):
    self.total_income = self.tax_return["income"]["salary"]
  if self.total_income <= 30000:
        return 0
    elif self.total_income <= 100000:
        return round(0.15 * (self.total_income - 30000))
    elif self.total_income <= 250000:
        return round(0.2 * (self.total_income - 100000) + 10500)
    elif self.total_income <= 500000:
        return round(0.3 * (self.total_income - 250000) + 40500)

    else:
        return round(0.4 * (self.total_income - 500000) + 115500)
…
```

Those tests are pretty ugly. Let's clean them up by using composed assertions.

———bobolia_taxes_test.py———

```
    …

    def assert_tax_with_badbobs(self, income, badbobs, tax):
        self.make_return({"income": {"salary": income},
                          "badbobs": badbobs})
        self.assertEqual(tax, self.get_tax())

    def test_class_1_badbobs(self):
        self.assert_tax_with_badbobs(50000,
                                     {"class1": (100,)},
                                     tax_50K + 50)

    def test_class_2_badbob_purchases_under_1001(self):
        self.assert_tax_with_badbobs(50000,
                                     {"class2": (400, 600)},
                                     tax_50K)

    def test_class_2_badbob_purchases_under_10001(self):
        self.assert_tax_with_badbobs(50000,
                                     {"class2": (10000,)},
                                     tax_50K + 2500)

    def test_class_2_badbob_purchases_under_50001(self):
        self.assert_tax_with_badbobs(50000,
                                     {"class2": (10000, 40000)},
                                     tax_50K + 5000)
```

```
def test_class_2_badbob_purchases_over_50000(self):
    self.assert_tax_with_badbobs(50000,
                                 {"class2": (10000, 40000, 100
                                 tax_50K + 7500)

def test_badbobs_do_not_reduce_after_tax_income_below_20000(se
    self.assert_tax_with_badbobs(20000,
                                 {"class2": (10000,)},
                                 0)

def assert_tax_for(self, income, tax):
    self.make_simple_return(income)
    self.assertEqual(tax, self.get_tax())

def test_0_30_tax_bracket(self):
    self.assert_tax_for(15000, 0)
    self.assert_tax_for(30000, 0)

def test_30_100_tax_bracket(self):
    self.assert_tax_for(30001, 0)
    self.assert_tax_for(50000, tax_50K)
    self.assert_tax_for(100000, 10500)

def test_100_250_tax_bracket(self):
    self.assert_tax_for(100001, 10500)
    self.assert_tax_for(200000, 30500)
    self.assert_tax_for(250000, 40500)
```

```
    def test_250_500_tax_bracket(self):
        self.assert_tax_for(250001, 40500)
        self.assert_tax_for(500000, 115500)

    def test_500_up_tax_bracket(self):
        self.assert_tax_for(500001, 115500)
        self.assert_tax_for(1000000, 315500)
    …
```

That's much better. Those tests are now in a position to become parametric (table driven) should the need arise—but I don't think we're there yet.

On the other hand, the production code is crying out for a table-driven approach. You don't think we've seen the last of the tax brackets, do you? So let's take care of that.

```
———bobolia_taxes.py———

…

def determine_base_tax(self):
    tax_brackets = ((30000, 0),
                    (100000, .15),
                    (250000, .20),
                    (500000, .30),
                    (None, .40))

    self.total_income = self.tax_return["income"]["salary"]

  previous_maximum = 0
    offset = 0
```

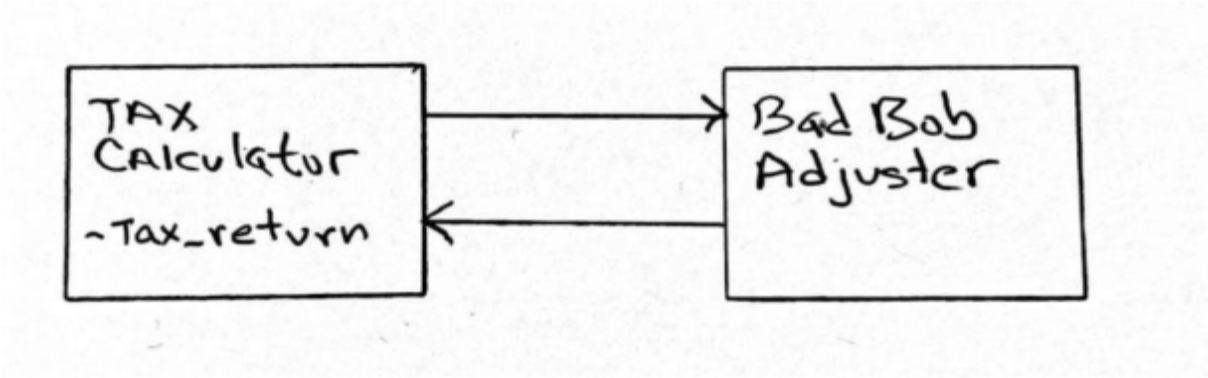
```
    for maximum, rate in tax_brackets:
        if maximum is None or self.total_income <= maximum:
            return round(
              rate *
              (self.total_income - previous_maximum) +
              offset)
        else:
            offset = (maximum - previous_maximum) * rate + offset
            previous_maximum = maximum
…
```

OK, that’s better. Any new bracket can be added easily.

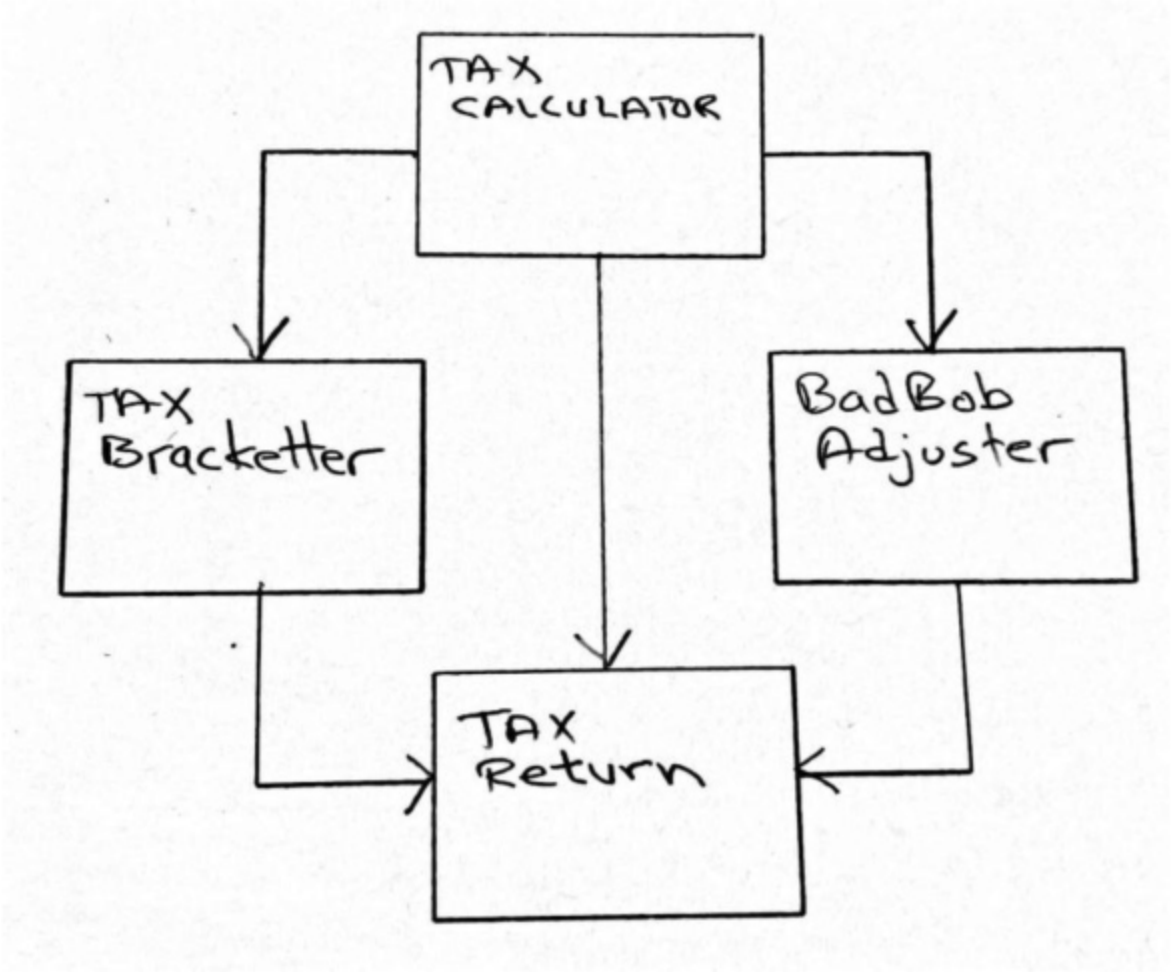
At this point, you might ask whether the tests ought to use the same `tax_brackets` table. I’m usually not in favor of allowing the tests to know the details of the production code. The whole point of tests is that they are a second statement of intent. It makes little sense to use the production code to test the production code. So I prefer to do the calculations manually and enter them into the tests.

Considering the Design and Architecture

Now let’s take a step back. So far you have watched me write a small module using a process that keeps it “clean” and test as it grows and is changed. I hope the process has been illuminating. The module has grown into two classes that have the following relationships.



First of all, I don’t like that reciprocal relationship. It only exists because the `BadBobAdjuster` uses the `tax_return` field of the `TaxCalculator`. Second, I think this is a bit asymmetric. Why does the `BadBobAdjuster` deserve its own special class whereas the tax bracket calculations are buried in the `TaxCalculator`? Let’s clean that up a bit. I’d like it to look like this.



This change requires only minor changes to the tests, so our test design is holding up.

```
——bobolia_taxes_test.py——

…
def make_return(self, args):
    tax_return_data = {"income": {"salary": 0},
                       "badbobs": {"class1": (),
                                   "class2": ()}}
    if "income" in args:
        tax_return_data["income"].update(args["income"])
    if "badbobs" in args:
        tax_return_data["badbobs"].update(args["badbobs"])
```

```
    self.tax_return = bobolia_taxes.TaxReturn(tax_return_data)
…

———bobolia_taxes.py———
class TaxCalculator:
    def get_tax(self, tax_return):
        total_income = tax_return.get_total_income()
        base_tax = TaxBracketter(tax_return).determine_base_tax()
        badbob_adjustment = BadBobAdjuster(tax_return).get_adjustm
        total_tax = base_tax + badbob_adjustment
        after_tax_income = total_income - total_tax
        if after_tax_income < 20000:
            total_tax = max(0, total_income - 20000)
        return total_tax


class TaxBracketter:
    tax_brackets = ((30000, 0),
                    (100000, .15),
                    (250000, .20),
                    (500000, .30),
                    (None, .40))

    def __init__(self, tax_return):
        self.tax_return = tax_return

    def determine_base_tax(self):
        total_income = self.tax_return.get_total_income()
```

```
        previous_maximum = 0
        offset = 0
        for maximum, rate in self.tax_brackets:
            if maximum is None or total_income <= maximum:
                return round(rate *
                             (total_income - previous_maximum) +
                             offset)
            else:
                offset = (maximum - previous_maximum) * rate + off
                previous_maximum = maximum


class BadBobAdjuster:
    def __init__(self, tax_return):
        self.tax_return = tax_return
        self.total_income = tax_return.get_total_income()

    def get_adjustment(self):
        class1_adjustment = self.class1_adjustment()
        class2_adjustment = self.get_class2_badbob_adjustment()
        return class1_adjustment + class2_adjustment

    def get_class2_badbob_adjustment(self):
        class2 = self.tax_return.get_total_class2_badbobs()
        class2_rate = self.get_class2_rate(class2)
        return class2_rate * self.total_income
```

```
    class2_table = ((1001, 0),
                    (10001, 0.05),
                    (50001, 0.1),
                    (None, 0.15))

    def get_class2_rate(self, class2):
        for limit, rate in self.class2_table:
            if limit is None or class2 < limit:
                return rate
        return None

    def class1_adjustment(self):
        class1 = self.tax_return.get_total_class1_badbobs()
        return class1 / 2


class TaxReturn:
    def __init__(self, tax_return_data):
        self.tax_return_data = tax_return_data

    def get_total_income(self):
        return self.tax_return_data["income"]["salary"]

    def get_total_class1_badbobs(self):
        return sum(self.tax_return_data["badbobs"]["class1"])

    def get_total_class2_badbobs(self):
        return sum(self.tax_return_data["badbobs"]["class2"])
```

At this point, you might be accusing this design of *classitis*[2]—the tendency to create too many classes and thereby complicate the design. I disagree. The three new classes do not add any complexity that wasn't already there. They just move that complexity into nicely named places.

[2] [APOD], p. 26.

A place for everything, and everything in its place!

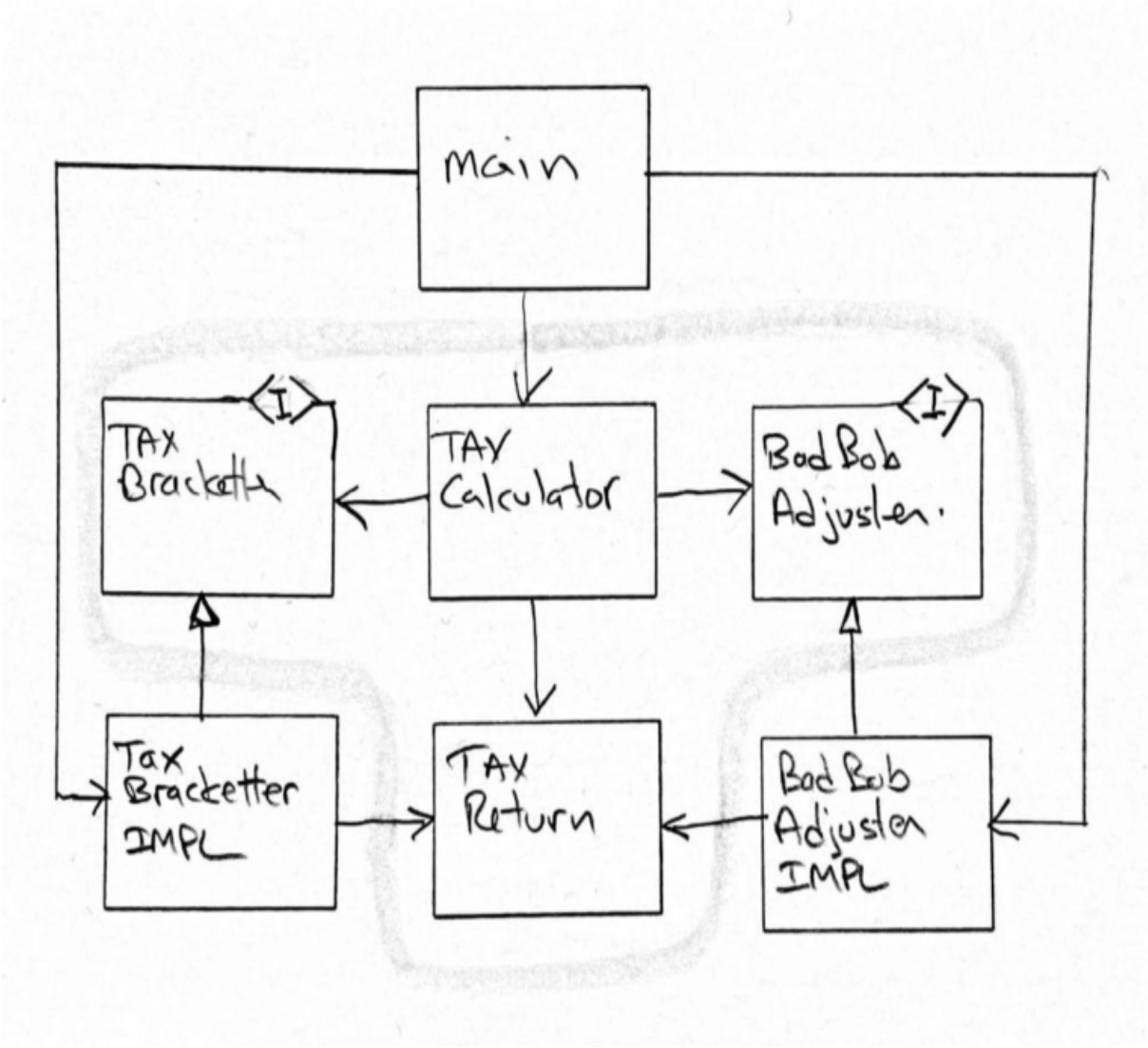
One major advantage of the `TaxReturn` class is that it hides the internal data structure of the `tax_return_data` from the rest of the application. When there are changes to that data structure, the other modules in the application will not need to be changed.

In any case, the advantage of those classes should become evident soon; because it's time for us to take yet another step back and think about the overall architecture of this system.

It seems to me that this system has two axes of change so far. There's the computation of the base tax, which is done by the `TaxBracketter`; and then there's the computation of the social demerit score, accomplished by the `BadBobAdjuster`. If we expect this system to grow (and what tax system doesn't grow?), then we may want to establish these axes of change within the architecture of our system. Or, to say that differently, we may wish to draw architectural boundaries that isolate and protect them.

We'll use the Single Responsibility Principle (SRP) to do that, and we'll use the Dependency Inversion Principle (DIP) to create a structure like this.[3]

[3] See Chapter 19, "The SOLID Principles."



Note the architectural boundary drawn around the four central classes. Notice also that every arrow that crosses that boundary points *inward*, following the Dependency Rule[4] of architecture. Inside that boundary is the high-level policy of this system. Outside are lower-level details. The `main` component is the lowest level in the system. It creates the implementations of the `TaxBracketter` and `BadBobAdjuster` and hands them to the `TaxCalculator` prior to any taxes being calculated.

[4] See Chapter 26, "The Clean Architecture."

The `TaxCalculator` uses the Strategy[5] pattern to delegate the bracket and demerit calculations to the appropriate objects. The canonical form of Strategy calls for the strategy implementations to derive from interfaces. However, in Python, interfaces do not need to exist as independent source code entities. Python is dynamically typed, and so polymorphic dispatch is through duck types. So the "interfaces" shown above are just the method signatures that the `TaxCalculator` expects to invoke.

[5] [GOF95].

The role of the main component will be played by our tests.

```
——bobolia_taxes_test.py——
def setUp(self):
    self.tax_bracketter = tax_bracketter.TaxBracketter()
    self.badbob_adjuster = badbob_adjuster.BadBobAdjuster()
    self.tax_calculator = tax_calculator.TaxCalculator(
        self.tax_bracketter, self.badbob_adjuster)
    self.tax_return = None
```

The `TaxCalculator` and `TaxReturn` exist in their own module.

```
——tax_calculator.py——
class TaxCalculator:
    def __init__(self, tax_bracketter, badbob_adjuster):
        self.tax_bracketter = tax_bracketter
        self.badbob_adjuster = badbob_adjuster

    def get_tax(self, tax_return):
        total_income = tax_return.get_total_income()
        base_tax = self.tax_bracketter.determine_base_tax(tax_retu
```

```
        badbob_adjustment =
          self.badbob_adjuster.get_adjustment(tax_return)
        total_tax = base_tax + badbob_adjustment
        after_tax_income = total_income - total_tax
        if after_tax_income < 20000:
            total_tax = max(0, total_income - 20000)
        return total_tax


class TaxReturn:
    def __init__(self, tax_return_data):
        self.tax_return_data = tax_return_data

    def get_total_income(self):
        return self.tax_return_data["income"]["salary"]

    def get_total_class1_badbobs(self):
        return sum(self.tax_return_data["badbobs"]["class1"])

    def get_total_class2_badbobs(self):
        return sum(self.tax_return_data["badbobs"]["class2"])
```

Then come the `TaxBracketter` and `BadBobAdjuster` modules.

```
———tax_bracketter.py———
class TaxBracketter:
    tax_brackets = ((30000, 0),
                    (100000, .15),
                    (250000, .20),
```

```
                    (500000, .30),
                    (None, .40))

    def determine_base_tax(self, tax_return):
        total_income = tax_return.get_total_income()

        previous_maximum = 0
        offset = 0
        for maximum, rate in self.tax_brackets:
            if maximum is None or total_income <= maximum:
                return round(rate *
                             (total_income - previous_maximum) +
                             offset)
            else:
                offset = (maximum - previous_maximum) * rate + off
                previous_maximum = maximum

———badbob_adjuster.py———
class BadBobAdjuster:
    def __init__(self):
        self.total_income = None
        self.tax_return = None

    def get_adjustment(self, tax_return):
        self.tax_return = tax_return
        self.total_income = tax_return.get_total_income()
        class1_adjustment = self.class1_adjustment()
        class2_adjustment = self.get_class2_badbob_adjustment()
```

```
        return class1_adjustment + class2_adjustment

    def get_class2_badbob_adjustment(self):
        class2 = self.tax_return.get_total_class2_badbobs()
        class2_rate = self.get_class2_rate(class2)
        return class2_rate * self.total_income

    class2_table = ((1001, 0),
                    (10001, 0.05),
                    (50001, 0.1),
                    (None, 0.15))

    def get_class2_rate(self, class2):
        for limit, rate in self.class2_table:
            if limit is None or class2 < limit:
                return rate
        return None

    def class1_adjustment(self):
        class1 = self.tax_return.get_total_class1_badbobs()
        return class1 / 2
```

Conclusion

Is this clean? I’m pretty happy with it, though I’m not a Python programmer, so I’ve likely violated a lot of conventions and probably have not used the most efficient techniques. Still, I think some major goals have been accomplished. We’ve separated the system into at least four

major components: `main`, `TaxCalculator`, `TaxBracketter`, and `BadBobAdjuster`. And we placed those components behind appropriate architectural boundaries.

Our tests have survived a significant amount of change and have showed themselves to be resilient. The same is true of our basic algorithms. Changes to one are very unlikely to affect the others. We've also built in some flexibility by using tables to drive the algorithms.

As the tests grew in specificity, the production code grew in generality.

Some folks might complain about the names. I dunno, I kinda like `TaxBracketter`[6] and `BadBobAdjuster`. But if you play with this code, you might find some better names.

[6] Despite the spelling error.

I have to say that I was pleased with IntelliJ's Python plug-in and Copilot's ability to help someone who hadn't written Python in over 20 years. I also have to say that I was impressed with the responsiveness of the tests. I'm used to JVM startup times, and it's a joy to have the test results pop up on the screen before my fingers have left the test button.

Chapter 10. One Thing



Do you remember what Bjarne Stroustrup said about clean code? He said, “Clean code does one thing well.” And, indeed, many other software authors have said things like that over the years. The problem with that statement is: What does *one thing* mean?

As I said in a previous chapter, I was once responsible for a C function that was 3,000 lines long. It was named `gi`, which stood for *graphic interpreter*. Had anyone challenged me about whether my 3,000-line function did one thing, I would have confidently replied: “Certainly! It interprets graphics.”

Of course, a 3,000-line function must be doing a lot of *things* while it interprets graphics. I’m quite convinced that the problem of interpreting graphics can be broken down into quite a few *things*.

So how do we resolve this issue? If the advice is to be followed, we must have some reasonable definition of *one thing*. I think I know what that definition is, and I'm going to tell you now.

But, like Deep Thought said in *Hitchhiker's Guide to the Galaxy*: "You're not going to like it."

Extract Method Refactoring

Does your IDE have a *Refactor* menu? Mine does. One of the menu items is *Extract Method*. IDEs have had this menu item for two decades now. I'm sure you've all used it. But just in case, let me describe it for you.

If you see a function that's a little too big, and you decide you are going to split it into two functions, you can use your mouse to select the code you want to pull out (extract) from the parent function.[1] Then, you select the *Extract Method* refactoring and up pops a dialog box that asks you what name you want for the new function. That box also tells you what arguments it plans to pass into the new function, and also what it plans to return. The dialog box gives you the option to change a few of these things. When you are ready, you hit the *Refactor* button on that dialog box, and the IDE cuts the selected code out of the original function and replaces it with a call to the new function. Then, it pastes the selected code into that new function and creates the glue code that allows it to be called.

[1] In 1974, no programmer could have made sense of that sentence. Forty years ago, some might have had a glimmer.

I mean . . . it's magic!

So, what does *one thing* mean?

A function does one thing if you cannot meaningfully extract another function from it.

OK, now disregard the word *meaningfully* for just a moment and think about that statement. Can you refute it? I don't think you can. If you can split one function into two, then the original function did two things. So, a function that does one thing is a function from which you cannot extract another.

OK, now, what does the word *meaningfully* mean in this context? Consider this function:

```
public void addRental (Rental rental) {
  rentals.addElement (rental);
}
```

It is not meaningful to extract the body of that function into a new function:

```
public void addRental (Rental rental) {
  doTheAdd(rental);
}

public void doTheAdd (Rental rental) {
  rentals.addElement (rental);
}
```

There is no point in extracting the entire body of a function, leaving behind a pure delegator. Such an extraction is not meaningful.[2]

[2] Except in those rare circumstances where you are creating a delegator so that you can move the extracted function to a new module.

Or consider this function:

```
void clearTotals() {
  totalAmount=0;
  renterPoints=0;
}
```

It would almost certainly be meaningless to extract those two lines into two different functions:

```
void clearAmount() {totalAmount=0;}
void clearPoints() (renterPoints=0;}
```

This has taken extraction too far. The `clearAmount` and `clearPoints` functions have names that are indistinguishable from their implementations. A meaningful abstraction is one that follows The Stepdown Rule, where the name of a function is more abstract than the implementation.

I'll leave the rest of the interpretation of the word *meaningful* up to you. Suffice it to say that you can extract too little, and you can extract too much.

So how do we ensure that all our functions do one thing? We extract, and extract, until we cannot meaningfully extract anymore. Or, in other words:

Extract till you drop!

OK, perhaps that's a little cheeky. I don't really want you to drop. On the other hand, I do want you to *consider* as many extractions as you can. You may not choose to do them all, but you should at least think about them.

As a consequence, the majority of your functions will be pretty small. How small? A few lines. That number might be three, or four, or six. It might be larger for certain `switch` statements, or certain formatting statements, or in certain

languages. But, in general, following this recommendation will cause you to create *a lot* of *small* functions.

And what will you be extracting? Often you'll be extracting the bodies of `if`/`else` statements and `for`/`while` loops. You'll also likely be extracting the predicates out of those statements so that those statements will be composed of little more than the keywords and the function calls.

For example:

```
if (shouldDeleteRecord(r))
  deleteRecord(r);
```

If you read that carefully, you'll see that it reads like *well-written prose*.

This Shouldn't Be Controversial

Over the years, this particular recommendation, from the first edition of this book, has been the most controversial. But why? There are a lot of reasons.

1. People are afraid that if they follow this rule, they will drown beneath a sea of tiny little functions.
2. People are afraid that following this rule will obscure the intent of a function because they won't be able to see it all in one place.
3. People worry that the performance penalty of calling all those tiny little functions will cause their applications to run like molasses in the wintertime.
4. People think they're going to have to be bouncing around, from function to function, in order to understand anything.

5. People are concerned that the resulting functions will be too entangled to understand in isolation.

Let’s address these one at a time.

1. Drowning

Long ago our languages were very primitive. The first version of Fortran, for example, allowed you to write one function. And it wasn’t called a function. It was just called a program. There were no modules, no functions, and no subroutines. As I said, it was primitive.

COBOL wasn’t much better. You could jump around from one *paragraph* to the next using a `goto` statement; but the idea of functions that you could call with arguments was still a decade in the future.

When Algol, and later, C, came along, the idea of functions burst upon us like a wave. Suddenly we could create modules that could be individually compiled. Modular programming became all the rage.[3]

[3] And as with all things in software, it was badly abused. Some managers insisted that no module should be longer than 500 bytes. So programmers were forced to put a `goto` at the end of their modules in order to call the next.

But as the years went by, and our programs got larger and larger, we began to realize that functions weren’t enough. Our names started to collide. We were having trouble keeping them unique. And we really didn’t like the idea of prefixing every name with the name of its module. I mean, nobody likes `io_files_open` as a function name.

So we added classes and namespaces to our languages. These gave us lovely little cubbyholes that had names into which we could place our functions and our variables. Moreover, these namespaces and classes quickly became

hierarchical. We could have functions within namespaces that were within yet other namespaces.

And this is the solution for the “sea of tiny little functions” fear. You aren’t going to drown beneath that sea, because there won’t be a sea. Instead, you’ll have a nicely arranged and nicely named hierarchy of namespaces and classes into which you can tuck your nicely named functions. Those nice names will be the guideposts that other programmers will use to locate your functions. Better still, those guideposts will provide the context that other programmers will use to understand your functions.

Let me say this a different way. Some of us really like working on largish functions. There’s something pleasing about the shape of those functions. We like the undulating in and out of the indentations. Something in our hindbrain finds comfort in it.

Why? Perhaps because, if you rotate the listing to the right by 90°, it looks like the horizon. Those undulations become peaks and valleys. And we, hominids who evolved on the savannah, used those shapes on the horizon to determine our location. We knew that between the two peaks to the east there was a watering hole; and the valley on the right was home to a ravenous saber-toothed tiger.

Remember my `gi` function? I knew that function like the back of my hand. I knew it *geographically*! If someone asked me to change the scaling on the *x*-axis, I would have said to myself: “The scaling on the *x*-axis . . . that’s in the third indent after the big comment block.” And I would have scrolled down to the big comment block and counted 1..2..3 and there it would be—the code that set the scaling on the *x*-axis.

Now take a new programmer who’d never seen `gi`. Ask that poor sod to change the scaling on the *x*-axis. Where the hell

should he go? He doesn’t know where the watering hole is. He’s got no idea that there’s a saber-toothed tiger on the loose in that valley over there. He’s just going to fumble around inside that 3,000-line morass of code until, by luck, he stumbles upon something that looks like it might set the scaling on the x-axis.

Wouldn’t it be great for that poor programmer if there were a function named `setScalingOnX`? Wouldn’t it be just spectacular if that function were located in the `Axis` namespace? I hope I’ve made my point. So, on to point 2.

2. Small Functions Don’t Obscure Intent

The answer to point 2 is the same as the answer to point 1. You cannot obscure the intent of the code by putting that code into a nicely named organizational structure. A well-designed and well-named structure of namespaces, classes, and functions *reveals* intent. It does not obscure it. More on this later.

3. Performance

OK, what about point 3? Performance: This is a real fear. But it is real only for those programmers who must preserve and protect every nanosecond under their control. If you are writing a first-person shooter game, or a high-speed trading app, or the guidance system for a hypersonic missile, then nanoseconds can be pretty important.

On the other hand, if you are writing a travel website, or a payroll system, or even the autopilot for a small airplane, nanoseconds are not your concern.

Long ago a function call took several microseconds. Machines were slow back then. So we were careful about how many functions we called. But nowadays, function calls take a few nanoseconds. There's almost no way that you can appreciably slow your application down by extracting methods. And, if your measurements show you that there is a slowdown, all you have to do is inline the most critical of those functions back again.

And remember, good compilers will often do that inlining for you, without your knowledge. If a good compiler sees that you've called a function from just one place, it's very likely to simply replace the call with the code within the function itself.

4. Bouncing Around

OK, then. Point 4: bouncing around. I agree with this point—or at least I *would* agree if the functions in question were randomly scattered hither and yon throughout the code. But who would do a thing like that? It'd be very rude.

I like to organize my extracted functions in the order that they are called. So, when one function calls another, all you have to do is scroll down to see the called function—right there. If function A calls B and C, then their order in the source file should be A, B, C.[4] How hard is that?

[4] In languages like Clojure, it would be C, B, A.

So, no, if the author is polite, you are not going to be bouncing all around inside the code.

Instead, you'll be able to read down the length of the code, from high level to low level. Not only that, but the high-level code will be organized and separated from the low-level code—because when you extract functions, you virtually always extract lower-level code from higher-level code.

Indeed, I'll show you an example of that later in this chapter. But first I want to talk about one last thing.

5. Entanglement

Of all the complaints, this is the most reasonable, and it is the point that John Ousterhout makes in the appendix. It is undeniably true that when you extract one function from another, you may be entangling the logic of those two functions. To say that differently, the lower-level function might not be independent of the operation of the higher-level function. To understand the lower-level function you must have a good understanding of the higher-level function.

If this entanglement is severe—that is if there are several different facts established by the higher-level function that you must keep in mind while reading the lower-level function—then there is a reasonable chance that the extraction might not be worth doing. However, if the entanglement is minimal, then the fact that the extracted function has a nice descriptive name and that fact that it appears directly after its parent function will likely make the extraction worthwhile.

Or to say this even more differently: It's a judgment call. Choose wisely.

What Are Large Functions Anyway?

Once again, imagine my `gi` function from long ago. Let's imagine that it was written in Java and that it began like this:

```
public class GraphicInterpreter {

public void gi() {
```

```
    int i;
    int j;

    ...
  }
}
```

As we scroll down through the 3,000 lines of the `gi` function, we notice an `if` statement:

```
    if (...) {
      ...
      i++;
      j++;
      ...
    }
```

We decide we'd like to extract the body of that `if` statement into a new method. So we select the body with our mouse, and we invoke the Extract Method menu item. But the IDE complains that it cannot extract that method because the code modifies *two* local variables. Had the code modified only one local variable, the IDE could have returned the new value from the extracted method. But the IDE has no good way to return two variables. (This is Java and not Go, after all).

So, what should we do? How are we going to extract the body of that `if` statement into a new method? Well, what if we did this:

```
public class GraphicInterpreter {
  private int i;
```

```
  private int j;


  public void gi() {
    ...
    if (...) {
      ...
      i++;
      j++;
      ...
    }
  }
}
```

I know what you are thinking: “Oh no! You promoted local variables into instance variables! You are violating encapsulation!” True. I am. But now I can extract the body of my `if` statement into a new function because the two variables remain in scope within the class:

```
public class GraphicInterpreter {
  private int i;
  private int j;
  public void gi() {


    ...
    if (...) {
      f();
    }
  }
```

```
  private void f() {
  ...
  i++;
  j++;
  ...
  }
}
```

Let’s keep doing this. We find more and more `if` and `while` statements whose bodies manipulate `i` and `j`, and that we can extract. Indent after indent we extract and extract and extract. And what is the result?

The result is a whole bunch of methods that use the instance variables `i` and `j`. What do you call it when you’ve got a bunch of methods that manipulate a set of variables? You call that a class, don’t you?

Now, think about this carefully.

Every large function is really hiding a class (or more) inside it.

Because every large function has a set of local variables and a set of indented regions that manipulate those variables. Those variables should really be fields of a class, and those indented regions should be methods of that class.

But to channel George Carlin, perhaps I’ve gone a bit too far, a bit too fast. So let’s look at an example.

Allow me to introduce the famous Video Store example from the first edition of Martin Fowler’s wonderful book, *Refactoring*. Back in 1999 Martin wrote this little example in Java. Here, we’re going to see it in Go. But never fear, if you don’t know Go, it’s pretty easy to understand.

First come the tests, which Martin did *not* include in his book so long ago.

```
———customer_test.go———
package GoVideoStore

import (
  "testing"
)

var customer *Customer

func setup() {
  customer = NewCustomer("Fred")
}

func assertEquals(t *testing.T, expected string, statement string)
  if statement != expected {
    t.Errorf("\nExpected:\n%s\n\nGot:\n%s", expected, statement)
  }
}

func TestSingleNewReleaseStatement(t *testing.T) {
  setup()
  customer.addRental(NewRental(NewMovie("The Cell", NewRelease), 3
  assertEquals(t,
    "Rental Record for Fred\n"+
      "\tThe Cell\t9.0\n"+
      "Amount owed is 9.0\n"+
```

```
      "You earned 2 frequent renter points",
    customer.statement())
}

func TestDualNewReleaseStatement(t *testing.T) {
  setup()
  customer.addRental(NewRental(NewMovie("The Cell", NewRelease), 3
  customer.addRental(NewRental(
                      NewMovie("The Tigger Movie", NewRelease), 3
  assertEquals(t,
    "Rental Record for Fred\n"+
      "\tThe Cell\t9.0\n"+
      "\tThe Tigger Movie\t9.0\n"+
      "Amount owed is 18.0\n"+
      "You earned 4 frequent renter points",
    customer.statement())
}

func TestSingleChildrensStatement(t *testing.T) {
  setup()
  customer.addRental(NewRental(
                      NewMovie("The Tigger Movie", Childrens), 3)
  assertEquals(t,
    "Rental Record for Fred\n"+
      "\tThe Tigger Movie\t1.5\n"+
      "Amount owed is 1.5\n"+
      "You earned 1 frequent renter points",
    customer.statement())
```

```
}

func TestMultipleRegularStatement(t *testing.T) {
  setup()
  customer.addRental(NewRental(
                       NewMovie("Plan 9 from Outer Space", Regular
                       1))
  customer.addRental(NewRental(NewMovie("8 1/2", Regular), 2))
  customer.addRental(NewRental(NewMovie("Eraserhead", Regular), 3)
  assertEquals(t,
    "Rental Record for Fred\n"+
      "\tPlan 9 from Outer Space\t2.0\n"+
      "\t8 1/2\t2.0\n"+
      "\tEraserhead\t3.5\n"+
      "Amount owed is 7.5\n"+
      "You earned 3 frequent renter points",
    customer.statement())
}
```

And then there's the code that passes these tests.

```
———Customer.go———
package GoVideoStore

import "fmt"

type Customer struct {
  name    string
  rentals []*Rental
```

```
}

func NewCustomer(name string) *Customer {
  return &Customer{name: name}
}

func (customer *Customer) addRental(rental *Rental) {
  customer.rentals = append(customer.rentals, rental)
}

func (customer *Customer) statement() string {
  totalAmount := 0.0
  frequentRenterPoints := 0
  result := "Rental Record for " + customer.name + "\n"
  for _, rental := range customer.rentals {
    thisAmount := 0.0
    switch rental.movie.movieType {
    case NewRelease:
      thisAmount += float64(rental.daysRented * 3)
    case Regular:
      thisAmount += 2
      if rental.daysRented > 2 {
        thisAmount += float64(rental.daysRented-2) * 1.5
      }
    case Childrens:
      thisAmount += 1.5
      if rental.daysRented > 3 {
        thisAmount += float64(rental.daysRented-3) * 1.5
```

```
      }
    }
    frequentRenterPoints++
    if rental.movie.movieType == NewRelease && rental.daysRented >
      frequentRenterPoints++
    }
    result += fmt.Sprintf("\t%s\t%.1f\n", rental.movie.title, this
    totalAmount += thisAmount
  }
  return result +
    fmt.Sprintf(
      "Amount owed is %.1f\nYou earned %d frequent renter points",
      totalAmount, frequentRenterPoints)
}
———Rental.go———
package GoVideoStore

type Rental struct {
  movie      *Movie
  daysRented int
}

func NewRental(movie *Movie, daysRented int) *Rental {
  return &Rental{movie: movie, daysRented: daysRented}
}

———Movie.go———
package GoVideoStore
```

```
const (
  Regular int = iota
  NewRelease
  Childrens
)

type Movie struct {
  title     string
  movieType int
}

func NewMovie(title string, movieType int) *Movie {
  return &Movie{title: title, movieType: movieType}
}
```

The first thing we're going to do is attack the tests. The tests need a better design, because those tests are a *mess*! Here's why.

- The tests are all conducted through the UI. The statement strings *are* the UI, and they are volatile. Marketing people will change those strings on a whim. For example, if a marketing person says that the title of each statement should be `Rental Statement for…` instead of `Rental Record for …`, every test will break and will have to be changed. That makes the tests fragile. So, instead of testing the format of the statement many times, we should only test it once. The other tests should focus on the values being computed.

- Names like `Fred` and `The Cell` tell us nothing about what is being tested. Those names should be changed to help the reader of the tests understand what the tests are about. In general, tests should not contain production data. They should contain data that explains the intent of the test.
- The tests are wordy. We should be able to tighten them up a bit by using some variables for the movies, and a *composed assertion*.

```
———customer_test.go———
package GoVideoStore

import (
  "testing"
)

var customer *Customer
var newRelease1 *Movie
var newRelease2 *Movie
var childrensMovie *Movie
var regular1 *Movie
var regular2 *Movie
var regular3 *Movie

func setup() {
  customer = NewCustomer("Customer Name")
  newRelease1 = NewMovie("New Release 1", NewRelease)
  newRelease2 = NewMovie("New Release 2", NewRelease)
  childrensMovie = NewMovie("Childrens", Childrens)
```

```
  regular1 = NewMovie("Regular 1", Regular)
  regular2 = NewMovie("Regular 2", Regular)
  regular3 = NewMovie("Regular 3", Regular)
}

func assertStringsEqual(t *testing.T, expected string, actual stri
  if actual != expected {
    t.Errorf("\nExpected:\n%s\n\nGot:\n%s", expected, actual)
  }
}

func assertIntsEqual(t *testing.T, expected int, actual int) {
  if actual != expected {
    t.Errorf("\nExpected:%d Got:%d", expected, actual)
  }
}

func assertFloatsEqual(t *testing.T, expected float64, actual floa
  if actual != expected {
    t.Errorf("\nExpected:%f Got:%f", expected, actual)
  }
}

func assertOwedAndPoints(t *testing.T, owed float64, points int) {
  customer.statement()
  assertFloatsEqual(t, owed, customer.totalAmount)
  assertIntsEqual(t, points, customer.frequentRenterPoints)
}
```

```
func TestTotalsForOneNewRelease(t *testing.T) {
  setup()
  customer.addRental(NewRental(newRelease1, 3))
  assertOwedAndPoints(t, 9.0, 2)
}

func TestTotalsForTwoNewReleases(t *testing.T) {
  setup()
  customer.addRental(NewRental(newRelease1, 3))
  customer.addRental(NewRental(newRelease2, 3))
  assertOwedAndPoints(t, 18.0, 4)
}

func TestTotalsForOneChildrensMovie(t *testing.T) {
  setup()
  customer.addRental(NewRental(childrensMovie, 3))
  assertOwedAndPoints(t, 1.5, 1)
}

func TestTotalsForManyRegularMovies(t *testing.T) {
  setup()
  customer.addRental(NewRental(regular1, 1))
  customer.addRental(NewRental(regular2, 2))
  customer.addRental(NewRental(regular3, 3))
  assertOwedAndPoints(t, 7.5, 3)
}
```

```
func TestStatementFormat(t *testing.T) {
  setup()
  customer.addRental(NewRental(regular1, 1))
  customer.addRental(NewRental(regular2, 2))
  customer.addRental(NewRental(regular3, 3))
  assertStringsEqual(t,
    "Rental Record for Customer Name\n"+
      "\tRegular 1\t2.0\n"+
      "\tRegular 2\t2.0\n"+
      "\tRegular 3\t3.5\n"+
      "Amount owed is 7.5\n"+
      "You earned 3 frequent renter points",
    customer.statement())
}
```

Remember, in the discussion about `gi`, how I broke encapsulation by moving some local variables into fields of the containing class? The changes I made to the tests caused me to do the same thing to the production code. I moved the `totalAmount` and `frequentRenterPoints` local variables into instance variables of the `Customer` class.

```
———Customer.go———
…
type Customer struct {
  name                 string
  rentals              []*Rental
  totalAmount          float64
  frequentRenterPoints int
```

```
}

…
```

You may recall that the reason I said we needed to break encapsulation was in order to support extraction. So let’s get to that extraction. You’re going to see me *extract till I drop*. Here’s the first pass.

```
——Customer.go——
package GoVideoStore

import "fmt"

type Customer struct {
  name                 string
  rentals              []*Rental
  totalAmount          float64
  frequentRenterPoints int
}

func NewCustomer(name string) *Customer {
  return &Customer{name: name}
}

func (customer *Customer) addRental(rental *Rental) {
  customer.rentals = append(customer.rentals, rental)
}

func (customer *Customer) makeStatement() string {
  customer.clearTotals()
```

```
  return
    customer.makeHeader() +
    customer.makeDetails() +
    customer.makeFooter()
}

func (customer *Customer) clearTotals() {
  customer.totalAmount = 0.0
  customer.frequentRenterPoints = 0
}

func (customer *Customer) makeHeader() string {
  return "Rental Record for " + customer.name + "\n"
}

func (customer *Customer) makeDetails() string {
  rentalDetails := ""
  for _, rental := range customer.rentals {
    rentalDetails += customer.makeDetail(rental)
  }
  return rentalDetails
}

func (customer *Customer) makeDetail(rental *Rental) string {
  amount := customer.determineAmount(rental)
  customer.frequentRenterPoints += customer.determinePoints(rental
  customer.totalAmount += amount
  return customer.formatDetail(rental, amount)
```

```
}

func (customer *Customer) determineAmount(rental *Rental) float64
  amount := 0.0
  switch rental.movie.movieType {
  case NewRelease:
    amount += float64(rental.daysRented * 3)
  case Regular:
    amount += 2
    if rental.daysRented > 2 {
      amount += float64(rental.daysRented-2) * 1.5
    }
  case Childrens:
    amount += 1.5
    if rental.daysRented > 3 {
      amount += float64(rental.daysRented-3) * 1.5
    }
  }
  return amount
}

func (customer *Customer) determinePoints(rental *Rental) int {
  if rental.movie.movieType == NewRelease && rental.daysRented > 1
    return 2
  }
  return 1
}
```

```
func (customer *Customer)
formatDetail(rental *Rental, amount float64) string {
  return fmt.Sprintf("\t%s\t%.1f\n", rental.getTitle(), amount)
}


func (customer *Customer) makeFooter() string {
  return fmt.Sprintf("Amount owed is %.1f\n"+
    "You earned %d frequent renter points",
    customer.totalAmount, customer.frequentRenterPoints)
}
```

I imagine that some of you who are fluent in Golang are now running away in horror.[5] If so, I apologize—I am not an accomplished Golang programmer. Others of you might be screaming as you run down the hall because of all those *little tiny functions*. Aaaarrrgggghhh!

[5] See the Future Bob note near the end of this example.

But wait. Go back, right now, and read that `makeStatement` function. I'll wait.

How do you make a statement? You clear the totals and then compose the header, details, and footer together into a single string.

Now, if you think about it, *that's the high-level procedure*. Indeed, that's the high-level procedure for most reports—because most reports are composed of a header, a set of details, and a footer.

So, to those of you who feared that all this extracting would hide the intent of the code, I suggest quite the opposite has happened. The extraction has *exposed* the high-level intent of the code and simply moved the lower-level details into

other functions, where they cannot distract you. If you have been told to fix a bug in the footer of this statement, you now know exactly where to go: the `makeFooter` function.

This code is polite. It is polite because, when the poor reader looks at it, there's little to decode. The reader does not have to pore and ponder over strings and variables and loops and `if` statements. Instead, the reader sees a simple statement of policy with directions pointing to the answers to any questions they might have.

Because this code is polite, it's fair to say that the original code was rude. And we don't want to be rude, do we?

Let's continue. It's pretty easy to see how to `clearTotals`. It's also pretty obvious how to make the header and the footer. Making the details is just a loop over all the rentals that simply calls `makeDetail`.

How do you make a detail? You determine the amount, then you determine the points, and then you format the detail. Yikes! How obvious can you get?

Formatting the detail is pretty obvious too. But those two `determine` functions are a bit problematic.

Look through all those methods. The first argument is always `customer *Customer`. This is just Go's way of saying that the function is part of the `Customer` class. Now look at the bodies of all the functions. Notice that they all use the `customer` argument—except for `formatDetail`, `determineAmount`, and `determinePoints`. We'll deal with `formatDetail` later. For now, there's an awful lot of code in those two `determine` functions that doesn't deal with the `Customer` class that they belong to. So . . . maybe they don't belong to that class.

What else gets passed into those `determine` functions? The `Rental`! So maybe those two `determine` functions belong in the

`Rental` class. Let’s move them there.

```
——Customer.go——
func (customer *Customer) makeDetail(rental *Rental) string {
  amount := rental.determineAmount()
  customer.frequentRenterPoints += rental.determinePoints()
  customer.totalAmount += amount
  return customer.formatDetail(rental, amount)
}



——Rental.go——
Package GoVideoStore

type Rental struct {
  movie      *Movie
  daysRented int
}

func (rental Rental) getTitle() string {
  return rental.movie.title
}

func (rental Rental) determineAmount() float64 {
  amount := 0.0
  switch rental.movie.movieType {
  case NewRelease:
    amount += float64(rental.daysRented * 3)
  case Regular:
```

```
    amount += 2
    if rental.daysRented > 2 {
      amount += float64(rental.daysRented-2) * 1.5
    }
  case Childrens:
    amount += 1.5
    if rental.daysRented > 3 {
      amount += float64(rental.daysRented-3) * 1.5
    }
  }
  return amount
}

func (rental Rental) determinePoints() int {
  if rental.movie.movieType == NewRelease && rental.daysRented > 1
    return 2
  }
  return 1
}

func NewRental(movie *Movie, daysRented int) *Rental {
  return &Rental{movie: movie, daysRented: daysRented}
}
```

The tests still pass, unchanged. The tests have no idea that this fundamental change in design has happened. That means the tests have a design that tolerates change!

However, something even more magical has taken place. All the business rules that were in the `Customer` class have fled to

the `Rental`. The `Customer` class is now all about the formatting of the statement—the business calculations, other than a few sums, are now done in the `Rental`.

This calls into question why the `Customer` class was given that name. It's not about the customer anymore—if it ever was to begin with. Now it's all about the rental statement. So that's what its name should be: `RentalStatement`.

```
——RentalStatement.go——

…
type RentalStatement struct {
  name                 string
  rentals              []*Rental
  totalAmount          float64
  frequentRenterPoints int
}
…
```

But now let's consider that `Rental` class. The `determine` functions are strongly coupled to the `Rental` class in that they use both the `movie` and the `daysRented` fields. That suggests a strong cohesion. However, the `determine` functions are very strongly coupled to the `Movie` class as well, and that's a bit troubling.

The coupling to the `Movie` is all about the `movieType`. That type field is used to select the various business rules for calculating the amount and the points. So why are the type fields and the type constants in the `Movie` class?

The original tests contain a hint. If you look back, you'll see that `"The Tigger Movie"` was considered both a `NewRelease` and a `Childrens` movie. That suggests that a movie can be rented as a new release and later rented as a children's movie. And

that suggests that the type field actually belongs to the `Rental`. Let's try that.

```
———Rental.go———
…
type Rental struct {
  movie      *Movie
  daysRented int
  rentalType int
}

const (
  Regular int = iota
  NewRelease
  Childrens
)
…
func NewRental(movie *Movie, rentalType int, daysRented int) *Rent
  return &Rental{
    movie:      movie,
    rentalType: rentalType,
    daysRented: daysRented}
}
…
```

This change affected the tests because the `NewRental` constructor was changed to include the type. But overall this was a pretty quick change. However, it begs another question. Whenever you have a `switch` statement that acts on

that type code, you might want to consider a polymorphic dispatch.

Why? Because `switch` statements are like gerbils—given enough time they'll reproduce all over your code and fill it to the brim. Indeed, there's already a baby `switch` statement growing in the `determinePoints` method.

In Chapter 12, "Objects and Data Structures," we'll talk about the pros and cons of `switch` statements. For now, let's see if we can get rid of these by creating a type hierarchy.

```
——RentalType.go——
package GoVideoStore

type RentalType interface {
  determineAmount(daysRented int) float64
  determinePoints(daysRented int) int
}

——Rental.go——
package GoVideoStore

type Rental struct {
  movie      *Movie
  daysRented int
  rentalType RentalType
}

func (rental Rental) getTitle() string {
  return rental.movie.title
}
```

```
func (rental Rental) determineAmount() float64 {
  return rental.rentalType.determineAmount(rental.daysRented)
}

func (rental Rental) determinePoints() int {
  return rental.rentalType.determinePoints(rental.daysRented)
}

func NewRental(movie *Movie,
               rentalType RentalType,
               daysRented int) *Rental {
  return &Rental{
    movie:      movie,
    rentalType: rentalType,
    daysRented: daysRented}
}

———NewReleaseRental.go———
package GoVideoStore

type NewReleaseRental struct{}

func (rentalType NewReleaseRental)
determineAmount(daysRented int) float64 {
  return float64(daysRented * 3)
}
```

```
func (rentalType NewReleaseRental) determinePoints(daysRented int)
  if daysRented > 1 {
    return 2
  }
  return 1
}

———ChildrensRental.go———
package GoVideoStore

type ChildrensRental struct{}

func (rentalType ChildrensRental)
determineAmount(daysRented int) float64 {
  amount := 1.5
  if daysRented > 3 {
    amount += float64(daysRented-3) * 1.5
  }
  return amount
}

func (rentalType ChildrensRental) determinePoints(daysRented int)
  return 1
}

———RegularRental.go———
package GoVideoStore
```

```
type RegularRental struct{}


func (rentalType RegularRental) determineAmount(daysRented int) fl
  amount := 2.0
  if daysRented > 2 {
    amount += float64(daysRented-2) * 1.5
  }
  return amount
}


func (rentalType RegularRental) determinePoints(daysRented int) in
  return 1
}
```

That’s a nice polymorphic dispatch. Now all the business rules for each type are safely encapsulated within their own type, and the `Rental` couldn’t care less which one it is using. Of course, the tests had to change, but only a little bit. Here’s just one to give you the idea.

```
——RentalStatementTest.go——
func TestTotalsForOneNewRelease(t *testing.T) {
  setup()
  statement.addRental(NewRental(newRelease1, NewReleaseRental{}, 3
  assertOwedAndPoints(t, 9.0, 2)
}
```

The `Movie` class is virtually empty at this point.

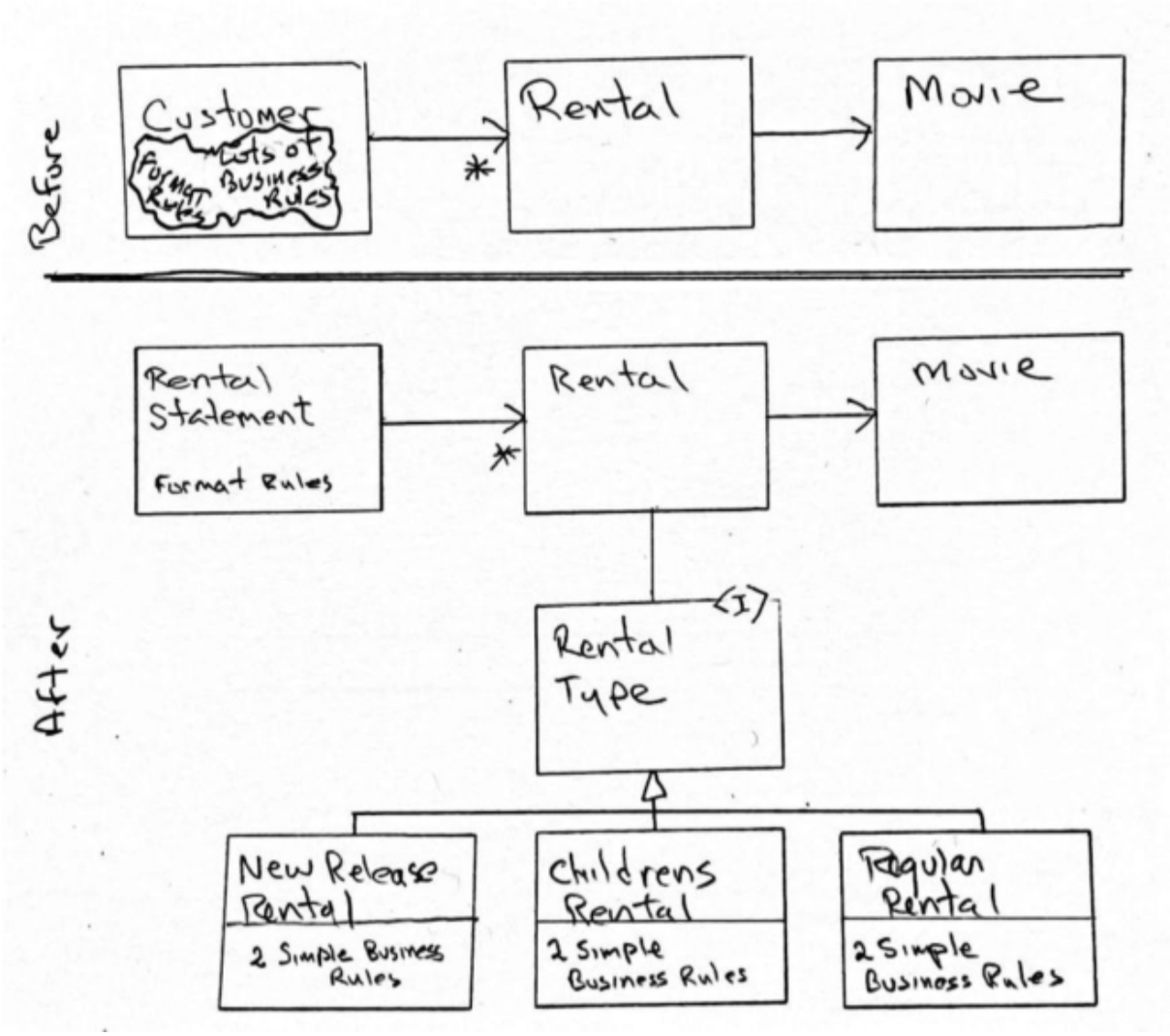
```
———Movie.go———
package GoVideoStore

type Movie struct {
  title string
}

func NewMovie(title string) *Movie {
  return &Movie{title: title}
}
```

And the `RentalStatement` class is concerned only with the formatting of the statement—which is why the `FormatDetail` function belongs there even though it doesn't use any of the `RentalStatement`'s fields.

Now, let's consider what we've just done. The diagram below shows a before-and-after picture.



Our cleanup procedure extracted the business rules and separated them from the formatting rules. Then, it moved those business rules, first into the `Rental` and then decomposed them down into the polymorphic hierarchy. Nice! And in the midst of all that, we realized that some names and concepts were wrong—and that's pretty typical. When you clean things up, you see things that were hidden before.

Finally, notice that there is an architectural boundary forming below the `RentalType`. I didn't draw it, but you ought to be able to sense it. All those little business rule details are isolated from the higher-level policies in `RentalStatement` and

`Rental`. Now when the requirements change, we can add lots of new `RentalType`s; and none of this code will need to change to accommodate those changes. That's the Open–Closed Principle (OCP) at work.

So, remember when I said that large functions are always hiding classes? That's what we found here. That one large `statement` function in the original code got broken up and distributed into three new classes and one existing class.

> **Future Bob:** Robert Laszczak happened to be perusing GitHub and saw the above meager attempt at Golang. He has kindly offered a blog post with improvements to this example that are more consistent with common Golang style. You can see this at https://threedots.tech/clean-code/. For my part, as a Golang novice, I agree with most of his changes, but I prefer to keep my names a bit longer, and prefix my getters with get.

Extraction and Classes

Consider the following Java code for solving the quadratic formula. The class is just a convenient namespace for the `solve` function.

```
import java.util.Arrays;
import java.util.List;


import static java.util.Arrays.asList;


public class QuadraticFormula {
  public static List<Double> solve(double a, double b, double c) {
    if (a == 0) {
```

```
      if (b == 0) {
        return asList();
      }
      return asList(-c / b);
    } else {
      double d = b * b - 4 * a * c;
      if (d > 0) {
        double sqrtd = Math.sqrt(d);
        double x1 = (-b + sqrtd) / (2 * a);
        double x2 = (-b - sqrtd) / (2 * a);
        return asList(x1, x2);
      } else if (d == 0) {
        return asList(-b / (2 * a));
      } else {
        return asList();
      }
    }
  }
}
```

This code is pretty straightforward, but let’s pretend that it’s more complicated and involved than shown here. Let’s also assume that it is going to *grow* with time. We might improve the readability by doing a few simple extractions and some renaming.

```
import java.util.List;
import static java.util.Arrays.asList;


public class QuadraticFormula {
```

```
public static List<Double> solve(double a, double b, double c) {
  if (a == 0)
    return linearSolution(b, c);
  return quadraticSolution(a, b, c);
}

private static List<Double> linearSolution(double b, double c) {
  if (b == 0)
    return asList();
  return asList(-c / b);
}

private static
List<Double> quadraticSolution(double a, double b, double c) {
  double discriminant = b * b - 4 * a * c;
  if (discriminant < 0)
    return complexSolution();
  if (discriminant == 0)
    return singleQuadraticSolution(a, b);
  return twoQuadraticSolutions(a, b, discriminant);
}

private static List<Double> complexSolution() {
  return asList();
}

private static
List<Double> singleQuadraticSolution(double a, double b) {
  return asList(-b / (2 * a));
```

```
  }

  private static List<Double>
  twoQuadraticSolutions(double a, double b, double discriminant) {
    double sqrtd = Math.sqrt(discriminant);
    double x1 = (-b + sqrtd) / (2 * a);
    double x2 = (-b - sqrtd) / (2 * a);

    return asList(x1, x2);
  }
}
```

Now again, for the sake of argument, we are pretending that this is a much more involved system that is likely to grow. I hope you agree that the extractions have lent some organization to the code that might make something more complicated a bit easier to read. But having done this, we are missing the opportunity of taking advantage of the class. Rather than passing arguments between all those extracted functions, we can load those values into variables within the class.

```
import java.util.List;

import static java.util.Arrays.asList;

public class QuadraticFormula {
  private static double a, b, c, discriminant;

  public static List<Double> solve(double a, double b, double c) {
    QuadraticFormula.a = a;
    QuadraticFormula.b = b;
```

```
  QuadraticFormula.b = b;
  QuadraticFormula.c = c;
  if (a == 0)
    return linearSolution();
  return quadraticSolution();
}

private static List<Double> linearSolution() {
  if (b == 0)
    return asList();
  return asList(-c / b);
}


private static List<Double> quadraticSolution() {
  discriminant = b * b - 4 * a * c;
  if (discriminant < 0)
    return complexSolution();
  if (discriminant == 0)
    return singleQuadraticSolution();
  return twoQuadraticSolutions();
}

private static List<Double> complexSolution() {
  return asList();
}

private static List<Double> singleQuadraticSolution() {
  return asList(-b / (2 * a));
}
```

```
  }

  private static List<Double> twoQuadraticSolutions() {
    double sqrtd = Math.sqrt(discriminant);
    double x1 = (-b + sqrtd) / (2 * a);
    double x2 = (-b - sqrtd) / (2 * a);
    return asList(x1, x2);
  }
}
```

This makes things just a little bit cleaner. In a more complex module, this could make things *a lot* cleaner. However, there is a risk. Those static variables could get corrupted in a multithreaded environment. This is easy enough to correct by changing all the functions and variables from `static` to instance methods and variables. But I'm not going to worry about that for this example.

The bottom line is that sometimes when we extract functions, it is convenient and clean to allow those functions to communicate through variables in the scope (the class) that contains them.

So now let's look at this problem in a functional language like Clojure.

```
(defn quad [a b c]
  (let [discriminant (- (* b b) (* 4 a c))]
    (cond
      (zero? a)
      (if (zero? b)
        []
        [(/ (- c) b)])
```

```
      (zero? discriminant)
      [(/ (- b) (* 2 a))]

      (neg? discriminant)
      []

      :else
      (let [sqrt-desc (Math/sqrt discriminant)
            x1 (/ (+ (- b) sqrt-desc) (* 2 a))
            x2 (/ (- (- b) sqrt-desc) (* 2 a))]
        [x1 x2])))
```

This is very pretty. I would not be tempted to improve this by extracting. But again, let's pretend that this is a much larger and more involved function. How might we extract functions to improve it?

```
(defn quad [a b c]
  (let [discriminant (- (* b b) (* 4 a c))
        linear-solution
        (fn []
          (if (zero? b)
            []
            [(/ (- c) b)]))

        single-solution
        (fn []
          [(/ (- b) (* 2 a))])

        t
```

```
    two-solutions
    (fn []
      (let [sqrt-desc (Math/sqrt discriminant)
            x1 (/ (+ (- b) sqrt-desc) (* 2 a))
            x2 (/ (- (- b) sqrt-desc) (* 2 a))]
        [x1 x2]))]

(cond
  (zero? a) (linear-solution)
  (zero? discriminant) (single-solution)
  (neg? discriminant) []
  :else (two-solutions))))
```

OK, now for the last time, keep in mind that we are pretending that this module is larger and more complicated. That last `cond` statement is pretty nice. Notice that we're not passing arguments to the extracted functions. That's because a function in Clojure creates a kind of object that has variables and functions scoped within it. Notice also that those variables are not static, and so our concurrency worries disappear.

Conclusion

So, should you practice Extract Till You Drop? And if so, how much?

To some extent this is a matter of style and preference. You may like your functions a bit bigger than I like mine. Perhaps you are more comfortable with 20-line functions or 25-line functions. That's up to you.

However, remember what you just saw. It was the extraction of all those methods in the Video Store example that allowed us to see that some of that code did not belong in the class it was in. That was not evident before we did the extractions. So, if you are going to leave your functions larger, make sure you don’t miss those kinds of opportunities. In fact, you may find it worthwhile to extract first and then do a little strategic inlining once you have determined where everything actually belongs.

But remember, when you inline a function, you destroy its name.

Chapter 11. Be Polite



We want our modules to be polite. Polite modules follow Ward Cunningham's definition of cleanliness. As you read the code, you find it to be "pretty much what you expected". A module is a bounded set of functions and variables. This

might be composed of one class, several classes, or no classes at all. What matters are the functions and variables. A module may also be contained in a single source file or several source files. What matters is the cohesion of the functions and variables, and not the artificial boundaries that separate them. Another common word for a module is a *component*.

If you follow something roughly akin to Extract Till You Drop, then your modules will be composed of a few public functions and the many private functions that were extracted from the public functions. Again, these may be organized into classes so that your module contains a few public classes and several private classes.

The public functions, possibly organized into public classes, represent the interface of your module. The private functions, also possibly organized into private classes, represent the implementation of your module.

To borrow a concept from John Ousterhout's wonderful book *A Philosophy of Software Design*,[1] that module has a narrow interface and a deep implementation—and that's a good thing.

[1] [APOD].

Now, pick one of those public functions. It performs a task at the highest level of the module. The name of that function should identify that task in abstract terms. For example, consider the `makeStatement` function from the `RentalStatement` class in the previous chapter.

```
func (statement *RentalStatement) makeStatement() string {
  statement.clearTotals()
  return
    statement.makeHeader() +
    statement.makeDetails() +
```

```
    statement.makeDetails() +
    statement.makeFooter()
}
```

The name of the function is abstract. It tells you what the function does, but not how the function does it. Looking inside the `makeStatement` function we see four calls to private functions that were extracted from it. Each of those names is also abstract, but at a lower level than `makeStatement`. Let's look at `clearTotals`.

```
func (statement *RentalStatement) clearTotals() {
  statement.totalAmount = 0.0
  statement.frequentRenterPoints = 0
}
```

We can go no further. We've reached the end of the chain of function calls. The implementation of `clearTotals` tells us exactly how the totals are cleared, in excruciating detail. There is no meaningful lower level to descend to.

Let's look instead at `makeDetails`.

```
func (statement *RentalStatement) makeDetails() string {
  rentalDetails := ""
  for _, rental := range statement.rentals {
    rentalDetails += statement.makeDetail(rental)
  }
  return rentalDetails
}
```

The name of this function tells us what the function does at this level of abstraction. The implementation creates a loop

that calls yet another private method, `makeDetail`, which is one level of abstraction lower.

Thus, every public function creates a tree of calls to private functions at lower levels of abstraction.

Here, again, we see The Stepdown Rule. Each such descending call goes *one level of abstraction* down. Not two, not three. Just one. That's why the body of the loop within `makeDetails` calls the next function down. If I had placed the implementation of `makeDetail` within `makeDetails`, then `makeDetails` would have contained at least two levels of abstraction.

Do I always follow that rule? No, sometimes I allow two levels of abstraction within a function. Generally, however, I consider that to be rude.

The Newspaper Metaphor

Think of a well-written newspaper article. You read it vertically. At the top, you expect a headline that will tell you what the story is about and allow you to decide whether it is something you want to read. The first paragraph gives you a synopsis of the whole story, hiding all the details while giving you the broad-brush concepts. As you continue downward, the details increase until you have all the dates, names, quotes, claims, and other minutiae.

Most of us never read the entirety of an article. We start at the headline, and if it is interesting we read on. If that paragraph interests us we read on. And we stay in that very simple loop:

```
while (interested) readMore;
```

As soon as the article becomes boring, we stop. Articles that allow us to use that loop are *polite*—they let us out early. Articles that force us to waste time reading a lot of unnecessary detail are rude.

A newspaper is composed of many articles; most are very short. Some are a bit longer. Very few contain as much text as a page can hold. This makes the newspaper *usable*. If the newspaper were just one long story containing a disorganized agglomeration of facts, dates, and names, then we simply would not read it.

We would like the source code of a module to be like a newspaper article. The module's name should be simple but explanatory. The name, by itself, should be sufficient to tell us whether we are in the right module or not. The topmost parts of the module should provide the high-level concepts and algorithms. Detail should increase as we move downward,[2] until at the end we find the lowest-level functions and details in the module.

[2] Some languages, such as Clojure, invert the vertical ordering because they require that declarations appear before their use. Thus, low-level details go at the top and the higher-level policies go toward the bottom.

A newspaper article that was a mixture of two different stories would be an abomination. So too is a module that contains two or more different topics.[3] So the classes and functions within a module should all be related to a single topic—a single responsibility—with a nice structure that proceeds from high-level policy to low-level detail.

[3] The Single Responsibility Principle (SRP).

Be Polite

When you write functions that encapsulate a single level of abstraction, it allows your readers to escape early. They

don’t have to read a lot of code to understand what the function does at that level. Sometimes that level is all they care about, and they don’t want to be bothered by all the details from the lower levels.

For example, I consider the `makeStatement` function above to be polite. The reader can glance at it and see what it does. If the reader is not worried about anything else, then they are done and go back to whatever else they were doing.

Now consider the original code that `makeStatement` evolved from.

```
func (customer *Customer) statement() string {
  totalAmount := 0.0
  frequentRenterPoints := 0
  result := "Rental Record for " + customer.name + "\n"
  for _, rental := range customer.rentals {
    thisAmount := 0.0
    switch rental.movie.movieType {
    case NewRelease:
      thisAmount += float64(rental.daysRented * 3)
    case Regular:
      thisAmount += 2
      if rental.daysRented > 2 {
        thisAmount += float64(rental.daysRented-2) * 1.5
      }
    case Childrens:
      thisAmount += 1.5
      if rental.daysRented > 3 {
        thisAmount += float64(rental.daysRented-3) * 1.5
      }
```

```
    }
    frequentRenterPoints++
    if rental.movie.movieType == NewRelease && rental.daysRented >
      frequentRenterPoints++
    }
    result += fmt.Sprintf("\t%s\t%.1f\n", rental.movie.title, this
    totalAmount += thisAmount
  }
  return result +
    fmt.Sprintf(
      "Amount owed is %.1f\nYou earned %d frequent renter points",
      totalAmount, frequentRenterPoints)
}
```

I consider this code to be rude. I consider the author to have been inconsiderate. To understand the high-level intent of this code I must deal with all the low-level details—and that takes time.

Think of it this way. Let’s say that you were reading code somewhere and you came across a call to the polite `makeStatement()`. The name of the function tells you what it does, but perhaps you’d like to check that the statement includes the footer. So you click over to the implementation of `makeStatement` and you see four simple lines of code that tell you exactly what you need to know. Two seconds later you click back to where you were and continue reading, suffering no feeling of interruption—just a simple question asked and answered in two seconds.

Now let’s say that you were reading code somewhere and you came across a call to the rude `statement()` function. The name is a noun, which describes no action. You have to infer

that the function builds the statement. Again, you're not sure if the statement includes the footer. So you click over to the implementation and are faced with 30 lines of deeply detailed code. Is the footer in there? Does the code that builds the footer get called at the right time? A minute or two later you've satisfied yourself that the footer is intact, but now you've forgotten why you were worried about that. You click back to where you were, and you've lost your train of thought and have to backtrack several lines, and several more minutes, to remind yourself of what you were doing and re-create that train of thought in your mind.

That's rude. It's as rude as the guy who bursts into your cubicle and throws a question at you without waiting for you to acknowledge him.

The Stepdown Rule: Once Again

So, the rule I like to follow, in order to be polite, is to descend one level of abstraction at a time. Each function has a name that defines what the function does, and each function consists of a set of lines that describe the implementation and that are all at the same level of abstraction. Each of those lines, if it is not already at the lowest level, calls functions at the next level down.

How do you determine how big one level of abstraction is? All I can say to that is to use your best judgment. For me, it's pretty simple. If I *can* meaningfully extract, then I do the extraction and check that it makes sense.

This is a polite practice, and it helps avoid the abstraction roller coaster.

The Abstraction Roller Coaster

Look again at that rude 30-line `statement` function. Right at the top, why do `totalAmount` and `frequentRenterPoints` need to be set to zero? What are we doing? This is the first line of a high-level function, and we are confronted with the lowest-level kind of detail. Then, we pop up a level to deal with the `result`, which we must infer to be the header, and then there's a loop. The first line of the loop descends immediately to another low-level detail and then enters a `switch` statement.

Can you feel the roller coaster? We are going up and down through levels of abstraction. Sometimes we are high, as we set up the loop through the rentals, and sometimes we are low, as we zero out totals and check individual cases of the switch.

Riding that roller coaster is not fun for the reader. The reason is simple. Every time you go from a line at a high level to a line at a low level, you have to push your mental model up on your mental stack to deal with that low-level concern. Once you understand it, then you pop your mental stack back to pick up where you left off.

The problem with that is that we do not have a mental stack. When we push our train of thought aside to deal with a detail, more often than not we lose that train of thought. Again, it's like that guy who bursts into your cubicle and shouts a question at you.

This Is How We Write, but Not How We Want to Read

The problem is that we write code on the roller coaster. We all do. We all must. Imagine the person writing that original rude `statement` function. Perhaps the first line they wrote was the loop, but then they remembered to zero out the

`totalAmount`. Later, after they wrote the switch, they realized they needed to deal with the frequent renter points, and so they went back to the top and added that line.

In short, we construct our code by rolling through the undulating track of abstractions and details. Our code comes out looking like the roller coaster. And this is where Kent Beck's advice becomes critical:

First, make it work. Then, make it right.

You are not done when the code works. You are done when the code reads well. You are done when you have extracted and organized and laid out the tree of functions descending from the public functions in a manner convenient to the reader.

Now, again, I don't follow these rules religiously or dogmatically. There are situations in which following them is inappropriate. But these rules are my default. I will follow them unless I have a good reason not to.

Chapter 12. Objects and Data Structures



Niklaus Wirth once wrote a lovely book titled *Algorithms + Data Structures = Programs*.[1] Though it wasn't his intent,

the title very neatly captures the fundamental concept behind object-oriented programming: that there is a deep relation between functions and the data they manipulate.

[1] [A+DS=P].

Object orientation was invented[2]/discovered by Ole-Johann Dahl and Kristen Nygaard in the mid- to late ’60s through the gradual evolution of their Simula language, culminating in Simula 67. Bjarne Stroustrup and Alan Kay both used Simula 67 in the early ’70s and then extended the ideas in C++ and Smalltalk, respectively.

[2] The story of this invention is fascinating. I wrote about it in [We Programmers].

From there, C++ evolved into Java and C# and then Go, while Smalltalk evolved into Objective-C, Ruby, and Python. Both of those lines of evolution borrowed from each other, causing a lot of cross-pollination. Today, virtually all of our languages are object oriented in one sense or another.

What Is an Object?

An object is a bundle of data manipulated by dedicated functions, some of which are public. From the outside looking in, only those public functions are visible.[3] The structure of the data within the object is hidden.

[3] Some languages enforce that visibility with keywords like `public` and `protected`. Other languages depend upon programmers to adopt conventions to determine which functions are public and private.

An object-oriented program is composed of a group of such objects, each with its own responsibilities and each collaborating with the others by calling each other’s public functions.

Alan Kay once imagined this using a biology analogy—a set of cells collaborating by passing molecular messages to

each other. The cells do not know of each other's internals, and may not even know the particular cells they are communicating with. They simply emit messages (molecules) that other cells detect and respond to. Notice that cells never ask other cells questions. They send messages but do not expect answers.

The message-passing concept is a very powerful principle for organizing behavior. If you only tell an object what to do and never ask that object anything, then the data in that object is not relevant to you.

Avoiding all queries is impractical in software; but as we shall see, there are ways to keep such queries to a minimum. This lack of dependence upon data means that object-oriented designs focus primarily on behavior. Objects are constructed and related so as to support the flow of messages that elicit the required behaviors of the system. The internal structure of the data within the individual objects is hidden from, and therefore nearly irrelevant to, the majority of that behavior.

But OO is not the only game in town.

In 1970, Edgar Codd published *A Relational Model of Data for Large Shared Databanks*.[4] From there he, Raymond Boyce, and Chris Date derived/discovered the concept that currently drives all relational databases. The overarching concept is that the structure of the data is critical and visible, and all programs that use it are subservient to it.

[4] *Communications of the ACM* 13, no. 6 (June 1970), 377–387.

A large enterprise database is a complex, sprawling data structure, whereas the programs that manipulate it are little more than minions that surround it. The schema of the database dominates those programs.

This is the opposite of OO. To recap: An object is a set of public functions with data hidden deep inside. A database is a published set of data structures with no intrinsic functions. Again, they are very nearly opposites. And, as we shall see, that opposing nature goes very deep indeed.

Data Abstraction

There is a reason that we tend to keep our variables private. We don't want anyone else to depend on them. We want to keep the freedom to change their type or implementation on a whim or an impulse. Why, then, do so many programmers automatically add getters and setters to their objects, exposing their private variables as if they were public?

Consider the difference between `Point 1` and `Point 2` below. Both represent the data of a point on the Cartesian plane. And yet one exposes its implementation and the other completely hides it.

```
——Point 1——
public class Point {
  public double x;
  public double y;
}


——Point 2——
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
```

```
  void setPolar(double r, double theta);
}
```

The beautiful thing about `Point 2` is that there is no way you can tell whether the implementation is in rectangular or polar coordinates. It might be neither! The interface represents an *abstract* data structure. We can tell that there must be a data structure in there somewhere, but we do not know what form that data structure takes.

But `Point 2` represents more than just a data structure. It also represents behavioral policies. Despite the fact that the getter methods allow you to access individual coordinate values, the two setter methods enforce the policy that the two coordinate values are set together as an atomic operation.

`Point 1`, on the other hand, is very clearly implemented in rectangular coordinates, and it forces us to manipulate the coordinate values independently. This exposes implementation. Indeed, it would expose implementation even if the variables were private and we were using single-variable getters and setters.

Hiding implementation is not just a matter of putting a layer of functions between the variables. Hiding implementation is about abstractions! A class does not simply push its variables out through getters and setters. Rather, it exposes abstract interfaces that allow its users to manipulate the *essence* of the data, without having to know how that data is implemented.

Consider `Vehicle 1` and `Vehicle 2` below. The first uses concrete terms to communicate the fuel level of a vehicle, whereas the second does so with the abstraction of percentage. In the concrete case, you can be pretty sure that these are just

accessors of variables. In the abstract case, you have no clue at all about the form of the data.

```
——Vehicle 1——

public interface Vehicle {

  double getFuelTankCapacityInGallons();

  double getGallonsOfGasoline();

}


——Vehicle 2——

public interface Vehicle {

  double getPercentFuelRemaining();

}
```

In both of the above cases, the second option is preferable. We do not want to expose the details of our data. Rather, we want to express our data in abstract terms. This is not merely accomplished by using interfaces and/or getters and setters. Serious thought needs to be put into the best way to represent the data that an object contains. The worst option is to blithely add getters and setters.

Data/Object Antisymmetry

These two examples show the difference between objects and data structures. Objects hide their data behind abstractions and expose functions that operate on that data. Data structures expose their data and have no meaningful functions. Go back and read that again. Notice the complementary nature of the two definitions. They are virtual opposites. This difference may seem trivial, but it has far-reaching implications.

Consider, for example, the `Procedural Shapes` example below. The `Geometry` class operates on the three shape classes. The shape classes are simple data structures without any behavior. All the behavior is in the `Geometry` class.

```
——Procedural Shapes——
public class Square {
  public Point topLeft;
  public double side;
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}

public class Geometry {
  public final double PI = 3.141592653589793;

  public double area(Object shape) throws NoSuchShapeException
  {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
```

```
    }
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

Object-oriented programmers might wrinkle their noses at this and complain that it is procedural—and they'd be right. But the sneer may not be warranted. Consider what would happen if a `perimeter()` function were added to `Geometry`. The shape data structures would be unaffected! Any other modules that depended upon those shapes would also be unaffected! On the other hand, if I add a new shape, I must change all the functions in `Geometry` to deal with it. Again, read that over. When you add a new function, the data structures remain unchanged; but when you add a new data structure, the functions must all be changed.

Now consider the `OO Shapes` example below. Here the `area()` method is polymorphic. No `Geometry` class is necessary. So, if I add a new shape, none of the existing functions are affected, but if I add a new function, all of the shapes must be changed![5]

[5] There are ways around this that are well known to experienced object-oriented designers: Visitor, or dual-dispatch, for example. But these techniques carry costs of their own and generally return the structure to that of a procedural program.

```
——OO Shapes———
public class Square implements Shape {
  private Point topLeft;
  private double side;
  public double area() {
    return side*side;
  }
}
public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;

  public double area() {
    return height * width;
  }
}
public class Circle implements Shape {
  private Point center;
  private double radius;
  public final double PI = 3.141592653589793;
  public double area() {
    return PI * radius * radius;
  }
}
```

Again we see the complementary nature of these two definitions; they are virtual opposites! This exposes the fundamental dichotomy between objects and data

structures: Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures. OO code, on the other hand, makes it easy to add new data structures (classes) without changing existing functions.

The complement is also true: Procedural code makes it hard to add new data structures, because all the functions must change. OO code makes it hard to add new functions, because all the classes must change.

So, the things that are hard for OO are easy for procedures, and the things that are hard for procedures are easy for OO.

In any complex system, there are going to be times when we want to add new data types rather than new functions. For these cases, objects and OO are most appropriate. On the other hand, there will also be times when we'll want to add new functions as opposed to data types. In these cases, procedural code and data structures will be more appropriate.

Mature programmers know that the idea that everything is an object is a myth. Sometimes you really do want simple data structures with procedures operating on them.

The Law of Demeter

There is a well-known heuristic called the Law of Demeter[6] that recommends that a module should not know about the innards of the objects it manipulates. As we saw in the last section, objects hide their data and expose operations. This means that an object should not expose its internal structure through accessors, because to do so is to expose, rather than to hide, its internal structure.

[6] http://en.wikipedia.org/wiki/Law_of_Demeter

More precisely, the Law of Demeter says that a method *f* of a class *C* should only call the methods of the following:

- *C*
- An object created by *f*
- An object passed as an argument to *f*
- An object held in an instance variable of *C*

The method should not invoke methods on objects that are returned by any of the allowed functions. In other words, be shy, talk to friends, and don't talk to strangers.

The following code[7] appears to violate the Law of Demeter (among other things) because it calls the `getScratchDir()` function on the return value of `getOptions()` and then calls `getAbsolutePath()` on the return value of `getScratchDir()`.

[7] Found somewhere in the Apache framework.

```
final String outputDir =

  ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Trainwrecks

This kind of code is often called a *trainwreck* because it look like a bunch of coupled train cars. Chains of calls like this are generally considered to be sloppy style and should be avoided. It is usually best to split them up as follows:

```
Options opts = ctxt.getOptions();


File scratchDir = opts.getScratchDir();

final String outputDir = scratchDir.getAbsolutePath();
```

Are these two snippets of code violations of the Law of Demeter? Certainly the containing module knows that the `ctxt` object contains options, which contain a scratch directory, which has an absolute path. That's a lot of knowledge for one function to know. The calling function knows how to navigate through a lot of different objects.

Whether this is a violation of Demeter depends on whether or not `ctxt`, `Options`, and `ScratchDir` are objects or data structures. If they are objects, then their internal structure should be hidden rather than exposed, and so knowledge of their innards is a clear violation of the Law of Demeter. On the other hand, if `ctxt`, `Options`, and `ScratchDir` are just data structures with no behavior, then they naturally expose their internal structure, and so Demeter does not apply.

The use of accessor functions in the examples above confuses the issue. If the code had been written as follows, then we probably wouldn't be worried about Demeter violations.

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

This issue would be a lot less confusing if we adopted the convention that data structures had public variables and no functions, whereas objects had private variables and public functions that abstracted the implementation.

Hybrids

This confusion sometimes leads to unfortunate hybrid structures that are half object and half data structure. They have functions that do significant things, and they also have either public variables or public accessors and mutators that, for all intents and purposes, make the private variables

public, tempting other external functions to use those variables the way a procedural program would use a data structure.[8]

[8] This is sometimes called *feature envy* in [Refactoring].

Such hybrids make it hard to add new functions but also make it hard to add new data structures. They are the worst of both worlds. Avoid creating them. They are indicative of a muddled design whose authors are unsure of—or worse, ignorant of—whether they need protection from functions or types.

Hiding Structure

What if `ctxt`, `options`, and `scratchDir` are objects with real behavior? Then, because objects are supposed to hide their internal structure, we should not be able to navigate through them. How, then, would we get the absolute path of the scratch directory?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

or

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

The first option could lead to an explosion of methods in the `ctxt` object. The second presumes that `getScratchDirectoryOption()` returns a data structure, not an object. Neither option feels good.

If `ctxt` is an object, we should be telling it to do something; we should not be asking it about its internals. So why did we want the absolute path of the scratch directory? What were we going to do with it? Consider this code from (many lines farther down in) the same module:

```
String outFile = outputDir + "/" + className.replace('.', '/') + "
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

The admixture of different levels of detail is a bit troubling. Dots, slashes, file extensions, and `File` objects should not be so carelessly mixed together, and mixed with the enclosing code. Ignoring that, however, we see that the intent of getting the absolute path of the scratch directory was to create a scratch file of a given name.

So, what if we told the `ctxt` object to do this?

```
BufferedOutputStream bos =
  ctxt.createScratchFileStream(classFileName);
```

That seems like a reasonable thing for an object to do! This allows `ctxt` to hide its internals and prevents the current function from having to violate the Law of Demeter by navigating through objects it shouldn't know about.

So, we can often fix trainwrecks and Law of Demeter violations by telling objects to do what we want, as opposed to traversing the network of objects to gather all the resources. This advice is called *Tell the Other Guy*.

Long ago Dave Thomas and Andy Hunt wrote a lovely article titled "OO in One Sentence: Keep it DRY, Shy, and Tell the Other Guy."[9] DRY, of course, is Don't Repeat Yourself, and being shy refers to the Law of Demeter. Telling the other guy is a reference to Alan Kay's message-passing idea or command/query separation (CQS). Good advice all around.

[9] *IEEE Software* 21, no. 3 (May/June 2004), 101–103.

Data Transfer Objects

The quintessential form of a data structure is a context, class, or struct with public variables and no functions. This is sometimes called a *data transfer object*, or *DTO*. DTOs are very useful structures, especially when communicating with databases, parsing messages from sockets, and so on. They often become the first in a series of translation stages that convert raw data in a database into objects in the application code.

Early in the Java era, the “bean” form shown below was popular. Beans have private variables manipulated by getters and setters. The quasi-encapsulation of beans seems to make some OO purists feel better but usually provides no other benefit. The `Address` DTO might as well have public fields.

```
public class Address {
  private String street;
  private String streetExtra;
  private String city;

  private String state;
  private String zip;
  public Address(String street, String streetExtra,
                 String city, String state, String zip) {
    this.street = street;
    this.streetExtra = streetExtra;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }
```

```
  public String getStreet() {
    return street;
  }
  public String getStreetExtra() {
    return streetExtra;
  }
  public String getCity() {
    return city;
  }
  public String getState() {
    return state;
  }
  public String getZip() {
    return zip;
  }
}
```

The Object/Relational “Impedance Mismatch”

In the late ’80s, as OO grew in popularity, the difference between the OO concept of public behavior and private data and the relational concept of public data and no behavior became clear. Programmers sought a merger of the two ideas; but they were disappointed with the result. For a while, object-oriented databases were tried; but these never really solved the issue. In the end, we fell back on object-relational mappers (ORMs).

The hope was that ORMs would allow programmers who were using object-oriented languages to conveniently

access the data within a relational database. The ORM would query the database, unpack the data from the resulting table rows, and repackage the data into the data structures used by the objects in their programs.

This works well enough as a way to convert tables into a more convenient form, but it does not resolve the dilemma, because objects are more about functions than data. In the end, there is no way to map a relational database to an object, because databases contain data structures, and data structures and objects are orthogonal. All we can really manage to do is extract data from a database and place it into a form that an object can internalize. ORMs don’t create objects, they just load data structures.

Using Objects and Data Structures

So why do we have these two opposite concepts? Why do we sometimes want the data structure to be hidden and the behavior exposed and other times want the behavior to be hidden while the data structure is exposed?

It all comes down to which is more likely to grow and change.

If there are likely to be new behaviors, and if existing behaviors are likely to change, then we want to hide the volatile behaviors while exposing the static data structure. If, on the other hand, it is the data structures that are likely to grow and change, then we want to expose the static behavior and hide the volatile data structures.

We hide the volatile so that the changes don’t affect the static. We build a barrier that protects the static from the volatile.

Example 1: The structure of a large enterprise database is relatively static. The data itself may change as new records are added and old records are updated or deleted, but the structure changes very infrequently. However, the behaviors of the programs that use that database change frequently. Business rules are volatile, but business structure is stable in comparison.

Example 2: The behavior of an IDE is relatively static. It is an editor of text files. But the structure of the data changes every time a new language is added to its capabilities. Users expect the IDE to behave consistently, whether programming in Ruby, Python, Java, or Clojure; and yet those languages have remarkably different structures. The overall data structure is volatile compared to the behavior.

Determining whether data or behavior is the more static is a decision that requires some thought and experience. It is also a decision that may have to be made differently in different parts of a system.

A complex system can have components of both kinds. Some components may have behavior that is static relative to the data structure, and others may have data structures that are static relative to the behavior.

Oddly enough, all this boils down to the decision to use `switch` statements.

`Switch` Statements

`Switch` statements and their siblings, `if`/`else` chains, are considered by some to be bad form and by others to be utterly essential. The reason behind this disagreement is the same as the decision over static behavior or data structures. Consider this simple program.

——Shape.java——

```
enum ShapeType {CIRCLE, SQUARE}

public class Shape {
  public ShapeType type;
}

——Square.java——
class Square extends Shape {
  public int side;
  public Point topLeft;
}

——Circle.java——
class Circle extends Shape {
  public int radius;
  public Point center;
}

——ShapeManager.java——
public class ShapeManager {
  public static void renderShapes(List<Shape> shapes) {
    for (Shape shape : shapes) {
      switch (shape.type) {
        case CIRCLE:
          renderCircle((Circle) shape);
          break;
        case SQUARE:
          renderSquare((Square) shape);
          break;
```

```
          break;
      }
    }
  }


  private static void renderSquare(Square square) {
    //...
  }


  private static void renderCircle(Circle circle) {
    //...
  }
}
```

If you are a Java programmer, this might look a little odd to you. If you are a C programmer, this might look quite normal to you.

Now consider the consequence of adding a `Triangle`.

The first file that needs to change is *Shape.java*, because `TRIANGLE` has to be added to the enum. Next, of course, the `Triangle` data structure (class) must be written. And then, finally, the `Triangle` case needs to be added to the `switch` statement in the `ShapeManager` class.

This design forces several modules to change, and others to be added, in response to one simple change to the requirements. This violates the Single Responsibility Principle (SRP) and the Open–Closed Principle (OCP). The number of changes also means that this is a rigid design. The more changes that are required, the more rigid the design becomes.

But it’s worse than that, because that `renderShapes` method in `ShapeManager` is not the only method of that class. To save space I didn’t write them all, but there’s `dragShapes`, `eraseShapes`, `rotateShapes`, `scaleShapes`, and a menagerie of other functions that manipulate shapes. And every one of them has that very same `switch` statement in it. And every one of them is going to need the `Triangle` case added to it.

Except that some of those methods don’t use `switch` statements. Some of them use `if`/`else` chains. What’s more, clever programmers may use logical optimizations to eliminate or modify some of the `switch` statements and `if`/`else` chains. For example, the `switch` statement in `rotateShapes` may not contain a `CIRCLE` case, because a clever programmer realized that circles have rotational symmetry.

So, we are going to have to find all those `switch`/`if`/`else` statements, decode all the logical optimizations, and figure out some way to get them all to properly deal with the new `TRIANGLE` case. And, in large systems, we’ll almost certainly miss one . . . or more.

A design that forces many modifications to be made within modules when only one simple requirement is changed or added is both rigid and fragile—it’s hard to change and it breaks easily. The more changes that are required, the more fragile the design is.

But it’s worse than that. Because, you see, our business has been trying to sell this shape management system for several months now, and without a lot of success. We were all starting to worry about our jobs. Life at a startup rests on a tenuous thread. Fortunately, our CEO has just made a critical announcement. He *knows* what the problem is, and he knows how to solve it.

“Our problem,” he says, “is that we’ve been forcing our customers to deal with circles and squares. Customers don’t

want that. They want a shape management system that allows them to choose whether they use circles or squares. They don't want to pay for circles if they only use squares. And who can blame them?

"So," he continues, "We are going to give the `ShapeManager` facility away for free. Customers can download it from our website. But. . . He pauses here for effect, because this is the insight that will save the company. ". . . We'll make them pay to download circles. And we'll make them pay to download squares. And if they want other shapes, we'll make them pay for those as well. Instead of selling one big system that manages shapes, we are going to sell the shapes themselves—and we're going to make a blithering fortune!"

And amid all the cheers, balloons, and popping of champaign corks, you realize that the CEO's dream is dead on arrival. It is dead because `Circle` and `Square` are woven through the entire fabric of this system. All those `switch` statements, all those `if`/`else` statements, all those clever logical optimizations: They all deal specifically with `Circle` and `Square`. There's just no way the architecture of this system is going to allow the CEO to give the `ShapeManager` away for free without including the `Circle` and `Square` with it. They are inextricably bound together.

A design that prevents its parts from being separated and deployed independently from its other parts violates the Dependency Inversion Principle (DIP) and is immobile. The fewer parts that are independently deployable, the more immobile it is.

This design is rigid, fragile, and immobile. It's trash. It's a horror scene. It's going to take the company down, and all of us with it!

The OO Solution

We hire a famous consultant who flies in, makes a lot of noise, craps all over everything, and then flies home.[10] His message is: “Your design is procedural. It should have been object oriented. Here, let me show you.” And what he shows is this:

[10] These are sometimes known as *seagull consultants*.

```
——Shape.java——
interface Shape {
  void render();
}


——Square.java——
class Square implements Shape {
  public int side;
  public Point topLeft;

  public void render() {
    //...
  }
}


——Circle.java——
class Circle implements Shape {

  public int radius;
  public Point center;

  public void render() {
```

```
  public void render() {
    //...
  }
}


——ShapeManager.java——
public class ShapeManager {
  public static void renderShapes(List<Shape> shapes) {
    for (Shape shape : shapes) {
      shape.render();
    }
  }
}
```

Not a `switch` statement in sight. Now consider what happens when you add the `TRIANGLE` case.

- `Shape.java` does *not* need to change.
- `Square.java` does *not* need to change.
- `Circle.java` does *not* need to change.
- `ShapeManager.java` does *not* need to change.

Nothing needs to be changed; except, perhaps, for the `main` program. But none of these modules need changing. And that means the design is not rigid.

There are no `switch`/`if`/`else` statements to find and change either. All the methods, like `erase`, `drag`, `scale`, and `rotate`, are contained within the `Shape` classes. This design is not fragile.

Best of all, we can deploy `Circle` and `Square` separately from `ShapeManager`. We could put `ShapeManager`, `Circle`, and `Square` into their own `jar` files and distribute them individually!

This design conforms to the OCP and the DIP. It is mobile, flexible, and robust. Hallelujah! We are saved!

Not So Fast, Johnson

What does this design protect us from? It protects us from new shapes. Whenever a customer wants a new shape, all we need to do is write a new derivative of the `Shape` class and put it into its own `jar` file that can be independently distributed and deployed.

But what if the customers don't ask for a new shape? What if they ask instead for a new method on `Shape` that renders drop shadows?

Now our `Shape` abstraction doesn't help us at all. We're going to have to take that lovely OO design and touch every single one of those classes. `Shape`, `Circle`, and `Square` will all need a `renderDropShadow` method; and the `ShapeManager` will have to call it. That `Shape` abstraction gave us no benefit at all.

But now look back at that first solution, the one with the `switch` statement. All we have to do to that one is add the `renderDropShadow` functions to the `ShapeManager`. We don't have to touch the `Circle` or `Square` at all!

The OO/Procedural Trade-off

If our design interprets changes to the requirements as new types, then OO is a good choice because it is easy to add new types, like `Triangle`, without changing anything else. However, if our design interprets changes as new functions, then the procedural `switch` statement is best because we don't have to change any of the types.

Or, to say this differently, we use OO when we want to add new data types to existing functions, and we use procedural

`switch` statements when we want to add new functions to existing data types.

Again we see this interesting opposition. Hiding data and exposing functions allows the data to change without significant impact. Hiding functions and exposing data allows the functions to be changed without significant impact.

So which is it? Which would we rather change, functions or data? And the answer to that is that it depends on how your application and your design interpret changes to that application.

In large enterprise models, we often write new applications against the existing data schema. Whereas when we add new capabilities to a given application, we often encapsulate that behavior into new data types.

Personally, I prefer the latter. Where possible, I create designs that interpret changes as new data types. And that means I prefer an OO design. For the most part, I find that works pretty well.

Thus, my general rule for `switch` statements and their ugly duckling sibling, `if`/`else` chains, is that they should be located at the periphery of the system (e.g., the `main` module); there should be, at most, one for each switchable type; and the cases within them must create the polymorphic objects that the rest of the system uses.

However, that preference and that rule do not mean that I won’t use a `switch` statement in those parts of the system where behavior is overwhelmingly more volatile than data. I won’t force an OO round peg into a procedural square hole, or vice versa.

But What About Performance?

It is fair to say that today's compilers can take advantage of the local nature of `switch` statements and specific processor architectures to create executable code that runs somewhat faster than a polymorphic dispatch. Thus, OO designs can incur a slight performance penalty. If you are working on a project where a few nanoseconds here or there are critical, then it may be that you will have to abandon the OCP and the DIP and use `switch` statements instead of polymorphism.

Most programmers have no need to care about stray nanoseconds. But if you do, be careful to concentrate them where they do you the most good. Don't abandon OO design throughout the entire system just because a few time-critical inner loops must use switches.

Conclusion

Objects expose behavior and hide data. This makes it easy to add new kinds of objects without changing existing behaviors. It also makes it hard to add new behaviors to existing objects. Data structures expose data and have no significant behavior. This makes it easy to add new behaviors to existing data structures but makes it hard to add new data structures to existing functions.

In any given system, we will sometimes want the flexibility to add new data types, and so we prefer objects for that part of the system. Other times we will want the flexibility to add new behaviors, and so in that part of the system, we prefer data types and procedures. Good software developers understand these issues without prejudice and choose the approach that is best for the job at hand.

Chapter 13. Clean Classes

by Jeff Langr

In Chapter 7, “Clean Functions,” you learned how to best organize code statements and expressions in the form of functions or methods. Hopefully you learned these important things about clean functions:

- Functions provide a way to declare, invoke, and reuse pieces of logical behavior, or concepts.
- While nothing prevents you from aggregating multiple concepts into a single function, you’re much better off if you compose methods that capture small, discrete concepts.

In this chapter, you’ll hear echoes of the advice from Chapter 7 at the next-higher-up organizational construct: classes or modules.

Your programming language allows you to group concepts into a single construct—a class, if you’re coding with an object-oriented language. If you program in a primarily functional language like JavaScript, Elixir, or F#, you might refer to a module instead.

Classes and Modules versus Files

In some object-oriented languages, such as Java, a file can declare only a single, publicly available class. Other

languages, such as C++, TypeScript, and Ruby, allow you to declare multiple classes within a single file, or even split a single class between multiple files.

To make matters more confusing, some languages (both OO and functional) allow you to declare namespaces. These can help you avoid name clashes across a larger system. They can also help you organize related concepts.

Most functional languages—for example, Clojure, Elixir, and Haskell—allow you to organize functions into multiple, separate modules within a file. Even for languages that don't directly support modules/namespaces within the language (JavaScript comes to mind), you can organize multiple "virtual" modules within a single source file.

Just because you can doesn't mean you should. Generally, stick to a single module or class per source file. You'll create fewer headaches for yourself and your dev team.

Everything is a trade-off in software development, of course, and thus, exceptions always exist. You might find it useful to declare other very closely related classes within the same source file, such as small structs (e.g., records in Java) or exception types. Perhaps they are consumed only by other types within the same file, or they always appear in client code that also involves the file's primary type.

(And, of course, all this means little to those devious Smalltalk programmers who have the audacity to not organize their code into files at all.)

But, yeah, for the most part, class <=> file, or module <=> file, or whatever. Consider the file part of things mostly not relevant in our discussions in this chapter. We're going to focus instead on how to properly group concepts in classes and modules, irrespective of how they are deployed into files.

What Should a Class Contain?

You could conceivably lump together your system's entire set of methods into a single class—a monolith. You could, similarly perversely, create a separate class for each and every method. Both are outrageous ideas, though I've done the former for very small systems, and perhaps the second isn't so outrageous for some Smalltalkers.

Classes provide ways to help you keep your sanity as your system grows. Without sensible organization, you will

- Have a harder time finding the code you're looking for.
- Bloat your system with considerable redundancy; doubling it in size is a likely possibility.
- Create defects around inconsistent behaviors, due to the redundancies.
- Create defects due to increasing complexity.
- Spend extensive amounts of time writing automated tests for it.

All those outcomes increase in likelihood with growth, in turn increasing your costs and frustrations.

We'll talk about clean class design throughout this chapter. In your head, translate that to "doing all the things that should make our development lives easier."

"Clean" isn't a ding at anyone. Our notion of "unclean" is a way of characterizing code that demands more effort from the average developer to understand or maintain.

Characterizing Class Design

The quality of a design can only truly be assessed in the face of change; change that occurs each time the system must take on new features. Is it getting increasingly costly to

introduce new features as the system grows and ages? Is the number of defects increasing?

Across seven decades of software development, we've gathered empirical data from our experiences with accommodating change. We've gleaned valuable insights from that data, and we've continually extrapolated those learnings into larger and newer contexts.

In the late 1940s, a typical system was implemented like our degenerate example, with the entire set of code essentially stored as a single function to be read and executed top to bottom. Let's call this a *monolith*.

John von Neumann and Herman Goldstine's article, "Planning and Coding of Problems for an Electronic Computing Instrument,"[1] explored the premise of a subroutine:

[1] H. H. Goldstine and J. von Neumann, "Planning and Coding of Problems for an Electronic Computing Instrument," Institute for Advanced Study, Princeton, New Jersey, 1947, archived at https://www.ias.edu/sites/default/files/library/pdfs/ecp/planningcodingof0103inst.pdf.

> *"We call the coded sequence of a problem a routine, and one which is formed with the purpose of possible substitution into other routines, a subroutine."*

The premise was that subroutines (particularly for smaller, common operations) would cut down on the amount of code by allowing reuse, and would also help reduce defects.

Pioneers Maurice Wilkes and David Wheeler, inspired by ideas found in the seminal article, implemented the concept of subroutines in the EDSAC computer in 1951.[2] In short, they believed that code modularization brought about by subroutines would lead to less costly software development. They didn't wait for research (though it did come, slowly, across later decades).

[2] [PPEDC].

Since the advent of subroutines, we've extrapolated the associated claimed value of modularization to larger contexts. We learned to organize collections of subroutines—functions or methods—into modules or classes. That, in turn, presented new challenges around the composition of such modules and classes, as well as the interrelationships (dependencies) between them. We began to derive various heuristics to guide other developers.

Heuristics and Characteristics

Heuristics help you identify the steps to take, and also the steps to avoid, to arrive at well-designed classes.

- Positive heuristics—steps to take. Kent Beck's four rules of emergent design,[3] for example, tell you to do the following.

[3] See Chapter 18, "Simple Design." See also https://martinfowler.com/bliki/BeckDesignRules.html.

  - Ensure that all code is testable as you go.
  - Eliminate logical redundancies.
  - Ensure that all programmatic elements are clearly and concisely named.
  - Minimize the number of elements.
- Negative heuristics—steps to avoid. Antipatterns describe bad paths to take while developing code. Copy-paste programming, for example, tells us to not habitually create new logic by first duplicating existing code and then changing it. (A more memorable name for this antipattern might be "grab and stab." Or "snatch and patch." Or "copy and corrupt." Or . . . oh, that's enough for now. Just avoid it.)

Characteristics help you assess the current quality of your classes.

- Positive characteristics—desired traits. Bob Martin's collection of five class design principles, known as *SOLID*, have withstood around three decades of scrutiny, and can be practically applied to functional code as well.
- Negative characteristics—undesired traits. Martin Fowler's code smells[4] are a collection of things to avoid in code, each identified with a memorable name. Shotgun surgery, for example, means that your code forces you to make updates to numerous classes in order to effect simple changes.

[4] [Refactoring] and https://martinfowler.com/bliki/CodeSmell.html.

Over time, folks in the software development arena have codified numerous collections of heuristics and characteristics. We'll refer to any of these as a design perspective.

Design perspectives can describe design at both the method (micro) level and the class (macro) level (or even higher levels). You might be able to consider some characteristics as heuristics, and vice versa; the categorization isn't terribly important.

For much of this chapter, we'll focus on the characteristics of an ideally sized class, because the idea is that you can glean them from looking at the code.

Such an ideal class is typically small. Its name concisely summarizes the small numbers of behaviors gathered within. And it is defined cohesively: Its methods are closely related and focused, and it performs a single, well-defined task (or set of related tasks) effectively. As a result, the class exhibits the characteristic of adhering to the Single Responsibility Principle (SRP): It has one reason to change.

Design perspectives most certainly overlap. Notably, outcomes for the more overarching perspectives—SOLID, Kent Beck's simple design, and Martin Fowler's code smells, for example—are comparable. They all foster small, cohesive modules and functions.

In nearly all of these perspectives, there's an acknowledgment, whether stated outright or implied, of the need for balance: Virtually no principle is absolute. Kent Beck's four rules of simple design, for example, drive us toward small, single responsibility classes. The resulting classes are easier to test (rule 1), they can help us eliminate redundancies more easily (rule 2), and they promote clear, intention-revealing names (rule 3). But rule 4—minimize the number of elements, for example, modules and functions—suggests that we can go too far and create tiny classes that add no advantages to the system.

> **Future Bob:** Indeed, this is one of John Ousterhout's fears regarding the "One Thing" rule.

Most of us needn't worry: The typical system has moved too far in the wrong direction, containing large, muddled, multipurpose modules and functions. Still, keep in mind that there are trade-offs for every choice in software. Choose the design perspective(s) you like, and focus on the outcome of creating code that is easy to maintain.

When Is a Class Too Large?

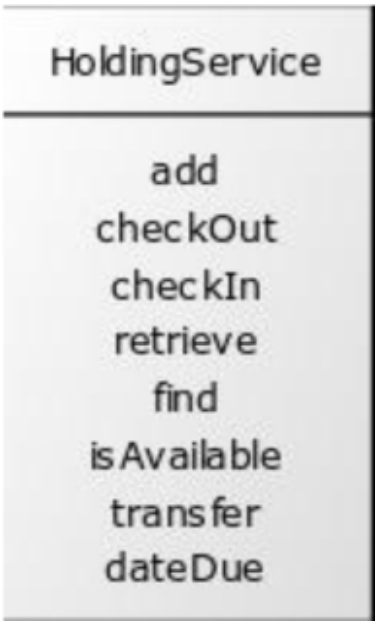
Let's take a look at the `HoldingService` class, used to manage the holdings (materials, such as books or physical audio recordings) of a library system and its patrons. Here are the public behaviors defined on `HoldingService`:

```
boolean isAvailable(String barCode)

Holding find(String barCode)
```

```
String add(String sourceId, String branchId)
void transfer(String barcode, String branchScanCode)
Date dateDue(String barCode)
void checkOut(String patronId, String barCode, Date date)
int checkIn(String barCode, Date date, String branchScanCode)
```

The following figure depicts that `HoldingService` interface (we'll be looking at more pictures later), using Unified Modeling Language (UML) diagramming guidelines:



The method parameters aren't usually interesting when we start looking at class-level design. We'll omit them until they become useful.

Our picture shows that the `HoldingService` allows holdings to be added to the system, borrowed ("checked out") by a patron, returned ("checked in") to a library location (branch), and transferred from one branch to another. The `HoldingService` also supports a few queries, including the ability to retrieve holding details, answer whether it's available for a patron to check out, and determine the date when the material is due to be returned.

The picture suggests that the core (single?) responsibility of the `HoldingService` is to manage the disposition of holdings. Each of the behaviors depicted directly supports that interest.

The SRP tells us to be a bit pickier, however: An SRP-compliant class should have only one reason to change. It's the opposite of a monolith, in which every single change requires updating the sole class in the system.

Why might a single-responsibility class be desirable?

- Its name can concisely reflect the behaviors contained within, making those behaviors easier to locate.
- It creates real possibilities for reuse.
- Its likelihood of adhering to the Open–Closed Principle (OCP; see Chapter 19), and thus being closed to future changes, increases.
- It's easier to test.
- It makes your system more flexible.

Each of the behaviors in `HoldingService` implements a policy—a set of rules established by the library system and implemented in the code. The `dateDue` policy, for example, specifies that books are due 21 days after they are checked out and that DVDs are due seven days after checkout.

The `checkIn` policy is more involved: When a book is checked in, it's marked as available for other patrons to check out. The patron returning the book is assessed a fine if material is late, that is, when it is returned after its due date. The fine amount varies depending on the material type and other possible factors.

Policies do change, and it's possible that the policy for check-outs might need to change exclusively from the policy for check-ins. Such independent variance suggests implementing each of the policies in its own class—`CheckOutService`,

`CheckInService`, and so on—at least if you want to be SRP compliant. At that point, the only job of the `HoldingService` would be as a consolidated interface for holdings interests that delegated to other objects to do the work.

> Future Jeff**:** Creating new classes is surprisingly easy in an IDE, yet we often resist as developers. It takes only moments and has few downsides, but we often pair with developers who do everything they can to avoid clicking a New Class menu item. Just do it.

We might not want a “ravioli” system where every class contains only one method, but there’s nothing wrong with tiny classes containing three or two methods, or even one.

While the SRP tells us to be picky, it also doesn’t tell us to speculate about the nexus of change. Your systems probably rampantly violate the SRP. Rather than go find code to change, wait for the next demand for change and ensure that you take that opportunity to shape your system to be more compliant.

Policies in Code

Let’s assume that the core library policies for check-in and check-out are stable. It’s easy to think of numerous reasons for `HoldingService` to potentially change. An example: The library wants to allow patrons to rent DVDs for a longer period—14 days instead of seven—because they’re an older technology and are in low demand nowadays.

Will `HoldingService` be impacted by that change? While looking at UML models can be very helpful, we can’t spot all change reasons by looking at UML alone.

The interface of a class only tells you about the behaviors it directly publicizes. To understand fully the extent to which a class violates the SRP, you have to open up the source file and

peruse the code. It doesn't take long to spot more reasons why a class might need to change.

Here's the `dateDue` method:

```
public Date dateDue(String barCode) {
   var holding = find(barCode);
   if (holding == null)
      throw new HoldingNotFoundException();
   return holding.dateDue();
}
```

The service delegates the responsibility for calculating a due date to the `Holding` class. As such, the DVD rental period change won't impact the `HoldingService` class, but it will impact the `Holding` class (or maybe another class it depends on).

"Managing dates due" is a responsibility that could disappear, because some libraries have actually eliminated fines. If and when that occurs, we'll look to shield `HoldingService` from similar, subsequent changes.

Where Reasons to Change Hide

Large classes tend to contain and obscure numerous responsibilities. Often, change reasons are buried within the body of a long method. Let's look at an example.

The `HoldingService` method `checkIn` does a pretty good job of declaring the policy for returning materials:

```
public int checkIn(String barCode, Date date, String branchScanCode)
   var branch = new BranchService().find(branchScanCode);
   var holding = find(barCode);
   if (holding == null)
      throw new HoldingNotFoundException();
```

```
    holding.checkIn(date, branch);

    var foundPatron = findPatronWith(holding);
    removeBookFromPatron(foundPatron, holding);

    if (holding.dateLastCheckedIn().after(holding.dateDue())) { //late
        foundPatron.addFine(calculateLateFine(holding));
        return holding.daysLate();
    }
    return 0;
}
```

While rich in policy, the `checkIn` method still begs for a bit of cleanup. One line of glaring implementation detail stands out as a potential problem—the `if` statement that determines whether the check-in is late:

```
if (holding.dateLastCheckedIn().after(holding.dateDue())) { //late?
```

That conditional represents a policy decision: A holding is late if it's returned after the date due. However, it's possible that the policy will become considerably more complex. Librarians might imagine no end of new policy changes: VIPs (very important patrons) might be exempt from fines, the library might offer grace periods, and so on. As a result, that seemingly innocuous single line of code represents another reason for `HoldingService` to change.

Because it exposes implementation detail, the conditional also slows comprehension time for readers. The detail demands closer inspection: We must carefully read it, then mentally assemble it into a singular concept.

A comment might help, but it's not the right solution, and can be a lie anyway. It's best if we can glean intent from scanning the code itself, rather than having to look elsewhere. Comments are the distracting footnotes of code.

Fixing the Problem

We want to move the specifics of "is it late" out of the `HoldingService` class. Where to? We could move it to a new class such as `LateHoldingService`, but the obvious short-term candidate appears to be the `Holding` class.

Step 1, however, is abstracting the concept of "is it late" by creating a new method that contains just the conditional. Yep—this notion of extracting methods should be familiar from Chapter 10, "One Thing." Extract till you drop!

```
public int checkIn(String barCode, Date date, String branchScanCode)
   // ...
   if (isLate(holding)) {
      foundPatron.addFine(calculateLateFine(holding));
      return holding.daysLate();
   }
   return 0;
}


private boolean isLate(Holding holding) {
   return holding.dateLastCheckedIn().after(holding.dateDue());
}
```

The silly line-level comment `//late?` has disappeared. The new query method's name, `isLate`, precisely imparts the same information. The implementation detail is encapsulated within.

Going forward, if you feel you need a comment to guide readers through a line or five of related code, consider extracting it into its own method. The “guiding” comment often provides the basis for the new method’s name, as it does here. A clear, concise method name obviates the need for a comment.

Predicates are particularly ripe for extraction. In fact, when faced with the challenge of trying to understand a long, daunting method, one of the best first steps you can take is to ensure that you understand the various paths through it (sometimes so that you can know which smaller piece you can focus on). Often the conditionals that control these pathways are complex, depending on combinations of method calls and variables—`if (x(a) && y || !z(c, d)`, for example. Those details don’t make it clear why we’re heading into a certain piece of code. An extracted predicate’s method name can impart immediate understanding.

Once isolated within `HoldingService`, the `isLate` method reveals a glaring code smell known as *feature envy*[5]: The method, apparently envious of the `Holding` class, asks it multiple questions (“What’s the date checked in? What’s the date due?”) in order to compute a result. The `isLate` method also shows disinterest in the `HoldingService` class on which it’s defined.

[5] [Refactoring].

We can soothe the method’s envy by moving it to the `Holding` class, where it can talk directly to its new peers:

```
public class HoldingService {
// ...
   public int checkIn(
      String barCode, Date date, String branchScanCode) {
      // ...
```

```
      if (holding.isLate()) {
         foundPatron.addFine(calculateLateFine(holding));
         return holding.daysLate();
      }
      return 0;
   }
}
public class Holding {
   // ...
   public boolean isLate() {
      return dateLastCheckedIn().after(dateDue());
   }
}
```

The calling code (in the `checkIn` method) now concisely states a piece of the overall policy:

```
if (holding.isLate())
```

The conditional is now immediately digestible. It doesn't require us to stop and pause. Instead, it helps us quickly understand the method's control flow, which, in turn, helps us know where we need to look next.

After moving the `checkIn` method to `Holding`, we look again at the method in its new context. Does the `Holding` class now contain too many reasons to change? Should we move the method again, perhaps this time to a brand-new class? What can be simplified within the method itself?

With every change, we continue to ask these kinds of questions as part of continuous design.

It might seem like we just passed the buck by moving a problem from `HoldingService` into `Holding`. But toward our primary

interest in SRP-compliant classes, we've removed one reason to change from `HoldingService`. We've also replaced unnecessary detail with an abstraction, which, in turn, helps us more easily consider the remainder of `HoldingService`.

An Overly Open Implementation

Opportunities to extract and move are rampant in a typical codebase. Extracting implementation detail from a method often makes it obvious that the code belongs elsewhere. Do this enough, and eventually most code makes it to the right place, and each such place is reasonably small, cohesive, and easy to work with. Another line in the `checkIn` method similarly suggests feature envy:

```
foundPatron.addFine(calculateLateFine(holding));
```

We're calling a local method, `calculateFine`, that interacts with code defined in `Holding`, again asking it questions in order to calculate a fine. We move the method into `Holding` (with only two keystrokes in IDEA![6]):

[6] The IDE from IntelliJ.

```
foundPatron.addFine(holding.calculateLateFine());
```

Here's what `calculateLateFine` looks like in its new home:

```
public int calculateLateFine() {
   var daysLate = daysLate();
   var fineBasis = getMaterial().materialType().dailyFine();

   var fine = 0;
   switch (getMaterial().materialType()) {
      case BOOK, NEW_RELEASE_DVD:
         fine = fineBasis * daysLate;
```

```
            break;

        case AUDIO_CASSETTE, VINYL_RECORDING, MICRO_FICHE,
             AUDIO_CD, SOFTWARE_CD, DVD, BLU_RAY, VIDEO_CASSETTE:
            fine = Math.min(1000, 100 + fineBasis * daysLate);
            break;
    }
    return fine;
}
```

Ugh, a `switch` statement, and an old-school one at that—the kind where control flow can accidentally drip from one case to the next if you're careless.

Even after being moved to `Holding`, the method still doesn't look at home. While `calculateLateFine` does interact directly with the `Holding` (asking it for `daysLate`), it predominantly interacts with the `Material` object returned by `getMaterial`.

The `calculateLateFine` method is the sort that will likely suffer many changes over time. It has at least three reasons to change: the addition of new material types (e-books, puzzles, STEM kits, etc.), new schemes to encourage patrons to return materials in high demand, and new rates to cover material price increases.

Such a method violates another SOLID principle—we want to minimize "opening up" existing classes to make changes, and instead find ways to enhance a system by adding new, single-purpose classes. This is the OCP.

Open classes introduce risk and cost. `Holding` is open. Changes to any of the three responsibilities in `calculateLateFine` carry the risk of breaking existing behaviors in `Holding`. With a drippy `switch` statement in the mix, anything can happen.

Should We Do Anything Now?

Typical systems being what they are, we already missed most of the opportunities to take advantage of the OCP. Our codebase is unashamed about its openness. Virtually no classes are closed. Rampant change greets everyone who must touch the code. If change is a given, the best approach is to wait for it. Speculative cleanup creates unnecessary risk from improvements no one yet needs. Let's wait for the next change.

What About Now?

OK, that didn't take too long. We were told, moments ago (between the prior paragraph and this one), that indeed we must support a couple of new material types. Specifically, we need to allow jigsaw puzzles and board games to be borrowed. They'll fall under a new fine scheme the library folks want to try.

We'll prepare for the new requirement by first factoring the code (prefactoring it) so that the change effort is smoother and has minimal impact.

Each of the two `switch` branches calculates an appropriate fine value using one of two strategies. We can extract the tiny bits of calculation logic to a couple of Strategy[7] classes, each implementing a common interface:

[7] [GOF95].

```
public interface LateStrategy {
   int calculateFine(int daysLate);
}

public class ConstrainedFineStrategy implements LateStrategy{
   public static final short BASE_FEE = 100;
```

```
   public static final short MAX_FINE = 1000;

   private final int fineBasis;

   public ConstrainedFineStrategy(int fineBasis) {
      this.fineBasis = fineBasis;
   }

   @Override
   public int calculateFine(int daysLate) {
      return Math.min(MAX_FINE, BASE_FEE + fineBasis * daysLate);
   }
}

public class DaysLateStrategy implements LateStrategy {
   private final int fineBasis;

   public DaysLateStrategy(int fineBasis) {
      this.fineBasis = fineBasis;
   }

   @Override
   public int calculateFine(int daysLate) {
      return fineBasis * daysLate;
   }
}
```

What cute, tiny little classes! Here's the updated `calculateLateFine` method.

```
public int calculateLateFine() {
```

```
   var daysLate = daysLate();
   var fineBasis = getMaterial().materialType().dailyFine();

   var fine = 0;

   switch (getMaterial().materialType()) {
      case BOOK, NEW_RELEASE_DVD:
         fine =
            new DaysLateStrategy(fineBasis).calculateFine(daysLate);
         break;

      case AUDIO_CASSETTE, VINYL_RECORDING, MICRO_FICHE,
           AUDIO_CD, SOFTWARE_CD, DVD, BLU_RAY, VIDEO_CASSETTE:
         fine = new ConstrainedFineStrategy(fineBasis)
                   .calculateFine(daysLate);
         break;
   }
   return fine;
}
```

Note that `calculateLateFine` first retrieves the material type (e.g., `BOOK`) from the material, in order to decide which strategy to apply.

Here's what `MaterialType` looks like:

```
public enum MaterialType {
   BOOK(21, 10),
   AUDIO_CASSETTE(14, 10),
   VINYL_RECORDING(14, 10),
   MICRO_FICHE(7, 200),
   AUDIO_CD(7, 100),
```

```
   SOFTWARE_CD(7, 500),
   DVD(3, 100),
   NEW_RELEASE_DVD(1, 200),
   BLU_RAY(3, 200),
   VIDEO_CASSETTE(7, 10);

   private final int checkoutPeriod;
   private final int dailyFine;

   MaterialType(int checkoutPeriod, int dailyFine) {
      this.checkoutPeriod = checkoutPeriod;
      this.dailyFine = dailyFine;
   }

   public int dailyFine() {
      return dailyFine;
   }

   public int checkoutPeriod() {
      return checkoutPeriod;
   }
}
```

If each material type can be initialized with the checkout period and late fine amounts, it can also be initialized with a Strategy object:

```
import domain.core.ConstrainedFineStrategy;
import domain.core.DaysLateStrategy;
import domain.core.LateStrategy;
```

```
public enum MaterialType {
   BOOK(21, new DaysLateStrategy(10)),
   AUDIO_CASSETTE(14, new ConstrainedFineStrategy(10)),
   VINYL_RECORDING(14, new ConstrainedFineStrategy(10)),
   MICRO_FICHE(7, new ConstrainedFineStrategy(200)),
   AUDIO_CD(7, new ConstrainedFineStrategy(100)),
   SOFTWARE_CD(7, new ConstrainedFineStrategy(500)),
   DVD(3, new ConstrainedFineStrategy(100)),
   NEW_RELEASE_DVD(1, new DaysLateStrategy(200)),
   BLU_RAY(3, new ConstrainedFineStrategy(200)),
   VIDEO_CASSETTE(7, new ConstrainedFineStrategy(10));

   private final int checkoutPeriod;
   private final LateStrategy lateStrategy;

   MaterialType(int checkoutPeriod, LateStrategy lateStrategy) {
      this.checkoutPeriod = checkoutPeriod;
      this.lateStrategy = lateStrategy;
   }

   public int checkoutPeriod() {
      return checkoutPeriod;
   }

   public int calculateFine(int daysLate) {
      return lateStrategy.calculateFine(daysLate);
   }
}
```

(Yes we could use Singletons[8] for each `LateStrategy` derivative, particularly since the subclasses contain no data. We could also specify a class type for each `enum` value, then use reflection to instantiate the Strategy object. But why make things more complicated?)

[8] [GOF95].

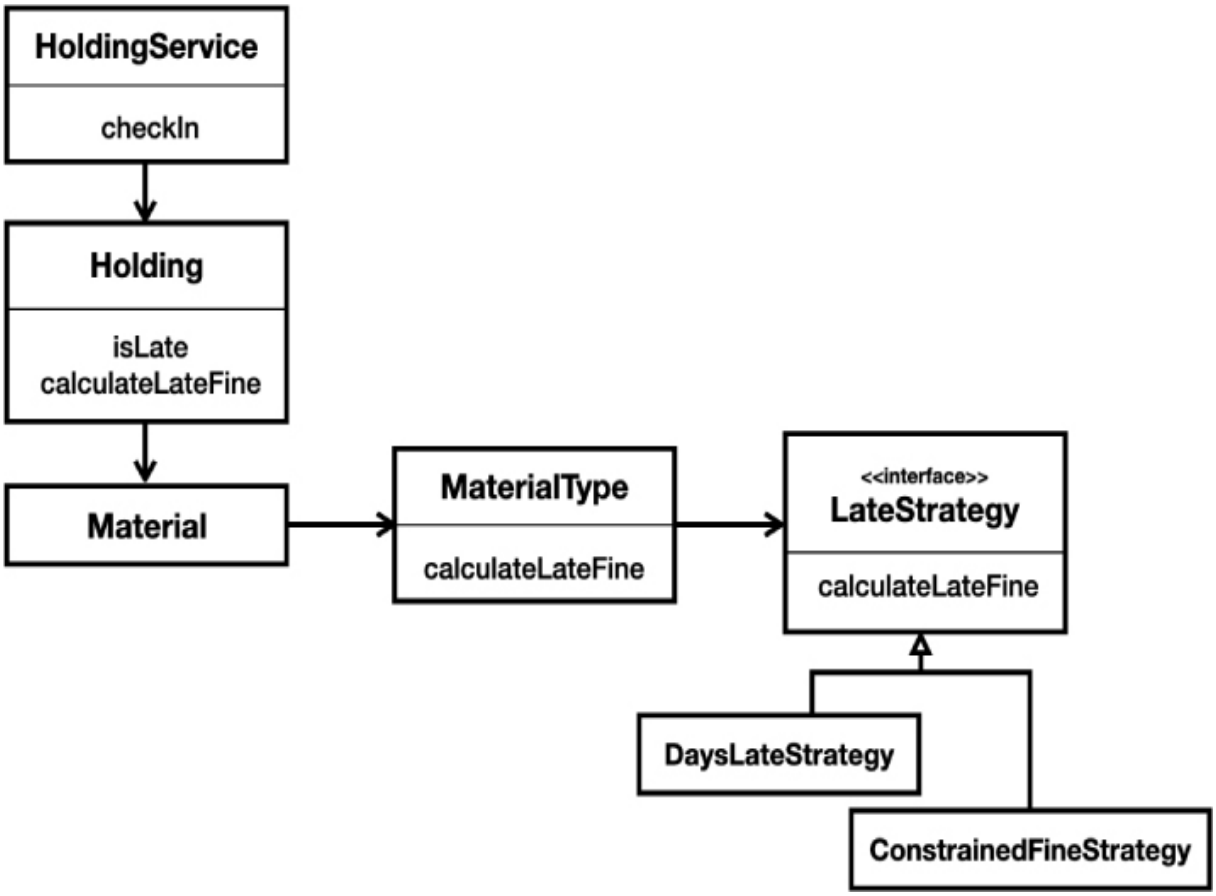
`MaterialType` can then supply a `calculateFine` implementation that delegates to the `lateStrategy` object:

```
public int calculateFine(int daysLate) {
   return lateStrategy.calculateFine(dailyFine, daysLate);
}
```

The best part? At this point, the `Holding` method `calculateLateFine` can simplify to a single line of code that delegates to `MaterialType`:

```
public int calculateLateFine() {
   return getMaterial().materialType().calculateFine(daysLate());
}
```

The following shows a picture of the updated solution:



Closed, Cohesive, Single-Responsibility Classes

We've been tasked with supporting a new feature. For jigsaw puzzles and board games, the library board has decided to test out a new fine scheme. We test-drive the corresponding Strategy class:

```
public class DegradingFineStrategy implements LateStrategy {
   private final int fineBasis;
   private final double degradationRate;

   public DegradingFineStrategy(
      int fineBasis, double degradationRate) {
      this.fineBasis = fineBasis;
```

```
      this.degradationRate = degradationRate;
   }

   @Override
   public int calculateFine(int daysLate) {
      double total = fineBasis *
         (1 - Math.pow(1 - degradationRate, daysLate)) /
         degradationRate;
      return (int) Math.round(total);
   }
}
```

Here are the tests, in case you thought we were only pretending to write them:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

class DegradingFineStrategyTest {
   int initialFine = 1000;
   double degradationRate = 0.1d;
   DegradingFineStrategy strategy =
      new DegradingFineStrategy(initialFine, degradationRate);

   @Test
   void firstDayIsFineBasis() {
      assertEquals(initialFine, strategy.calculateFine(1));
   }

   @Test
```

```
   void nextDayReducedUsingDegradationRate() {
      assertEquals(1000 + 900, strategy.calculateFine(2));
   }

   @Test
   void multipleDays() {
      assertEquals(1000 + 900 + 810 + 729 + 656,
         strategy.calculateFine(5));
   }
}
```

For now, we must add a line to the `MaterialType` class to accommodate the new category:

```
public enum MaterialType {
   GAMES_AND_PUZZLES(21, new DegradingFineStrategy(500, 0.20)),
   BOOK(21, new DaysLateStrategy(10)),
   AUDIO_CASSETTE(14, new ConstrainedFineStrategy(10)),
   // ...
}
```

As a result of having to modify an existing class, our system isn’t quite fully closed with respect to accommodating new fine schemes. But we realize that it wouldn’t be difficult to replace the `MaterialType enum` with a class that reads the necessary information from a database. Once built, our system would be fully OCP compliant: Code a new class, seed the database with sufficient configuration data, and go—no changes to existing classes required!

When Policies Change

Many service classes must orchestrate numerous disparate behaviors. They interact with business logic regarding various domains (holdings, patrons, branches), they might persist or retrieve data, and they might interact with external services. The `HoldingServiceClass` is one example of an orchestrating class.

Some services might have minimal orchestration needs. An example is a collection of behaviors concerned with date calculations.

A well-defined SRP-compliant service, as such, should either:

- Contain only orchestration logic that declares policy and delegates to other classes for implementation specifics.
- Contain only implementation specifics for related domain or utility behaviors.

In other words, don't mix policy with implementation detail. Doing so is costlier to understand, navigate, and maintain. It's also not SRP compliant: A policy change is one reason to make a code change, and implementation details about a step in the policy is another reason.

Is This Overengineering?

What would happen if we took the SRP and the OCP to the extreme? Virtually everything in our system would be "plug and play." In a service, we wouldn't have a single controller class with a half dozen routes; we'd create a separate class for each route. It would register itself (perhaps like what an `@PostMapping` annotation accomplishes in Spring Boot). Its sole job would be to delegate requests to a similarly single-purpose service class (or to reject them as a result of validation, which is likewise implemented elsewhere). Each microservice facet would represent either a policy—whereby all specifics are delegated to—or a chunk of code that accomplished a singular need.

And so on for each class down the line. Pretty much all conditionals would be implemented polymorphically; that is, both branches would be implemented in separate classes, much like states in a State[9] pattern. Each step in a policy (a workflow) would be implemented in a separate class as well.

[9] [GOF95].

Ridiculous? It may sound like it, but such a system would truly support the OCP, whereby all new behavior was accomplished by dropping a new class in place. All classes would be small and easily tested in isolation. You'd write tests for each new class and perhaps would update an end-to-end test. Otherwise, you'd make no changes to the system.

Almost all systems don't look like this, and they are harder to work with as well as riskier.

A highly SRP/OCP-compliant system can be initially challenging for developers not familiar with the idea—it takes a while to figure out where things are at first. ("There's all these objects and none of 'em are doing anything!") Long term, however, it dramatically streamlines the change process.

It's not a new idea. We worked with systems approaching this ideal 30 years ago in Smalltalk. Most classes were small, and the majority of methods were one to three lines of code each. Life was bliss.

As with most principles, consider both the SRP and the OCP as ideals: You want to almost always move in their direction, not away from them. It might be OK that you'll never hit the ideal. The typical system, however, moves in the opposite direction, degrading toward larger, more monolithic classes. That's why they're harder to work with as they grow.

For now, forgive your system for its flaws. Moving forward, assess each change you make—how many existing classes did you have to open for change? Think of a reshaped, SRP/OCP-

compliant system that would have instead accommodated that new feature as an extension of the existing code. Expend at least some effort to move your system in that direction.

Seek to create smaller, single-purpose classes. They're easier to work with (test, understand, change), and they're easier to close.

Seek to close classes. They're something we can consider "out of sight, out of mind." We seek to have less code that we must concern ourselves with.

Simpler Testing

You could choose to retain whatever tests you had in place for `HoldingService`. When adding the new behavior for fines, you'd add to these tests. That's one acceptable route.

However, you can also distribute the testing as well. Yes, the individual fine policies aren't publicly exposed to the ultimate client, and might be considered "private behavior." But the policies are also isolated, closed concepts that can be reused—and more easily tested than within the larger context of `HoldingService`. They are also public in the sense that they could conceivably be consumed by a different client.

Creating unit tests focused directly on a now-public fine policy Strategy class makes it officially documented. Such focused unit tests describe all the intentional behavior that developers coded in the class. Clean (clear, concise, and well-abstracted) unit tests provide far more useful documentation than method-level doc comments: We can trust them, as long as the tests pass, and we can see exactly how to interact with the class as a direct client.

We avoid testing implementation details. We test the outcomes of (small) behavioral concepts instead. Small, closed units should be concepts that can be moved about and work in any context where they're useful. A days-late fine policy can

be tested in isolation without exposing any implementation specifics. It is a “sub-behavioral unit.”

Corollary: If you extract to a small, single-purpose class, but don’t view it as useful in any other context, don’t make it publicly accessible. You’ll still need to write tests that indirectly involve these classes at a higher level. Test such private details directly only as a last resort.

Our new fine strategy took less than 20 minutes to test-drive—we created both unit tests and the production code in that time. We didn’t have to worry about changing tests for, or adding tests to, an existing class.

Enter AI

We (humans) are and have been advancing by orders of magnitude in many ways, technologically, every five to ten years or so for the past 90 years. AI will likely continue that amazing trend, accelerating us toward the singularity by the 2040s. Per Ray Kurzweil, “We are now entering the steep part of the exponential.”[10]

[10] [Singularity], pp. 164–165.

Today, we can ask an LLM to produce code at the class level, and it will do an OK job, maybe getting 80% of the code correct. (There’s that good, old 80/20 rule again.[11]) We can help an LLM do a better job by:

[11] https://www.investopedia.com/terms/1/80-20-rule.asp

- Providing it with examples
- Providing it with a simple style that tells it to (a) produce small, single-purpose modules/classes and small, single-purpose functions/methods, (b) produce more-functional solutions with less state management and side effects, and (c) emphasize clarity

What's cool is that this simple style matches how we've been told to code all along. It's also the style that has the broadest support, with few serious detractors. We have concepts like SOLID and an exhaustively examined body of work in design patterns; these things have been around for decades now.

It Will Be Wrong

Unfortunately, we cannot trust that an LLM will get 100% of the code correct. The easy solution is to have it generate tests from the examples provided, then vet the tests for fidelity with our examples, then run them.

In fact, comprehensive testing for behavioral intent is *essential* when using AI. AI will quickly generate more code than has been produced by humans to this point in history. It should create fear in everyone that 20% of that code is likely to be defective.

When the AI gets the code wrong—and it will—we need to have a conversation with it. That conversation is far more effective with a clean design: The source of the problem will be easier to pinpoint, and the resolution can involve focusing on that one small class or method. (If it's a method, sometimes the best route is to have the AI extract what's known as a *method object*—a new class that embodies the single troubled method.[12])

[12] [WELC].

While the current trend is to view AI as most useful as a line-by-line coding assistant, it's already at the point where it's more effective to have it produce code at a modular level. At this point, the code content of the module matters far less. Need to change a class? Pull up the list of examples, update them, and regenerate both the tests and production code. That's not so easy if the modules aren't SRP compliant.

Don’t get us wrong. We love mucking with code on a line-by-line basis. We are gratified by streamlining functions to make them elegant—that is, clear and concise. But ultimately, the contents of any well-composed function are irrelevant once we begin generating wholesale modules.

Our focus begins to shift. We concern ourselves with the overall architecture/design of the system, and we seek to create a system based on the OCP and SRP. When we need new behavior, we generate the appropriate module using AI, we vet the tests it generates, and we drop it into place.

Our system designs once again take center stage. We find value in employing time-honored design patterns that directly speak toward pluggable designs: Functional Pipeline, Command, Factory, Decorator, Strategy, Composite, Chain of Responsibility, and Bridge.

Our focus on clean code remains important for the near future. At some point, AI will be able to completely implement an entire system from a set of examples/tests. We’ll be happy to have retired by then. You’ll still have opportunities as someone who knows how to design and deliver a system as an orchestration of well-tested, AI-implemented pieces.

Chapter 14. Testing Disciplines



Our profession has come a long way in the last three decades. In 1997, no one had heard of testing disciplines such as TDD[1] or TCR.[2] For the vast majority of us, unit tests were short bits of throwaway code that we wrote to make sure our programs “worked.” We would painstakingly write

our classes and methods, and then we might concoct some ad hoc code to test them. Typically, this would involve some kind of simple driver program that would allow us to manually interact with the program we had written.

[1] Test-driven development.

[2] Test && commit || revert.

I remember writing a C++ program for an embedded real-time system back in the mid-'90s. The program was a simple timer with the following signature:

```
void Timer::ScheduleCommand(Command* theCommand, int milliseconds)
```

The idea was simple; the `execute` method of the `Command`[3] would be executed in a new thread after the specified number of milliseconds. The problem was how to test it.

[3] See the Command pattern in [GOF95].

I cobbled together a simple driver program that listened to the keyboard. Every time a character key was pressed, it would schedule a command that would print the same character five seconds later. Then, I tapped out a rhythmic melody on the keyboard and waited for that melody to replay on the screen five seconds later.

"I . . . want-a-girl . . . just . . . like-the-girl-who-marr . . . ied . . . dear . . . old . . . dad."

I actually sang that melody while typing the "." key, and then I sang it again as the dots appeared on the screen.

That was my test! Once I saw it work and demonstrated it to my colleagues, I threw the test code away.

As I said, our profession has come a long way.

Nowadays, I write tests that make sure every nook and cranny of the code works as I expect it to. I isolate my code

from the operating system rather than creating hard dependencies upon it. If I were to write the above example today, I would use test doubles to “mock” out the OS timing functions so that I had absolute control over the time. I would schedule commands that set boolean flags, and then I would use my tests to step the time forward, watching those flags and ensuring that they went from false to true just as I changed the time to the right value.

Nowadays, once I get a set of tests to pass, I clean those tests to make sure that they are well designed and are strongly decoupled from the production code. I also ensure that they are convenient to run for anyone else who needs to work with the code. I keep the code and tests together in the same source package.

In the early days of Extreme Programming and Agile, we only talked about one testing discipline: TDD. When I wrote the first edition of this book, TDD was the only testing discipline that I presented. In the intervening years, it has become clear that there are other testing disciplines that are as effective, or nearly as effective, as TDD. It has also become clear that there are many programmers who prefer those other disciplines for one reason or another.

And so while I will remain an adherent to the discipline of TDD, I feel that it is appropriate to talk about two other disciplines that are compatible with writing clean code. And, although I am only discussing three such disciplines, there may well be others.

Discipline 1: Test-Driven Development (TDD)

This discipline was proposed by Kent Beck in the mid- to late ’90s and became strongly associated with Extreme

Programming and Agile development.

By now, everyone knows that TDD asks us to write unit tests first, before we write production code. But that rule is just the tip of the iceberg.

TDD is not just writing tests first. TDD is an intricate discipline, not a simple rule of thumb. We practitioners do not write a whole batch of tests up front and then, one by one, make them pass. The discipline is far more intricate and intimate than that.

The Three Laws of TDD

What follows are the three laws of TDD. These laws are the basic foundation of the discipline. Following them is very hard, especially at first. Following them also requires some skill and knowledge that are hard to come by. If you try to follow these laws without that skill and knowledge, you will almost certainly become frustrated and abandon the discipline. We will address that skill and knowledge in subsequent chapters. For the moment, be warned. Following these laws without proper preparation can be very difficult.

The First Law

Write no production code until you have first written a test that fails due to the lack of that production code.

If you are a programmer of any years' experience, this law may seem foolish. You might wonder: What test am I supposed to write if there's no code to test? This question comes from the common expectation that tests are written after code. But, if you think about it, you'll realize that if you can write the production code, you can also write the code that tests the production code. It may seem out of order, but

there’s no lack of information preventing you from writing the test first.

The Second Law

Write no more of a test than is sufficient to fail, or fail to compile. Resolve the failure by writing some production code.

Again, if you are an experienced programmer, then you likely realize that the very first line of the test will fail to compile because that first line will be written to interact with code that does not yet exist. And that means, of course, that you will often be unable to write more than one line of a test before having to switch over to writing production code.

The Third Law

Write no more production code than will resolve the currently failing test. Once the test passes, write more test code.

And now the cycle is complete. It should be obvious to you that these three laws lock you into a cycle that is just a few seconds long. It looks like this:

- You write a line of test code, but it doesn’t compile (of course).
- You write a line of production code that makes the test compile.
- You write another line of test code that doesn’t compile.
- You write another line or two of production code that makes the test compile.
- You write another line or two of test code that compiles, but fails an assertion.
- You write another line or two of production code that passes the assertion.

You saw this cycle being played out in the previous chapter. Watching that cycle is the best way to understand the tight feedback loop. I wrote a book sometime back titled *Clean Craftsmanship*. It has many examples of the TDD cycle. There are also some videos of those examples that you can see here: <<the urls of those videos from Clean Craftsmanship>>.

Discipline 2: Test && Commit || Revert (TCR)

This discipline was proposed[4] by Kent Beck in 2018. TCR does not insist that tests be written first. Rather, it insists that code that passes tests should be immediately committed, and code that fails tests should be immediately reverted. This is generally enforced with a script that runs the tests whenever source files are saved and automatically executes the commit or revert based on the test results.

[4] https://medium.com/@kentbeck_7670/test-commit-revert-870bbd756864

The immediate reversion on failure forces the programmers to move very carefully, and in very small steps. The programmer can write as much test and production code as they like, but they are liable to lose what they wrote if they have not written tests that pass. Thus, they will write tests and code together in very short cycles. Those cycles are as small as those in TDD.

Personally, I find TCR to be stressful, while I find TDD to be relaxing. But others may find TCR to be better for their mindset because they are not forced into writing tests first.

Discipline 3: Small Bundles

This discipline was proposed to me by John Ousterhout while I was in the process of writing this second edition. It forces neither the ordering of the tests and code, nor the intensely small cycle of TDD and TCR. The cycle time is longer, probably by an order of magnitude. Minutes as opposed to seconds.

The idea is simple. Programmers write a small bundle of code and tests to achieve a certain result. They might write the code first. They might write the tests first. They might intersperse the writing of the code and tests. But they will conclude with that small bundle having high test coverage and all tests passing.

If you are used to TDD or TCR, you might feel that the bundling discipline is a watered-down compromise. I disagree. It is perhaps a softer ritual; but it can be just as effective. Any discipline can be abused, and any discipline can be honored. If programmers honor this discipline by keeping bundles small and testing every branch and condition, then they will produce a test suite that they can trust with their lives. Therefore, it is entirely compatible with writing clean code.

Design

You might object that disciplines like these are too tactical because they make no room for strategic design. However, testing disciplines are not intended to supplant strategic thinking. I, for example, am a big believer in hammock-driven development.[5] Thinking up front is vital in all software endeavors—so long as that up-front thinking is brought back to reality on a frequent basis. Months of strategic planning, without any coding, is suicide. Coding without any strategic planning is equally suicidal.

[5] https://www.youtube.com/watch?v=f84n5oFoZBc

After an appropriate amount of strategic design, programmers must become tactical. After all, the lines of code have to get written, and writing them is tactical. The testing disciplines described above are tactical disciplines—they augment the tactic of writing code.

Moreover, these disciplines test the design of the code by imposing the viewpoint of users—after all, the tests are a kind of user.

This imposition of viewpoint has led many practitioners to consider these disciplines to be design disciplines as well as coding disciplines. Personally, I find that TDD assists in verifying my lowest-level designs, but it does not help much in higher-level design and architecture. I have seen some pretty bad designs implemented with TDD. Some are in this very book.

Discipline

Disciplines like these should be taken seriously; but they are not universally applicable and should not be followed blindly. For example, I am a pilot; I follow the discipline of using checklists instead of committing many complex procedures to memory. I take that discipline very seriously. There are, however, certain procedures that I must commit to memory because time is of the essence, and the time it would take to pull out and follow a checklist would be unsafe. Pilots call these procedures *memory items*, and the learning and refreshing of memory items is yet another discipline.

Tedious, Boring, and Slow

Any programmer who is unfamiliar with these disciplines, and who has been a programmer for more than a few years, may find these descriptions horrifying. They may sound tedious, boring, and slow. You might infer that you'd never be able to think through a problem. With TDD and TCR, it even seems likely that you'd never be able to write a little `while` loop or `if` statement without interrupting yourself. And all three will certainly break your concentration and interrupt your flow. In short, you could conclude that these disciplines are stupid.

But consider.

Debugging

Imagine a room full of programmers following one of these disciplines. Walk up to any programmer you like, at any time you like. Everything they are working on executed and passed their tests sometime in the last few minutes. And it doesn't matter who you pick or when you pick them: Everything worked a few minutes ago.

What would your life be like if everything worked a few minutes ago—and this was always true? How much debugging would you do?

It's hard to believe you would debug much if everything worked so recently. The most probable debugging technique would be to revert back to the last working version and just try again.

Those of us who follow these disciplines have little use for debuggers. Oh, I'll fire one up from time to time. I mean, this is still software, and it's still hard. Weird things still happen. But the time I spend in the debugger is vanishingly small. So small, in fact, that I don't remember the hot keys.

Every time I try to use the debugger I have to reacquaint myself with how to operate it.

Following one these disciplines (TDD) has reduced my debugging time by a very significant factor. I contend that any of these disciplines will do the same for every adherent who masters them.

Documentation

Have you ever integrated a third-party package? Perhaps you downloaded a zip file from a website. Upon unzipping, you find the artifacts you need to integrate, but also a PDF manual with instructions to help with the integration. At the end of that manual, there is an ugly appendix with all the code examples. And, of course, that’s the first place you go.

You go to the code examples because you hope that the example code is current and that it will tell you more truth than the words within the manual. If you are very lucky, you can copy and paste that code from the manual into your project and fiddle it into working.

When you follow a testing discipline, you write a series of tiny little tests. Each one of those tests describes how a part of the system works. If you want to know how to create an object, there are tests that create that object every way it can be created. If you need to know how to call a certain API, there are tests that call that API every way it can be called. These tests are the code examples for the whole system.

What’s more, each one of those tests is isolated from all the others. The tests themselves are not a system. Each is understandable in its own right.

The tests produced are a document that describes the low-level design of the system. And that document is written in a language that you intimately understand. It is utterly unambiguous, it is so formal that it executes, and it cannot get out of sync with the application that it describes. The tests are an almost perfect form of low-level documentation.

That's not to say that they are high-level documentation—they are not. But at the lowest level of the system, the tests are about as good a document as can be created.

Reliability

Imagine a programmer who does not follow one of these disciplines. He has just finished writing a module and has run it through a battery of manual tests. It seems to work just fine. But then the process proctor reminds him that he needs to write unit tests. He feels that this is redundant make-work since he's already tested the module, so his effort is less than stellar.

He writes a few tests here and there, and they pass just fine. But then he encounters the module that's hard to test. It's hard to test because it was not designed to be testable. It is coupled to things that should not be executed in the context of a test.

He could redesign that module to make it testable; but that would take a lot of time that he doesn't think he has. And besides, his manual tests proved to him that everything already works. So he walks away without writing that test.

Now, if you've been a programmer for more than a few years, you may recognize this scenario. Many of us have written tests after the fact; and many of us have walked away from a difficult test because we didn't want to change a "working" design.

Unfortunately, this leaves a hole in the test suite. And if you have left holes in the test suite, you know everyone else has left some holes as well. And so everyone knows that the test suite is full of holes.

What happens when you run such a test suite and see it pass? Everyone smiles knowingly and no other action is taken. When you know that the test suite is full of holes, there is no safe decision you can make when it passes. All you can do is smile, knowing all too well that the passing of the test suite does not mean that the system works properly.

When you follow a testing discipline, things are very different. When the tests pass, you know with a high degree of confidence that the system works as it should. And that allows you to make critical decisions.

For example, you could decide to *deploy* the system. I've worked on systems like this. When the tests passed, we deployed because we had very high confidence in our test suite.

Even if deployment is not an option, the passing of the tests could allow you to move the system into a new stage. For example, you could move it from DEV to QA.

Design

Something else happens when you follow these disciplines. You cannot write the module that's hard to test. By writing the test along with the code, you are automatically designing every module to be easy to test.

A module that is easy to test is decoupled. It is the decoupling that makes the module easy to test. So the end result is a less-coupled design. The mere act of writing the

tests along with the code improves the overall design of the system.

Reprise

So, by following these disciplines, you will in all likelihood:

- Significantly reduce your debugging time.
- Create a stream of nearly perfect low-level documentation.
- Create a test suite that you trust so much that you can deploy when it passes.
- Create a system with a less coupled design.

And while these four things are good—very good, in fact—they are not the reason that we follow a testing discipline.

The Angel and the Devil

In Chapter 1, "Clean Code," we discussed the cost of making a mess. I described the way that the ever-increasing mess degrades productivity. I introduced the Boy Scout Rule and told you that keeping the code clean was the only way to prevent that loss.

So how do you keep the code clean?

Imagine that you are sitting at your computer, scrolling through some code you need to work on. As you scroll, the blood drains from your face and you realize that this code is an ungodly mess that is going to be hell to work with.

Just then, a little angel appears on your right shoulder and whispers in a still small voice: "You could clean it."

But then, in a gout of flame, a devil appears on your left shoulder and screams: “NO! Don’t touch it! If you touch it, you will break it; and if you break it, it will become yours FOREVER. Muahahahahahaha!”

And so you ignore the angel and back away from the code. You leave it in its current state. You leave it for someone else to deal with. You leave it to fester and to rot and to pull down the productivity of the team.

This is a fear reaction. You are afraid to clean the code.

I want you to think very carefully about how wildly unprofessional and irresponsible it is to have allowed the code to get so far out of your control that you are afraid of it. You react to it, rather than it reacting to you. You defer to its demands, rather than forcing it to conform to yours. You, the creator, have lost control of your creation and have become a minion of its whims.

And because of that fear, the only thing that can happen to the code in that system is that it *must* rot. No one dares do the one thing that might reverse that rot—cleaning. And so down it will go, with every month getting worse and worse, and dragging the productivity of the team down with it.

And if you think that’s an exaggeration, talk to some of the older programmers you know. They’ll tell you.

Muzzling the Devil

Let’s revisit that scenario again. But this time let’s say that the team has been practicing a good testing discipline and has built up a very good test suite.

There you are, looking at your screen, scrolling through the code. You see that bad code and the blood drains from your face as you stare into the depths of its horror. And, as

expected, the little angel whispers in that still small voice: “Clean it!”

And the devil does not appear. The devil has no power. Because you have the test suite.

So you make a small change, perhaps just the name of a variable. And you run the tests. And the tests pass and you know you haven’t broken anything. So you make another small change, perhaps splitting a longer function in two. And again, the tests pass. So you move one of those new functions to a different class, and the tests fail! So you quickly revert and see the tests pass. Then, you realize your mistake and you move the function again, but this time correctly, and the tests pass.

If you have a test suite, and if you trust that test suite, and if that test suite runs quickly, then you *will* clean the code. You will, because cleaning the code is virtually risk free. And it won’t be just you cleaning the code—everyone on the team will clean the code. Everyone on the team will follow the Boy Scout Rule and continuously improve the quality of the code, keeping productivity high.

Complications and Loopholes

If that last discussion sounded too good to be true, it is. Oh, it can be true for a very large fraction of a system; but the devil still hides in the details. There are situations where writing tests is either impossible or at least impractical.

For example, it is impractical, if not downright impossible, to write tests for code that communicates outside the hardware boundary of the machine. For example, unless you somehow use a camera that’s staring at the screen, it is impossible to test the final output of a graphical user interface. The same holds true for testing the movement of

the mouse, or the data going in and out over a socket. Automated tests for these things are not quite impossible; but in most cases, writing them is deeply impractical.

Therefore, seasoned developers will design their systems such that those untestable stretches of code are very thin. They will move as much intelligence as possible away from the hardware boundaries and ensure that the code that does finally touch those boundaries is as anemic as possible.

This technique is called *The Humble Object Pattern*.[6] I talk in much more detail about these kinds of situations in my book *Clean Craftsmanship*.

[6] http://xunitpatterns.com/Humble%20Object.html

Another complication can occur when using third-party frameworks. If a framework was not written to be testable, then the code that uses that framework may also be impractical to test. Again, good designers will anticipate this and attempt to push as much intelligence away from the framework as possible and build an isolation layer between the framework and the code that needs to be tested. Care should be taken when committing to frameworks, because they have the potential to prevent very large swathes of code from being properly tested.

> Experience Report: I recently wrote a significant application that used the Java Swing framework. Trying to test any of the code that touched Swing was impractical. I had to separate the application into a tested component and an untested (Swing) component. That untested component grew fairly large because the Swing API forces the construction of large, interconnected data structures that contain callbacks. This left my

> application with ~70% test coverage. I could easily refactor the tested component; but touching the Swing component was always very risky. I won't use Swing again.

Still another complication occurs when you write a function, but you don't know the correct results of that function. For example, it is hard to test whether a font looks "right," or whether a field is positioned "correctly," or whether the format of a report looks "good." In all of these cases, the results are subjective. The only way to ascertain whether the code is correct is to look at the results and make a judgment. There's no way to write a test for that. So there's no way to write the code and tests together. It's also doubtful that writing a test long after the fact is very helpful, since all it can manage to do is cement a subjective decision.

These three cases are the only situations that I have encountered in which tests are impractical. There are, however, likely to be others. For example, how do you test the results of an AI?

In the end, when faced with areas of the system that are impractical to test, try to make those areas as small and simple as possible, and protect the rest of the system from them.

Costs and Repercussions

If we follow these disciplines, we will write dozens of tests every day, hundreds of tests every month, and thousands of tests every year. Those tests will cover virtually all of our production code. The sheer bulk of those tests, which can rival the size of the production code itself, can present a daunting management problem.

Keeping Tests Clean

Many years ago I was asked to coach a team who had explicitly decided that their test code should not be maintained to the same standards of quality as their production code. They gave one another license to break all the rules of clean code and good design in their unit tests. “Quick and dirty” were the watchwords. Their variables did not have to be well named; their test functions did not need to be short and descriptive. Their test code did not need to be well designed and thoughtfully partitioned. So long as the test code worked, and so long as it covered the production code, it was good enough.

Some of you reading this might sympathize with that decision. Perhaps, long in the past, you wrote tests of the kind that I wrote for that `Timer` class mentioned at the beginning of this chapter. It’s a huge step from writing that kind of throwaway test to writing a suite of well-designed, well-partitioned, automated unit tests. So, like the team I was coaching, you might decide that having dirty tests is better than having no tests.

What that team did not realize was that having dirty tests is equivalent to, if not worse than, having no tests. The problem is that tests must change as the production code evolves. The dirtier the tests, the harder they are to change. The more tangled the test code, the more likely it is that you will spend more time cramming new tests into the suite than it takes to write the new production code. As you modify the production code, old tests start to fail, and the mess in the test code makes it hard to get those tests to pass again. So the tests become viewed as an ever-increasing liability.

From release to release the cost of maintaining this team’s test suite rose. Eventually, it became the single biggest

complaint among the developers. When managers asked why their estimates were getting so large, the developers blamed the tests. In the end, they were forced to discard the test suite entirely.

But, without a test suite, they lost the ability to make sure that changes to their codebase worked as expected. Without a test suite, they could not ensure that changes to one part of their system did not break other parts of their system. So their defect rate began to rise. As the number of unintended defects rose, the devil's voice grew louder, and they started to fear making changes. They stopped cleaning their production code because they feared the changes would do more harm than good. Their production code began to rot. In the end, they were left with no tests, tangled and bug-riddled production code, frustrated customers, and the feeling that their testing effort had failed them.

In a way, they were right. Their testing effort *had* failed them. But it was their decision to allow the tests to be messy that was the seed of that failure. Had they kept their tests clean, their testing effort would not have failed. I can say this with some certainty because I have participated in, and coached, many teams who have been successful with *clean* unit tests.

The moral of the story is simple: Test code is just as important as production code. It is not a second-class citizen. It requires thought, design, and care. It must be kept as clean as production code.

Tests Enable the -ilities

If you don't keep your tests clean, you will lose them. And without them, you lose the very thing that keeps your production code flexible. Yes, you read that correctly. It is

tests that keep our code flexible, maintainable, and reusable. The reason is simple. If you have tests, you do not fear improving the code! Without tests, every change is a possible bug. No matter how flexible your architecture is, no matter how nicely partitioned your design is—without tests, you will be reluctant to make changes due to the fear that you will introduce undetected bugs. Without tests, you fear to clean the code.

But with tests, that fear virtually disappears. The higher your test coverage, the lower your fear. You can make changes with near impunity to code that has a less-than-stellar architecture and a tangled and opaque design. Indeed, you can improve that architecture and design without fear!

So, having an automated suite of tests that cover the production code is the key to keeping your design and architecture as clean as possible. Tests enable all the -ilities, because tests enable change.

So, if your tests are dirty, then your ability to change your code is hampered, and you begin to lose the ability to improve the structure of that code. The dirtier your tests are, the dirtier your code becomes. Eventually, you lose the tests, and your code rots.

Chapter 15. Clean Tests



Clean tests are readable, fast, isolated, repeatable, self-verifying, timely, and designed. Readability is perhaps even more important in unit tests than it is in production code.

What makes tests readable? The same things that make all code readable: clarity, simplicity, and density of expression. In a test, you want to say a lot with as few expressions as reasonable.

Consider this code from FitNesse. These three tests are difficult to understand and can certainly be improved. First, there is a terrible amount of duplicate code in the repeated calls to `addPage` and `assertSubString`. More importantly, this code is just loaded with details that interfere with the expressiveness of the test.

```
public void testGetPageHierarchyAsXml() throws Exception {
  crawler.addPage(root, PathParser.parse("PageOne"));
  crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
  crawler.addPage(root, PathParser.parse("PageTwo"));

  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
       new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
}
```

```
public void testGetPageHierarchyAsXmlDoesntContainSymbolicLinks
throws Exception {
  WikiPage pageOne =
    crawler.addPage(root, PathParser.parse("PageOne"));
  crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
  crawler.addPage(root, PathParser.parse("PageTwo"));

  PageData data = pageOne.getData();
  WikiPageProperties properties = data.getProperties();
  WikiPageProperty symLinks =
    properties.set(SymbolicPage.PROPERTY_NAME);
  symLinks.set("SymPage", "PageTwo");
  pageOne.commit(data);

  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
       new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
  assertNotSubString("SymPage", xml);
}
```

```
public void testGetDataAsHtml() throws Exception {
  crawler.addPage(root, PathParser.parse("TestPageOne"), "test

  request.setResource("TestPageOne");
  request.addInput("type", "data");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
       new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubString("test page", xml);
  assertSubString("<Test", xml);
}
```

For example, look at the `PathParser` calls. They transform strings into `PagePath` instances used by the crawlers. This transformation is completely irrelevant to the test at hand and serves only to obfuscate the intent. The details surrounding the creation of the responder and the gathering and casting of the response are also just noise. Then there's the ham-handed way that the request URL is built from a resource and an argument. (I helped write this code 20 years ago, so I feel free to roundly criticize it.)

In the end, this code was not designed to be read. The poor reader is inundated with a swarm of details that must be understood before the tests make any real sense.

Now consider the improved tests below. These tests do the exact same thing, but they have been refactored into a much cleaner and more explanatory form.

```
  public void testGetPageHierarchyAsXml() throws Exception {
    makePages("PageOne", "PageOne.ChildOne", "PageTwo");

    submitRequest("root", "type:pages");

    assertResponseIsXML();
    assertResponseContains(
      "<name>PageOne</name>",
      "<name>PageTwo</name>",
      "<name>ChildOne</name>");
  }

  public void
  testSymbolicLinksAreNotInXmlPageHierarchy() throws Exception {
    WikiPage page = makePage("PageOne");
    makePages("PageOne.ChildOne", "PageTwo");

    addLinkTo(page, "PageTwo", "SymPage");

    submitRequest("root", "type:pages");

    assertResponseIsXML();
    assertResponseContains(
      "<name>PageOne</name>",
      "<name>PageTwo</name>",
      "<name>ChildOne</name>"
```

```
    );
    assertResponseDoesNotContain("SymPage");
  }

  public void testGetDataAsXml() throws Exception {
    makePageWithContent("TestPageOne", "test page");

    submitRequest("TestPageOne", "type:data");

    assertResponseIsXML();
    assertResponseContains("test page", "<Test");
  }
```

The Arrange/Act/Assert (AAA)[1] pattern is made obvious by the structure of these tests. Each of the tests is clearly split into three parts. The first part (*Arrange*) builds up the test data, the second part (Act) operates on that test data, and the third part (*Assert*) checks that the operation yielded the expected results.

[1] https://automationpanda.com/2020/07/07/arrange-act-assert-a-pattern-for-writing-good-tests/

Notice that the vast majority of annoying detail has been eliminated. The tests get right to the point and use only the data types and functions that they truly need. Anyone who reads these tests should be able to work out what they do very quickly, without being misled or overwhelmed by details.

Domain-Specific Testing Language

The tests above demonstrate the technique of building a domain-specific language for your tests. Rather than using the APIs that programmers use to manipulate the system, we build up a set of functions and utilities that make use of those APIs and that make the tests more convenient to write and easier to read. These functions and utilities become a specialized API used by the tests. They are a testing language that programmers use to help them write their tests and to help those who must read those tests later on.

Composed Assertions

The functions `assertResponseIsXML` and `assertResponseContains` are part of that domain-specific testing language. These kinds of assertion functions are called *composed assertions*.

Testing APIs like these are not designed up front; rather, they evolve from the continued refactoring of test code that has gotten too tainted by obfuscating detail. Just as you saw me refactor the original tests, so too will disciplined developers refactor their test code into more succinct and expressive forms.

Composed Test Results

Another technique for creating a domain-specific testing API is to create composed test results. These are used when the test results are needlessly complex and can be compressed down into a smaller form that is easy to understand.

Consider the tests I wrote as part of an environment control system I was prototyping. Without going into the details, you can tell that this test checks that the low-temperature alarm, the heater, and the blower are all turned on when the temperature is “way too cold.”

```
@Test
```

```
public void turnOnLoTempAlarmAtThreshold() throws Exception {
  hw.setTemp(WAY_TOO_COLD);
  controller.tic();
  assertTrue(hw.heaterState());
  assertTrue(hw.blowerState());
  assertFalse(hw.coolerState());
  assertFalse(hw.hiTempAlarm());
  assertTrue(hw.loTempAlarm());
}
```

There are, of course, lots of details here. For example, what is that `tic` function all about? In fact, I'd rather you not worry about that while reading this test. I'd rather you just worry about whether you agree that the end state of the system is consistent with the temperature being "way too cold."

Notice, as you read the test, that your eye needs to bounce back and forth between the name of the state being checked and the *sense* of the state being checked. You see `heaterState`, and then your eyes glissade left to `assertTrue`. You see `coolerState`, and your eyes must track left to `assertFalse`. This is tedious and unreliable. It makes the test hard to read.

To make matters worse, this is one of 14 very similar tests. Imagine how your eyes will cross as you try to read all that.

I improved the reading of these tests greatly by transforming them as follows.

```
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
  wayTooCold();
  assertEquals("HBchL", hw.getState());
}
```

Of course, I hid the detail of the `tic` function by creating a `wayTooCold` function. But the thing to note is the strange string in the `assertEquals`. Uppercase means “on,” lowercase means “off,” and the letters are always in the following order: {heater, blower, cooler, hi-temp-alarm, lo-temp-alarm}.

Once you know the meaning, your eyes glide across that string and you can quickly interpret the results. Reading the test becomes almost a pleasure. Just take a look at a few of these tests and see how easy they are to comprehend.

```
@Test
public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
  tooHot();
  assertEquals("hBChl", hw.getState());
}
@Test
public void turnOnHeaterAndBlowerIfTooCold() throws Exception {
  tooCold();
  assertEquals("HBchl", hw.getState());
}
@Test
public void turnOnHiTempAlarmAtThreshold() throws Exception {
  wayTooHot();
  assertEquals("hBCHl", hw.getState());
}
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
  wayTooCold();
  assertEquals("HBchL", hw.getState());
}
```

```
}
```

The `getState` function that creates the composed test result is shown below.

```
public String getState() {
  String state = "";
  state += heater ? "H" : "h";
  state += blower ? "B" : "b";
  state += cooler ? "C" : "c";
  state += hiTempAlarm ? "H" : "h";
  state += loTempAlarm ? "L" : "l";
  return state;
}
```

Dual Standard

You might complain that composing the test result in this fashion is wasteful of computing resources. After all, I didn't even use a `StringBuffer` or `StringBuilder` to do the concatenation.

This application is clearly an embedded real-time system, and it is likely that computer and memory resources are very constrained. However, the test environment runs in my laptop and is not constrained at all.

That is the nature of the dual standard between test and production. There are things that you might never do in a production environment that are perfectly fine in a test environment. Usually they involve issues of memory or CPU efficiency. But they never involve issues of cleanliness.

The Single Assert Rule

This rule states that a good test asserts one and only one thing. This is often misunderstood to mean that a test should have only a single assertion statement. In reality, a test can have many assertion statements and still assert one logical fact.

For example, consider the first set of environment controller tests above. They may be hard to read with all those assertions, but they are asserting a single logical fact. Indeed, that is why we could refactor them with composed test results. Or, consider the refactored assertions in in the `pageHierarchy` example above. They are asserting the single logical fact that the response is an XML document that has certain contents.

You might find this confusing, or at least subjective. After all, the contents of the document and the type of the document are different facts, aren't they?

The confusion arises because the Single Assert Rule is poorly named.

The Single Act Rule

The rule should really be called the Single Act Rule. The Arrange-Act-Assert pattern says that every test should be composed of three elements: the arrangement, the action, and the assertion. The first arranges the data to be tested. The second is the action upon that data that we wish to test. The third is the assertion that the data has been properly transformed by the action.

The goal of the Single Assert Rule is to make sure that each test is testing one and only one action. We want to avoid tests that arrange, act, assert, act, assert. We also want to avoid tests that arrange, act, act, assert, assert. We want every action tested individually because we don't want

downstream assertions to be corrupted by upstream actions. We want every test to stand alone.

F.I.R.S.T.

Here is a simple memory aid for the desired characteristics of tests.

Fast

Tests should be fast. They should run quickly. When tests run slow, you won't want to run them frequently. If you don't run them frequently, you won't find problems early enough to fix them easily. You won't feel as free to clean up the code. Eventually the code will begin to rot. So consider the speed of the tests to be a design imperative. Design your tests to be fast. Very fast.

Isolated

This is a restatement of the Single Act Rule. Tests should not depend on each other. One test should not set up the conditions for the next test. You should be able to run each test independently and run the tests in any order you like. When tests depend on each other, the first one to fail causes a cascade of downstream failures, making diagnosis difficult and hiding downstream defects.

Repeatable

Tests should be repeatable in any environment. You should be able to run the tests in the production environment, in the QA environment, and on your laptop while riding home on the train without a network. If your tests aren't repeatable in any environment, then you'll always have an

excuse for why they fail. You'll also find yourself unable to run the tests when the environment isn't available.

Self-Validating

The tests should have a boolean output. Either they pass or they fail. You should not have to read through a log file to tell whether the tests pass. You should not have to manually compare two different text files to see whether the tests pass. If the tests aren't self-validating, then failure can become subjective and running the tests can require a long manual evaluation.

Timely

The tests need to be written in a timely fashion. Unit tests should be written along with the production code that makes them pass. If you write tests long after the production code, then you may find the production code to be too hard to test. Timely tests cause the design of the production code to be testable.

Test Design

As we saw in Chapter 9, "The Clean Method," tests need to be designed to reduce their coupling to the production code. This makes sure that changes to the requirements affect as few tests as possible.

Suppose you are working on a large system composed of hundreds of modules. Suppose a single change to that system forces you to make changes to a majority of those modules. That system is poorly designed—by definition.

The principles and rules of software design, when carefully followed, prevent—or at least mitigate—that nightmare

scenario.

Now imagine you have a well-designed system with hundreds of modules and a test suite with thousands of unit tests. Suppose a single change to that system breaks hundreds of those tests. That is a very poorly designed suite of tests.

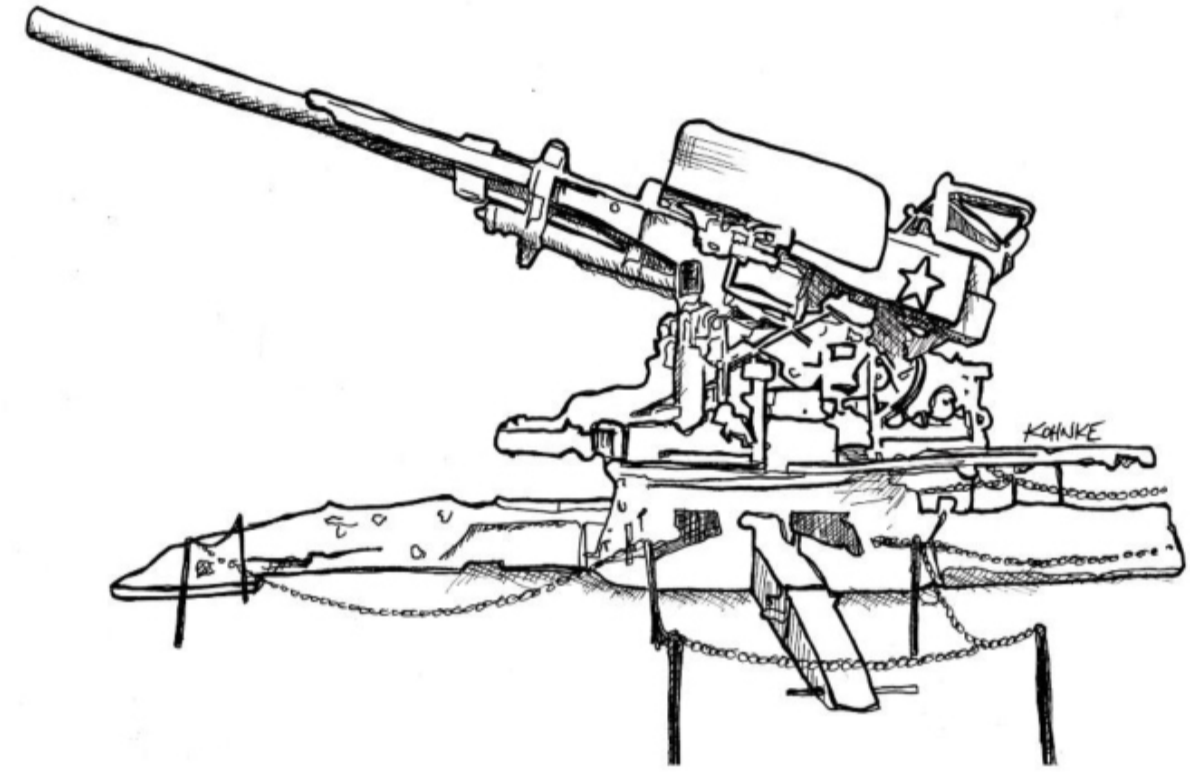
The principles and rules of software design apply to the tests as much as they apply to the production code. Coupling must be minimized. One change to the signature of a function in the production code must not cause hundreds of tests to be altered, just as it must not cause hundreds of other modules to be altered.

Conclusion

We have barely scratched the surface of this topic. My book *Clean Craftsmanship* goes very deep into the topic of clean and well-designed tests. Tests are as important to the health of a project as the production code is. Perhaps they are even more important, because tests preserve and enhance the flexibility, maintainability, and reusability of the production code. So keep your tests constantly clean and well designed. Work to make them expressive and succinct. Invent testing APIs that act as domain-specific languages that help you write the tests.

If you let the tests rot, then your code will rot too. Maintain your disciplines and keep your tests clean.

Chapter 16. Acceptance Testing



Of all the disciplines of clean code, this is the one that programmers have the least control over. Fulfilling this discipline requires the participation of the business. Unfortunately, many businesses have, so far, proven unwilling to properly engage.

How do you know when a system is ready to deploy? Organizations around the world frequently make this

decision by engaging a QA department or group to “bless” the deployment. Typically, this means that the QA folks run a rather large battery of manual tests that walk through the various behaviors of the system until they are convinced that the system behaves as specified. When those tests “pass,” the system may be deployed.

This means that the true requirements of the system are those tests. It does not matter what the requirements document says, it is only the tests that matter. If QA signs off after running their tests, the system is deployed. Therefore, it is those tests that are the requirements.

The Acceptance Testing Discipline

The discipline of acceptance testing recognizes this simple fact and recommends that all requirements be specified as tests. Those tests should be written by business analysts (BA) and QA on a feature-by-feature basis, shortly before each feature is implemented. QA is not responsible for running those tests. Rather, that task is left to the programmers; and therefore, the programmers will very likely automate those tests.

No programmer in their right mind wants to manually test the system over and over again. Programmers automate things. Thus, if the programmers are responsible for running the tests, the programmers *will* automate those tests.

However, since BA and QA author the tests, the programmers must be able to prove to BA and QA that the automation actually performs the tests that were authored. Therefore, the language in which the tests are automated must be a language that BA and QA understand. Indeed, BA and QA ought to be able to *write* the tests in that automation language.

There are several tools that have been invented over the years to help with this problem: FitNesse,[1] JBehave, SpecFlow, Cucumber, and others. But tools are not really the issue. The specification of software behavior is always a simple function of specifying input data, the action to perform, and the expected output data. This is the well-known AAA[2] pattern of Arrange/Act/Assert, or the Given-When-Then discipline of behavior-driven development (BDD).

[1] https://fitnesse.org/

[2] Credit: Bill Wake.

All tests begin by arranging the input data for the test. Then, the test causes the tested action to be performed. Finally, the test asserts that the output data from that action matches the expectation.

These three elements can be specified in a variety of different ways; but the easiest approach is a simple tabular format.

The following figure, for example, is a portion of one of the acceptance tests within the FitNesse tool. FitNesse is a wiki, and this test checks that the various markup gestures are properly translated into HTML. The action to be performed is "`widget should render`," the input data is the "`wiki text`," and the output is the "`html text`."

widget should render		
wiki text	html text	
normal text	normal text	
this is ''italic'' text	this is italic text	italic widget
this is '''bold''' text	this is bold text	bold widget
!c This is centered text	This is centered text	
!1 header	header	
!2 header	header	
!3 header	header	
!4 header	header	
!5 header	header	
!6 header	header	
http://files/x	http://files/x	file link
http://fitnesse.org	http://fitnesse.org	http link
SomePage	SomePage\[\?\]	missing wiki word

Following is an example of the Given-When-Then style:

```
Given a page with the wiki text: !1 header
When that page is rendered.
Then the page will contain: <h1>header</h1>
```

It should be clear that these formalisms, whether they are written in an acceptance testing tool or whether they are written in a simple spreadsheet or text editor, are relatively easy to automate.

The Discipline

In the strictest form of the discipline, the acceptance tests are written by BA and QA. BA focuses on the happy path scenarios, while QA focuses on exploring the myriad of ways that the system can fail.

These tests are written at the same time, or just before, the features they test are developed. In an Agile project, divided up into sprints or iterations, the tests are written during the first few days of the sprint. They should all pass by the end of the sprint.

BA and QA provide the programmers with these tests, and the programmers automate them in a manner that keeps BA and QA engaged.

These tests become the definition of done. A feature is not complete until all its acceptance tests pass. And when all the acceptance tests pass, the feature is done.

This, of course, puts a huge responsibility on BA and QA. The tests that they write must be full specifications of the features being tested. The suite of acceptance tests *is* the requirements document for the entire system. By writing those tests, BA and QA are certifying that when they pass, the specified features are done and working.

In some cases, BA and QA may not be accustomed to writing such formal and detailed documents. In that case, the programmers may wish to write the acceptance tests with guidance from BA and QA. The intermediate goal is to

create tests that BA and QA can read and bless. The ultimate goal is to get BA and QA comfortable enough to write the tests.

The Continuous Build

Once an acceptance test passes, it goes into the suite of tests that is run during the continuous build.

The continuous build is an automated procedure that runs every time[3] a programmer checks code into the source code control system. This procedure builds the system from source and then runs the suite of automated programmer unit tests and the suite of automated acceptance tests. The results of that run are visibly posted, often in an email to every programmer and interested party. The state of the continuous build is something everyone should be continuously aware of.

[3] Within a few minutes.

The continuous running of all these tests ensures that subsequent changes to the system do not break working features. If a previously passing acceptance test fails during the continuous build, the team must immediately respond and repair it before making any other changes. Allowing failures to accumulate in the continuous build is suicidal.

Conclusion

Acceptance testing is one of the weakest aspects of software development in the first half of the twenty-first century. This must change. The businesses and the developers must learn to collaborate to create a language and a procedure that allow requirements to be formally specified by BA and QA and that allow programmers to quickly verify that those

requirements have been satisfied. The advent of AI makes this need doubly important.

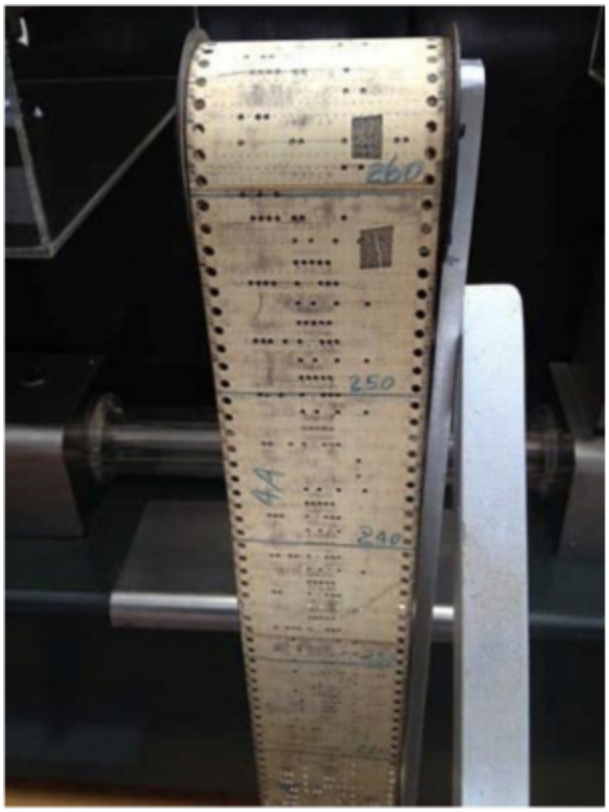
Chapter 17. AIs, LLMs, and God Knows What



In the late 1940s, Grace Hopper wrote her first lines of code. Each line looked something like this[1]:

[1] [We Programmers].

Each digit represented a particular hole on a paper tape that looked like this:



Each row of that paper tape was an instruction for the Harvard Mark I to execute. The instructions were very simple—something like “Add Register 10 to Register 12.”

I’ll leave you to imagine just how hard programming was back then.

Within a few years, it became clear that a computer program could be used to generate codes like that from a more abstract textual representation like “add 10, 12.” Hopper called this *automatic programming*.

Before a decade had passed from Hopper’s first line of code, John Backus and his team had produced the first Fortran compiler, which used statements like “a = b + c.” Algol came a handful of years later, and within a decade, Dahl and

Nygaard had created Simula-67, the first object-oriented language. C, Unix, Smalltalk, and C++ were born within a few years. And within the next few, Java and C# appeared.

At each one of those transitions, the productivity of programmers increased. The first leap in productivity was the greatest. In the ’50s, the productivity of programmers using a Fortran-like compiler was shown to be 45 times that of programmers using machine code.

Each productivity leap thereafter was smaller than the previous. C was probably twice that of Fortran. C++ was likely a 20% improvement on C. Java/C# was perhaps on the order of another 20%. Ruby/Python are likely 20% better than Java, and Clojure likely adds another 10%.

Each of these transitions in language was driven by an increase in abstraction. Each new language in that sequence was another step further removed from Grace Hopper’s holes in that paper tape. And each of those steps was accompanied by the fear that programming was being ruined.

Yes, you read that correctly. At every stage, there were those who predicted the end of programming. Books were published that predicted the doom of the industry. They claimed that programmers would become as rare as dodo birds.

Every time we made programming a little easier, those of us who were invested in the current complexity were liable to fear that unwashed masses would take our jobs. And so we predicted the doom of the profession. And every time, those fears and predictions were the exact opposite of the actual outcomes. With every new advance, the number of projects and the demand for programmers grew.

Today, we face yet another such transition—the transition of AI. And there is, of course, rampant fear that programming is dead and that programmers will soon go extinct.

Baloney! This is just another increase in the level of abstraction of our languages—another step away from the holes in the paper tape. And as we come to grips with it, the number of projects and the demand for programmers will continue to grow.

Programming by Prompt

This is a very new technology, and we don't have a good handle on how to use it yet. This is reminiscent of our first forays into Java and the Web. Does anybody remember applets? It took several years before we worked out the rudiments of how to use Java/C# well. It took several more to refine those rudiments.

The same happened with Ruby and Python. At first we stumbled and bumbled around, and then we gradually figured out the best ways to proceed. And believe me, when Fortran first came out, there was plenty of fear, anguish, doubt, and bumbling and stumbling.

And so it will be with AIs. At this point, we have no idea what to do. We are just now barely able to bumble and stumble our way to write programs with prompts. We have no idea how to do it well.

So, for example, consider the following prompt that I gave to Grok[2] a few days ago:

[2] Grok 3 beta (19 March, 2025).

```
write a clojure program that finds all sentences
in a given file that contain a given substring.
Choose one of those sentences at random and find
```

```
choose one of those sentences at random and find
the next word in the substring, or a . if no word follows.
Add the found word to the substring and remove the
first if the substring is longer than three words.
Repeat the entire process until the substring
ends in a . or has exceeded 50 words.

Accumulate all the words that were found and
return them appended to the initial substring.
```

Here’s the code that Grok generated. It works—after a fashion.

```
(ns markov
  (:require [clojure.string :as str]))

(defn read-file [filename]
  "Reads the content of a file and splits it into sentences."
  (try
    (let [content (slurp filename)
          sentences (filter not-empty
                            (map str/trim
                                 (str/split content #"[.!?]")))]
      sentences)
    (catch Exception e
      (throw (Exception. (str "Error reading file: " (.getMessage

(defn find-sentences-with-substring [sentences substring]
  "Finds all sentences containing the substring."
  (filter #(str/includes? (str/lower-case %) (str/lower-case subst
```

```
          sentences))

(defn get-next-word [sentence substring]
  "Finds the next word after the substring
  in the sentence, or '.' if none."
  (let [index (str/index-of (str/lower-case sentence)
                            (str/lower-case substring))]
    (if index
      (let [post-substring (subs sentence (+ index (count substrin
            words (str/split (str/trim post-substring) #"\s+")]
        (if (seq words)
          (first words)
          "."))
      "."))))

(defn update-substring [substring next-word]
  "Updates the substring by adding the next word
   and trimming if longer than 3 words."
  (let [words (str/split substring #"\s+")
        new-words (conj (vec words) next-word)
        trimmed-words (if (> (count new-words) 3)
                        (subvec new-words 1)
                        new-words)]
    (str/join " " trimmed-words)))

(defn sentence-walk [filename initial-substring]
  "Main function to perform the sentence walking process."
  (let [sentences (read-file filename)]
```

```
    (loop [substring initial-substring
           accumulated-words []
           iterations 0]
      (if (or (= (last (str/split substring #"\s+")) ".")
              (> iterations 50))
        (str/join " " (concat (str/split initial-substring #"\s+")
                              accumulated-words))
        (let [matching-sentences (find-sentences-with-substring
                                  sentences substring)]
          (if (empty? matching-sentences)
            (str/join " " (concat (str/split initial-substring #"\
                                  accumulated-words))
            (let [random-sentence (rand-nth matching-sentences)
                  next-word (get-next-word random-sentence substri
              (recur (update-substring substring next-word)
                     (conj accumulated-words next-word)
                     (inc iterations)))))))))


(defn -main [& args]
  (if (< (count args) 2)
    (println "Usage: program <filename> <initial-substring>")
    (let [filename (first args)
          initial-substring (second args)]
      (try
        (println (sentence-walk filename initial-substring))
        (catch Exception e
          (println (.getMessage e)))))))
```

It is easy to be impressed by this. I certainly am. But it also terrifies me. I am not terrified by the power of the AI to generate code. I am terrified by the structure of that horrible prompt.

Look carefully at that prompt. What is the definition of `sentence`? What is the definition of `word`? What does the clause `the found word` mean? Are those dot characters punctuation or nouns? What does it mean to `accumulate words`?

Yes, Grok figured all this out well enough for this little experiment. But the program that resulted was by no means ready for production. It makes lots of dumb little mistakes, most of which have to do with the questions I raised in the previous paragraph.

Here, for example, is a sample run:

```
>clj -M -m markov
~/Documents/Books/Clean\ Code\ 2ded\ 20250311.txt "The Single"

The Single Responsibility Principle, and the Dependency
relationships and wind up back at that first solution, the one wit
the project you were working on  it every day  business and person
```

Notice the double spaces in the third and fourth lines. Why is it doing that?

I could fix those mistakes by adding clarification to the prompt. For example, I might add:

```
Make sure all accumulated words are separated by just one space.
```

But there are a number of problems with this approach. First, I haven’t figured out why those double spaces are there, so this change to the prompt is just a patch to fix the

symptom. Second, am I sure that the clause `all accumulated words` will properly refer to the return value?

I could try it to find out; but that causes another problem. With each new change to the prompt, Grok *regenerates* the program from scratch as opposed to modifying the existing program. With all the ambiguities in the prompt, how can I be sure that the regenerated program doesn't break something that worked in the previous one? Will it use the same definitions of `word` and `sentence` that it did before? Will it interpret `the found word` the same as it did before? Given the statistical and fuzzy nature of the LLM algorithms, there's a good chance it won't!

Here is what Grok generated for that modified prompt. Notice that the two programs look like they were written by two different programmers.

```
(ns sentence-walker
  (:require [clojure.string :as str]
            [clojure.java.io :as io]))


(defn get-sentences [filename]
  "Read file and split into sentences"
  (-> (slurp filename)
      (str/replace #"[.!?]\s+" "$0|")
      (str/split #"\|")
      (->> (map str/trim)
           (filter not-empty))))


(defn find-matching-sentences [sentences substring]
  "Find all sentences containing the substring"
  (filter #(str/includes? (str/lower-case %)
```

```
                          (str/lower-case substring))
          sentences))

(defn get-next-word [sentence substring]
  "Find word following substring in sentence, or . if at end"
  (let [idx (str/index-of (str/lower-case sentence)
                          (str/lower-case substring))
        substr-end (+ idx (count substring))
        remainder (subs sentence substr-end)]
    (if (empty? (str/trim remainder))
      "."
      (let [next-word (first (str/split (str/trim remainder) #"\s+
        (if (or (nil? next-word) (empty? next-word))
          "."
          next-word)))))

(defn update-substring [substring next-word]
  "Add next word and trim if longer than 3 words"
  (let [words (str/split substring #"\s+")
        new-words (conj (vec words) next-word)]
    (if (> (count new-words) 3)
      (str/join " " (subvec new-words (- (count new-words) 3)))
      (str/join " " new-words))))

(defn walk-sentences [filename initial-substring]
  "Main function to walk through sentences"
  (let [sentences (get-sentences filename)]
    (loop [current-substr initial-substring
```

```
            accumulated-words [initial-substring]
            iterations 0]
       (if (or (str/ends-with? current-substr ".")
               (> iterations 50))
         (str/join " " accumulated-words)
         (let [matching-sentences (find-matching-sentences
                                   sentences current-substr)]
           (if (empty? matching-sentences)
             (str/join " " accumulated-words)
             (let [random-sentence (rand-nth matching-sentences)
                   next-word (get-next-word random-sentence current
                   new-substr (update-substring current-substr next
               (recur new-substr
                      (conj accumulated-words next-word)
                      (inc iterations)))))))))

;; Example usage:
;; (walk-sentences "sample.txt" "the")
```

This new program solves the double-space problem but creates a new problem at the end of the sentence. It also doesn’t deal with uppercase and lowercase very well.

```
clean code is hard Writing a system that handles one user at a tim
```

Infancy

These are problems related to the infancy of the approach. In order to effectively program large systems by prompt, we are going to need AI tools that allow for incremental

changes. We are also going to need ways to eliminate the ambiguities in our prompts.

In short, we are going to need a formalism—a prompt language that is unambiguous and precise and that cannot be misinterpreted by the statistical and fuzzy nature of the LLM algorithm. We'll need to come up with a way to precisely say what we mean.

We'll also need to come up with a way to mean what we say. Or, in other words, even if we have a perfectly precise language, it is easy to precisely say something that is wrong. To ensure that our precise statements are accurate descriptions of our intent, we should state our intent in at least two different ways (just as I did in this paragraph!).

When you say something in two different ways, you are overloading the concept you are trying to communicate. Such overloading is a way to ensure that any error in either of the two statements will be detected by the other. We error-prone humans absolutely require overloading.

Overloading is the strategy behind double-entry bookkeeping and the testing disciplines described in earlier chapters. It would be supremely irresponsible to trust a single set of precise statements written by a human if those statements aren't overloaded. What we are going to need is a way to create precise and overloaded prompts and a way to quickly verify their consistency.

A SWAG

So let me make a Scientific Wild-Axx Guess.

Imagine a prompt language that contains formal definitions and constraints overloaded by a set of BDD-style[3] `Given-When-Then` scenarios.

[3] [RSpec].

Let’s give that a try with a slightly simpler problem. Notice how much it looks like COBOL. That similarity might give us pause.

```
Language: Clojure

Name: Markov

Command line arguments:

  f a text file.

  s a text string.


Definitions:

  Word: A string of alphabetic characters.

  Sentence: One or more words separated by spaces and ending with

  a period, question mark, or exclamation mark.


Constraints:

  Ignore case in all comparisons.


Description:

  Find all sentences in f that contain s.

  Choose a sentence c at random with equal probability.

  Return the word in c that immediately follows s;

    if none return a period.


Scenarios to be tested with speclj:

Given f contains the following lines:

  Oh, Hello there Bob, how are you today? I'm fine.

  Hello there Bill, I trust you are well.  How's the wife?

  Ah yes, hello there John, great day!  Kowabunga!
```

```
  Ah yes, hello there John, great day!  Kowabunga!

When s is "not in any sentence"
Return "."

When s is "hello"
Then return "there"

When s is "are well"
Then return "."

When s is "hello there"
Then return "Bob", "Bill", or "John".
Assert a uniform distribution after 100 iterations.
```

Grok was quite happy to generate the following code.

```
(ns markov.core
  (:require [clojure.string :as str]
            [clojure.java.io :as io]))

(defn get-sentences [file]
  (->> (slurp file)
       (re-seq #"[A-Za-z\s]+[.!?]")
       (map str/trim)))

(defn words [sentence]
  (str/split (str/lower-case sentence) #"\s+"))
```

```
(defn find-following-word [sentences search-str]
  (let [search-str (str/lower-case search-str)
        matching-sentences (filter
                            #(str/includes? (str/lower-case %)
                                            search-str)
                            sentences)]
    (if (empty? matching-sentences)
      "."
      (let [chosen (rand-nth matching-sentences)
            word-seq (words chosen)
            search-words (str/split search-str #"\s+")
            idx (.indexOf word-seq (last search-words))]
        (if (or (< idx -1) (>= (+ idx 1) (count word-seq)))
          "."
          (nth word-seq (+ idx 1)))))))

(defn -main [f s]
  (let [sentences (get-sentences f)
        result (find-following-word sentences s)]
    (println result)))
```

And here are the tests Grok generated from the scenarios.

```
;; speclj tests
(ns markov.core-spec
  (:require [speclj.core :refer :all]
            [markov.core :refer :all]
            [clojure.java.io :as io]))
```

```
(def test-content
  "Oh, Hello there Bob, how are you today? I'm fine.
   Hello there Bill, I trust you are well.  How's the wife?
   Ah yes, hello there John, great day!  Kowabunga!")

(describe "Markov"
  (before-all
    (spit "test.txt" test-content))

  (after-all
    (.delete (io/file "test.txt")))

  (it "returns period when string not found"
    (should= "." (find-following-word (get-sentences "test.txt")
                                      "not in any sentence")))

  (it "returns 'there' when searching for 'hello'"
    (should= "there" (find-following-word (get-sentences "test.txt
                                          "hello")))

  (it "returns period when searching for 'are well'"
    (should= "." (find-following-word (get-sentences "test.txt")
                                      "are well")))

  (it "returns Bob, Bill, or John for 'hello there'
       with uniform distribution"
    (let [results (repeatedly 100
```

```
                #(find-following-word
                  (get-sentences "test.txt") "hello there"))
        freqs (frequencies results)
        expected ["bob" "bill" "john"]]
    (should= 3 (count freqs))
    (should (every? #(contains? freqs %) expected))
    (doseq [name expected]
      (let [count (get freqs name)]
        ;; Check if count is reasonably close to 33 (100/3)
        ;; with some tolerance
        (should (> count 20))
        (should (< count 50)))))))
```

Despite a valiant effort by Grok, these tests do not pass. That should drive home just how important it is to overload a prompt with testing scenarios!

```
Failures:


  1) Markov returns 'there' when searching for 'hello'
     Expected: "there"
          got: "." (using =)


  2) Markov returns period when searching for 'are well'
     Expected: "."
          got: "i" (using =)


  3) Markov returns Bob, Bill, or John for
     'hello there' with uniform distribution
```

```
    Expected: 3
         got: 1 (using =)
```

Debugging the code is relatively simple; but can you debug the prompt? And even if you can “fix” the prompt, are you sure it will generate code and tests that pass if you apply the prompt more than once?

The fact that Grok wrote the tests and yet could not use them to infer the desired behavior exposes the limitation of the LLM approach. LLMs are statistical in nature; they create their responses through probability and not inference. It is true that many inferences are trained into the statistical model; but the evaluation of the prompt does not infer in real time. Inference is the primary operation of reason. If LLMs do not actively infer, then LLMs do not reason!

If you debug the code that Grok produced, you will find that Grok apparently ignored the provided definition of “word” and did not properly interpret the case insensitivity constraint. The fact that Grok got confused by a formal prompt overloaded with test scenarios is not encouraging. It seems obvious, at least to me, that without overloaded formality, the prospect of using AIs to write our systems is doomed to spectacular failure. After all, programming is formal, and humans require overloading.

Conclusion

In the end, I expect that the LLMs and AIs will improve to the point that long, formal specifications will be within their grasp. I also expect that we programmers will design formal and overloaded specification languages that the AIs can consume to produce complex systems.

And who will learn and apply these formalisms? Who will provide the ability to reason that the AIs lack? Programmers, of course. We programmers will become the lawyers of AI, responsible (as we always have been) to reason through and draw up the precise and accurate formal requirements and contracts for the applications our clients need.

Part II

Design

In this part of the book, we're going to raise our perspective slightly above coding issues and look at the principles of clean design. You'll still see plenty of code—but the level will continuously rise as each chapter goes by. We'll cover the basics of lower-level boundaries and systems, Kent Beck's rules of simple design, the SOLID principles, and the principles of component design.

Portions of the following chapters are edited and abridged excerpts from my books *Clean Craftsmanship* and *Clean Architecture*.

Chapter 18. Simple Design



Kent Beck

This chapter is an abridged and edited excerpt from the book Clean Craftsmanship*.*

It should be obvious, on the face of it, that the best design for a system is the simplest design that supports all the required features of that system while simultaneously affording the greatest flexibility for change. However, that leaves us to ponder the meaning of simplicity.[1] Simple does not mean easy. Simple means untangled; and untangling things is *hard*.

[1] In 2012, Rich Hickey gave a wonderful talk titled “Simplicity Made Easy.” I encourage you to listen to it.

What things get tangled in software systems? The most expensive and significant entanglements are those that convolve high-level policies with low-level details. You create terrible complexities when you conjoin SQL with HTML, or frameworks with core values, or the format of a report with the business rules that calculate the reported values. These entanglements are easy to write, but they make it hard to add new features, hard to fix bugs, and hard to improve and clean the design.

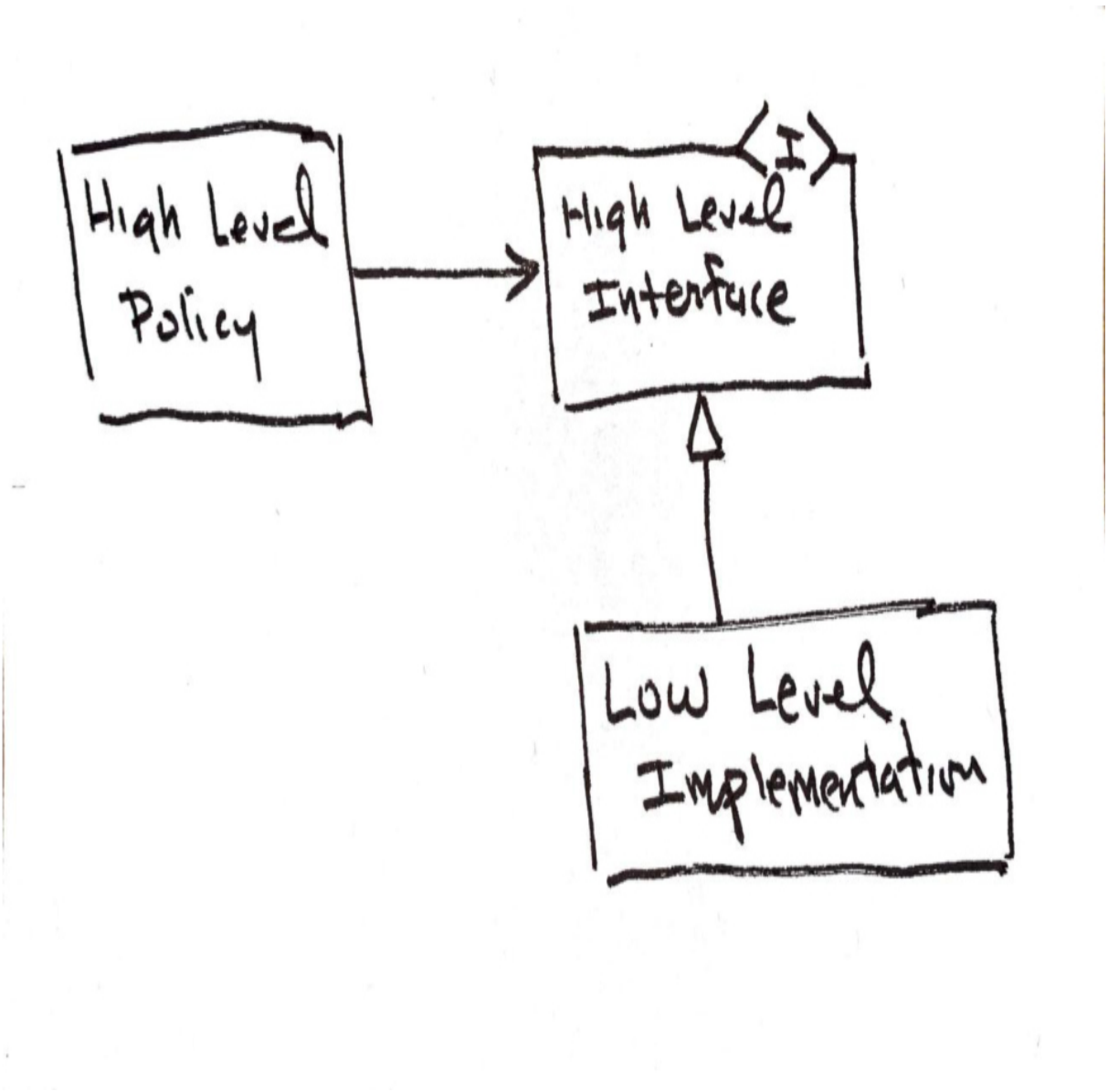
A simple design is a design in which high-level policies are ignorant of low-level details. Those high-level policies are sequestered and isolated from low-level details such that changes to the low-level details have no impact upon the high-level policies.[2]

[2] I write a great deal about this in [Clean Arch].

The primary means for creating this separation and isolation is *abstraction*. Abstraction is the amplification of the essential and the elimination of the irrelevant. High-level policies are essential, so they are amplified. Low-level details are irrelevant at that level, and so they are isolated and sequestered.

One of the physical means we employ for this abstraction is polymorphism. We arrange high-level policies to use

polymorphic interfaces to manage the low-level details. Then, we arrange the low-level details as implementations of those polymorphic interfaces. This keeps all source code dependencies pointing from low-level details to high-level policies and keeps high-level policies ignorant of the implementations of the low-level details. Low-level details can be changed without affecting the high-level policies. This is the essence of the Dependency Inversion Principle (DIP).



Pursuant to the goal of creating good software designs, we will first discuss YAGNI and then Kent Beck's four principles of simple design:

1. Covered by tests
2. Reveals intent
3. Minimizes duplication
4. Minimizes size

YAGNI

What if you aren't gonna need it*?*

In 1999, I was teaching an Extreme Programming course with Martin Fowler, Kent Beck, Ron Jeffries, and a host of others. The topic turned to the dangers of overdesign, and premature generalization. Someone wrote YAGNI on the whiteboard and said, "You aren't gonna need it." Kent Beck interrupted and said something to the effect of you might, in fact, need it; but you should ask yourself: "What if you aren't going to need it?"

That was the original question that YAGNI posed: Every time you thought to yourself "I'm going to need this hook," you then asked yourself what would happen if you didn't put the hook in. If the cost of leaving the hook out was tolerable, then you probably shouldn't put it in. If the cost of carrying the hook in the design year after year would be high while the odds that you'd eventually need that hook were low, you probably shouldn't put that hook in.

It is always wise to think of the future, and there are times when putting a particular hook in is a good idea. But be sure to count the costs and advantages on both sides of the

argument. Make sure you ask yourself the question: *What if I'm not gonna need it?*

This kind of discernment is one of the most important disciplines of good software design. If you have a good suite of tests and you are skilled at the discipline of refactoring, then the cost of adding a new feature and updating the design to support that new feature will very often be smaller than the cost of implementing and maintaining all the hooks you might need one day.

Covered by Tests

What is a good test coverage number: 80%? 90%? Many teams treat such numbers as laudable goals. I disagree. The only reasonable coverage goal is 100%.

What other number could possibly make sense? If you are happy with 80% coverage, it means you don't know if 20% of your code works. How could you possibly be happy with that? How could your customers be happy with that?

An Asymptotic Goal

You might complain that 100% is an unreachable goal. I might even agree with you. Achieving 100% line and branch coverage is no mean feat. It may, in fact, be impractical depending on the situation. But that does not mean that the coverage you currently have cannot be improved.

Think of 100% as an asymptotic goal. You may never reach it, but that's no excuse for not trying to get closer and closer with every check-in.

I have personally participated in projects that grew to many tens of thousands of lines of code while constantly keeping the code coverage percentage in the very high 90s.

Design?

But what does high code coverage have to do with simple design? Why is coverage the first rule?

Testable code is decoupled code.

In order to achieve high line and branch coverage of each individual part of the code, each of those parts must be made accessible to the test code. That means those parts must be so well decoupled from the rest of the code that they can be isolated and invoked from an individual test. Therefore, those tests are not only tests of behavior, they are also tests of decoupling. The act of writing isolated tests is an act of design, because the code being tested must be designed to be tested.

Tests don’t just drive you to create decoupled and robust designs. They also allow you to improve those designs with time. As we have discussed many times before in these pages, a trusted suite of tests vastly reduces the fear of change. If you have such a suite, and if that suite executes quickly, then you can improve the design of the code every time you find a better approach. When the requirements change in a way that the current design does not easily afford, the tests will allow you to fearlessly shift the design to better match those new requirements.

And this is why this rule is the first and most important rule of simple design. Without a suite of tests that covers the system, the other three rules become impractical, because those rules are best applied *after the fact*. Those other three rules are rules that involve refactoring. And refactoring is virtually impossible without a good, comprehensive suite of tests.

Maximize Expression

In the early decades of programming, the code we wrote could not reveal intent. Indeed, the very name “code” suggests that intent is obscured. Back in the late ’60s, code looked like this:

```
------


/ROUTINE TO TYPE A MESSAGE                    PAL8-V10D NO DATE   PAGE 1

                /ROUTINE TO TYPE A MESSAGE
        0200            *200
        7600            MONADR=7600
00200   7300    START,  CLA CLL         /CLEAR ACCUMULATOR AND LINK
00201   6046            TLS             /CLEAR TERMINAL FLAG
00202   1216            TAD BUFADR      /SET UP POINTER
00203   3217            DCA PNTR        /FOR GETTING CHARACTERS
00204   6041    NEXT,   TSF             /SKIP IF TERMINAL FLAG SET
00205   5204            JMP .-1         /NO: CHECK AGAIN
00206   1617            TAD I PNTR      /GET A CHARACTER
00207   6046            TLS             /PRINT A CHARACTER
00210   2217            ISZ PNTR        /DONE YET?
00211   7300            CLA CLL         /CLEAR ACCUMULATOR AND LINK
00212   1617            TAD I PNTR      /GET ANOTHER CHARACTER
00213   7640            SZA CLA         /JUMP ON ZERO AND CLEAR
00214   5204            JMP NEXT        /GET READY TO PRINT ANOTHER
00215   5631            JMP I MON       /RETURN TO MONITOR
00216   0220    BUFADR, BUFF            /BUFFER ADDRESS
00217   0220    PNTR,   BUFF            /POINTER
00220   0215    BUFF,   215;212;"H;"E;"L;"L;"O;"!;0
00221   0212
00222   0310
00223   0305
00224   0314
00225   0314
00226   0317
00227   0241
00230   0000
00231   7600    MON,    MONADR          /MONITOR ENTRY POINT
```

Notice the ubiquitous comments. These were absolutely necessary because the code itself revealed nothing at all about the intent of the program.

However, we do not work in the 1960s anymore. The languages that we use are *immensely* expressive. With the proper discipline, we can produce code that reads like “well-written prose.”

As an example of such code, consider this little bit of Java:

```
public class RentalCalculator {
  private List<Rental> rentals = new ArrayList<>();

  public void addRental(String title, int days) {
    rentals.add(new Rental(title, days));
  }

  public int getRentalFee() {
    int fee = 0;
    for (Rental rental : rentals)
      fee += rental.getFee();
    return fee;
  }

  public int getRenterPoints() {
    int points = 0;

    for (Rental rental : rentals)
      points += rental.getPoints();
    return points;
  }
```

```
}
```

If you were not a programmer on this project, you might not understand everything that is going on in this code. However, after even the most cursory glance, the basic intent of the designer is easy to identify. The names of the variables, functions, and types are deeply descriptive. The structure of the algorithm is easy to see. This code is expressive. This code is simple.

The Underlying Abstraction

Lest you think that expressivity is solely a matter of nice names for functions and variables, I should point out that there is another concern: the separation of levels and the exposition of the underlying abstraction.

A software system is expressive if each line of code, each function, and each module lives in a well-defined partition that clearly depicts the level of the code and its place in the overall abstraction.

You may have found that last sentence difficult to parse, so let me be a bit clearer by being much more long-winded.

Imagine an application that has a complex set of requirements. The example I like to use is a payroll system.

- Hourly employees are paid every Friday, based upon the time cards they have submitted. They are paid time-and-a-half for every hour they work after 40 hours in a week.
- Commissioned employees are paid on the first and third Friday of every month. They are paid a base salary plus a commission on the sales receipts they have submitted.

- Salaried employees are paid on the last day of the month. They are paid a fixed monthly salary.

It should not be hard for you to imagine a set of functions with a complex `switch` statement or `if`/`else` chain that captures these requirements. However, such a set of functions is likely to obscure the underlying abstraction. What is that underlying abstraction?

```
public List<Paycheck> run(Database db) {
  Calendar now = SystemTime.getCurrentDate();
  List<Paycheck> paychecks = new ArrayList<>();

  for (Employee e : db.getAllEmployees()) {
    if (e.isPayDay(now))
      paychecks.add(e.calculatePay());
  }

  return paychecks;
}
```

Notice that there is no mention of any of the hideous details that dominate the requirements. The underlying truth of this application is that we need to pay all employees on their payday. Separating the high-level policy from the low-level detail is the most fundamental part of making a design simple and expressive.

Tests: The Other Half of the Problem

When Kent Beck originally wrote this rule, he phrased it this way:

The system (code and tests) must communicate everything you want to communicate.

There is a reason he phrased it that way: No matter how expressive you make the production code, it cannot communicate the context in which it is used. That's the job of the tests.

Every test you write, especially if those tests are isolated and decoupled, is a demonstration of how the production code is intended to be used. Well-written tests are example use cases for the parts of the code that they test.

Thus, *taken together*, the code and the tests are an expression of what each element of the system does and how each element of the system should be used.

What does this have to do with design? Everything, of course. Because the primary goal we wish to achieve with our designs is to make it easy for other programmers to understand, improve, and upgrade our systems. And there is no better way to achieve that goal than to demonstrate the intended usage of the system's abstractions through a comprehensive suite of executable tests.

Minimize Duplication

As I have told you in several previous chapters, in the earliest days of software, there was no way to call one program from another. If you wanted to do something more than once, you had to duplicate the code. Programmers wrote code snippets into their notebooks so that they could copy them into their programs when needed. They called those snippets *subroutines*.

In the 1950s, it became possible for programs to call subroutines and have those subroutines return to the point of call. In those days, we had no source code editors at all.

We wrote our code, using #2 pencils, on preprinted coding forms. We therefore avoided most duplication because it was easier for us to create a single instance of a code snippet and call it. The calls took many CPU cycles, but the lack of duplication saved a lot of memory. It was often a good trade-off.

Then came the source code editors. And with those editors came copy/paste operations. Suddenly it became easy to copy a snippet of code from one routine and paste it into a new routine and then fiddle with it until it worked. With memory becoming cheaper and bigger, the cost of duplication became less of a concern. Thus, as the years went by, more and more systems exhibited massive amounts of duplication in the code.

Duplication is usually problematic. Two or more similar stretches of code will often need to be modified together. Finding those similar stretches is hard. Properly modifying them is even harder because they exist in different contexts. Thus, duplication leads to fragility.

In general, it is best to reduce similar stretches of code into a single instance by abstracting the code into a new function and providing it with appropriate arguments that communicate any differences in context.

Sometimes that strategy is not workable. For example, sometimes the duplication is in code that traverses a complex data structure. Many different parts of the system may wish to traverse that structure, and they will use the same looping and traversal code only to then operate on the data structure in the body of that code.

As the structure of the data changes over time, programmers will have to find all the duplications of the traversal code and properly update them. The more the traversal code is duplicated, the higher the risk of fragility.

The duplications of the traversal code can be eliminated by encapsulating it in once place, and using lambdas, Command objects, the Strategy pattern, or even the Template Method pattern[3] to pass the necessary operations into the traversal.

[3] [GOF95].

Accidental Duplication

Not all duplication should be eliminated. There are situations in which two stretches of code may be very similar, even identical, but will change for very different reasons.[4] I call this *accidental duplication*. Accidental duplicates should not be eliminated. The duplication should be allowed to persist. As the requirements change, the duplicates will evolve separately and the accidental duplication will dissolve.

[4] See “SRP: The Single Responsibility Principle” in Chapter 19, “The SOLID Principles.”

It should be clear that managing duplication is nontrivial. Identifying which duplications are real and which are accidental and then encapsulating and isolating the real duplications requires a significant amount of thought and care.

Determining the real duplications from the accidental duplications depends strongly on how well the code expresses its intent. Accidental duplications have divergent intent. Real duplications have convergent intent.

Encapsulating and isolating the real duplications, using abstraction, lambdas, and design patterns, involves a substantial amount of refactoring. And refactoring requires a good, solid suite of tests.

Therefore, eliminating duplication is third in the priority list of the rules of simple design. First come the tests, then the expression, and *then* the duplication.

Minimize Size

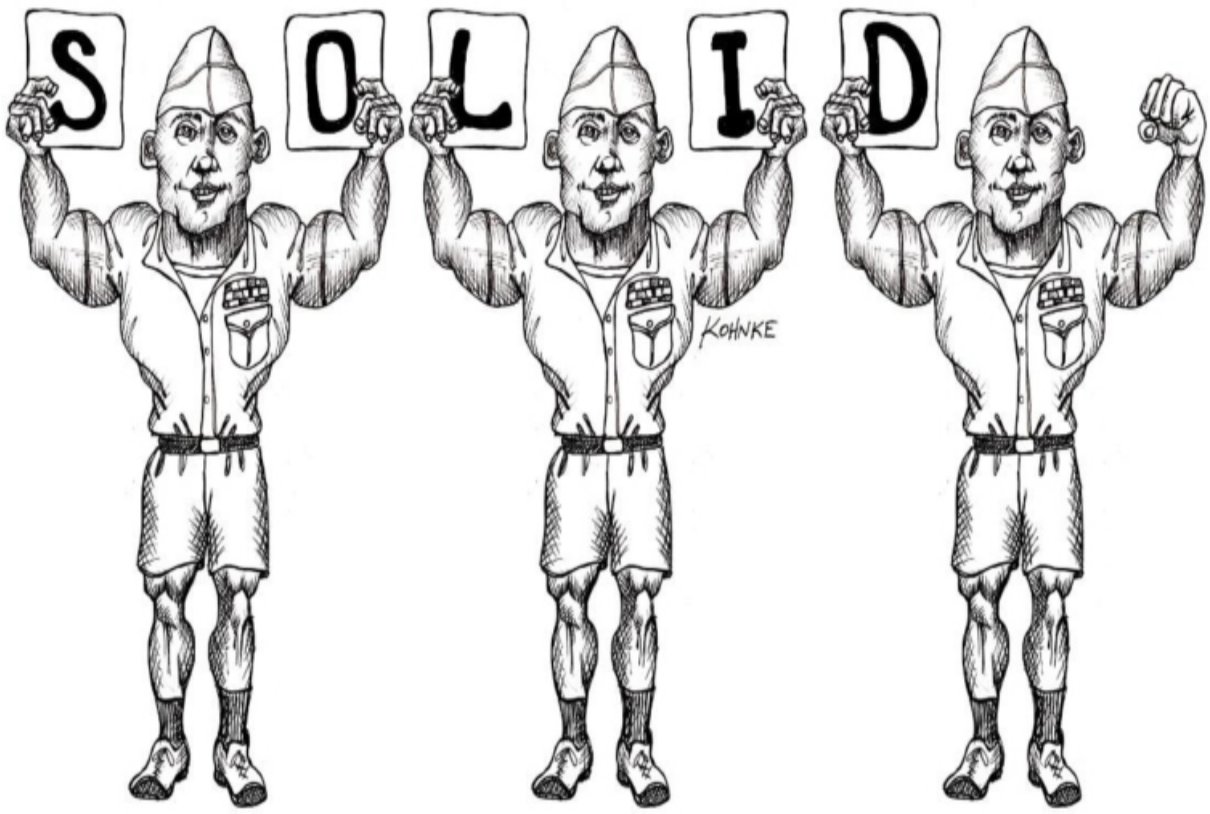
A simple design is composed of as few simple elements as possible without compromising test coverage and expression. Simple elements are small. The last rule of simple design suggests that after you've gotten all the tests to pass, and after you have made the code as expressive as possible, and after you have minimized duplication, *then* you should work to decrease the number of modules, classes, functions, and lines.

Simple Design

Many years back, Kent Beck and I were having a discussion on the principles of design. He said something that has always stuck with me. He said that if you followed these four things as diligently as possible, all other design principles would be satisfied—the principles of design can be reduced to coverage, expression, singularization, and reduction.

I don't know if this is true or not. I don't know if a perfectly covered, expressed, singularized, and reduced program necessarily conforms to the Open–Closed Principle (OCP) or the Single Responsibility Principle (SRP). What I am very sure of, however, is that knowing and studying the principles of good design and good architecture (e.g., the SOLID principles) makes it much easier to create well-partitioned and simple designs.

Chapter 19. The SOLID Principles



This chapter is a significantly abridged and edited set of excerpts from my book Clean Architecture*. It is here to give you a quick reference to the principles mentioned elsewhere in the book, and to provide you with directions for further study.*

Good software systems begin with clean code. If the bricks aren’t well made, the architecture of the building doesn’t matter much. On the other hand, you can make a substantial mess with well-made bricks. This is where the SOLID principles come in.

The SOLID principles tell us how to arrange our functions and data structures into classes, and how those classes should be interconnected. The use of the word *class* does not imply that these principles are only applicable to object-oriented software. A class is simply a coupled grouping of functions and data. Every software system has such groupings, whether they are called classes or not. The SOLID principles apply to those groupings.

The goal of the principles is the creation of mid-level software structures that

- Tolerate change
- Are easy to understand
- Are the basis of components that can be used in other software systems

The term *mid-level* refers to the fact that these principles are applied by programmers working at the module level. They are applied just above the level of functions and help define the kinds of software structures used within modules and components.

Just as it is possible to create a substantial mess with well-made bricks, it is also possible to create a systemwide mess with well-designed mid-level components. So, once we have covered the SOLID principles, we will move on to their counterparts in the component world, and then, in Part III, to the principles of high-level architecture.

The history of the SOLID principles is long. I began to assemble them in the late ’80s while debating software design principles with others on Usenet (an early social media platform). Over the years, the principles have shifted and changed. Some were deleted. Others were merged. Still others were added. The final grouping stabilized in the early 2000s, though I presented them in a different order.

Then, in 2004 or thereabouts, Michael Feathers sent me an email saying that if I rearranged the principles, their first letters of their initialisms would spell the word *solid*. And thus, the SOLID principles were born. The sections that follow describe each principle in detail. Here is the executive summary:

- **SRP**—The Single Responsibility Principle: The best structure for a software system is heavily influenced by the social structure of the organization that uses it, such that each software module has one, and only one, reason to change.
- **OCP**—The Open–Closed Principle: Bertrand Meyer made this famous in the ’80s. The gist is that in order for software systems to be easy to change, they must be designed to allow the behavior of those systems to be changed by adding new code, rather than by changing existing code.
- **LSP**—The Liskov Substitution Principle: This is Barbara Liskov’s famous definition of subtypes, from 1988. In short, this principle says that in order to build software systems from interchangeable parts, those parts must adhere to a contract that allows them to be substituted one for another.
- **ISP**—The Interface Segregation Principle: This principle advises software designers to avoid depending on things that they don’t use.

- **DIP**—The Dependency Inversion Principle: This principle says that the code that implements high-level policy should not depend on the code that implements low-level detail. Rather, low-level details should depend upon high-level policies.

These principles have been described in detail in many different publications[1] over the years. The sections that follow are not meant to replace those detailed writings. If you are not already familiar with these principles, what follows will be a good introduction, but insufficient for a deep understanding. You would be well advised to study them in the footnoted documents.

[1] For example, [PPP02], http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod, and https://en.wikipedia.org/wiki/SOLID_(object-oriented_design) (or just google SOLID).

SRP: The Single Responsibility Principle

Of all the SOLID principles, this might be the least well understood. That's likely because it has a particularly inappropriate name. It is too easy for programmers to hear the name and then assume that it means that every module should do just one thing.

There *is* a principle like that, as we have seen in previous chapters. It is true that a *function* should do one, and only one, thing. We use that principle when we are refactoring large functions into smaller functions. We use that principle at the lowest levels. But it is not one of the SOLID principles. It is not the SRP.

Historically, the SRP has been described this way:

A module should have one, and only one, reason to change.

Software systems are changed in order to satisfy users and stakeholders. So those users and stakeholders *are* the "reason to change" that the principle is talking about. Indeed, we can rephrase the principle to say this:

A module should be responsible to one, and only one, user or stakeholder.

Unfortunately, the words *user* and *stakeholder* aren't really the right words to use here. This is because there will likely be more than one user or stakeholder who wants the system changed in the same way. So we're really referring to a group of one or more people who require changes. We shall refer to that group as an *actor*.

Thus, the final version of the SRP is:

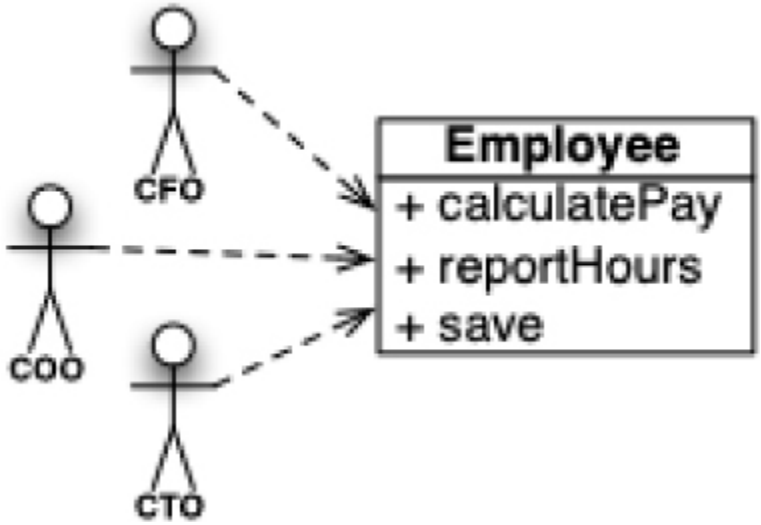
A module should be responsible to one, and only one, actor.

Now, what do we mean by the word *module*? A module is just a cohesive set of functions and data structures. That word *cohesive* implies the SRP. Cohesion is the force that binds together the code responsible to a single actor.

Perhaps the best way to understand this principle is by looking at one of the many symptoms that result from violating it.

Accidental Duplication

Consider the `Employee` class from a payroll application. It has three methods: `calculatePay()`, `reportHours()`, and `save()`.

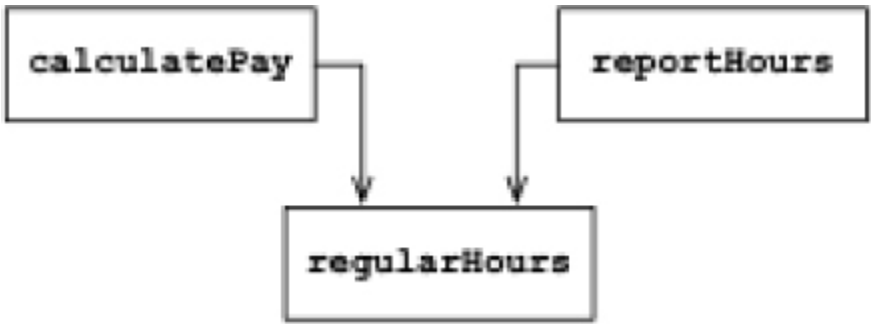


This class violates the SRP because those three methods are responsible to three very different actors.

- The `calculatePay()` method is specified by the accounting department, which reports up to the CFO.
- The `reportHours()` method is specified and used by human resources, which reports up to the COO.
- The `save()` method is specified by the database administrators (DBAs), who report up to the CTO.

By putting the source code for these three methods into a single `Employee` class, the developers have coupled each of these actors to the others. This coupling can cause the actions of the CFO's team to affect something that the COO's team depends upon.

For example, let's say that the `calculatePay()` function and the `reportHours()` function share a common algorithm for calculating nonovertime hours. Let's also say that the developers, who are careful not to duplicate code, put that algorithm into a function named `regularHours()`.



Now let’s say that the CFO’s team decides that the way nonovertime hours are calculated needs to be tweaked. But the COO’s team in HR does not want that particular tweak, because they use nonovertime hours for a different purpose.

A developer is tasked to make the change and sees the convenient `regularHours()` function called by the `calculatePay()` method. Unfortunately, that developer does not notice that the function is also called by the `reportHours()` function.

The developer makes the required change and carefully tests it. The CFO’s team validates that the new function works as desired, and the system is deployed.

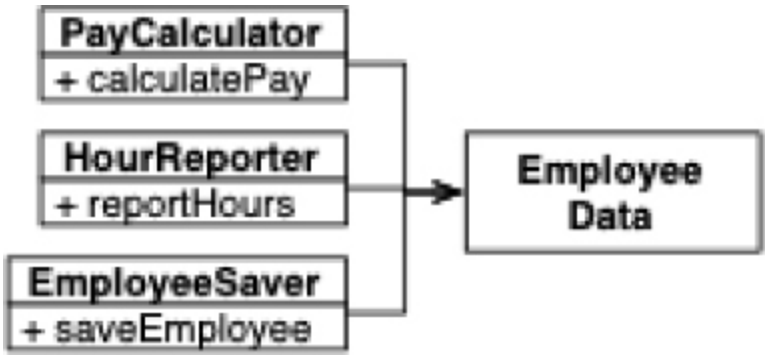
Of course, the COO’s team doesn’t know that this is happening. The COO’s team continues to use the reports generated by the `reportHours()` function—but now they contain incorrect numbers. Eventually the problem is discovered, and the COO is livid because the bad data has cost his budget millions of dollars.

We’ve all seen things like this happen. It occurs because we put code that different actors depend upon into close proximity. The SRP says to *separate the code that different actors depend upon*.

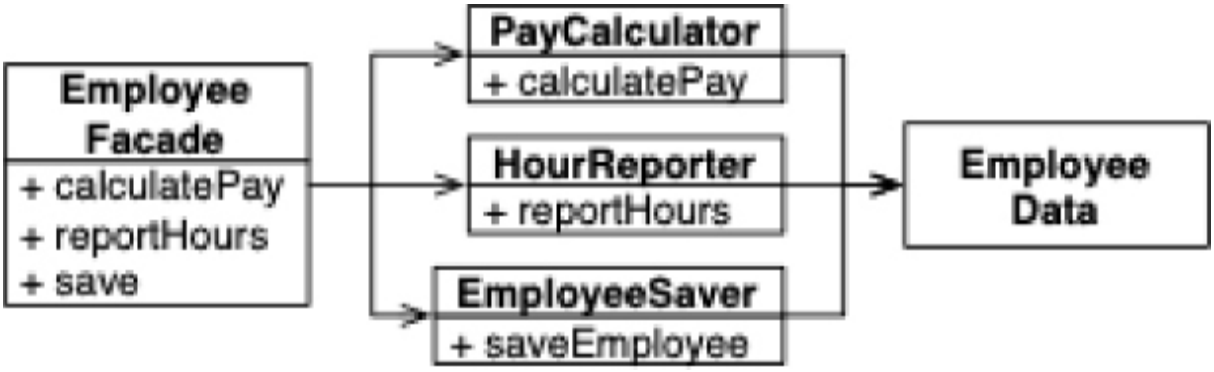
Solutions

There are many different solutions to this problem. Each moves the functions into different classes.

Perhaps the most obvious way to do this is to separate the data from the functions. The three classes share access to `EmployeeData`, which is a simple data structure with no methods. Each class holds only the source code necessary for its particular function. The three classes are not allowed to know about each other. Thus, any accidental duplication is avoided.



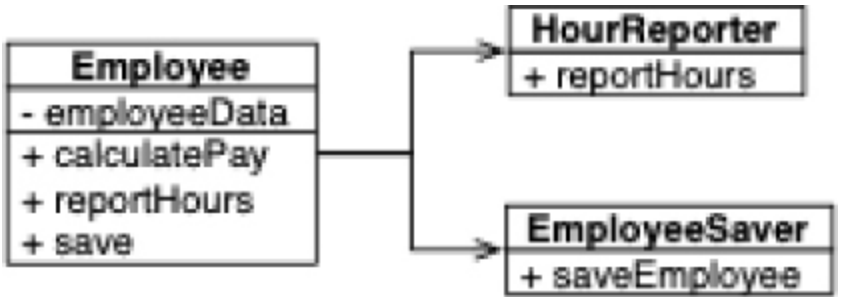
Of course, the downside of this solution is that the developers now have three classes that they have to instantiate and keep track of. A common solution to this dilemma is to use the Facade pattern.



The `EmployeeFacade` contains very little code. It is responsible for instantiating and delegating to the classes with the functions.

Some developers prefer to keep the most important business rules closer to the data. This can be done by keeping the most important method in the original `Employee`

class and then using that class as a facade for the lesser functions.



You might object to these solutions on the basis that every class would just contain one function. This is hardly the case. The number of functions required to calculate pay, generate a report, or save the data is likely to be large in each case. Each of those classes would likely have many other public methods, and many more private methods in them.

Higher Levels

The SRP is about functions and classes. But the principle reappears in a different form at two more levels.

As we shall see, at the level of components it becomes the Common Closure Principle. And at the architectural level, it becomes the axis of change responsible for the creation of architectural boundaries. We'll be studying all of this in the chapters to come.

OCP: The Open–Closed Principle



The OCP was coined in 1988 by Bertrand Meyer.[2] It says:

[2] [OOSC], p. 23.

A software artifact should be open for extension but closed for modification.

In other words, the behavior of a software artifact ought to be extendible, without having to modify that artifact.

This, of course, is the most fundamental reason that we study software design. When simple extensions to the requirements force massive changes to the software, the architects of that software system have engaged in a spectacular failure.

Most students of software design recognize the OCP as a principle that guides them in the design of classes and modules. But the principle takes on even greater significance when we consider the level of architectural components.

A Thought Experiment

A thought experiment will make this clear.

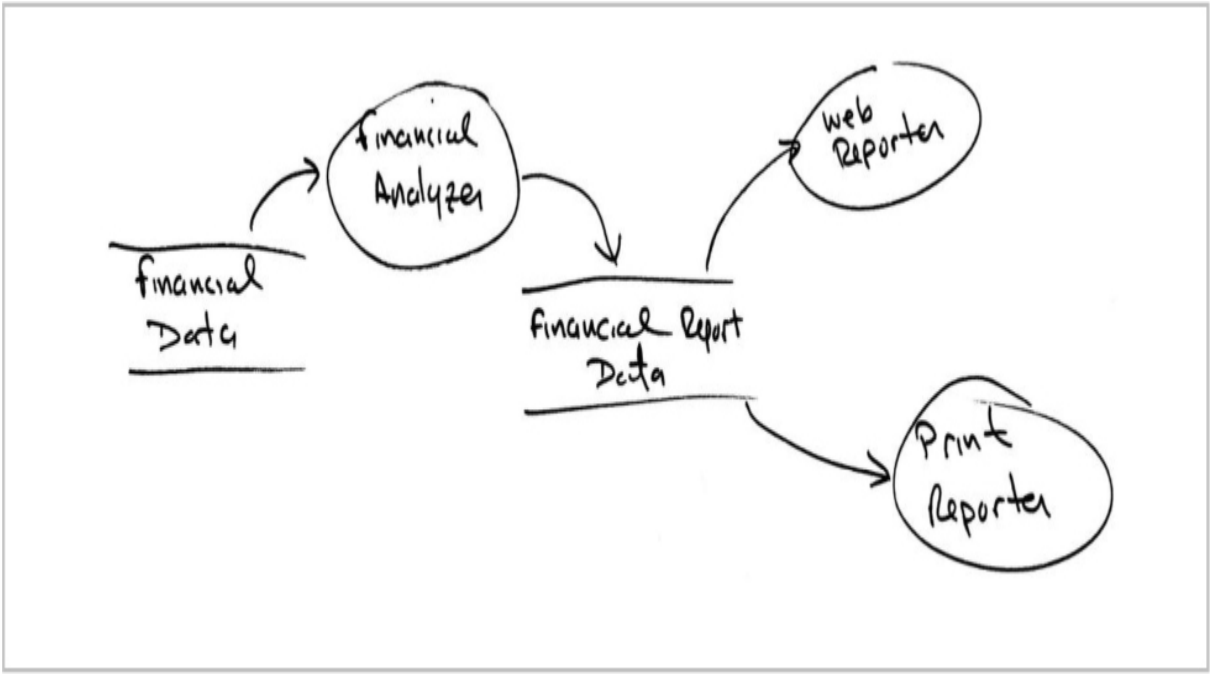
Imagine, for a moment, that we have a system that displays a financial summary on a web page. The data on the page is scrollable, and negative numbers are rendered in red.

Now imagine that the stakeholders ask that this same information be produced into a report to be printed on a black-and-white printer. The report should be properly paginated, with appropriate page headers, page footers, and column labels. Negative numbers should be surrounded by parentheses.

Clearly there will be some new code to write. But how much old code will have to change?

A good software architecture would reduce the amount of changed code to the barest minimum; ideally, zero.

How? By properly separating the things that change for different reasons (the SRP), and then organizing the dependencies between those things properly (the DIP).

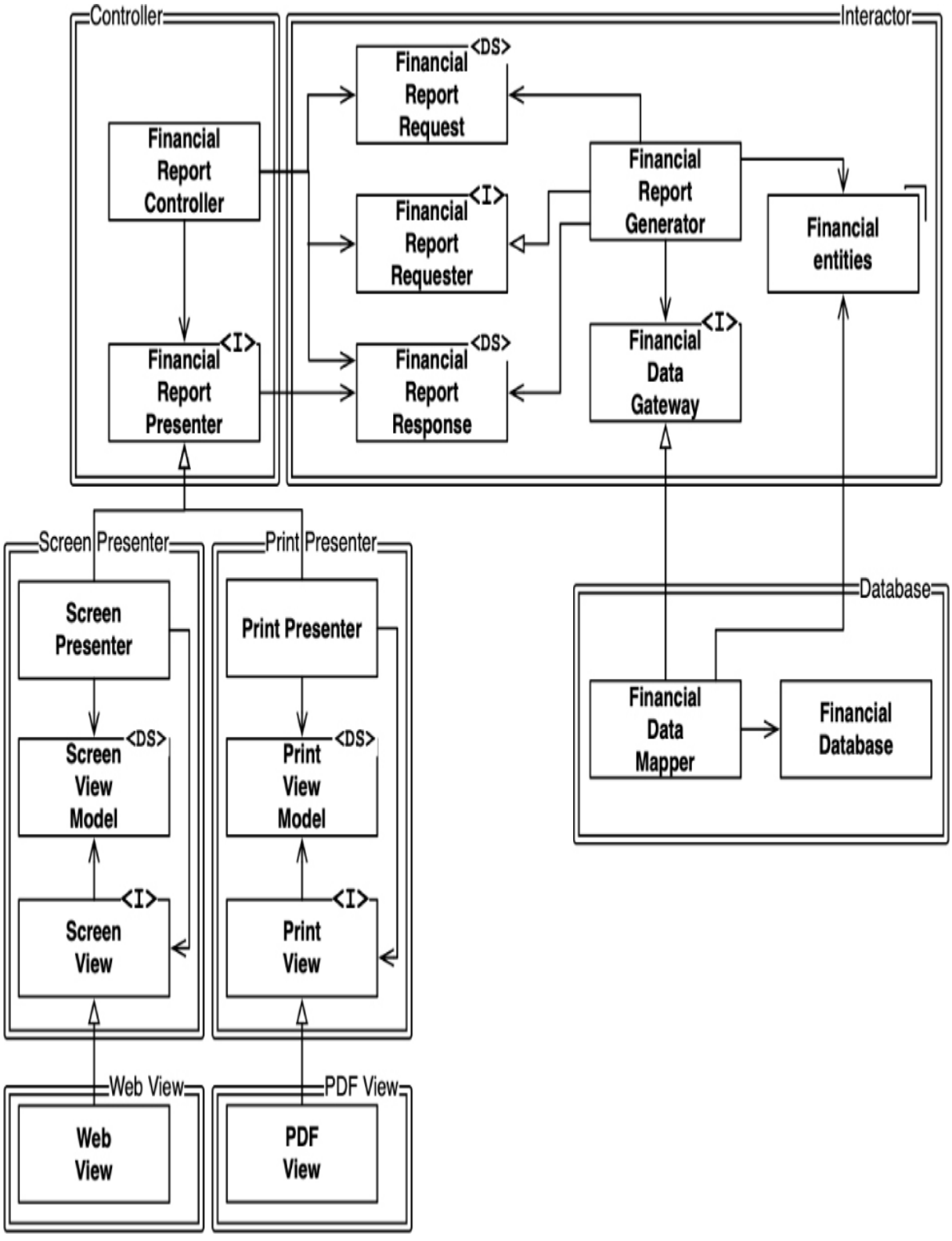


Applying the SRP, we might come up with the data-flow view above. Some analysis procedure inspects the financial data and produces reportable data, which is then formatted appropriately by the two reporter processes.

The essential insight here is that the report consists of two separate responsibilities: the calculation of the reported data and the presentation of that data into a web and printer form.

Having made this separation, we need to organize the source code dependencies in order to ensure that changes to one of those responsibilities do not cause changes in the other, and that the behavior can be extended without undo modification.

We accomplish this by partitioning the processes into classes, and separating those classes into components as shown by the double lines in the following diagram.



Future Bob: To be clear, I would never expect you to draw a diagram like this. I certainly don’t when

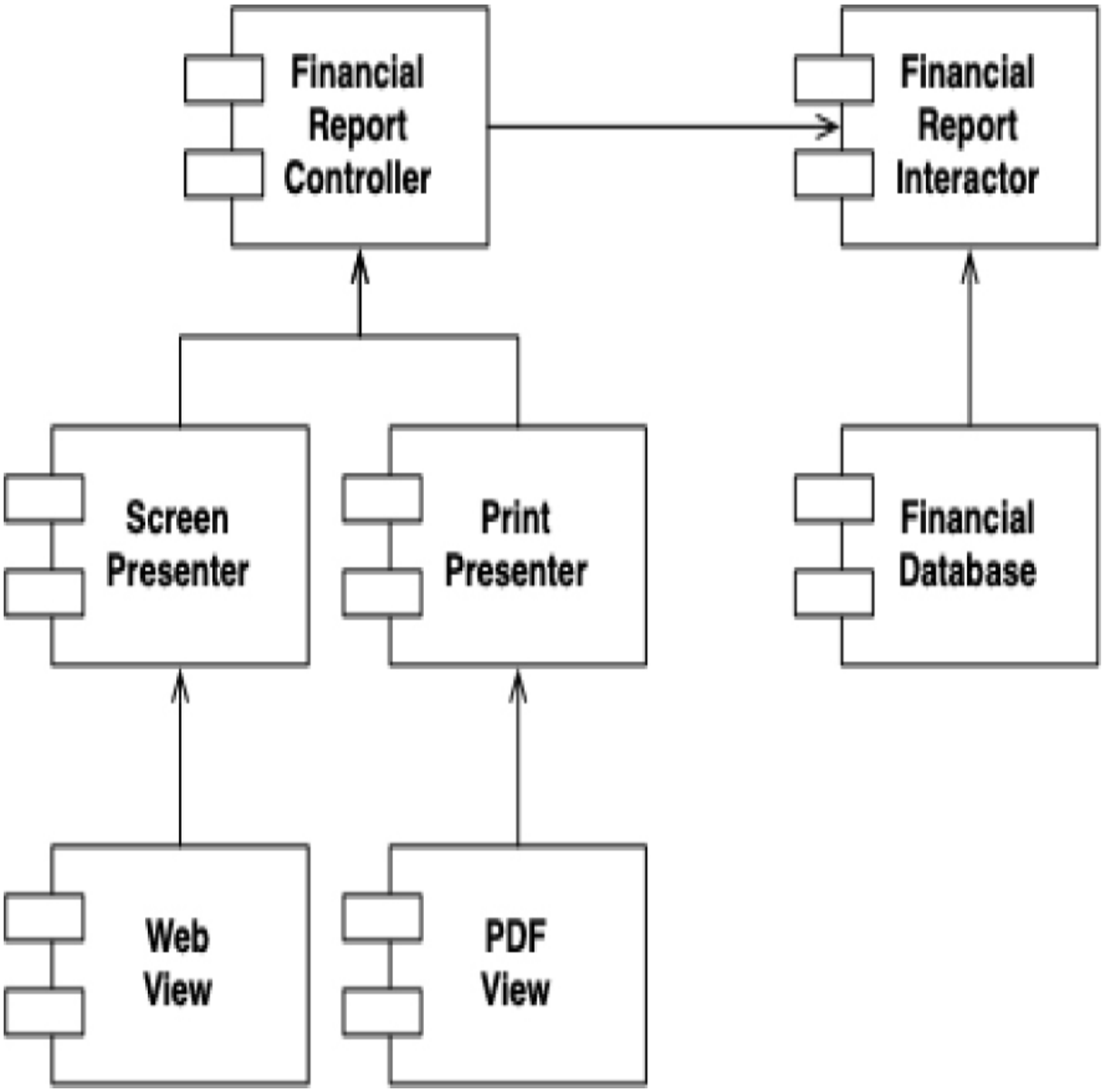
> I’m thinking through issues like this. I might draw the abbreviated view that you see below on a whiteboard in order for the team to discuss it. Most of the classes within those components would be so obvious to the team that drawing them would be superfluous. If any team member had a particular question, someone else on the team could scribble a portion of the diagram on the whiteboard or on a napkin.

The component at the upper left is the Controller. On the upper right we have the Interactor. On the lower right there is the Database. And on the lower left there are four components that represent the Presenters and the Views.

Classes marked with `<I>` are interfaces and with `<DS>` are data structures. Open arrowheads are *using* relationships. Closed arrowheads are *implements* or *inheritance* relationships.

The first thing to notice is that all the dependencies are *source code* dependencies. An arrow pointing from class A to class B means that the source code of class A mentions the name of class B; but class B mentions nothing about class A. So, in the diagram above, `FinancialDataMapper` knows about `FinancialDataGateway` with an implements relationship, but `FinancialDataGateway` knows nothing at all about `FinancialDataMapper`.

The next thing to notice is that each double line is crossed *in one direction only*. This means that all component relationships are unidirectional, as shown in the following diagram.



These arrows point toward the components that we want to protect from change.

Let me say that again. If component A should be protected from changes in component B, then component B should depend upon component A.

We want to protect the Controller from changes in the Presenters. We want to protect the Presenters from changes in the Views. We want to protect the Interactor from changes in—anything.

The Interactor is in the position that best conforms to the OCP. Changes to the Database, or the Controller, or the Presenters, or the Views will have no impact upon the Interactor.

Why should the Interactor hold such a privileged position? Because it contains the business rules. The Interactor contains the highest-level policies of the application. All the other components are dealing with peripheral concerns. The Interactor deals with the central concern.

Even though the Controller is peripheral to the Interactor, it is nevertheless central to the Presenters and Views. And while the Presenters might be peripheral to the Controller, they are central to the Views.

Notice how this creates a hierarchy of protection based upon the notion of “level.” Interactors are the highest-level concept, so they are the most protected. Views are among the lowest-level concepts, so they are the least protected. Presenters are higher level than Views, but lower level than the Controller or the Interactor.

This is how the OCP works at the architectural level. Architects separate functionality based on how, why, and when it changes, and then they organize that separated functionality into a hierarchy of components. Higher-level components in that hierarchy are protected from the changes made to lower-level components.

Directional Control

If you recoiled in horror from the class design above, look again. Much of the complexity in that diagram was to make sure that the dependencies between the components pointed in the correct direction.

For example, the `FinancialDataGateway` interface between the `FinancialReportGenerator` and the `FinancialDataMapper` exists in order to invert the dependency that would otherwise have pointed from the Interactor component to the Database component. The same is true of the `FinancialReportPresenter` interface and the two View interfaces.

Information Hiding

The `FinancialReportRequester` interface serves a different purpose. It is there to protect the `FinancialReportController` from knowing too much about the internals of the Interactor. If that interface were not there, then the Controller would have transitive dependencies upon the `FinancialEntities`.

Transitive dependencies are a violation of the general principle that software entities should not depend upon things they don't directly use. We'll see that principle covered later when we talk about the ISP and the CRP.

So, even though our first priority is to protect the Interactor from changes to the Controller; we also want to protect the Controller from changes to the Interactor by hiding the internals of the Interactor.

Conclusion

The OCP is one of the driving forces behind the design of systems. The goal is to make the system easy to extend without incurring a high impact of change. This is accomplished by partitioning the system into components and arranging them into a dependency hierarchy that protects higher-level components from changes in lower-level components.

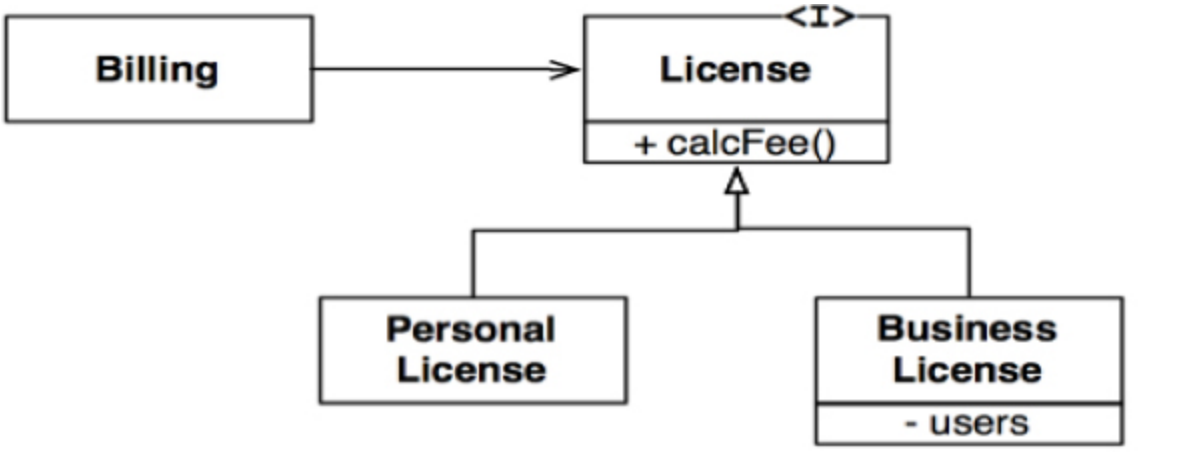
LSP: The Liskov Substitution Principle

In 1988, Barbara Liskov wrote the following as a way of defining subtypes.

What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.[3]

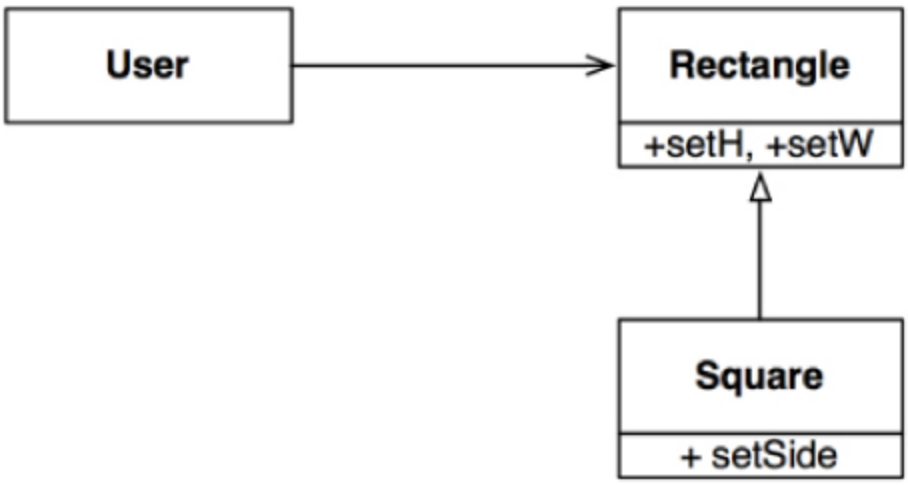
[3] Barbara Liskov, "Data Abstraction and Hierarchy," *ACM* SIGPLAN Notices 23, no. 5 (May 1988): 17–34.

For example, imagine that we have a class named `License`, as shown below. This class has a method named `calcFee()` that is called by the `Billing` application. There are two "subtypes" of `License`: `PersonalLicense` and `BusinessLicense`. They use different algorithms to calculate the license fee.



This design conforms to the LSP because the behavior of the `Billing` application does not depend, in any way, upon which of the two subtypes it uses. Both of the subtypes are substitutable for the `License` type.

The canonical example of a violation of the LSP is the famed `Square`/`Rectangle` problem.



In this example, `Square` is not a proper subtype of `Rectangle`, because the height and width of the `Rectangle` are independently mutable, but the height and width of the `Square` must change together. Since the `User` believes it is communicating with a `Rectangle`, it could easily get confused. The following code shows why.

```
Rectangle r = . . .
r.setW(5);
r.setH(2);
assert(r.area() == 10);
```

If the . . . produced a `Square`, then the assertion would fail.

The only way to defend against this kind of LSP violation is to add mechanisms to the `User` (such as an `if` statement) that detect whether the `Rectangle` is, in fact, a `Square`. Since the behavior of the `User` depends on the types it uses, those types are not substitutable.

Putting an `if` statement like that into the `User` violates the OCP. Thus, every violation of the LSP is a latent violation of the OCP.

LSP and Design

In the early years of the object-oriented revolution, we thought of this principle as a way to guide the use of inheritance, as shown in the previous section. However, over the years, we have morphed the LSP into a broader principle of software design that pertains to interfaces and implementations.

The interfaces in question can be of many forms. We might have a Java-style interface, implemented by several classes. Or we might have several Ruby classes that share the same method signatures. Or we might have a set of services that all respond to the same REST interface.

In all of these situations, and more, the LSP is applicable because there are users who depend upon well-defined interfaces and upon the substitutability of the implementations of those interfaces.

Taxi Aggregator

The best way to understand the LSP from a software design viewpoint is to look at what happens to the design of a system when the principle is violated.

Let’s assume we that we are building an aggregator for many taxi dispatch services. Customers use our website to find the most appropriate taxi to use, regardless of the taxi company. Once the customer decides, our system dispatches the taxi by using a RESTful service.

Now let’s assume that the URI for the RESTful dispatch service is part of the information contained in the driver database. Once our system has chosen a driver appropriate for the customer, it gets that URI from the driver record and then uses it to dispatch the driver.

For example, let’s say that Driver Bob has a dispatch URI that looks like this:

```
purplecab.com/driver/Bob
```

Our system will append the dispatch information onto this URI and send it with a `PUT`, as follows:

```
purplecab.com/driver/Bob
  /pickupAddress/24 Maple st.
  /pickupTime/1530
  /destination/ORD
```

Clearly this means that all the dispatch services for all the different companies must conform to the same REST interface. They must treat the `pickupAddress`, `pickupTime`, and `destination` fields identically.

Now let’s say the Acme taxi company hired some programmers who didn’t read the spec very carefully, and they abbreviated the destination field to just `dest`. And let’s say that Acme was the largest taxi company in our area, and Acme’s CEO’s ex-wife is our CEO’s new wife, and . . . Well, you get the picture. What would happen to the architecture of our system?

Clearly we’d need to add a special case. The dispatch request for any Acme driver would have to be constructed using a different set of rules from all the other drivers.

The simplest way to accomplish this would be to add an `if` statement to the module that constructed the dispatch command.

```
if (driver.getDispatchUri().startsWith("acme.com"))…
```

But, of course, no developer worth their salt would allow such a construction to exist in the system. Putting the word *acme* into the code itself creates an opportunity for all kinds of horrible and mysterious errors, not to mention security breaches.

For example, what if Acme got very successful and bought the Purple Taxi company? What if they maintained the brand and the website but unified all their systems? Would we have to add another `if` statement for the word *purple*?

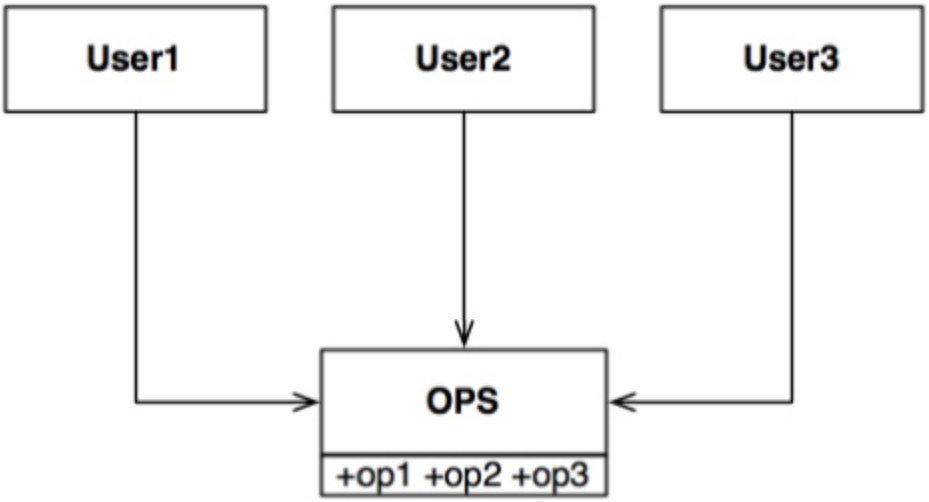
So, our architect would have to insulate the system from bugs like this by creating some kind of dispatch command creation module that was driven by a configuration database keyed by the dispatch URI. The configuration data might look something like this:

URI	Dispatch format
Acme.com	/pickupAddress/%s/pickupTime/%s/dest/%s
.	/pickupAddress/%s/pickupTime/%s/destination/%s

The point of all this is that a simple violation of the LSP for a RESTful service can cause our architecture to be polluted with a significant number of extra mechanisms.

ISP: The Interface Segregation Principle

The ISP derives its name from the following:

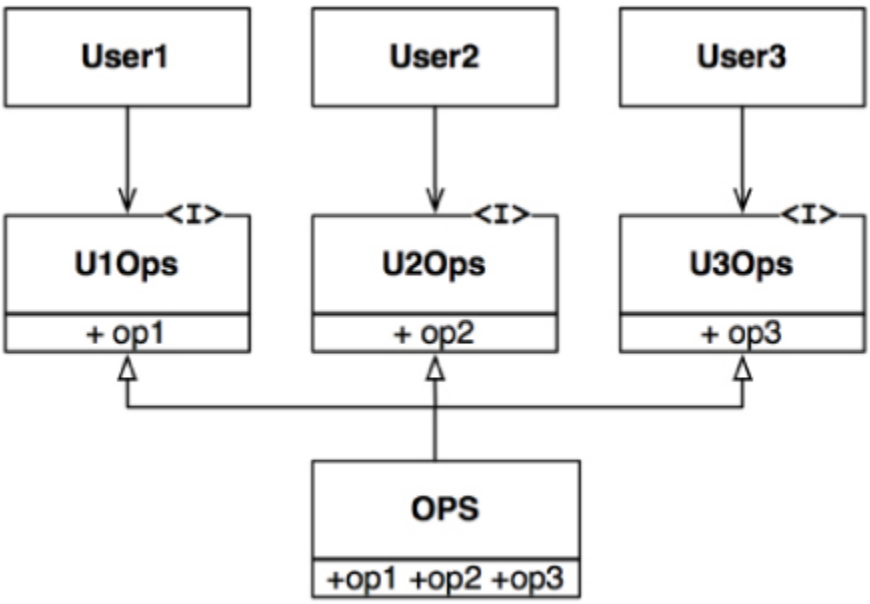


In the situation above, there are several users that use the operations of the `OPS` class. Let's assume that `User1` only uses `op1`, `User2` only uses `op2`, and `User3` only uses `op3`.

Now imagine that `OPS` is a class written in a language like C++.[4] Clearly, in that case, the source code of `User1` will inadvertently depend upon `op2` and `op3`, even though it doesn't call them. This dependence means that a change to the source code of `op2` in `OPS` will force `User1` to be recompiled and redeployed, even though nothing that it cared about has actually changed.

[4] The situation in Java and C# is a bit more complicated. The compiler does not recompile modules based solely upon dates. Rather, it looks for changed declarations.

This can be resolved by segregating the operations into interfaces, as shown below.



Again, if we imagine that this is implemented in a statically typed language like C++, then the source code of `User1` will depend upon `U1Ops` and `op1`, but it will not depend upon `OPS`. Thus, a change to `OPS` that `User1` does not care about will not cause `User1` to be recompiled and redeployed.

ISP and Language

Statically typed languages like C++ force programmers to create declarations that users must `#include`. It is these included declarations in the source code that create the source code dependencies that can force recompilation and redeployment.

In languages like Java and C#, the dependencies are softer but not absent. The compilers follow the dependency tree using individual declarations rather than source files. So recompilation and redeployment are only necessary if declarations change.

In dynamically typed languages like Ruby and Python, interfaces don’t exist in source code. Rather, they are inferred at runtime. Thus, there are no source code

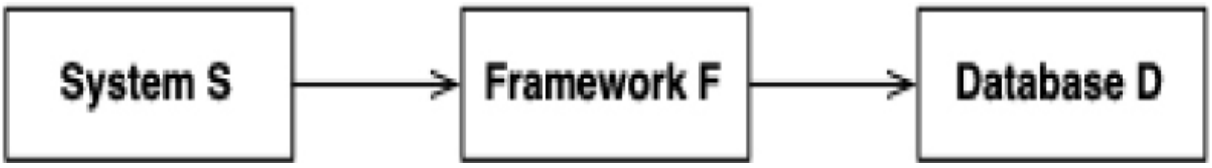
dependencies nor any interface declarations to force recompilation and redeployment. This is one reason that dynamically typed languages create systems that are more flexible and less coupled than statically typed languages.

This fact could lead you to conclude that the ISP is a language issue and not a design issue.

ISP and Design

However, if you take a step back and look at the root motivations of the ISP, you can see that there is a deeper concern lurking there. In general, it is harmful to depend upon modules that contain more than you need. This is obviously true for source code dependencies that can force unnecessary recompilation and redeployment. However, it is also true at a much higher level.

Consider, for example, a developer working on a system, S. He wants to include a certain framework, F, in the system. Now suppose that the authors of F have bound it to a particular database, D. So S depends upon F, which depends upon D.



Now suppose that D contains features that F does not use and that therefore S does not care about. Changes to those features within D may well force the redeployment of F, and therefore S. Worse, a failure of one of the features within D may cause failures in F and S.

The lesson here is that depending upon something that carries baggage that you don’t need can cause you troubles that you didn’t expect.

The bottom line is:

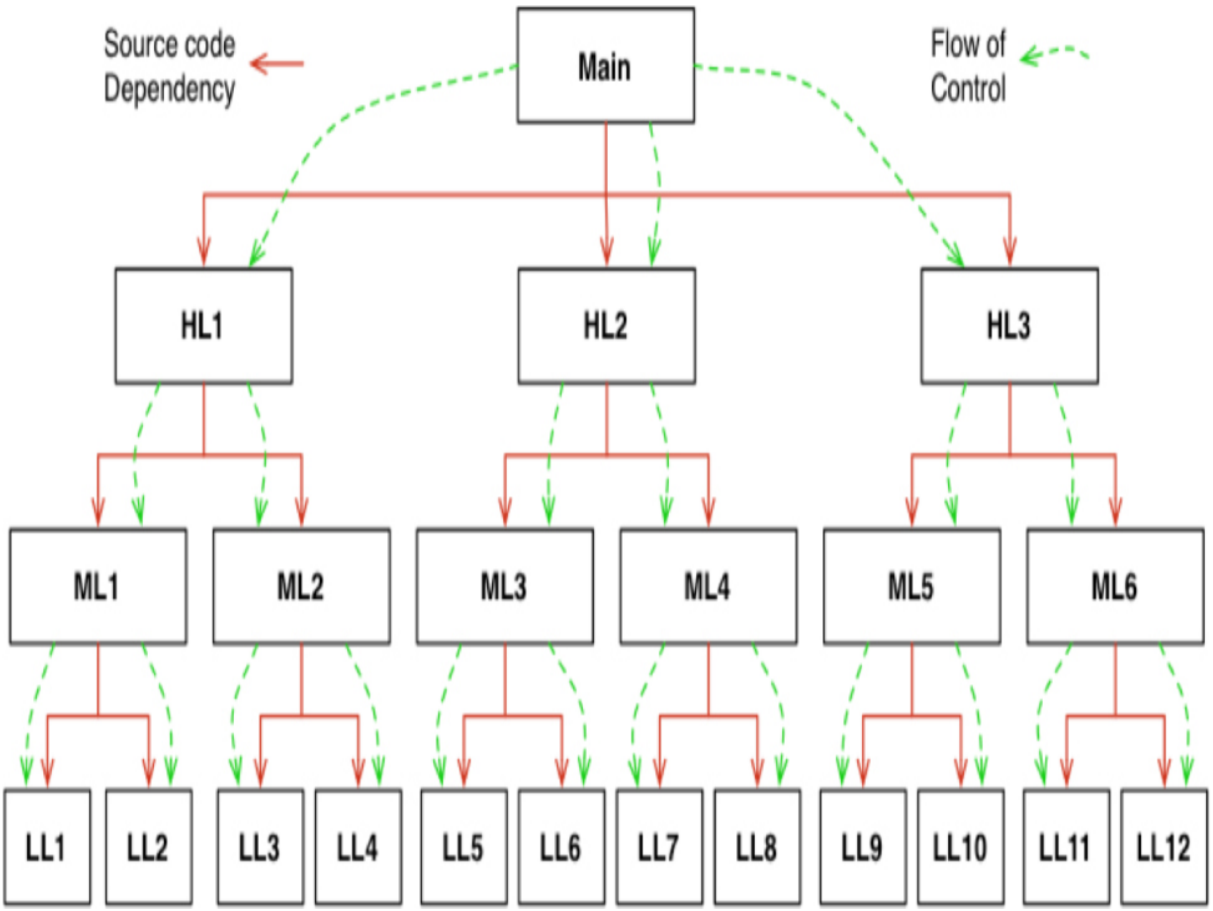
Don’t depend on things you don’t need.

We’ll see more about that when we discuss the Common Reuse Principle in Chapter 20, “Component Principles.”

DIP: The Dependency Inversion Principle

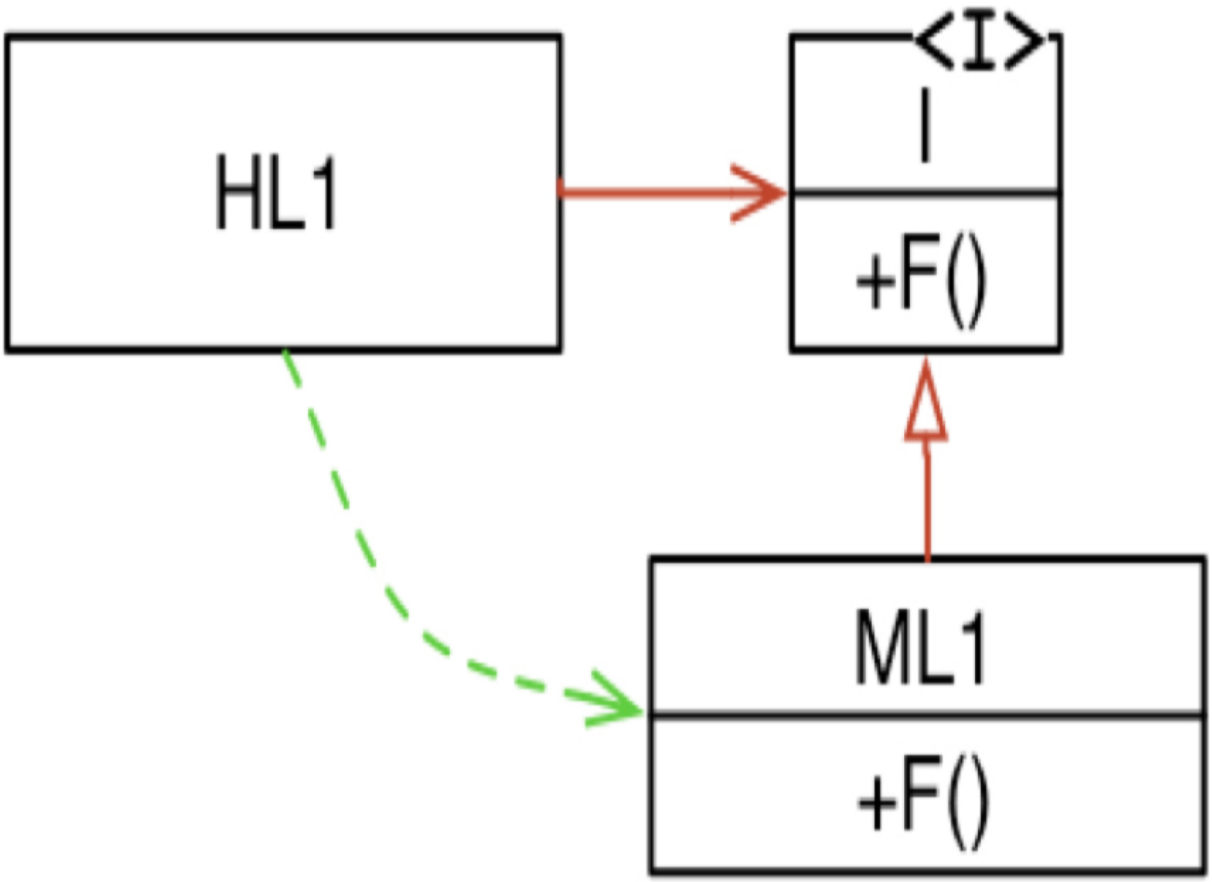
> *“It’s better to depend on something you control than on something you don’t control, lest it end up controlling you.”*
>
> —James Grenning

In the ’60s and ’70s, virtually all programs were built according to the following pattern.



The source code dependencies followed the flow of control, without exception. High-level modules depended upon low-level modules because the high-level modules *called* the low-level modules.

OO gave us a different option: polymorphic interfaces.



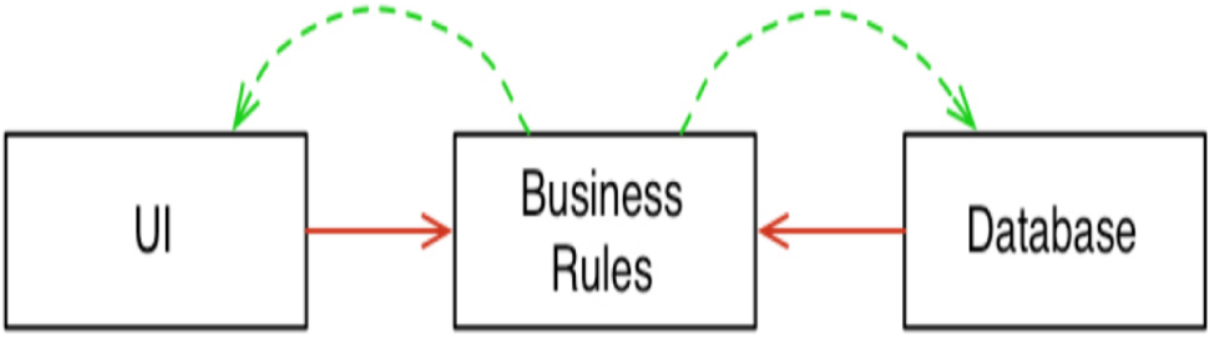
The insertion of an interface into the calling path inverts the source code dependency against the flow of control. In the above diagram, the high-level module `HL1` calls the `f` function in the lower-level module `ML1`; but it does so through the interface `I`. The source code dependency from `ML1` to `I` points against the flow of control.

The ability to invert dependencies in this way gives the software designer absolute control over every source code dependency in the system. If any are problematic (and they often are), they can be quickly inverted by inserting an interface.

This ability is power!



It allows us to create plug-in architectures that allow high-level policies, like the business rules, to be independent of low-level details, like the UI and the database.



In the above diagram, all source code dependencies (solid arrows) point from lower-level modules to higher-level modules. Both `UI` and `Database` depend upon and implement interfaces within `BusinessRules`. In other words, the concrete details depend upon abstractions. This tells us that the most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions.

In a statically typed language, like Java, this means that the `use`, `import`, or `include` statements should refer only to source modules containing interfaces, abstract classes, or some

other kind of abstract declaration. Nothing concrete should be depended upon.

The same rule applies for dynamically typed languages, like Ruby or Python. Source code dependencies should not refer to concrete modules. However, in these languages, it is a bit harder to define what a concrete module is. In particular, it is any module in which the functions being called are implemented.

Treating this as a hard-and-fast rule is unrealistic. There are many concrete facilities that software systems must depend upon. For example, the `String` class in Java is concrete. It would be unrealistic to try to force it to be abstract. The source code dependency upon the concrete `java.lang.string` cannot, and should not, be avoided.

On the other hand, the `String` class is very stable. Changes to that class are very rare and tightly controlled. Programmers and architects do not have to worry about frequent and capricious changes to `String`.

And so we tend to ignore the stable background of operating system and platform facilities when it comes to the DIP. We tolerate those concrete dependencies because we know we can rely upon them not to change.

It is the volatile concrete elements of our system that we want to avoid depending upon. It is those modules that we are actively developing and that are undergoing frequent change.

Stable Abstractions

Every change to an abstract interface corresponds to a change to its concrete implementations. On the other hand, changes to concrete implementations do not always, or even

usually, require changes to the interfaces that they implement. Therefore, interfaces are generally less volatile than implementations.

Indeed, good software designers and architects work hard to reduce the volatility of interfaces. They try to find ways to add functionality to implementations without making changes to the interfaces. This is Software Design 101.

The implication, therefore, is that stable software designs are those that avoid depending on volatile concretions, and favor the use of stable abstract interfaces.

This implication boils down to a set of very specific coding practices.

- **Don’t refer to volatile concrete classes.** Refer to abstract interfaces instead. This rule applies in all languages, whether statically or dynamically typed. It also puts severe constraints on the creation of objects and generally enforces the use of Abstract Factories.
- **Don’t derive from volatile concrete classes.** This is a corollary to the previous rule, but it bears special mention. In statically typed languages, inheritance is the strongest and most rigid of all the source code relationships, and it should be used with great care. In dynamically typed languages, inheritance is less of a problem; but it is still a dependency, and caution is always the wisest choice.
- **Don’t override concrete functions.** Concrete functions often require source code dependencies. When you override those functions, you do not eliminate those dependencies—indeed, you *inherit* them. To manage those dependencies you should make the function abstract and create multiple implementations.

- **Never mention the name of anything concrete and volatile.** This is really just a restatement of the principle itself.

While these are written as rules, they are actually more like warnings. A pragmatic programmer will violate this principle frequently. But a wise programmer will consider the warnings first.

Slavishly obeying this principle can lead to an explosion of interfaces that no one really needs. On the other hand, blithely ignoring this principle can cause your software to become tangled and rigid.

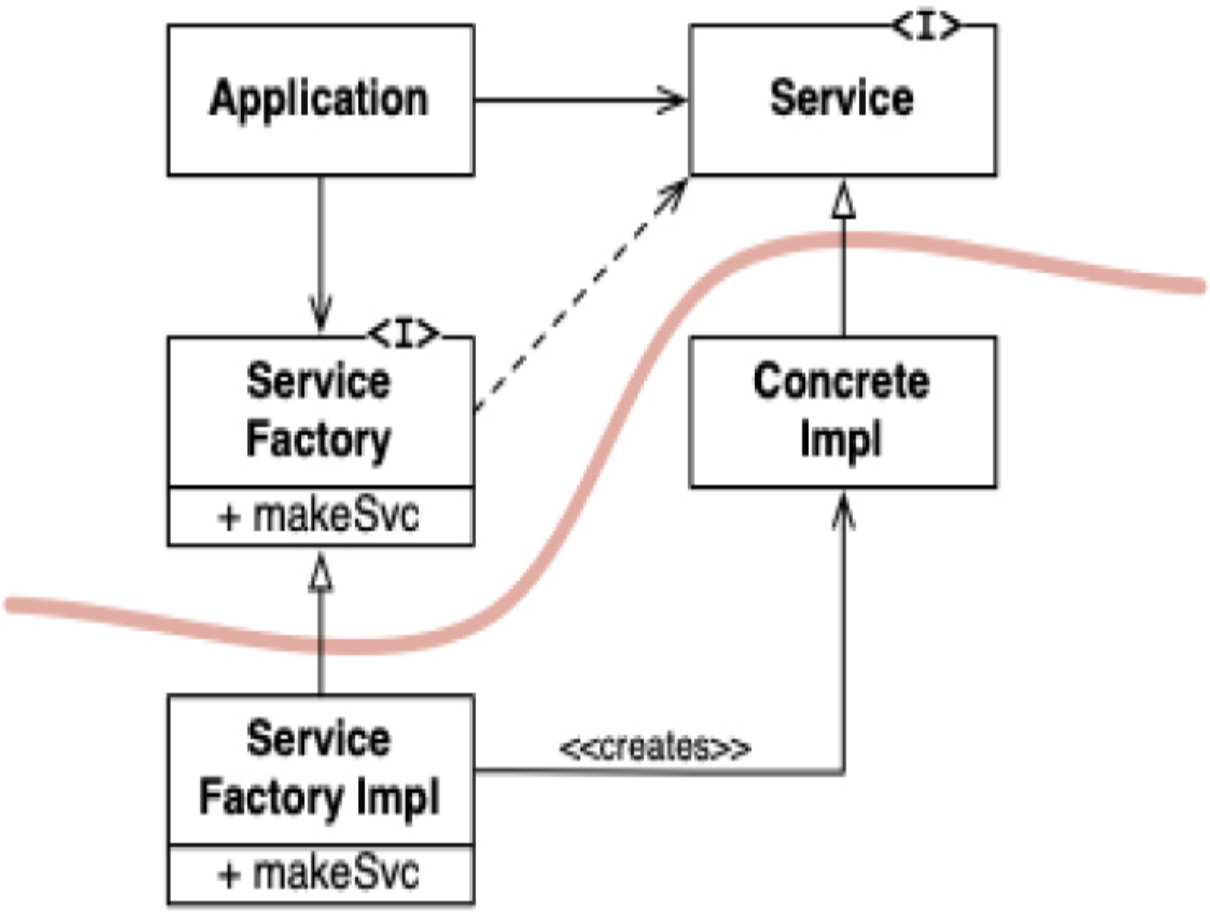
Finally, wise programmers will conform to this principle *gradually*, as the system evolves and the needs arise. With a test suite that they trust with their lives, they know that when interfaces need to be added in order to invert dependencies, the changes will be relatively easy to make and free of risk.

Factories

In order to comply with the DIP, the creation of volatile concrete objects may require special handling. This is because, in virtually all languages, the creation of an object requires a source code dependency upon the concrete definition of that object.

In most OO language, like Java, we would use an Abstract Factory[5] to eliminate this undesirable dependency.

[5] [GOF95].



The preceding diagram shows the structure. The `Application` uses the `ConcreteImpl` through the `Service` interface. However, the `Application` must somehow create instances of the `ConcreteImpl`. To achieve this without creating a source code dependency upon the `ConcreteImpl`, the `Application` calls the `makeSvc` method of the `ServiceFactory` interface. This method is implemented by the `ServiceFactoryImpl` class that derives from `ServiceFactory`. That implementation instantiates the `ConcreteImpl` and returns it as a `Service`.

Note the curved line. This is an architectural boundary. It separates the abstract from the concrete. All source code dependencies cross that line pointing in the same direction: toward the abstract side.

The line divides the system into two components: one abstract and the other concrete. The abstract component

contains all the high-level business rules of the application. The concrete component contains all the implementation details that those business rules manipulate.

Note that the flow of control crosses the line in the opposite direction of the source code dependencies. The source code dependencies are inverted against the flow of control. That is why we refer to this principle as *dependency inversion*.

Concrete Components

The concrete component in the above diagram contains a single dependency that violates the DIP. This is typical. DIP violations cannot be entirely removed; but they can be gathered into a small number of concrete components and kept separate from the rest of the system.

Most systems will contain at least one such concrete component, often called `main`, because it contains the `main`[6] function. In the above case, the `main` function would instantiate the `ServiceFactoryImpl` and place that instance in a global variable of type `ServiceFactory`. The `Application` would then access the factory through that global.

[6] That is, the function that is invoked by the operating system when the application is first started up.

As we proceed in this book and cover higher-level architectural principles, the DIP will show up again and again. It will be the most visible organizing principle in our architecture diagrams. The curved line in the diagram above will become the architectural boundaries in later chapters. The way the dependencies cross that line in one direction and toward more abstract entities will become a new rule that we will call the Dependency Rule.

Chapter 20. Component Principles



If the SOLID principles tell us how to arrange the bricks into walls and rooms, then the component principles tell us how to arrange the rooms into buildings. Large software systems, like large buildings, are built out of smaller components.

In this chapter, we will discuss what software components are, what elements they should be composed of, and how they should be composed together into systems.

This chapter is an overview. The principles are described in detail in [PPP02] and again in [Clean Arch].

Components

Components are the units of deployment. They are the smallest entities that can be deployed as part of a system. In Java, they are `jar` files. In Ruby, they

are `gem` files. In .NET, they are DLLs. In compiled languages, they are aggregations of binary files. In interpreted languages, they are aggregations of source files. In all languages, they are the granule of deployment.

Components can be linked together into a single executable. Or they can be aggregated together into a single archive such as a *.war* file. Or they can be independently deployed as separate, dynamically loaded plug-ins, such as *.jar* or *.dll* or *.exe* files. However, regardless of how they are eventually deployed, well-designed components always retain the ability to be *independently deployable* and therefore independently developable.

A Brief History of Components

In the early years of software development, programmers controlled the memory location and layout of their programs. One of the first lines of code in a program would have been the origin statement, which declared the address at which the program was to be loaded. Consider this simple PDP-8 program. It consists of a subroutine named `GETSTR`, which inputs a string from the keyboard and saves it in a buffer. It also has a little unit test program to exercise `GETSTR`.

```
        *200
        TLS
START,  CLA
        TAD BUFR
        JMS GETSTR
        CLA
        TAD BUFR
        JMS PUTSTR
        JMP START

BUFR,   3000

GETSTR, 0
        DCA PTR
NXTCH,  KSF
        JMP .-1
        KRB
        DCA I PTR
        TAD I PTR
        AND K177
```

```
          ISZ PTR
          TAD MCR
          SZA
          JMP NXTCH

K177,     177
MCR,      -15
```

Note the `*200` at the start of this program. This tells the compiler to generate code that will be loaded at address 200_8.[1]

[1] For example, [PPP02], http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod, and https://en.wikipedia.org/wiki/SOLID_(object-oriented_design) (or just google SOLID).

This is a foreign concept for most programmers today. They don't usually have to think about where a program is loaded in the memory of the computer. But in the early days, this was one of the first decisions a programmer needed to make. In those days, programs were not *relocatable*.

How did you access a library function in those days? The code above shows you how we did it. We included the source code of the library functions with our application code and compiled them all as a single program.[2] Libraries were kept in source, not in binary.

[2] My first employer kept several dozen decks of the subroutine library source code on a shelf. When you wrote a new program, you simply grabbed one of those decks and slapped it onto the end of your deck.

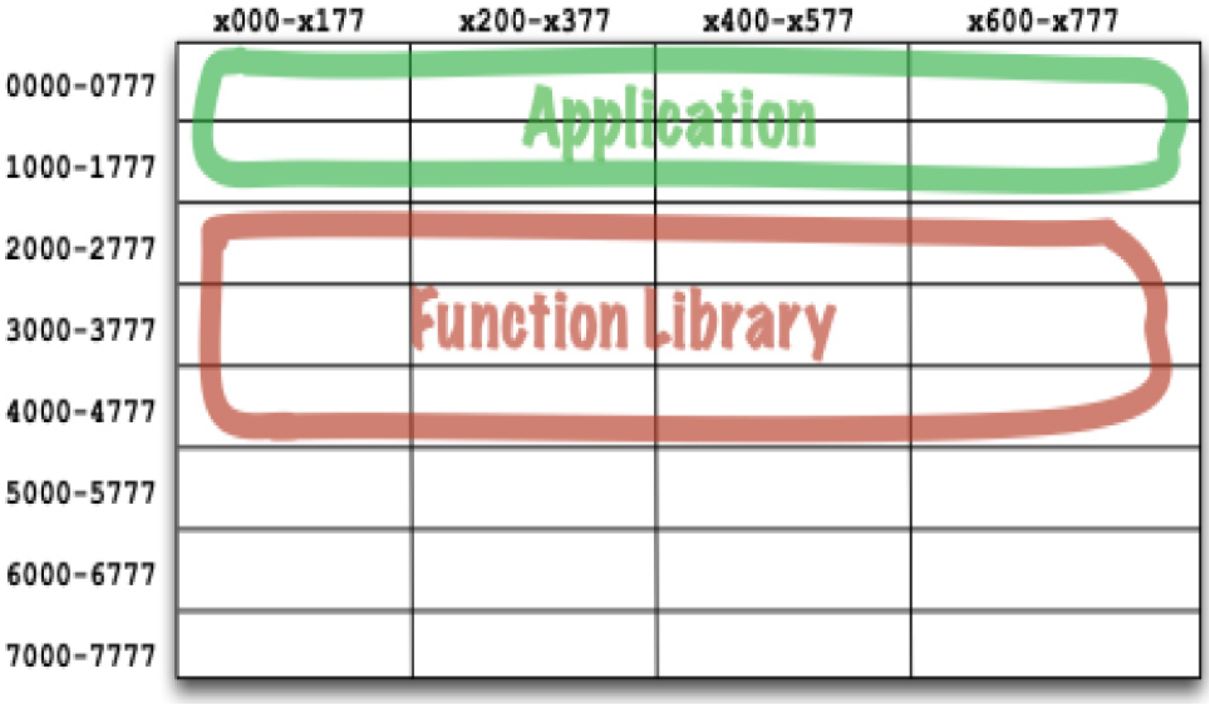
The problem with this was that devices in those days were slow, and memory was expensive and therefore limited. Compilers needed to make several passes over the source code, but memory was too limited to keep all the source code resident. So the compiler had to read in the source code several times using those slow devices.

This took a long time; and the larger your function library was, the longer the compiler took. Compiling a large program could take hours.[3] In order to shorten the compile times, we separated the source code of the function library from the applications. We compiled the function library separately and loaded the binary at a known address; say, 2000_8. We created a symbol table for the function library and compiled that with our application code. When we wanted to run an application, we would load the binary function library,[4] and then load the application.

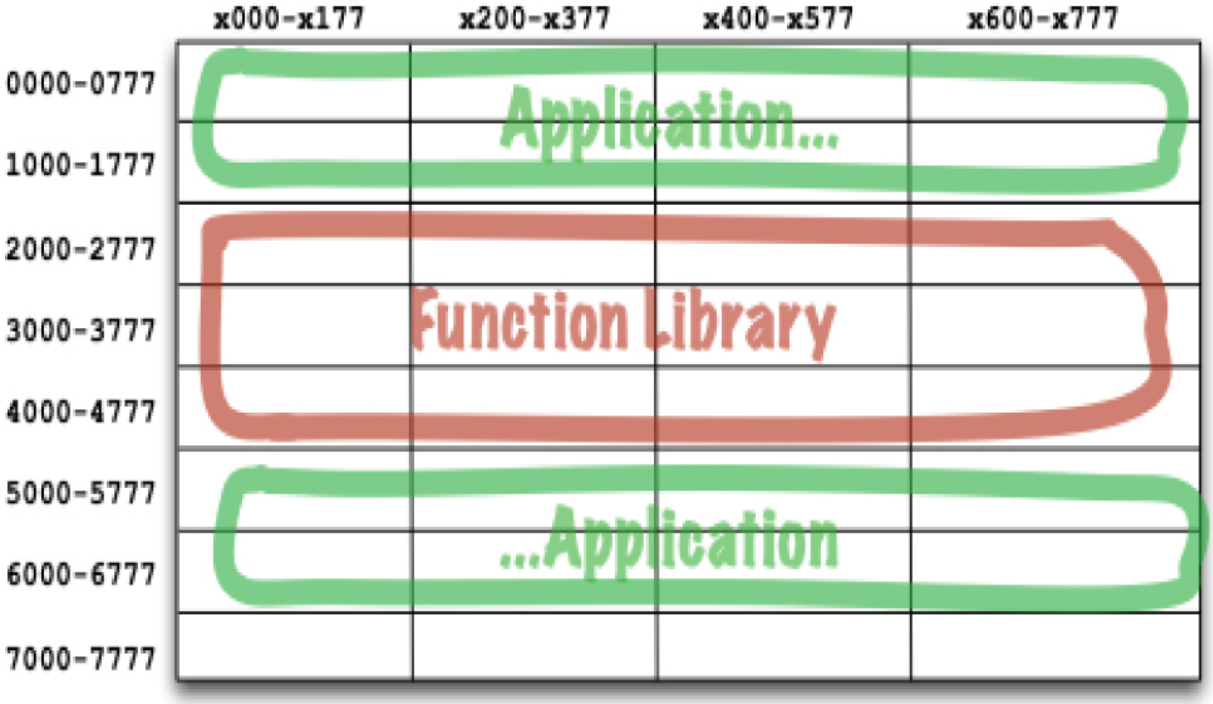
[3] This video, which I uploaded to YouTube in 2022, shows the process: https://www.youtube.com/playlist?list=PLINHgpdx-_Cq0blmYF-IWVO8U2XERZuit.

[4] Actually, most of those old machines used core memory, which did not get erased when you powered the computer down. So we often just left the function library loaded for days at a time.

Memory looked like this:



This worked fine so long as the application could fit between 0000_8 and 1777_8. But soon applications grew to be larger than the space allotted. So then, programmers had to split their applications into two address segments, jumping around the function library.



It should be clear that this was not a sustainable situation. As we added more functions to the function library, it exceeded its bounds, and we had to allocate more space for it near 7000_8. And this fragmentation of programs and libraries necessarily continued as computer memory grew.

Clearly, something had to be done.

Relocatability

The solution was relocatable binaries. The idea was very simple. We changed the compiler to output binary code based at address zero. The loader would be told where to load the relocatable code, and the loader would add the load address to all the addresses in the binary code. This was a little more complicated than it sounds, but we found ways to make it work.

Now we could tell the loader where to load the function library and where to load the application. In fact, the loader would accept several binary inputs and simply load them in memory one right after the other, relocating them as it loaded them. This allowed us to load only those functions that we needed.

The compiler was also changed to emit the names of the functions as metadata in the relocatable binary. If a program called a library function, the compiler would emit that name as an external reference. If a program defined a function, the compiler would emit that name as an external definition. Then, the loader could *link* the external references to the external definitions once it had determined where it had loaded those definitions.

And the linking loader was born.

Linkers

The linking loader allowed us to divide our programs onto segments that can be separately compiled and loaded. This worked great when we had relatively small programs being linked with relatively small libraries. However, in the late ’60s and early ’70s, we got more ambitious, and programs got a lot bigger.

Eventually, the linking loaders were too slow to tolerate. This was because our function libraries were stored on slow devices such a magnetic tape. Even the disks back then were quite slow. Using these relatively slow devices, the linking loaders had to read dozens, if not hundreds, of binary libraries in order to resolve the external references. As programs grew larger and larger, and more library functions accumulated in our libraries, a linking loader could take well over an hour just to load the program.

So the loader and the linker were separated into two phases. We took the slow part, the part that did that linking, and we put it into a separate application called the *linker*. The output of the linker was a linked relocatable

that a relocating loader could load very quickly. This allowed us to prepare an executable using the slow linker; but then we could load it quickly, anytime we wanted to.

Then came the ’80s. We were working in C or some other high-level language. As our ambitions grew, so did our programs. Programs containing hundreds of thousands of lines of code were not unusual.

We compiled our source modules from *.c* files into *.o* files, and fed them into the linker to create executable files that could be quickly loaded. Compiling each individual module was relatively fast; but compiling all the modules took a bit of time. And then the linker would take the most time. Turnaround had again grown to an hour or more in many cases.

It seemed we were doomed to endlessly chase our tail. Throughout the ’60s, ’70s, and ’80s, all the changes we had made to speed up our workflow were thwarted by our ambitions and the size of the programs we wrote. We could not seem to escape from the hourlong turnaround times. Loading time remained fast; but compile-link times were the bottleneck.

We were, of course, experiencing Murphy’s law of program size:

Programs will grow to fill all available compile and link time.

But Murphy was not the only contender in town. Along came Moore,[5] and in the late ’80s, the two battled it out. Moore won that battle. Disks started to shrink and get exponentially faster. Computer memory started to get so ridiculously cheap that much of the data on disk could be cached in RAM. Computer clock rates increased from 1MHz to 100MHz.

[5] Moore’s law: Computer speed, memory, and density double every 18 months. This law held from the 1950s to 2000; but then stopped cold.

By the mid-’90s, the time spent linking had begun to shrink faster than our ambitions could make programs grow. In many cases, link time dropped to a matter of *seconds*. For small jobs, the idea of a linking loader became feasible again.

This was the era of ActiveX, shared libraries, and the beginnings of *.jar* and *.dll* files. Computers and devices had gotten so fast that we could once again do the linking at load time. We could link together several *.jar* files or several shared libraries in a matter of seconds, and execute the resulting program.

And so the component plug-in architecture was born.

Today, we routinely ship *.jar* files or DLLs or shared libraries as plug-ins to existing applications. If you want to create a mod to *Minecraft*, for example, you simply include your custom *.jar* files in a certain folder. If you want to plug *ReSharper* into *Visual Studio*, you simply include the appropriate DLLs.

These dynamically linked files, which can be plugged together at runtime, are the software components of our architectures. It has taken 50 years, but we have arrived at a place where component plug-in architecture can be the casual default as opposed to the herculean effort it once was.

Component Cohesion

Which classes belong in which components? This is an important decision, and it requires guidance from good software engineering principles. Unfortunately, over the years, this decision has been made in an ad hoc manner based almost entirely on context.

In this section, we will discuss the three principles of component cohesion:

- **REP:** The Reuse/Release Equivalence Principle
- **CCP:** The Common Closure Principle
- **CRP:** The Common Reuse Principle

The Reuse/Release Equivalence Principle (REP)

The granule of reuse is the granule of release.

The past decade has seen the rise of a menagerie of module management tools, such as Maven, Leiningen, the `tools.deps` library, and RVM. These tools have grown in importance because, during that time, a vast number of reusable components and component libraries have been created. We are now living in the age of software reuse—a fulfillment of one of the oldest promises of OO.

The REP is a principle that seems obvious, in hindsight. People who want to reuse software components cannot, and will not, do so unless those components are tracked through a release process and are given release numbers.

This is not simply because, without release numbers, there would be no way to ensure that all the reused components are compatible with each other. It is also because software developers need to know when new releases are coming and what changes those new releases will bring.

It is not uncommon for developers to be alerted about a new release and decide, based on the changes made in that release, to continue to use the old release instead. Therefore, the release process must produce the appropriate notifications and release documentation so that users can make informed decisions about when and whether to integrate.

From a software design and architecture point of view, this principle means that the classes and modules that are formed into a component must belong

to a cohesive group. The component cannot simply consist of a random hodgepodge of classes and modules. There must be some overarching theme or purpose that those modules all share.

Of course, this should be obvious. However, there is another way to look at this that is perhaps not quite so obvious. Classes and modules that are grouped together into a component should be *releasable* together. The fact that they share the same version number and the same release tracking and are included under the same release documentation should make sense both to the author and the users.

Clearly this is weak advice. Saying that something should “make sense” is just a way of waving your hands in the air and trying to sound authoritative. The advice is weak because it is hard to precisely explain the glue that holds the classes and modules together into a single component. Weak though the advice may be, the principle itself is important, because violations are easy to detect—they don’t “make sense.” If you violate the REP, your users will know, and will not be impressed with your architectural skills.

The weakness of this principle is more than compensated for by the strength of the next two. Indeed, the next two principles strongly constrain this principle.

The Common Closure Principle (CCP)

Gather into components those classes that change for the same reasons and at the same times. Separate into different components those classes that change for different reasons or at different times.

This is the Single Responsibility Principle (SRP) restated for components. Just as the SRP says that a *class* should not contain multiples reasons to change, this principle says that a component should not have multiple reasons to change.

For most applications, maintainability is more important that reusability. If the code in an application must change, you would rather the changes occur all in one component than being distributed through many components. If changes are focused into a single component, then we need only redeploy the one changed component. Other components that don’t depend upon the changed component do not need to be revalidated or redeployed.

The CCP prompts us to gather together in one place all the classes that are likely to change for the same reasons. If two classes are so tightly bound, either physically or conceptually, that they always change together, then they belong in the same component. This minimizes the workload related to releasing, revalidating, and redeploying the software.

This principle is closely associated with the Open–Closed Principle (OCP). For it is “closure” in the OCP sense of the word that this principle is dealing with. The OCP states that classes should be closed for modification but open for extension. But 100% closure is not attainable. Closure must be strategic. We design our classes such that they are closed to the most common kinds of changes that we expect or have experienced.

The CCP amplifies this by gathering together into the same component those classes that are closed to the same types of changes. Thus, when a change in requirements comes along, that change has a good chance of being restricted to a minimal number of components.

Similarity with SRP

As I stated above, the CCP is the component form of the SRP. The SRP tells us to separate methods into different classes if they change for different reasons. The CCP tells us to separate classes into different components if they change for different reasons.

Different Times?

The caption at the start of this section implies that a component should not have elements that change at different times, even though they might change for the same reasons. Some changes are very high priority. Other changes are lower priority. Some changes are volatile with a high frequency. Some have a much lower volatility. We’d like to avoid components that have both high- and low-volatility elements.

The Common Reuse Principle (CRP)

Don’t force users of a component to depend on things they don’t need.

This is yet another principle that helps us decide which classes and modules should be placed into a component. It states that classes and modules that tend to be reused together belong in the same component.

Classes are seldom reused in isolation. Generally, reusable classes collaborate with other classes that are part of the reusable abstraction. The CRP states that these classes belong together in the same component. In such a component, we would expect to see classes that have lots of dependencies upon each other.

A simple example might be a container class and its associated iterators. These classes are reused together because they are tightly coupled to each other. Thus, they ought to be in the same component.

But the CRP tells us more than just what classes to put together into a component. It also tells us what classes not to keep together in a component.

When one component uses another, a dependency is created between the components. It may be that the using component only uses one module within the *used* component; but that doesn't weaken the dependency at all. The using component still depends upon the used component.

Because of that dependency, every time the used component is changed, the using component will likely need corresponding changes. Even if no changes are necessary to the using component, it will likely still need to be recompiled, revalidated, and redeployed. This may even be true if the change to the used component is something that the using component doesn't care about.

Thus, we want to make sure that when we depend upon a component, we depend upon the majority of classes in that component. To say this another way, we want to make sure that the classes that we put into a component are inseparable; that it is unlikely that we can depend upon some and not the others. Otherwise, we will be redeploying more components than is necessary, and wasting significant effort.

Therefore, the CRP tells us more about what classes shouldn't be together than what classes should be together. The CRP says that classes that are not tightly bound to each other should not be in the same component.

Relation to ISP

The CRP is the generic version of the ISP. The ISP advises us not to depend upon classes that have methods we don't use. The CRP advises us not to depend upon components that have modules we don't use. All of this advice can be reduced to a single sound bite:

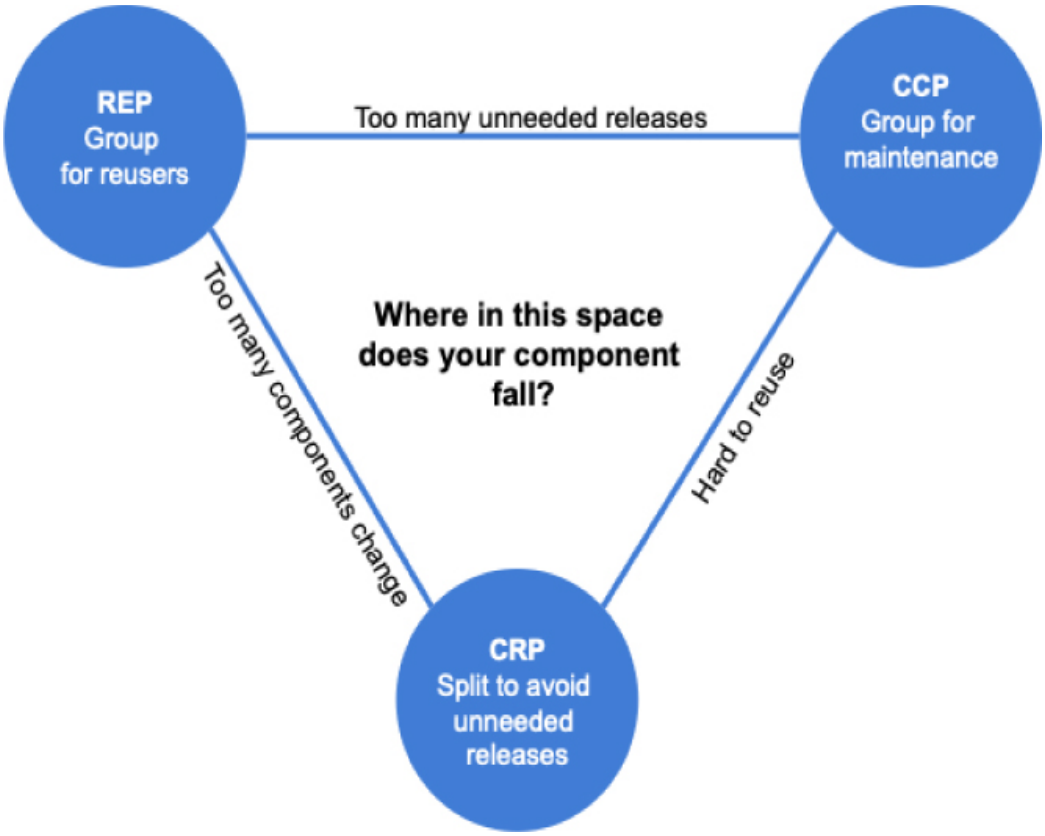
Don't depend on things you don't need.

The Tension Diagram for Component Cohesion

You may have already realized that the three cohesion principles tend to fight each other. The REP and CCP are inclusive principles. They both tend to make components larger. The CCP is an exclusive principle, driving components to be smaller. It is the tension between these principles that we seek to resolve.

Below is a tension diagram[6] that shows how the three principles of cohesion interact with each other. The edges of the diagram describe the *cost* of abandoning the principle on the opposite vertex.

[6] Thanks to Tim Ottinger for this idea.



So, a programmer who focuses on the REP and the CRP will find that too many components are impacted when simple changes are made. An architect who focuses too strongly on the CCP and the REP will cause too many unneeded releases to be made.

The goal is to find a position in that tension triangle that meets the *current* concerns of the development team, and shift that position as the system evolves over time. For example, early in the development of a project, the CCP is much more important than the REP, because developability is more important than reuse.

Generally, projects tend to start on the right-hand side of the triangle, where the only sacrifice is reuse. As the project matures and other projects begin to draw from it, the project will slide over to the left.

This means that the component structure of a project can vary with time and maturity. It has more to do with the way that project is developed and used than with what the project actually does.

In Summary . . .

In the past, our view of cohesion was much simpler than the last three principles have implied. We used to think that cohesion was simply the attribute of a module to perform one, and only one, function. However, the

three principles of component cohesion describe a much more complex variety of cohesion. In choosing the classes to group together into components, we must consider the opposing forces involved in reusability and developability. Balancing these forces with the needs of the application is nontrivial. Moreover, the balance is almost always dynamic. That is, the partitioning that is appropriate today might not be appropriate next year. Thus, the composition of the components will likely jitter and evolve with time as the focus of the project changes from developability to reusability.

This means that the component structure of the system cannot be determined up front, and must incrementally evolve as the system grows and changes. And this, of course, means that the changes to the component structure will be small but frequent. They become a normal part of any Agile iterative cycle.

Component Coupling

The next three principles deal with the relationships between components. Here again we will run into the tension between developability and logical design. The forces that impinge upon the architecture of a component structure are technical, political, and volatile.

The Acyclic Dependencies Principle (ADP)

Allow no cycles in the component dependency graph.

Have you ever worked all day, gotten some stuff working and then gone home, only to arrive the next morning to find that your stuff no longer works? Why doesn’t it work? Because somebody stayed later than you and changed something you depend upon! I call this *the morning-after syndrome*.

The morning-after syndrome occurs in development environments where many developers are modifying the same source files. In relatively small projects with just a few developers, it isn’t too big a problem. But as the sizes of the project and the development team grow, the mornings after can get pretty nightmarish. It is not uncommon for weeks to go by without being able to build a stable version of the project. Instead, everyone keeps on changing and changing their code, trying to make it work with the last changes that someone else made.

Eliminating Dependency Cycles

The solution to this problem is to partition the development environment into releasable components. The components become units of work, which can be the responsibility of a developer or a team of developers. When developers get a component working, they release it for use by the other developers. They give it a release number and move it into a directory for other teams to

use. They then continue to modify their component in their own private areas. Everyone else uses the released version.

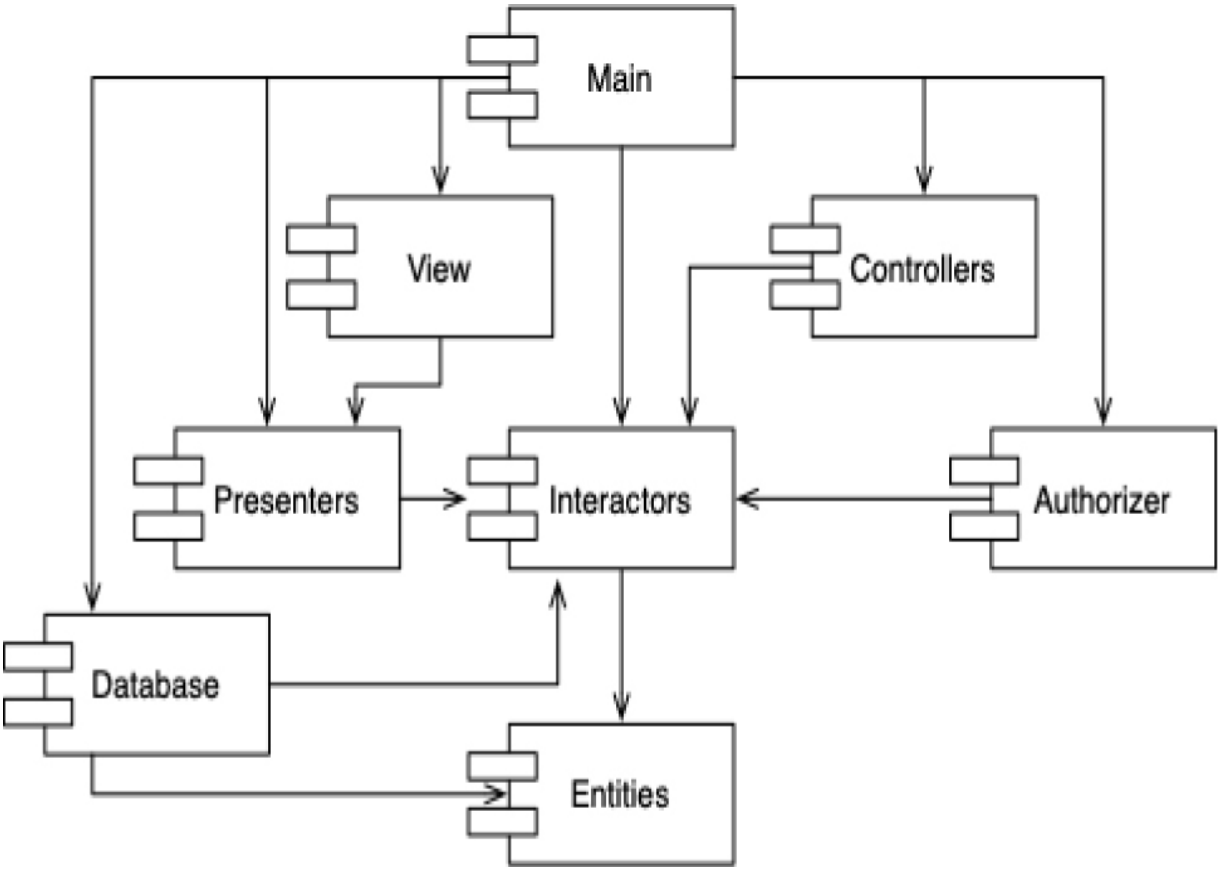
As new releases of a component are made, other teams can decide whether or not to immediately adopt the new release. If they decide not to, they simply continue using the old release. Once they decide that they are ready, they begin to use the new release.

Thus, none of the teams are at the mercy of the others. Changes made to one component do not need to have an immediate effect on other teams. Each team can decide for itself when to adapt its components to new releases of the components they use. Moreover, integration happens in small increments. There is no single point in time when all developers must come together and integrate everything they are doing.

This is a very simple and rational process; and it is widely used. However, to make it work you must manage the dependency structure of the components. There can be no cycles.

If there are cycles in the dependency structure, then the morning-after syndrome cannot be avoided.

Consider the component diagram below.



Here we see a rather typical structure of components assembled into an application. The function of this application is unimportant for the purpose of this example. What *is* important is the dependency structure of the components. Notice that this structure is a directed graph. The components are the *nodes*, and the dependency relationships are the directed edges.

Now notice one more thing. Regardless of which component you begin at, it is impossible to follow the dependency relationships and wind up back at that component. This structure has no cycles. It is a directed acyclic graph (DAG).

Now notice what happens when the team responsible for `Presenters` makes a new release of their component. It is easy to find out who is affected by this release; you just follow the dependency arrows backward. Thus, `View` and `Main` are both going to be affected. The developers currently working on those components will have to decide when they should integrate with the new release of `Presenters`.

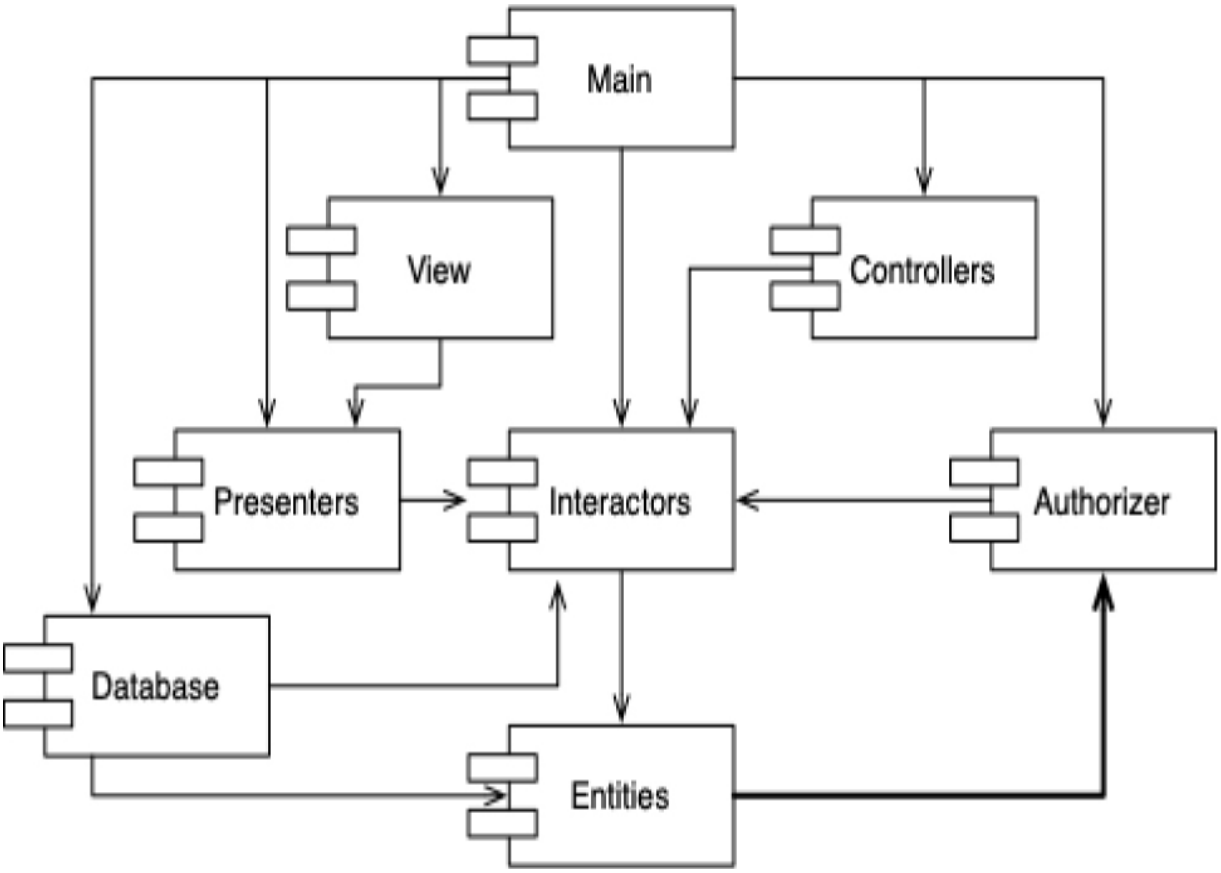
Notice also that when `Presenters` is released, it has utterly no effect upon many of the other components in the system. They don't know about `Presenters`, and they don't care when it changes. This is nice. It means that the impact of releasing `Main` is relatively small.

When the developers working on the `Presenters` component would like to run a test of that component, all they need to do is build their version of `Presenters` with the versions of the `Interactors` and `Entities` components that they are currently using. None of the other components in the system need be involved. This is nice. It means that the developers working on `Presenters` have relatively little work to do to set up a test and that there are relatively few variables for them to consider.

When it is time to release the whole system, it is done from the bottom up. First, the `Entities` component is compiled, tested, and released; then, `Database` and `Interactors`. These are followed by `Presenters`, `View`, `Controllers`, and then `Authorizer`. `Main` is last. This process is very clear and easy to deal with. We know how to build the system because we understand the dependencies between its parts.

The Effect of a Cycle in the Component Dependency Graph

Let us say that a new requirement forces us to change one of the modules in `Entities` such that it makes use of a module in `Authorizer`. For example, let's say that the `User` class in `Entities` uses the `Permissions` class in `Authorizer`. This creates a dependency cycle, as shown below.

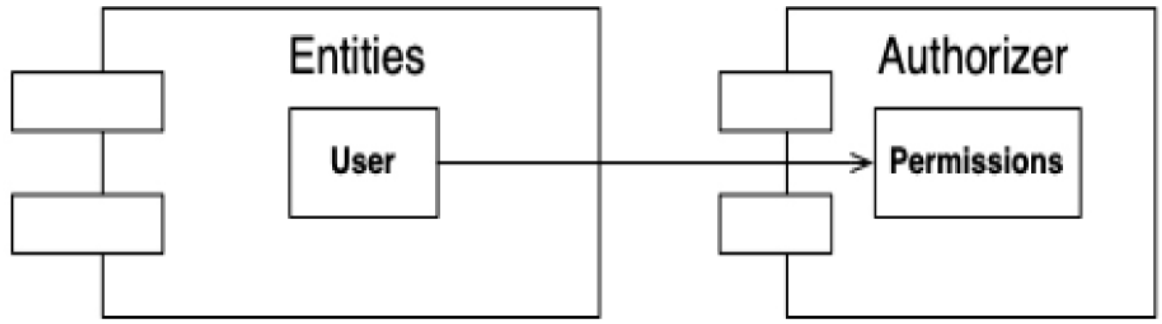


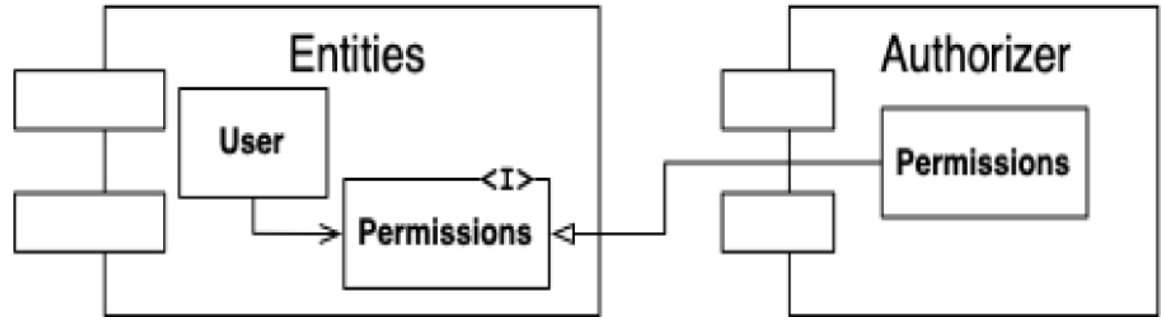
This cycle effectively combines `Interactors`, `Entities`, and `Authorizer` into a single component. Any component that uses one depends upon them all. And this makes building and testing much more difficult. It also begs the question of the correct build order. You can't build `Entities` first, because it depends upon `Authorizer`. But `Authorizer` depends upon `Entities` through `Interactors`. So there is no correct build order. Any system built without a correct build order functions by accident rather than by plan.

Breaking the Cycle

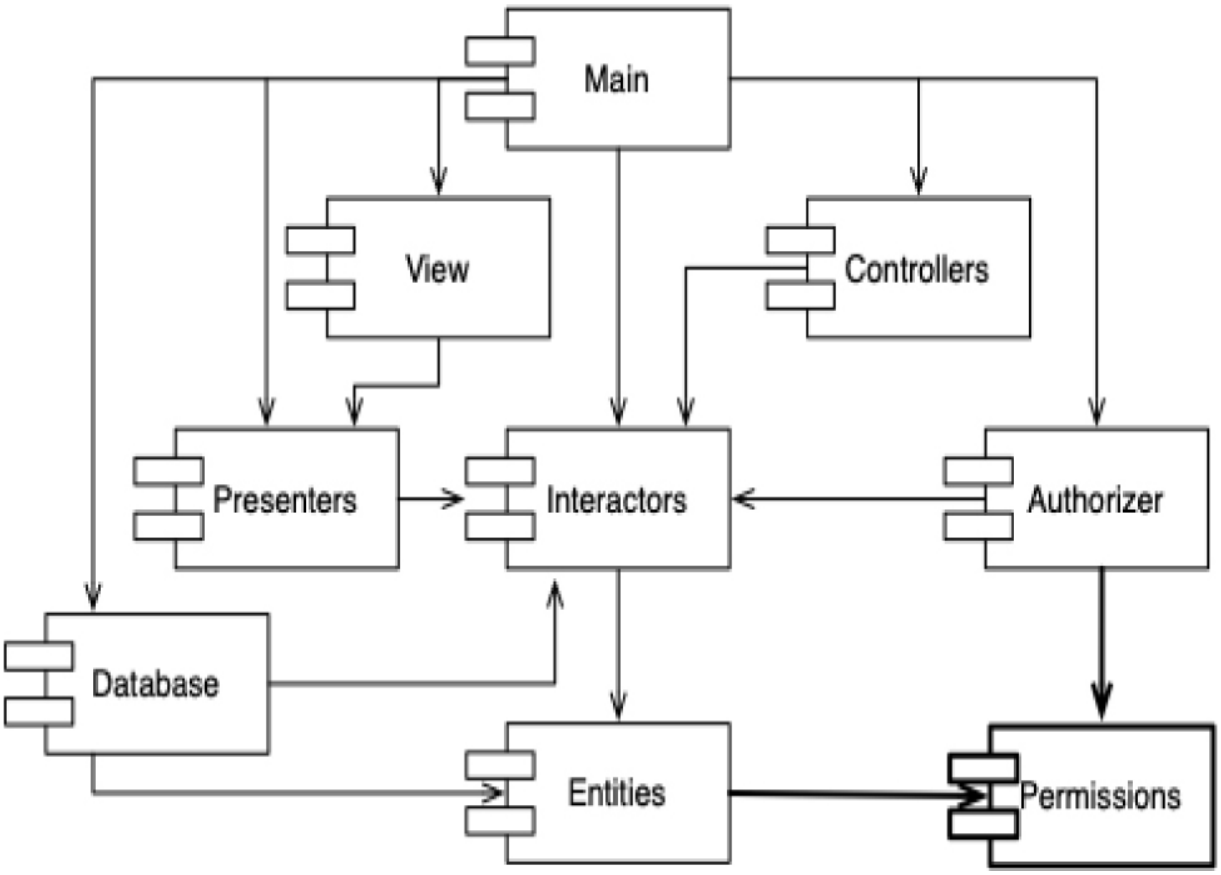
It is always possible to break a cycle of components and reinstate the dependency graph as a DAG. There are two primary mechanisms.

1. **Apply the Dependency Inversion Principle (DIP).** In the case below, we could create an interface that has the methods that `User` needs. We could then put that interface into `Entities` and inherit it into `Authorizer`. This inverts the dependency between `Entities` and `Authorizer`, thus breaking the cycle. See the before-and-after view below.





2. **Create a new component that both `Entities` and `Authorizer` depend upon.** Move the class(es) that they both depend upon into that new component. See below.



The “Jitters”

The second solution implies that the component structure is volatile in the presence of changing requirements. Indeed, as the application grows, the component dependency structure jitters and grows. Thus, the dependency structure must always be monitored for cycles. When cycles occur, they must be broken somehow. Sometimes this will mean creating new components, making the dependency structure grow.

Top-Down Design

The issues we have discussed so far lead to an inescapable conclusion. The component structure cannot be designed from the top down. This means that is it not one of the first things about the system that is designed. Indeed, it evolves as the system grows and changes.

Some of you may find this to be counterintuitive. We have come to expect that large-grained decompositions, like components, are also high-level functional decompositions.

When we see a large-grained grouping, such as a component dependency structure, we feel that the components ought to somehow represent the

functions of the system. Yet this does not seem to be an attribute of component dependency diagrams.

In fact, component dependency diagrams have very little do to with describing the function of the application. Instead, they are a map to the buildability and maintainability of the application. This is why they aren't designed at the start of the project. There is no software to build or maintain, and so there is no need for a build and maintenance map. But as more and more modules accumulate in the early stages of implementation and design, there is a growing need to manage the dependencies so that the project can be developed without the morning-after syndrome. Moreover, we want to keep changes as localized as possible, so we start paying attention to the SRP and the CCP and collocate classes that are likely to change together.

One of the overriding concerns of this dependency structure is the isolation of volatility. We don't want components that change frequently and for capricious reasons to affect components that otherwise ought to be stable. For example, we don't want cosmetic changes to the GUI to have an impact on our business rules. We don't want the addition or modification of reports to have an impact on our highest-level policies. So the component dependency graph is created and molded by developers to protect stable high-value components from volatile lower-level components.

As the application continues to grow, we start becoming concerned about creating reusable elements. That is when the CRP begins to influence the composition of the components. Finally, as cycles appear, the ADP is applied and the component dependency graph jitters and grows.

If we were to try to design the component dependency structure before we had designed any classes, we would likely fail rather badly. We would not know much about common closure, we would be unaware of any reusable elements, and we would almost certainly create components that produced dependency cycles. Thus, the component dependency structure grows and evolves with the logical design of the system.

The Stable Dependencies Principle (SDP)

Depend in the direction of stability.

Designs cannot be completely static. Some volatility is necessary if the design is to be maintained. We accomplish this by conforming to the CCP. Using this principle, we create components that are sensitive to certain kinds of changes but immune to others. Some of these components are designed to be volatile. We expect them to change.

Any component that we expect to be volatile should not be depended upon by a component that is difficult to change! Otherwise, the volatile component will also be difficult to change.

It is the perversity of software that a module that you have designed to be easy to change can be made hard to change by someone else who simply hangs a dependency upon it. Not a line of source code in your module need change; and yet your module will suddenly be hard to change. By conforming to the SDP, we ensure that modules that are intended to be easy to change are not depended upon by modules that are harder to change.

Stability

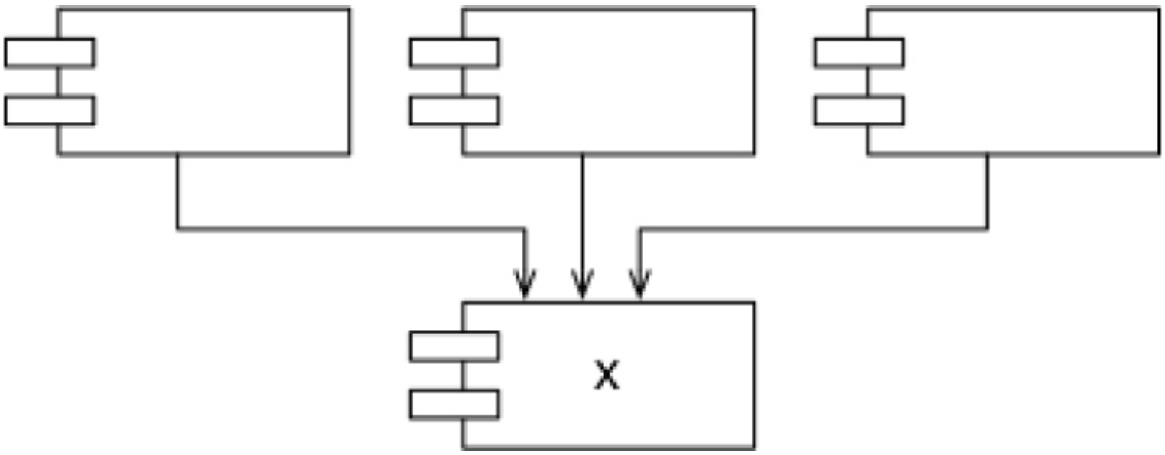
What is meant by stability? Stand a penny on its side. Is it stable in that position? You'd likely say that it was not. However, unless disturbed, it will remain in that position for a very long time. Thus, stability has nothing directly to do with frequency of change. The penny is not changing, but it is hard to think of it as stable.

Webster says that something is stable if it is "not easily moved." Stability is related to the amount of work required to make a change. The penny is not very stable, because it requires very little work to topple it. On the other hand, a table is very stable because it takes a considerable amount of effort to turn it over.

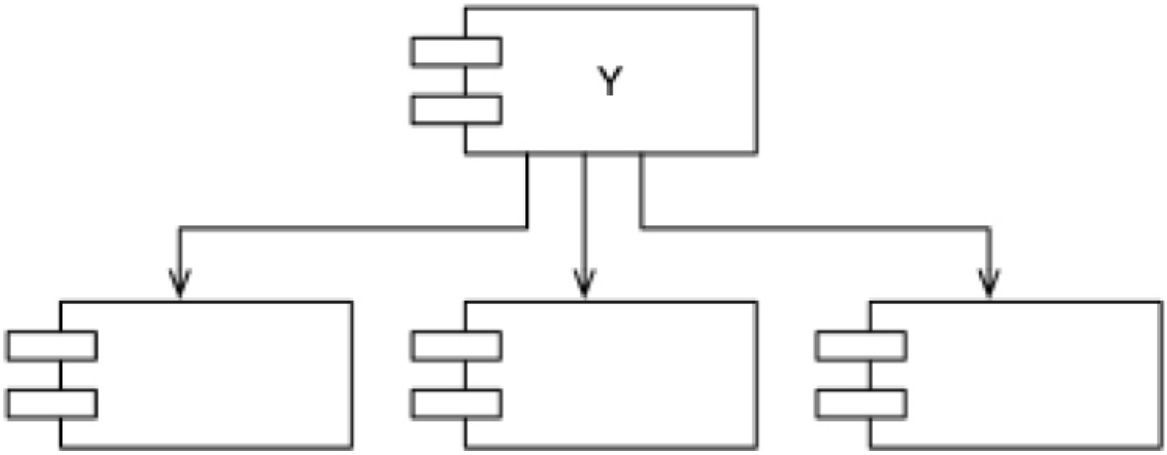
Stability is a continuum, not a boolean value.

How does this relate to software? There are many factors that make a software component hard to change—its size, complexity, clarity, and so on. We are going to ignore all those factors and focus upon something different. One sure way to make a software component difficult to change is to make lots of other software components depend upon it. A component with lots of incoming dependencies is very stable because it requires a great deal of work to reconcile any changes with all the dependent components.

The following diagram shows X, a stable component. This component has three components depending upon it, and therefore it has three good reasons not to change. We say that it is responsible to those three components. On the other hand, X depends upon nothing, so it has no external influence to make it change. We say that X is responsible and independent: It is an adult.



The next diagram, on the other hand, shows Y, a very unstable component. Y has no other components depending upon it; we say that it is irresponsible. Y also has three components that it depends upon, so changes may come from three external sources. We say that Y is dependent and irresponsible: It is a teenager.



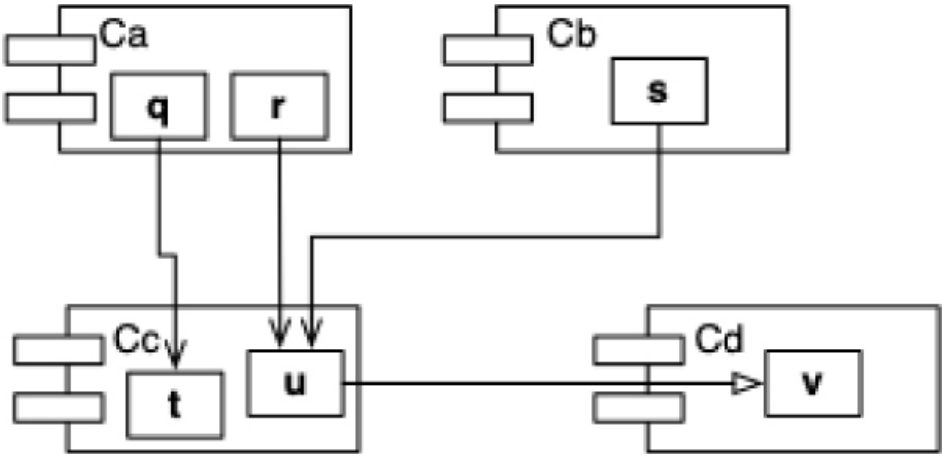
Stability Metrics

How can we measure the stability of a component? One way is to count the number of dependencies that enter and leave that component. These counts will allow us to calculate the positional stability of the component.

- Fan-in: Incoming Couplings—This is the number of modules outside this component that depend upon modules within this component.
- Fan-out: Outgoing Couplings—This is the number of modules inside this component that depend upon modules outside this component.
- *I*: Instability: $I = \text{Fan-out} \div (\text{Fan-in} + \text{Fan-out})$. This metric has the range [0,1]. $I = 0$ indicates a maximally stable component. $I = 1$ indicates a maximally unstable component.

The Fan-in and Fan-out metrics[7] are calculated by counting the number of modules outside the component in question that have dependencies on the modules inside the component in question. Consider the example below:

[7] In previous publications I used the names Efferent and Afferent couplings (Ce, and Ca) for Fan-out and Fan-in respectively. That was just hubris on my part… I liked the metaphor of the central nervous system.



Let's say we want to calculate the stability of the component Cc. We find that there are three classes outside Cc that depend upon classes in Cc. Thus, Fan-in = 3. Moreover, there is one class outside Cc that classes in Cc depend upon. Thus, Fan-out = 1, and *I* = 1/4.

In C++, these dependencies are typically represented by `#include` statements. Indeed, the *I* metric is easiest to calculate when you have organized your source code such that there is one class in each source file. In Java, the *I* metric can be calculated by counting `import` statements and qualified names. Other languages have similar mechanisms such as `using` or `require`.

When the *I* metric is 1, it means that no other component depends upon this component (Fan-in = 0); and this component depends upon other components (Fan-out > 0). Thus, it is as unstable as a component can get; it is a teenager, irresponsible and dependent. Its lack of dependents gives it no reason not to change, and the components that it depends upon may give it ample reason to change.

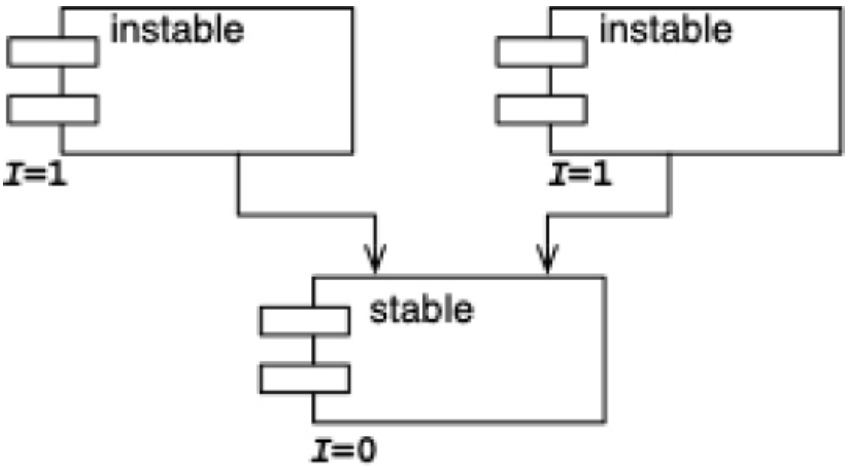
On the other hand, when the *I* metric is zero, it means that the component is depended upon by other components (Fan-in > 0), but it does not itself depend upon any other components (Fan-out = 0). It is a responsible and independent adult. Such a component is as stable as it can get. Its dependents make it hard to change, and it has no dependencies that might force it to change.

The SDP says that the *I* metric of a component should be larger than the *I* metrics of the components that it depends upon.

I metrics should decrease in the direction of dependency.

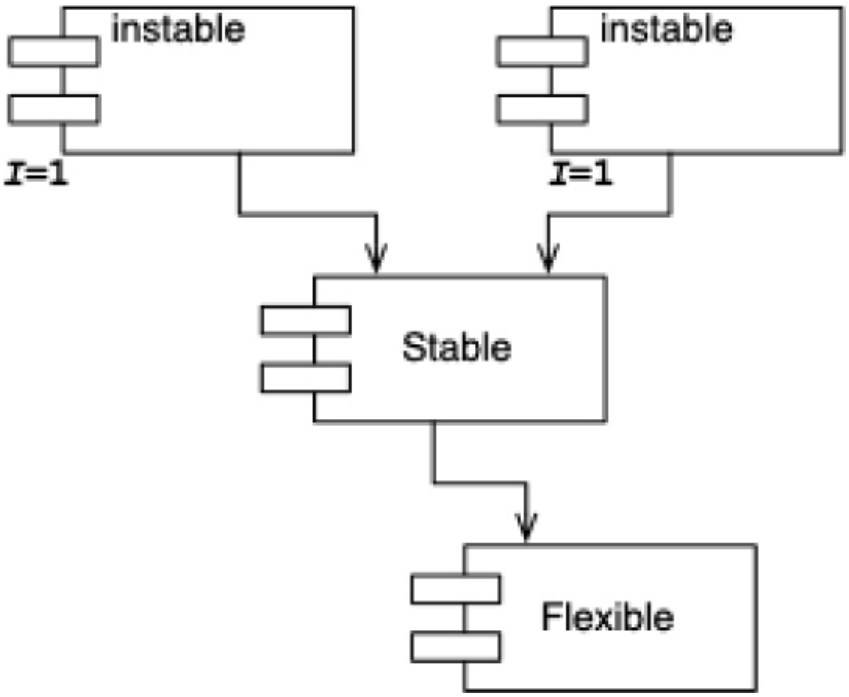
Not All Components Should Be Stable

If all the components in a system were maximally stable, the system would be unchangeable. This is not a desirable situation. Indeed, we want to design our component structure so that some components are unstable and some are stable. The following diagram shows a conformant configuration for a system with three components.



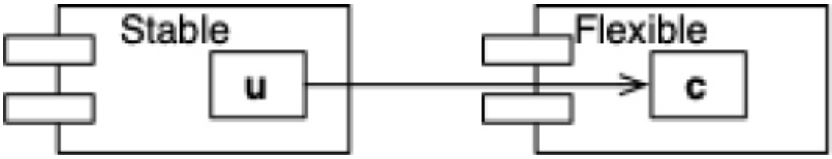
The changeable components are on top and depend upon the stable component at the bottom. Putting the unstable components at the top of the diagram is a useful convention since any arrow that points *up* is violating the SDP (and, as we shall see in the next chapter, the ADP).

The next diagram shows how the SDP can be violated.

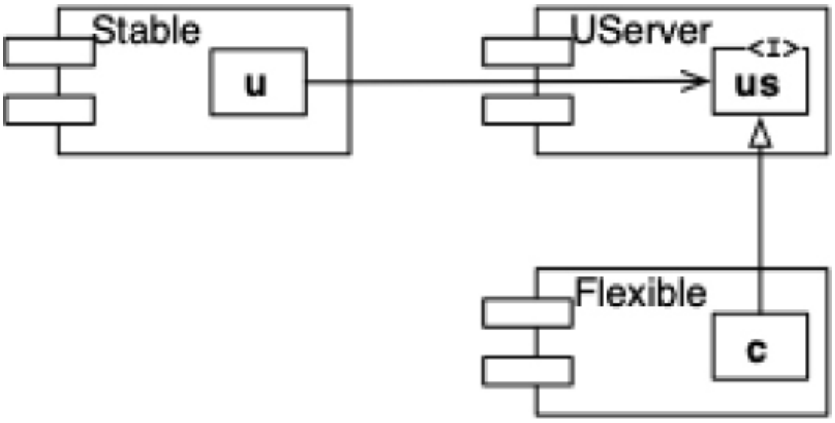


`Flexible` is a component that we have designed to be easy to change. We want `Flexible` to be unstable. However, some developer, working in the component named `Stable`, has hung a dependency upon `Flexible`. This violates the SDP since the *I* metric for `Stable` is much lower than the *I* metric for `Flexible`. As a result, `Flexible` will no longer be easy to change. A change to `Flexible` will force us to deal with `Stable` and all its dependents.

To fix this, we somehow have to break the dependence of `Stable` upon `Flexible`. Why does this dependency exist? Let's assume that there is a class `C` within `Flexible` that another class `U` within `Stable` needs to use.



We can fix this by employing the DIP. We create an interface class called `US` and put it in a component named `UServer`. We make sure that this interface declares all the methods that `U` needs to use. We then make `C` implement this interface, as shown below. This breaks the dependency of `Stable` upon `Flexible`, and it forces both components to be dependent upon `UServer`. `UServer` is very stable (*I* = 0), and `Flexible` retains its necessary instability (*I* = 1). All the dependencies now flow in the direction of decreasing *I.*



Abstract Components

You may find it strange that we have created a component, like `UService`, that contains nothing but an interface. Such a component contains no executable code! It turns out, however, that this is a very common, and necessary, tactic when using statically typed languages like Java or C#. These abstract components are very stable and are therefore ideal targets for less stable components to depend upon.

When using dynamically typed language like Ruby or Python, these abstract components don't exist at all. Nor do the dependencies that would have

targeted them. Dependency structures in these languages are much simpler because dependency inversion does not require either the declaration or the inheritance of interfaces.

The Stable Abstractions Principle (SAP)

A component should be as abstract as it is stable.

What follows is going to get a bit mathematical and theoretical. Don't let that scare you off. The theory is important, even if the practice is only loosely bound to it. Read through to the end and you'll see what I mean.

Where Do We Put the High-Level Policy?

Some software in the system should not change very often. This software represents high-level architecture and policy decisions. We don't want these business and architectural decisions to be volatile. Thus, the software that encapsulates the high-level policies of the system should be placed into stable components ($I = 0$). Unstable components ($I = 1$) should contain only the software that is volatile—software that we want to be able to quickly and easily change.

However, if the high-level policies are placed into stable components, then the source code that represents those policies will be difficult to change. This could make the overall architecture inflexible. How can a component that is maximally stable ($I = 0$) be flexible enough to withstand change? The answer is to be found in the OCP. This principle tells us that it is possible and desirable to create classes that are flexible enough to be extended without requiring modification. What kinds of classes conform to this principle? Abstract classes.

The SAP sets up a relationship between stability and abstractness. It says that a stable component should also be abstract so that its stability does not prevent it from being extended. On the other hand, it says that an unstable component should be concrete since it its instability allows the concrete code within it to be easily changed.

Thus, if a component is to be stable, it should also consist of interfaces and abstract classes so that it can be extended. Stable components that are extensible are flexible and do not overly constrain the architecture.

The combination of the SAP and the SDP amounts to the DIP for components. This is true because the SDP says that dependencies should run in the direction of stability, and the SAP says that stability implies abstraction. Thus, dependencies run in the direction of abstraction.

Measuring Abstraction

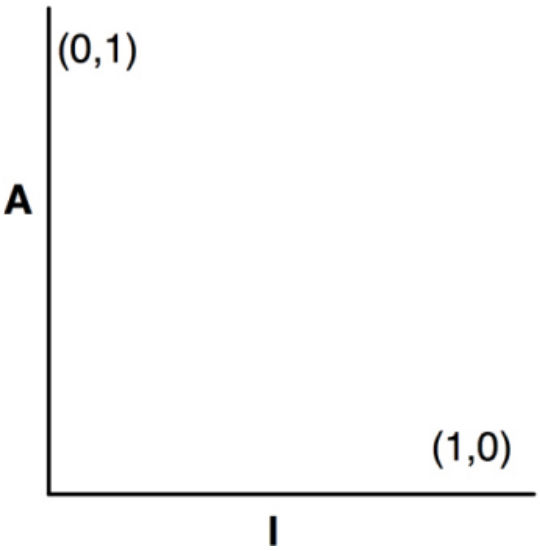
The *A* metric is a measure of the abstractness of a component. Its value is simply the ratio of interfaces and abstract classes in a component to the total number of classes in the component.

- Nc*:* The number of classes in the component.
- Na*:* The number of abstract classes and interfaces in the component.
- A: Abstractness.

  A = Na ÷ Nc. The A metric ranges from 0 to 1. Zero implies that the component has no abstract classes at all. A value of 1 implies that the component contains nothing but abstract classes and/or interfaces.

The Main Sequence

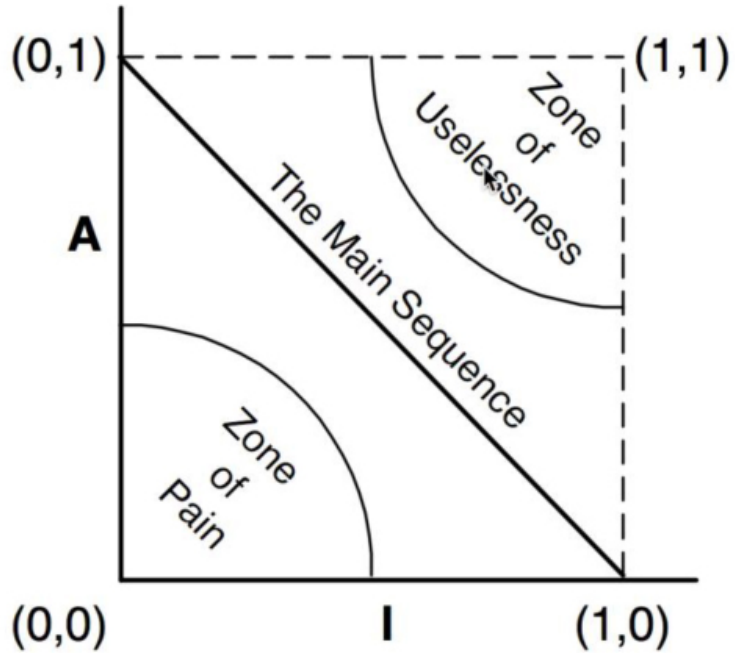
We are now in a position to define the relationship between stability (*I*) and abstractness (*A*). We can create a graph with *A* on the vertical axis and *I* on the horizontal axis. If we plot the two “good” kinds of components on this graph, we will find the components that are maximally stable and abstract (the adults) at the upper left at (0,1). The components that are maximally unstable and concrete (the teenagers) are at the lower right at (1,0).



Not all components can fall into one of these two positions. Components have degrees of abstraction and stability. For example, it is very common for one abstract class to derive from another abstract class. The derivative is an abstraction that has a dependency. Thus, though it is maximally abstract, it will not be maximally stable. Its dependency will decrease its stability.

Since we cannot enforce that all components sit at either (0,1) or (1,0), we must assume that there is a locus of points on the *A*/*I* graph that defines reasonable positions for components. We can infer what that locus is by

finding the areas where components should not be—that is, zones of exclusion.



The Zone of Pain

Consider a component in the area of (0,0). This is a highly stable and concrete component. Such a component is not desirable, because it is rigid. It cannot be extended, because it is not abstract. And it is very difficult to change because of its stability. Thus, we do not normally expect to see well-designed components sitting near (0,0). The area around (0,0) is a zone of exclusion called the *Zone of Pain.*

It should be noted that certain software entities do, in fact, fall within the Zone of Pain. An example would be a database schema. Database schemas are notoriously volatile, extremely concrete, and highly depended upon. This is one of the reasons that the interface between `OO` applications and databases is so difficult, and that schema updates are generally painful.

Another example of software that sits on (0,0) is a concrete utility library. Although such a library has an *I* metric of 1, it may in fact be nonvolatile. Consider the `String` component, for example. Even though all the classes within it are concrete, it is so commonly used that changing it would create chaos. Therefore, `String` is nonvolatile.

Nonvolatile components are harmless in the (0,0) zone since they are not likely to be changed. For that reason, it is only software components that are volatile that are also problematic in the Zone of Pain. The more volatile a component in the Zone of Pain is, the more “painful” it is. Indeed, we might

consider volatility to be a third axis of the graph; and then the graph above shows only the most painful plane, where volatility = 1.

The Zone of Uselessness

Consider a component near (1,1). This location is undesirable because it is maximally abstract and yet has no dependents. This is like an interface that has no implementation. Such components are useless. Thus, this is called the *Zone of Uselessness*.

The software entities that inhabit this region are a kind of detritus. They are often leftover abstract classes that no one ever implemented. We find them in systems from time to time, sitting in the codebase, unused.

Or they may be superfluous interfaces—interfaces that a few classes implemented but no class ever used.

A component that has a position deep within the Zone of Uselessness must contain a significant fraction of such entities. The presence of such useless entities is undesirable.

The Main Sequence

We'd like our most volatile components to be as far from both zones of exclusion as possible. The locus of points that is maximally distant from each zone is the line that connects (1,0) and (0,1). I call this line the *main sequence*.[8]

[8] The author begs the reader's indulgence for the arrogance of borrowing such an important term from astronomy.

A component that sits on the main sequence is not "too abstract" for its stability, nor is it "too unstable" for its abstractness. It is neither useless nor particularly painful. It is depended upon to the extent that it is abstract, and it depends upon others to the extent that it is concrete.

The most desirable positions for a component are at one of the two endpoints of the main sequence. We strive to position the majority of components at those endpoints. However, in my experience, some small fraction of the components in a large system are neither perfectly abstract nor perfectly stable. Those components have the best characteristics if they are on or close to the main sequence.

Distance from the Main Sequence

This leads us to our last metric. If it is desirable for components to be on or close to the main sequence, we can create a metric that measures how far away a component is from this ideal.

- *D*[9]: Distance. *D* = |A+I–1|. This metric ranges from [0,1]. Zero indicates that the component is directly on the main sequence. One indicates that the component is as far away as possible from the main sequence.

[9] In previous publications, I called this metric D'. I see no reason to continue that practice.

Given this metric, a design can be analyzed for its overall conformance to the main sequence. The *D* metric for each component can be calculated. Any component that has a *D* value that is not near zero can be reexamined and restructured.

Conclusion

If your head is reeling at this point, I suggest you breathe. I am not suggesting that you should measure and calculate all these metrics. It's good to know that F = ma, but you don't often do that math when you drive your car. Theory is one thing. Practice is another.

I certainly don't make a practice of gathering the necessary statistics, doing all the math, and generating plots. In most projects, I can *see* these relationships in my head and do a simple mental evaluation—and I don't mean that I'm doing the math in my head either. Rather, I simply envision the relationships and infer how stable and how abstract each component is. If you wish, consider it a "feeling."

It's good to understand the above theory. There are cases where it may even be beneficial to do the above calculations. There have been a few extreme cases where I certainly have. But in most cases, understanding the theory and using it to "feel" the relationships is good enough.

You may find that your IDE does many of these calculations for you. I've found it helpful, in larger projects, to look over the numbers that the IDE has calculated. But my "feel" is pretty good, and I am seldom surprised.

Chapter 21. Continuous Design



By Jeff Langr

Product owners (and others responsible for helping to define a software product) determine the set of end goals that a system must accomplish for its users. Developers, in turn,

build support for each of these goals in the system, in the form of code and configuration. As Jack W. Reeves taught us over three decades ago, this code (and configuration[1]) is the ultimate and definitive representation of "the design."[2]

[1] For example, [PPP02], http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod, and https://en.wikipedia.org/wiki/SOLID_(object-oriented_design) (or just google SOLID).

[2] https://www.developerdotstar.com/mag/articles/reeves_design.html. Republished in [PPP02].

The word *design* classically has two primary meanings:

- The plan for building something
- The form and functionality exuded by the current state of that something

Continuous design, also abbreviated as . . . Wait, that abbreviation is already taken. We could just utter the full phrase "continuous design" throughout this chapter. Ultimately, we'll just call it "design," because that's how we best build software systems: one conscious design decision after another.

As we were saying: Continuous design means that we continually plan how to accommodate new behaviors and that our system has a new design each time we add a new behavior. Most of that planning occurs as we begin writing the code that will provide the new behaviors.

Adding new code or updating existing code always changes the design, often only in a very small way.

Design is not a nebulous phase of development that occurs at the outset of a project. It is not solely the activity of drawing high-level sketches about what the system needs to look like.

Design is a continuous activity. Our system's design should command our constant attention, because it changes continuously, by definition. Sounds daunting, but in reality, it's easier than the alternative of generating an initial set of design models, then pretending we can ignore design after that.

Continuous Change

While software is made up of code, it's more useful to say that software is made of thousands of small behaviors (sometimes called *units*), each implemented in the form of code. Our job as programmers is to create and orchestrate these behaviors in a way that supports meeting the end goals of a system's users.

We must first translate end goals into a number of smaller software behaviors. Given the goal "present a random list of flash cards for learning Czech nouns," we might implement the following sequence of behaviors.

1. Retrieve a random subset of all nouns.
2. For each noun:
    a. Present a question: A fill-in-the-blank sentence along with a multiple-choice listing of the four possible noun forms.
    b. Capture the user's choice.
    c. Show the user the correct answer if their choice is incorrect.
3. Repeat steps 2a through 2c for each question that the user chose incorrectly.

There are usually many ways to skin a cat, so we might instead support an alternate set of steps. For example, the

user might be able to continue selecting an answer for a single question until they get it correct. We might present hints with each failed answer. We might replace multiple-choice questions with fill-in-the blank questions as the user improves in skill.

If there are many design choices for an application itself, there are orders of magnitude more choices for how we write the underlying code. No, that’s not right. We have infinite design choices. Most of those choices will be suboptimal.

Our product managers and owners shape our software products to meet the perceived needs of the marketplace. Most software products continue to demand constant change in the form of new needs, and enhancements to existing needs.

In turn, we must constantly be able to meet the demand by shaping and enhancing existing software. A software system is a unique kind of product: It must not only continuously present a design that supports all current needs to date, but also accept that this design will need to be different tomorrow.

We might be able to build a slick, useful flash card system in a short period of time, perhaps a month or two. After an initial release, users will demand new behaviors. We’re selling new features—great! But we must also continuously return to the current system’s inherent design—its codebase. In order to add new features, we’re forced to figure out how to accommodate new concepts into the existing design, things that might have never been considered before.

Continuous Design

When it comes to making choices in software, we have only a few kinds of primary building blocks: modules (classes), functions (methods), data references (variables), and statements/expressions. The first three of these constructs can be named, to give us humans a better chance of understanding what concepts exist in the codebase, and also a better chance of finding the code we need to change or extend.

How we choose to manage and organize these building blocks is the activity of software design. Our system's design is the collection of choices we've made to this point in time.

We can easily make many bad choices when assembling these constructs, and in a very short time too. After making changes to our flash card app, it might still work, but our changes might have unnecessarily increased the cost for any future modifications.

If changing our flash card app results in unorganized, convoluted, and unclear code, accommodating new needs will be slow. We'll expend too much time navigating code to find all the places it must change. Once we find those locations (usually plural), we'll expend too much time understanding what the current code does. We'll then expend too much time carefully inserting our changes and ensuring that all the existing behaviors remain unbroken.

Each design choice we make impacts the cost for future choices. We succeed only if we minimize the negative design impact of each choice. This is the critical concept known as *continuous design*.

Sailing on the Four Cs of Continuous Design

An Unclean Code system makes it harder to

- Find code you're looking for.
- Understand where you need to make changes.
- Make changes without creating defects.
- Write tests to verify its behaviors.

You might recognize these characteristics from Chapter 13, "Clean Classes."

We have many design perspectives to help guide us toward a Clean Code system. As we change our system, we continue to reflect on one or more of these perspectives—SOLID, code smells, simple design, design patterns, and others, including various one-off principles such as the Law of Demeter or Tell-Don't-Ask.

For the concept of continuous design, we'll try—as many have before—to codify it all in a simple, hopefully memorable set of rules. Consider this YADP: Yet Another Design Perspective.

Our four Cs design considerations are

- Clarity: The programmer's intents in the system are all clearly stated.
- Conciseness: The programmer's intents in the system are implemented in a minimal amount of code.
- Confirmability: All unit behaviors in the system can be easily tested, in a way that also provides "living documentation" of the behavioral choices that were made.
- Cohesion: Each module in the system has a high level of cohesion—all its elements maximally relate to one another.

These considerations don’t live in isolation. We’ll talk about them one by one, but as we discuss, think about how they can both support and oppose each other.

Clarity

Getting code working is always our first step. We’re challenged to add a small, new piece of behavior to the code. We think about how we want to solve the problem for a small amount of time, then quickly splash some code into our editor. We whittle the code a bit until we finally manage to get it working, then move on to the next slice of logic needed.

If we never had to read that code again, we might consider ourselves “done.” Code complete! It ain’t broke, and so we ain’t gonna try ’n fix what’s already working. Why risk breaking things?

But we will have to read that code again, when we’re later required to change nuances of the logic! Or when we’re asked “What does that code do in this case?” Or when we’re required to add new related behaviors. Or in a half hour when we have to return to the code because it’s not quite done yet.

For each of these needs, we want the code to be as clearly stated as possible. Anything unclear wastes time.

A challenge: Decipher all the nuances of the `retrieveWords` function. It might take you a half minute or two.

```
export const retrieveWords = async words => {
  const wordPrefix = words.length === 1 ? 'word' : 'words'
  let wordList = "";
  for (let i = 0; i < words.length; i++) {
```

```
    const currentWord = words[i];
    const quotedWord = '"' + currentWord + '"';
    wordList += quotedWord;
    if (i !== words.length - 1) {
      wordList += ",";
    }
  }

  const finalPrompt = `Given a list of nouns separated by commas,
    `provide appropriate Czech language information for the ` +
    `${wordPrefix} ${wordList}`
  const response = await sendPrompt(`${format} ${finalPrompt}`)
  const startIndex = response.indexOf('[')
  const endIndex = response.lastIndexOf(']') + 1
  const jsonText = response.slice(startIndex, endIndex)
  return JSON.parse(jsonText)
}
```

This is not necessarily bad code, but it could easily be better. It took more time for you to decipher than necessary, so we'll call it unclean code.

A continual focus on code clarity tells us to produce code that we can all consume quickly. In the following refactored code, `retrieveWords` appears as the bottommost function. Start your reading there.

```
const sliceJSONArrayFrom = response => {
  const startIndex = response.indexOf('[')
  const endIndex = response.lastIndexOf(']') + 1
```

```
  return response.slice(startIndex, endIndex)
}

const joinQuoted = words =>
  words.map(word => `"${word}"`).join(',')

const pluralizeIfMany = (word, list) =>
  list.length === 1 ? word : `${word}s`

export const retrieveWords = async words => {
  const finalPrompt = `Given a list of nouns separated by commas,
    `provide appropriate Czech language information for the ` +
    `${(pluralizeIfMany('word', words))} ${(joinQuoted(words))}`

  const response = await sendPrompt(`${format} ${finalPrompt}`)

  return JSON.parse(sliceJSONArrayFrom(response))
}
```

If we're tasked with making changes to `retrieveWords`, understanding its overall intent and flow now occurs more quickly. The function initializes a `finalPrompt` as a long text string with a couple of embedded concepts:

- Either the text "word" or "words," depending on the size of the `words` array
- A joined string of the quoted words in that array

Those two concepts are now calls to tiny, isolated functions.

We then send a longer prompt string to an LLM, receiving a text response. Finally, we slice out the JSON text from the response string (which exists because we sent the LLM a desired format as part of the prompt), then parse it.

We can understand such a function in about fifteen seconds; it is largely a statement of policy. It uses extracted functions with names that capture their intent. Each extracted function is implemented in one to three lines of code. If we needn't change any of those implementations, that's one to three lines of hidden detail we can ignore for now and maybe forever.

If we need to change one of the helper functions, the streamlined policy declaration allows us to find it quickly and focus on a tiny bit of implementation detail. We can make our changes without considering any other code.

Editing Our Code

Crafting code with high clarity demonstrates that we care enough about our teammates (and ourselves) to spend a few moments editing our code before moving on.

As developers, we're good about getting things to work—our #1 job. We need to also remind ourselves that we're writers as well, and that others must consume what we write. Once we capture our ideas on paper, good writing demands that we revisit our spewage and edit it to emphasize clarity.

The sad fact is that we're not typically taught to edit our code. The vast majority of code out there shows it, and wastes copious amounts of your time as a result. WTFs per minute are high.

In fact, we're often told expressly not to edit code. "If it ain't broke, don't fix it." There's that lame, misapplied mantra again.

If other people can’t make quick sense of your code, it *is* broken.

If you picked up a poorly written book, one that required you to reread sentences and paragraphs before they made sense, you’d consider it a bad, broken book (we bet you had at least one of these at university).

The mantra exists because we fear breaking things. Yes, code is a brittle material. It’s pretty easy to make a dumb coding mistake in just about any programming language, yet not even spot it. As a result, we’re inclined to avoid cleaning things up, despite how easy it usually is.

Fearing making changes to our code is a quality smell. There are simple paths for creating the controls that tell us, with every tiny change, whether or not we’ve broken already-working logic. Visit Chapter 14, “Testing Disciplines,” to learn how.

Quick Steps to Clarity

- Extract implementation detail from multipurpose functions to create new functions with concise names.
- Move functions as appropriate to other modules. This results in increased cohesion, which helps readers find code where they might expect it.
- Replace comments with clear declarations.
- Write tests that double as documentation on all behavioral intents coded into the system.

Objections to Clarity

Seasoned programmers are able to immediately digest short code phrases. An experienced JavaScript developer can quickly comprehend the following couple of examples:

```
words.map(word => `"${word}"`).join(',')
list.length === 1 ? word : `${word}s`
```

Such phrases still demand careful reading, though. We must mentally assemble the tokens involved into the behavioral concept they represent. A small potential for misinterpretation exists, as does a small potential for a defect. The following line of seemingly idiomatic code should give us pause:

```
for (let i = 0; i <= words.length; i++)
```

Yet for many, it does not. That’s not the idiom; this is:

```
for (let i = 0; i < words.length; i++)
```

The first `for` loop is likely a defect.

In any case, extracting small implementation details into intention-revealing functions streamlines code in `retrieveWords`:

```
const joinQuoted = words =>
  words.map(word => `"${word}"`).join(',')


const pluralizeIfMany = (word, list) =>
  list.length === 1 ? word : `${word}s`


export const retrieveWords = async words => {
  const finalPrompt =
    `Given a list of English nouns, separated by commas, ` +
    `provide appropriate Czech language information for the ` +
    `${(pluralizeIfMany('word', words))} ${(joinQuoted(words))}`
    // ...
```

Declare Intent, Don’t Ooze Details

We use functional pipelines partly because they allow us to replace details with declarations:

```
words.map(word => `"${word}"`).join(',')
```

That’s loads easier to follow than the old-school procedural equivalent:

```
let wordList = "";
for (let i = 0; i < words.length; i++) {
  const currentWord = words[i];
  const quotedWord = '"' + currentWord + '"';
  wordList += quotedWord;
  if (i !== words.length - 1) {
    wordList += ",";
  }
}
```

The old-school code also has a dramatically increased possibility of hiding defects (hearken back to the `for` loop example).

Keeping Pipelines Streamlined

While it’s reasonable to code short in-line functions within a pipeline:

```
export const mostExpensiveHighlyRatedBookInEachCategory = (categor
  categories.flatMap(category =>
    category.books
```

```
      .filter(book => !!book.title)
      .filter(book => book.rating > 4.0)
      .sort((a, b) => b.price - a.price)
      .map((book) =>
        ({ category: book.category, title: book.title }))
      .slice(0, 1))
```

they break the flow and require careful stepwise reading.

Replacing all the in-line lambdas—even the very short functions—with named functions does wonders for our ability to quickly follow the entire flow:

```
export const
mostExpensiveHighlyRatedBookInEachCategory = (categories) => {
  const byPrice = (a, b) => b.price - a.price
  const highlyRated = book => book.rating > 4.0
  const hasTitle = book => !!book.title
  const categoryAndTitle = (book) =>
    ({ category: book.category, title: book.title })

  return categories.flatMap(category =>

    category.books
      .filter(hasTitle)
      .filter(highlyRated)
      .sort(byPrice)
      .map(categoryAndTitle)
      .slice(0, 1))
}
```

When reading the larger function, we scan past the one-liner function declarations and focus on the `return` statement. We're able to read and mentally assemble what's going on more quickly now, because the pipeline is abstracted and not full of detail.

Hmm. The updated solution contains an idiomatic expression: `.slice(0, 1)`. It returns the first element of an array (less idiomatic) or an empty array if the array is empty (yes idiomatic). We don't eliminate our idioms unless they're causing readers to come to a grinding halt. If so, we find a way to abstract them. (Here, we might create a `firstOrDefault` function to supplant the slice call.)

Note that as we extract functions, we find insufficient cohesion in the form of misplaced bits of logic. Typical. A couple of functions extracted from this example's pipeline might find more appropriate homes in a book module.

Conciseness

A functional pipeline can provide code that is not only very clear, but also concise. The solution that requires the fewest possible tokens is maximally concise—but this is almost never what we really want.

Code that is maximally concise without consideration for clarity is obfuscated. The only legitimate time when we might deliberately obscure code is as part of a post-development deployment process (perhaps for compression or security interests), or as challenge/entertainment (see https://www.ioccc.org).

The absolute need for clarity means we must find a balance between conciseness and clarity. Finding the balance isn't tough, though—just ask your teammates. They'll let you know the moment something doesn't make sense. Watch

them read through your code, and it'll usually be obvious when they're struggling with your ultra-optimized logic.

Let's start with an opposite example, however—code that's clear but not concise. The following `generateCard` function, along with a couple of helper functions, assembles a text-based flash card (complete with the answer on it—how curious!).

```
// generate integer from 0 to n-1
export const randomInt = n => {
  const randomValue = Math.random(); // Generate a random decimal
  const scaledValue = randomValue * n; // scale to desired range
  const flooredValue = Math.floor(scaledValue); // floor to int
  return flooredValue;
}

// return a word in its plural form if needed
const pluralizeIf = (word, isPlural) => {
  if (isPlural) {
      return `${word}s`; // Append 's' to make the word plural
  } else {
      return word; // Return the original word if not plural
  }
}

// generates prompt text for a language card game
const generateCard = (adjectives, nouns) => {
  // Select a random adjective and noun from the provided lists
  const randomAdjectiveIndex = randomInt(adjectives.length);
  const randomNounIndex = randomInt(nouns.length);
```

```
  const adjective = adjectives[randomAdjectiveIndex];
  const noun = nouns[randomNounIndex];

  // Determine the case and number randomly
  const nominativeOrAccusative =
    randomInt(2) === 0 ? 'nominative' : 'accusative';
  const isPlural = randomInt(2) === 0;
  const singularOrPlural = isPlural ? 'plural' : 'singular';

  // Get correct forms of the noun & adjective based on case/numbe
  const correctNoun = noun[singularOrPlural][nominativeOrAccusativ
  const nounGender = noun.gender.toLowerCase();
  const correctAdjective =
    adjective[nominativeOrAccusative][singularOrPlural][nounGender

  // Construct the prompt text for the game card
  const promptText = `Adjective: ${adjective.word}
Noun: ${noun.word}

Determine the ${singularOrPlural} ${nominativeOrAccusative} case.

Correct answer: ${correctAdjective} ${correctNoun}`;

  // Return the constructed prompt
  return promptText;
}
```

The code appears typical to us, and that’s not a good thing. The core function, `generateCard`, is a top-to-bottom affair. It’s not terribly long, but it includes comments to guide readers through what’s happening in the next few statements. Each of the function’s statements uses clear variable names and is readily comprehensible. We can carefully read through the code statement by statement, mentally assembling groups of statements into unit behaviors. What’s not to like?

Indeed, the above example is how many developers happily continue coding for most of their career. But it’s suboptimal for numerous reasons.

A reworked version results in additional small helper functions.

(Again, start your reading at `generateCard` at the bottom.)

```
export const randomInt = n => Math.floor(Math.random() * n)


const pluralizeIf = (word, isPlural) => isPlural ? `${word}s` : wo


const randomElement = (array) => array[randomInt(array.length)]


const cases = ["nominative", "accusative"]
const numbers = ["singular", "plural"]


const generateRandomCardData = (adjectives, nouns) => ({
  adjective: randomElement(adjectives),
  noun: randomElement(nouns),
  grammaticalCase: randomElement(cases),
  number: randomElement(numbers)
})
```

```
const formatCard =
  (adjective, noun, number, grammaticalCase,
   correctAdjective, correctNoun) =>
  `${adjective.word} ${pluralizeIf(noun.word, number === 'plural')

  Provide the ${number} ${grammaticalCase} case.

  Correct answer: ${correctAdjective} ${correctNoun}`

const generateCard = (adjectives, nouns) => {
  const { adjective, noun, grammaticalCase, number } =
        generateRandomCardData(adjectives, nouns)

  const correctNoun = noun[number][grammaticalCase]
  const correctAdjective = adjective[grammaticalCase]
                                    [number]
                                    [noun.gender.toLowerCase()]

  return formatCard(adjective, noun, number, grammaticalCase,
                    correctAdjective, correctNoun)
}
```

Clarity is increased in a few ways. Distilling `generateCard` into three top-level chunks allows readers to quickly identify the overall policy.

- Generate random flash card elements using the adjectives and nouns passed in.

- Determine the correct answers to the challenge on the flash card by dereferencing the noun and adjective structures passed in.
- Return a formatted card containing all the relevant information.

We can immediately understand and trust each supporting function or piece of data in this solution. We might not even need to look at any supporting code or data.

The separate functions enhance clarity by allowing us to focus on small, understandable chunks. And while it would help to see the data structures involved (individual noun and adjective objects specifically), every statement otherwise stands on its own. No statements demand a comment. Comments would only increase the development, comprehension, and maintenance costs for this code.

As for understanding the data structures, that's what unit tests are for. Still, we might make a case for extracting a couple of functions to get just an ounce more isolation from implementation specifics in `generateCard`:

```
const generateCard = (adjectives, nouns) => {
  const { adjective, noun, grammaticalCase, number } =
    generateRandomCardData(adjectives, nouns)

  const correctNoun = selectNoun(noun, number, grammaticalCase)
  const correctAdjective =
    selectAdjective(adjective, grammaticalCase, number, noun)

  return formatCard(
    adjective,
    noun,
```

```
    number,
    grammaticalCase,
    correctAdjective,
    correctNoun)
}
```

We also spotted opportunities to cohere seemingly disparate elements, by introducing a new abstraction—the notion of an object that captures the underlying data elements of a random card. Yet there's still something amiss with our function. The excessive number of parameters returned and passed about not only muddles the solution, but also decreases its concision.

Let's make a final cleanup pass.

```
const generateRandom = (adjectives, nouns) => ({
  adjective: randomElement(adjectives),
  noun: randomElement(nouns),
})


const formatCard =
(phrase, caseAgreement, correctAdjective, correctNoun) =>
  `${phrase.adjective.word} ${pluralizeIf(
                                phrase.noun.word,
                                caseAgreement.number === 'plural')


  Provide the ${caseAgreement.number} ` +
              `${caseAgreement.grammaticalCase} case.


  Correct answer: ${correctAdjective} ${correctNoun}`
```

```
const selectNoun = (noun, caseAgreement) =>
  noun[caseAgreement.number][caseAgreement.grammaticalCase]

const selectAdjective = (adjective, gender, caseAgreement) =>
  adjective[caseAgreement.grammaticalCase]
           [caseAgreement.number]
           [gender.toLowerCase()]

const cases = ["nominative", "accusative"]
const numbers = ["singular", "plural"]

const generateCaseAgreementChallenge = () => ({
  grammaticalCase: randomElement(cases),
  number: randomElement(numbers)
})

const generateCard = (adjectives, nouns) => {
  const phrase = generateRandom(adjectives, nouns)
  const caseAgreement = generateCaseAgreementChallenge()

  const correctAdjective = selectAdjective(
    phrase.adjective, phrase.noun.gender, caseAgreement)
  const correctNoun = selectNoun(phrase.noun, caseAgreement)

  return formatCard(
    phrase, caseAgreement, correctAdjective, correctNoun)
}
```

Slightly more-thoughtful data groupings make `generateCard` seem less of a haphazard bustle of wayward local variables. Each of its handful of steps emphasizes the key elements involved. We also reworked `selectAdjective` to accommodate a gender parameter instead of a noun. Our choice emphasizes that a noun's gender, specifically, is key to determining the proper adjective.

Code Duplication

Being concise in code means implementing concepts once and only once. We often find small chunks of implementation detail replicated through a module or codebase—usually from a couple to a handful of lines that are oft repeated. Here's a commonly found example—repetitive error handling:

```
export const postItem = (request, response) => {
  const itemDetails = retrieveItem(request.body.upc)
  if (!itemDetails) {
    response.status = 400
    const errorMessage = 'unrecognized UPC code';
    logError(errorMessage)

    return response.send({error: errorMessage})
  }


  const checkout = Checkouts.retrieve(request.params.id)
  if (!checkout) {
    response.status = 400
    const errorMessage = 'nonexistent checkout';
    logError(errorMessage)
    return response.send({error: errorMessage})
  }
```

```
  }

  const newCheckoutItem = Checkouts.addItem(checkout, itemDetails)

  sendResponse(response, newCheckoutItem, 201)
}
```

Repetition increases costs for a number of reasons. The essence of the controller function `postItem` is obscured by repetitious error handling code. This duplication also decreases clarity, by mixing implementation details into policy flow.

```
const sendRequestError = (response, message) => {
  response.status = 400
  logError(errorMessage)
  response.send({error: message})
}

export const postItem = (request, response) => {
  const itemDetails = retrieveItem(request.body.upc)
  if (!itemDetails)
    return sendRequestError(response, 'unrecognized UPC code')

  const checkout = Checkouts.retrieve(request.params.id)
  if (!checkout)
    return sendRequestError(response, 'nonexistent checkout')

  const newCheckoutItem = Checkouts.addItem(checkout, itemDetails)
```

```
  sendResponse(response, newCheckoutItem, 201)
}
```

Often, eliminating duplication involves replacing concrete details with abstractions. It’s generally a win-win-win-win situation (increased conciseness, clarity, and ease of confirmability, plus a springboard to increased cohesion), but we don’t go overboard by obsessing over two lines of code that happen to look the same. We seek to find and eradicate duplicate implementations of concepts, not incidentally common lines of code.

Duplication fosters many increases in cost/effort:

- Time to read and comprehend cost
- Time to search code in general (we’re always wading through increased amounts of irrelevant results)
- Time to test
- Effort to find all points of change for a replicated concept
- Risk of not finding all necessary points of change
- Time to analyze whether variances found in duplicated code are deliberate or if they represent defects
- Effort to change duplicated code
- Cost to refactor code
- Propagation of defect costs, when common code is defective

We’ve worked on many systems that were two-to-three times as large as they might have been due to rampant code duplication. Oh, the days of getting paid by the line of code! (Just kidding. It never happened for any of us.)

One example included an operating room scheduling system, which by definition needed to do a lot of work with dates and timestamps. Among many other amusements, we found a common five-line piece of (old-school) Java date handling logic. These five lines were repeated in over fifty places throughout the codebase. Our costs increased dramatically when we sought to replace the old-school logic with a modern library.

Extract-and-move is once again our trustworthy workhorse for tackling duplication problems.

- Identify a clump of implementation detail that represents a singular concept.
- Extract it to a function.
- Focus on the new function in isolation. Move it to another module if appropriate.

As awareness of the existence of a new module increases, we start to spot more opportunities to move additional related behaviors into it. The cohesion of our modules increases as a result.

Conciseness Nits

Every unnecessary token adds to the clutter of our code. We can and should often use things like idiomatic constructs and syntactic sugar to reduce the amount of unnecessary code that we must scan and mentally assemble. A few pet nits of ours:

Return Boolean Expressions

Learning a language well means we must not only learn its syntax, but also learn the most concise way of representing common concepts. Some of these streamlined constructs might go so far as to be classified as idiomatic. (Yes, we’ve

used that word a number of times; we guess we should finally define it: Idiomatic code isn’t obvious the first time you see it.)

A ternary operator is idiomatic: In an isolated context, it doesn’t inherently provide all the information you need to immediately decipher it. The second or third time you see it, however, you’ll know how to interpret it—it has become obvious to you. It’s like riding a bicycle; you won’t likely ever need to be reminded after that.

Sometimes we see anti-idioms:

```
const hasTitle = book => {
  if (book.title !== null) {
    return true
  } else {
    return false
  }
}
```

Anti-idiom? Despite the fact that this construct demands five precious vertical source lines, we usually digest it as a single chunk when we see it (because unfortunately, we see it a lot). Sometimes we even notice when a chucklehead has sneakily reversed the order of the `true` and `false` returns.

We rarely say that things are flat-out wrong in programming, but this is one of those cases.

Simply (and we don’t use that word lightly) return the boolean expression:

```
const hasTitle = book => book.title !== null
```

Unnecessary `Else`

Maybe we needn’t embrace the ternary operator—worse things than `if-else` statements pollute our coding world—but if we’re going to code a conditional return:

```
const pluralizeIf = (word, isPlural) => {
  if (isPlural) {
      return `${word}s`
  } else {
      return word
  }
}
```

we should at least have the decency to not include the unnecessary `else` keyword. An early return suffices:

```
const pluralizeIf = (word, isPlural) => {
  if (isPlural) {
    return `${word}s`;
  }
  return word
}
```

No, we really don’t need the braces, though odds are good our team has adopted a coding standard that insists on them. But it’s really just overprotective clutter. It’s hard to mess up if you start with a short, single-line `if` statement:

```
const pluralizeIf = (word, isPlural) => {
  if (isPlural) return `${word}s`
  return word
}
```

We write tests if we remain worried. In any case, there's honestly not much good reason to eschew a ternary for this example:

```
const pluralizeIf = (word, isPlural) => isPlural ? `${word}s` : wo
```

We get it, though, if you object on aesthetic grounds.

And So On

We could dedicate an entire book to clutter in code. Most of it, however, you'll figure out on your own. We don't seek clever code; we instead seek elegant code. Elegant code says exactly what it needs, without too few or too many words. It is both clear and concise.

Confirmability

Our continuous design journey requires that we know what our code is intended to do. Without that knowledge, any change is unsafe. A system may contain many thousands of behavioral units. An unnoticed change in behavior to any one of them can allow us to unleash a costly defect in production.

Fortunately, we can write thousands of small, fast unit tests that verify whether we've created any regressions. These tests can tell us within seconds the moment we broke something. We'll need other kinds of tests at higher levels, of course, such as end-to-end functional tests, performance tests, load tests, and contract tests. But if we want to move fast, we need to be able to rapidly create and manage tests around the unit implementations.

Fear Degrades Design

In contrast, the lack of such tests gradually slows us down. As the amount of (not unit-tested) code increases, our costs to verify it increase. We can manually step through only so many test cases.

We can write integration tests, but they're generally costlier to build and maintain. They also create a slower feedback loop. Finally, it's unrealistic to expect that you'll be able to cover all the thousands of intentional decisions and logic variants that went into the codebase by virtue of integration tests (which also can increase the challenge of pinpointing the defect when they do fail).

Without fast feedback in the form of unit tests, we relegate ourselves to that "it ain't broke, don't fix it" mentality. Our codebase degrades by definition. We don't have the confidence to ensure that the code remains cohesive, clear, and concise. Instead, we habitually slap out "first drafts": We get our code working, then immediately move on to the next thing without making any edits.

It seems fast at first, yet the results of fear-driven coding quickly become apparent in the codebase. Developers learn to do the worst possible thing to a system, rather than what they know to be the best thing for the system.

A real example from a high-performance, historically famous large system: Developers needed to create variants of behaviors existing within typically long (100+ lines) C++ member functions. Shipping defective versions of these existing behaviors could cost the company millions of dollars per minute.

As a result, developers habitually replicated (i.e., copied/pasted) entire member functions, then made their changes within these copies. They didn't change the original functions—no one wanted to be the one who broke code

that was already working. (“Why did you even touch that code?”)

Fear significantly increases code duplication and costs.

Conflated Units Imply Tough Tests

If we write code like the following `postCheckoutTotal` function, writing tests for it will be daunting:

```
export const postCheckoutTotal = (request, response) => {
  const checkoutId = request.params.id
  const checkout = Checkouts.retrieve(checkoutId)
  if (!checkout) {
    response.status = 400
    response.send({error: 'nonexistent checkout'})
    return
  }

  const messages = []
  const discount = checkout.member ? checkout.discount : 0

  let totalOfDiscountedItems = 0
  let total = 0
  let totalSaved = 0

  checkout.items.forEach(item => {
    let price = item.price
    const isExempt = item.exempt
    if (!isExempt && discount > 0) {
      const discountAmount = discount * price
```

```
  const discountedPrice = price * (1.0 - discount)

  // add into total
  totalOfDiscountedItems += discountedPrice

  let text = item.description
  // format percent
  const amount = parseFloat(
    (Math.round(price * 100) / 100).toString()).toFixed(2)
  const amountWidth = amount.length

  let textWidth = LineWidth – amountWidth
  messages.push(pad(text, textWidth) + amount)

  total += discountedPrice

  // discount line
  const discountFormatted = '-' + parseFloat(
    (Math.round(discountAmount * 100)/100).toString()).toFixed
  textWidth = LineWidth - discountFormatted.length
  text = `   ${discount * 100}% mbr disc`
  messages.push(`${pad(text, textWidth)}${discountFormatted}`)

  totalSaved += discountAmount
}
else {
  total += price
  const text = item.description
```

```
    const amount = parseFloat(
      (Math.round(price * 100) / 100).toString()).toFixed(2)
    const amountWidth = amount.length

    const textWidth = LineWidth - amountWidth
    messages.push(pad(text, textWidth) + amount)
  }
})

total = Math.round(total * 100) / 100

// append total line
const formattedTotal = parseFloat(
  (Math.round(total * 100) / 100).toString()).toFixed(2)
const formattedTotalWidth = formattedTotal.length
const textWidth = LineWidth - formattedTotalWidth
messages.push(pad('TOTAL', textWidth) + formattedTotal)

if (totalSaved > 0) {
  const formattedTotal = parseFloat(
    (Math.round(totalSaved * 100) / 100).toString()).toFixed(2)
  console.log(`formattedTotal: ${formattedTotal}`)
  const formattedTotalWidth = formattedTotal.length
  const textWidth = LineWidth - formattedTotalWidth
  messages.push(pad('*** You saved:', textWidth) + formattedTota
}

totalOfDiscountedItems =
```

```
    Math.round(totalOfDiscountedItems * 100) / 100


  totalSaved = Math.round(totalSaved * 100) / 100


  response.status = 200
  // send total saved instead
  response.send(
    { id: checkoutId,
      total,
      totalOfDiscountedItems,
      messages,
      totalSaved })
}
```

Capturing all the nuances and writing tests for all relevant cases might take a few hours. Also, there's a good possibility that we don't cover all the cases, which increases our risk of shipping defects.

Small, Isolated Units Imply Simple Tests

It is much easier to write small, simple tests as we craft the logic for assembling a checkout receipt. No doubt you can spot a few of the bits of duplication in the midst of `postCheckoutTotal`, as well as the number of opportunities for extracting conceptual units to new functions.

Here's an improved solution, a work in progress.

```
const sendSuccessResponse = (response, body) => {
  response.status = 200
  response.send(body)
}
```

```
}

const sendErrorResponse = (response, message) => {
  response.status = 400
  response.send({ error: message })
}


export const postCheckoutTotal = (request, response) => {
  const checkoutId = request.params.id
  const checkout = Checkouts.retrieve(checkoutId)
  if (!checkout)
    return sendErrorResponse(response, 'nonexistent checkout')

  const { totals, messages } = createReceipt(checkout)

  sendSuccessResponse(response, {
    id: checkoutId,
    messages,
    total: round2(totals.total),
    totalOfDiscountedItems: round2(totals.totalOfDiscountedItems),
    totalSaved: round2(totals.totalSaved)
  })
}
```

The function `postCheckoutTotal`, now a declaration of policy, can be covered with a couple of “end-to-end” tests.

The module `receipt.js` exposes code that can be directly and easily unit tested:

```
import { pad } from '../../util/stringutil'
import { calculateTotal,
         calculateTotalOfDiscountedItems,
         calculateTotalSaved } from './checkout-model'

const LineWidth = 45
const Indent = `   `

export const round2 = amount => Math.round(amount * 100) / 100

const formatAmount = amount => parseFloat(round2(amount)).toFixed(

const shouldApplyDiscount = (checkout, item) =>
  !item.exempt && memberDiscountPct(checkout) > 0

const calculateDiscountAmount = (checkout, item) =>
  memberDiscountPct(checkout) * item.price

const memberDiscountPct =
  checkout => checkout.member ? checkout.discount : 0

const lineItem = (text, lineAmount) => {
  const amount = formatAmount(lineAmount)
  const amountWidth = amount.length
  const textWidth = LineWidth - amountWidth
  return pad(text, textWidth) + amount
}
```

```
export const createReceiptMessages = (checkout, totals) => {
  const messages = []
  checkout.items.forEach(item => {
    messages.push(lineItem(item.description, item.price))
    if (shouldApplyDiscount(checkout, item)) {
      messages.push(lineItem(
        `${Indent}${memberDiscountPct(checkout) * 100}% mbr disc`,
        -calculateDiscountAmount(checkout, item)))
    }
  })
  messages.push(lineItem('TOTAL', round2(totals.total)))
  if (totals.totalSaved > 0)
    messages.push(lineItem('*** You saved:', totals.totalSaved))
  return messages
}

export const createReceipt = checkout => {
  const totals = ({
    total: calculateTotal(checkout),
    totalSaved: calculateTotalSaved(checkout),
    totalOfDiscountedItems: calculateTotalOfDiscountedItems(checko
  })
  return {
    totals,
    messages: createReceiptMessages(checkout, totals)
  }
}
```

By separating out `receipt.js`, we've also increased cohesion. The `postCheckoutTotal` function is primarily concerned with orchestrating the creation of a response to a request. It delegates the calculation specifics of assembling a receipt and totals to the `receipt.js` module.

The `receipt.js` still includes a few utility/helper functions that could be again moved, then tested in isolation (notably `round2` and `formatAmount`).

Here's the final cohesive module, `checkout-model.js`:

```
export const calculateTotal = checkout => {
  let total = 0
  checkout.items.forEach(item => {
    if (!item.exempt && (checkout.member ? checkout.discount : 0) :
      total += item.price * 1.0 –
                 (checkout.member ? checkout.discount : 0))
    Else
      total += item.price
  })
  return total
}


export const calculateTotalSaved = checkout => {
  let totalSaved = 0
  checkout.items.forEach(item => {
    if (!item.exempt && (checkout.member ? checkout.discount : 0) :
      totalSaved +=
        (checkout.member ? checkout.discount : 0) * item.price
  })
  return totalSaved
```

```
}

export const calculateTotalOfDiscountedItems = checkout =>
  checkout.items
    .filter(item => !item.exempt &&
                    (checkout.member ? checkout.discount : 0) > 0)
    .reduce((total, item) => total + item.price *
       (1.0 - (checkout.member ? checkout.discount : 0)), 0)
```

Now that it is isolated, we can focus on improving clarity and conciseness within these calculate functions. We can put tests in place in short order. It took ten minutes to create these tests for `calculateTotal`:

```
import {calculateTotal} from "./checkout-model";


describe('calculate total', () => {
  it('is zero when no items exist', () => {
    const checkout = {
      items: []
    }

    const result = calculateTotal(checkout)

    expect(result).toBe(0)
  })

  it('is price of single undiscounted item', () => {
    const checkout = {
```

```
    items: [{ price: 42 }]
  }

  const result = calculateTotal(checkout)

  expect(result).toBe(42)
})

it('discounts item when member attached & item not exempt', () =
  const checkout = {
    member: {},
    discount: 0.1,
    items: [{
      exempt: false,
      price: 100
    }]
  }

  const result = calculateTotal(checkout)

  expect(result).toBe(90)
})

it('does not discount item when exempt', () => {
  const checkout = {
    member: {},
    discount: 0.1,
    items: [{
```

```
      exempt: true,
      price: 100
    }]
  }

  const result = calculateTotal(checkout)

  expect(result).toBe(100)
})

it('does not discount item when no member attached', () => {
  const checkout = {
    discount: 0.1,
    items: [{
      exempt: true,
      price: 100
    }]
  }

  const result = calculateTotal(checkout)

  expect(result).toBe(100)
})

it('totals discounted and non-discounted items', () => {
  const checkout = {
    discount: 0.1,
    member: {},
```

```
      items: [
        { exempt: true, price: 5 },
        { exempt: true, price: 10 },
        { exempt: false, price: 100 },
        { exempt: false, price: 200 },
      ]
    }

    const result = calculateTotal(checkout)

    expect(result).toBe(5 + 10 + 90 + 180)
  })
})
```

The tests gave us the confidence to fearlessly refactor `calculateTotal` to a clearer solution:

```
const memberDiscountPct = checkout =>
  checkout.member ? checkout.discount : 0

const shouldApplyDiscount = (checkout, item) =>
  !item.exempt && memberDiscountPct(checkout) > 0

const calculateDiscountedPrice = (checkout, item) =>
  item.price * (1.0 - memberDiscountPct(checkout))

export const calculateTotal = checkout => {
  const totalDiscountable = checkout.items
    .filter(item => shouldApplyDiscount(checkout, item))
    .reduce((total, item) => total +
```

```
    .reduce((total, item) => total +
              calculateDiscountedPrice(checkout, item), 0);
  const totalNonDiscountable = checkout.items
    .filter(item => !shouldApplyDiscount(checkout, item))
    .reduce((total, item) => total + item.price, 0);
  return totalDiscountable + totalNonDiscountable;
}
```

It looks like our refactoring increased the lines of code, for `calculateTotal` at least. However, the overall number of lines in the module decreases by a handful after we made similar changes to the other calculation functions as well.

When Should We Ensure Confirmability?

Many developers prefer to write a significant chunk of code, then later come back and meet what they view as their obligation to achieve code coverage mandates. The mandate for code coverage—the measure of what percent of code is exercised by tests—becomes a self-fulfilling prophecy: We get what we ask for. If a development team is mandated a minimum coverage metric of 75%, we'll get that and no more.

Seventy-five percent coverage doesn't sound bad until you invert it: One-quarter of our system has insufficient unit tests. They're also often the more complex parts of our system: They're harder to test, and when meeting a metric we don't really believe in, we'll do it in the most expeditious way we can imagine.

The reason so much of our code is harder to test is because testing it wasn't a paramount concern while we were coding it. Our primary interest was to just get the damn thing working. As a result, the complex parts and even the not-so-

complex parts are tougher to test. Our code is not easily confirmable.

Writing tests after the fact is harder because the existing code

- Lacks clarity. It is overly complex, because it wasn't edited. It's tougher to figure out the basic question of what the functions do.
- Lacks cohesion. It becomes increasingly difficult to discern all the actual developer intents, because they aren't all in one place, and they aren't decoupled from other intents.
- Lacks conciseness. It contains excessive code, as well as excessively repeated constructs that must be retested in multiple contexts.
- Contains deep dependency chains. As a result, tests demand increased effort to set up the appropriate data/context. The existence of what are typically private dependencies means that we must reshape the code to allow tests to inject the dependencies.

One more problem with that 75% or 80% or whatever code metric: The remaining portion of our code is a 20%+ blob of "who really knows what this stuff does." Good tests not only provide us with the ability to edit code without fear, but they also document all the developer intents.

If writing tests later creates so many problems (or, more positively stated, doesn't help us sufficiently enough), the obvious answer is to start with tests. With each new behavior we must add, we write a test that captures that behavior. Our test's name summarizes that behavior (which helps us later find the behaviors we seek). Our test's code provides an example that pins down the design for how we effect that behavior.

Our code coverage becomes a self-fulfilling prophecy: We attain complete coverage on all the behaviors we needed and designed into the system. We get high confidence to continuously shape its design to be clear, cohesive, and concise. Our system is eminently confirmable.

You might be wondering about what that coverage metric mandate should be. The right answer: (a) Mandates are a bad idea, and (b) the coverage number is never what's important. What's important is that developers are sold on the value of delivering what we know to be working software.

Cohesion

It was infeasible to get to this last C criterion for design without previously mentioning cohesion. Our continuous design criteria necessarily intertwine to support each other, and at times they can be at odds with one another. Increasing clarity might require diminishing conciseness, for example.

But here we are, and cohesion deserves its own focused discussion.

We're building `Czecher`, a flash card application designed to help us ingrain challenging aspects of the Czech language. As a first step, we've created the module `words.js` to allow us to add new nouns to our collection of words to practice:

```
import { retrieveWord } from '../prompts/languageClient'


const definitions = {}


export const clearWords = () =>
  Object.keys(definitions).forEach(key => delete definitions[key])
```

```
export const loadDefinition = (word, definition) =>
  definitions[word] = { ...definition, word }

export const addWord = async word => {
  if (definitions[word]) return

  const definition = await retrieveWord(word)
  definitions[word] = { ...definition, word }

}

export const definition = word => definitions[word]

export const allDefinitions = () =>
  Object.values(definitions)
```

The `addWord` function takes an English word as an argument (e.g., “dog”). If a definition has already been loaded for the word, the function does nothing. Otherwise, `addWord` calls `retrieveWord` to look up information using a third-party resource (OpenAI in this case). The resulting definition is added to a local store. Calling the definition function with a word returns the detailed information.

Some tests might add a little color:

```
import { when } from 'jest-when'
import { addWord, clearWords, definition,
         allDefinitions, loadDefinition }
  from './words'
import { retrieveWord } from '../prompts/languageClient'
```

```
jest.mock('../prompts/languageClient')

describe('addWord', ()  => {
  const orangeDefinition = {
    word: 'orange',
    gender: 'MI',
    singular: { nominative: 'pomeranč', accusative: 'pomeranč' },

    plural: { nominative: 'pomeranče', accusative: 'pomeranče' }
  }
  const dogDefinition = {
    word: 'dog',
    gender: 'MA',
    singular: { nominative: 'pes', accusative: 'psa' },
    plural: { nominative: 'psi', accusative: 'psy' }
  }

  beforeEach(() => {
    clearWords()
    jest.resetAllMocks()
  })

  it('retrieves and saves definition on add', async () => {
    when(retrieveWord).calledWith('orange')
      .mockResolvedValueOnce(orangeDefinition)

    await addWord('orange')
```

```
    expect(definition('orange')).toEqual(orangeDefinition)
  })

  it('does nothing if already added', async () => {
    loadDefinition('orange', orangeDefinition)

    const word = definition('orange')


    expect(word).toEqual(orangeDefinition)
    expect(retrieveWord).not.toHaveBeenCalled()
  })

  const add = async (word, definition) => {
    when(retrieveWord).calledWith(word)
        .mockResolvedValueOnce(definition)
    await addWord(word)
  }

  it('persists multiple words', async () => {
    await add('orange', orangeDefinition)
    await add('dog', dogDefinition)

    const definitions = allDefinitions()

    expect(definitions).toEqual([orangeDefinition, dogDefinition])
  })
})
```

The `words` module is our first incremental step toward an MVP.[3] As a quick measure, we chose to persist the words in an in-memory JavaScript object (`{}`), where the keys are the words and the values provide the language detail we need. The constant `orangeDefinition` provides one example.

[3] Minimum Viable Product.

Here's our continuous design report card.

- Clarity: Pass. All the operations appear in short, JavaScript-idiomatic functions, without any C-for-Cleverness (for which we would immediately get a failing mark).
- Confirmability: Pass. All code was driven into existence by behavioral tests.
- Conciseness: Pass. The code seems about as short as it could get.
- Cohesion: Fail. Wait, fail?

Our intent is to replace the JavaScript object with a proper persistence mechanism. Unfortunately, we've conflated our little bit of definition-management logic with logic supporting storage of the definitions. When we do change, we'll have to open up the `words` module and poke through most of its methods to update implementations.

You'll likely recognize that we're describing a Single Responsibility Principle (SRP) violation. SRP and cohesion both say the same thing, minus some nuances. Cohesion is how well the elements of a module align to the same purpose.

Had we adhered to a cohesive solution, we might have found it a bit easier to build the solution in the first place. We'd also have been ready to immediately accommodate

changes, rather than have to prefactor our code to support the change.

We'll refactor the code now, to demonstrate how a cohesive solution increases the clarity of the code.

```
import { retrieveWord } from '../prompts/languageClient'
import * as Data from '../persistence/database'

export const clearWords = () =>
  Data.deleteAll()

export const loadDefinition = (word, definition) =>
  Data.add(word, definition)

export const addWord = async word => {
  if (Data.containsKey(word)) return

  const definition = await retrieveWord(word)
  Data.add(word, { word, ...definition})
}

export const definition = word => Data.get(word)

export const allDefinitions = () => Data.allValues()
```

The implementation specifics around JavaScript objects have been replaced with abstractions: `add`, `get`, `allValues`, `containsKey`, and `deleteAll`. The underlying implementation could change from an object, to a list, to interaction with a third-party key/value store or relational database. The now-

cohesive `words` module wouldn’t be impacted by these changes.

The module `database.js` provides an interface that abstracts the notion for persisting data, with nothing exported that betrays the underlying data structure. Future changes to its implementation don’t impact other code in the system.

```
const data = {}


export const containsKey = key => data[key]


export const deleteAll = () =>
  Object.keys(data).forEach(key => delete data[key])


export const add = (key, value) =>
  data[key] = value


export const get = word => data[word]


export const allValues = () => Object.values(data)
```

When Else Do We Design?

With a continuous design mentality, we seek to keep costs low by reviewing our code’s adherence to the four Cs, a collection of code criteria that we can continuously consult. When do we do that? Why, continuously of course, or at least continually.

But we don’t only consider design when we’re writing production code. It’s a part of virtually every step of software development.

Up-front Design

We initiate a project by determining what new capabilities our product will bring to its users. Necessarily this requires some level of understanding about the design of the system—what it accomplishes and what its constraints are, for example.

As part of answering our analysis and design efforts, we might first produce summary diagrams that capture salient elements of the system's current design. We would then also create speculative sketches for what the design will need to look like in order to accommodate the desired new behaviors.

Stakeholders usually want to know when the project will be done at this point in the planning process. Estimates are a design consideration: Not only must we have a sense of how much time it would take to develop a new feature, but we also must know the extent to which the current design resists the new feature. Ideally, the amount of extra effort needed is "zero," which can be true if we've adhered to the four Cs well enough.

The estimates themselves might feed back into the project plan. The powers that be might decide to trim their wishlist due to unexpectedly high estimates. Given these changes to their desired scope, we reconsider the design and provide updated estimates.

Our project estimates at this time cover larger initiatives that might take weeks or months to complete. The estimates are high level as a result; we invest enough time in them to get a ballpark sense of the initiative. We might use relative sizes—small, medium, large, extra-large—to figure out which ones we can guarantee we'll deliver in the next quarter.

As we ready for development, we seek to slice these larger challenges into thinner pieces of work that we can start and deliver within shorter periods of time (perhaps hours or days). An understanding of the system's design is once again required at this point, to ensure that our ideas around slicing up work align with the realities of the system.

Readying for Work

As we get ready to code a feature, the product folk (product owners, product managers, business analysts, UX designers, and others) are working to ensure that they've captured all the actual specifics needed for us to get started. When they meet with us to describe the new feature, we ask questions, and we dig a bit again into the system's design.

Sometimes we uncover the devil when the product people reveal the details for the feature. We discover that things aren't as ideal as we'd speculated earlier. It's also possible that the system has changed enough in the interim to make things more difficult.

Sometimes answers to our questions reveal hidden complexity. "Oh, I didn't think that bell or whistle would be a big deal," they might tell us. It might not be, from their perspective; from ours, it might be something that will be very difficult to fit into the existing design.

Starting Work

We break down our work first into smaller behavioral units. If our work is to add support for adverbs to the `Czecher` app, we might break that effort into the following smaller slices.

- Support adding a single adverb to the word list.
- Bulk-load a CSV file of adverbs.

- Add flash card examples containing adverbs.

Those slices might still be a little too large. We might split “support adding an adverb” into a few technical challenges.

- Create and incorporate LLM prompt text to specify the format for adverbs.
- Add persistence support for adverbs.
- Rework card building logic to provide examples involving adverbs.

Such decomposition necessarily involves an awareness of design, and may also spur a discussion around how the current and/or speculative design must change.

Doing Work

We write tests to drive in each unit behavior. A unit test describes the behavior and provides a “live” example demonstrating how clients interact with the system to effect that behavior. The test, as a result, is our primary mechanism for designing and documenting the interface to the system.

For each test we write, we code the behavior that we hope realizes the need specified in the test. We run our tests, correcting the code (i.e., the design) until the tests pass. We revisit the coded behavior, and we edit it for clarity, conciseness, and cohesion before moving on to the next test.

We ship when we’ve written tests for all the behaviors, as well as gotten all the tests to pass.

Design is omnipresent.

Chapter 22. Concurrency

By Brett L. Schuchert



“Objects are abstractions of processing. Threads are abstractions of schedule.”

—James O. Coplien[1]

[1] For example, [PPP02], http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod, and https://en.wikipedia.org/wiki/SOLID_(object-oriented_design) (or just google “SOLID”).

> Future Bob: Everything in this chapter applies equally to services and microservices.

Writing clean concurrent programs is hard—very hard. It is much easier to write code that executes in a single thread. It is also easy to write multithreaded code

that looks fine on the surface but is broken at a deeper level. Such code works fine until the system is placed under stress.

In this chapter, we discuss the need for concurrent programming and the difficulties it presents. We then present several recommendations for dealing with those difficulties and writing clean concurrent code. Finally, we conclude with issues related to testing concurrent code.

Clean concurrency is a complex topic, worthy of a book by itself. Our strategy in *this* chapter is to present an overview.

Why Concurrency?

Concurrency is a decoupling strategy. It helps us decouple *what* gets done from *when* it gets done. In single-threaded applications, *what* and *when* are so strongly coupled that the state of the entire application can often be determined by looking at the stack backtrace. A programmer who debugs such a system can set a breakpoint, or a sequence of breakpoints, and *know* the state of the system by which breakpoints are hit.

Decoupling *what* from *when* can dramatically improve both the throughput and structure of an application. From a structural point of view, the application looks like many little collaborating computers rather than one big main loop. This can make the system easier to understand and offers some powerful ways to separate concerns.

Consider, for example, the standard “servlet” model of Web applications. These systems run under the umbrella of a Web server that *partially* manages concurrency for you. The servlets are executed asynchronously whenever Web requests come in. The servlet programmer does not have to manage all the incoming requests. *In principle*, each servlet execution lives in its own little world and is decoupled from all the other servlet executions.

Of course, if it were that easy, this chapter wouldn’t be necessary. In fact, the decoupling provided by Web servers is far less than perfect. Servlet programmers have to be very aware, and very careful, to make sure their concurrent programs are correct. Still, the structural benefits of the servlet model are significant.

But structure is not the only motive for adopting concurrency. Some systems have response time and throughput constraints that require hand-coded concurrent solutions.

For example, consider a single-threaded information aggregator that acquires information from many different websites and merges that information into a daily summary. Because this system is single threaded, it hits each website in turn, always finishing one before starting the next. The daily run needs to be executed in less than 24 hours. However, as more and more websites are added, the time grows until it takes more than 24 hours to gather all the data. The single thread involves a lot of waiting at Web sockets for IO to complete. We could improve the performance by using a multithreaded algorithm that hits more than one website at a time.

Or consider a system that handles one user at a time and requires only one second of time per user. This system is fairly responsive for a few users, but as the number of users increases, the system's response time increases. No user wants to get in line behind 150 others! We could improve the response time of this system by handling many users concurrently.

Or consider a system that interprets large data sets but can only give a complete solution after processing all of them. Perhaps each data set could be processed on a different computer so that many data sets are being processed in parallel.

Myths and Misconceptions

And so there are compelling reasons to adopt concurrency. However, as we said before, concurrency is *hard*. If you aren't very careful, you can create some very nasty situations. Consider these common myths and misconceptions.

- **Concurrency always improves performance.** Concurrency can *sometimes* improve performance, but only when there is a lot of wait time that can be shared between multiple threads or multiple processors. Neither situation is trivial.
- **Design does not change when writing concurrent programs.** In fact, the design of a concurrent algorithm can be remarkably different from the design of a single-threaded system. The decoupling of *what* from *when* usually has a huge effect on the structure of the system.
- **Understanding concurrency issues is not important when working with a Web server.** In fact, you'd better know just what your server is doing and how to guard against the issues of concurrent update and deadlock, described later in this chapter.

Here are a few more balanced sound bites regarding writing concurrent software.

- Concurrency incurs some overhead, both in performance as well as writing additional code.
- Correct concurrency is complex, even for simple problems.
- Concurrency bugs aren't usually repeatable, so they are often ignored as one-offs[2] instead of actual defects.

[2] Cosmic rays, glitches, and so on.

- Concurrency often requires a fundamental change in design strategy.

Challenges

What makes concurrent programming so difficult? Consider the following trivial class:

```
public class X {
   private int lastIdUsed;
```

```
   public int getNextId() {
        return ++lastIdUsed;
    }
}
```

Let's say we create an instance of `X`, set the `lastIdUsed` field to `42`, and then share the instance between two threads. Now suppose that both of those threads call the method `getNextId()`; there are three possible outcomes:

1. Thread one gets the value `43`, thread two gets the value `44`, `lastIdUsed` is `44`
2. Thread one gets the value `44`, thread two gets the value `43`, `lastIdUsed` is `44`
3. Thread one gets the value `43`, thread two gets the value `43`, `lastIdUsed` is `43`

The surprising third result occurs when the two threads step on each other. This happens because there are many possible paths that the two threads can take through that one line of code, and some of those paths generate incorrect results. How many different paths are there? To really answer that question, we need to understand what the compiler does with the generated bytecode, and understand what the Java memory model considers to be atomic.

A quick answer, working with just the generated bytecode, is that there are 12,870 different possible execution paths for those two threads executing within the `getNextId` method. If the type of `lastIdUsed` is changed from `int` to `long`, the number of possible paths increases to 2,704,156. Of course, most of those paths generate valid results. The problem is that some of them don't.

Concurrency Defense Principles

What follows is a series of principles and techniques for defending your systems from the problems of concurrent code.

Single Responsibility Principle

The SRP states that a given method/class/component should have a single reason to change. Concurrency design is complex enough to be a reason to change in its own right and therefore deserves to be separated from the rest of the code. Unfortunately, it is all too common for concurrency implementation details to be embedded directly into other production code. Here are a few things to consider.

- Concurrency-related code has its own life cycle of development, change, and tuning.
- Concurrency-related code has its own challenges, which are different from and often more difficult than nonconcurrency-related code.
- The number of ways in which miswritten concurrency-based code can fail makes it challenging enough without the added burden of surrounding application code.

Recommendation: Keep your concurrency-related code separate from other code.

Corollary: Limit the Scope of Data

As we saw, two threads modifying the same field of a shared object can interfere with each other, causing unexpected behavior. One solution is to use something like Java's `synchronized` keyword to protect a critical section in the code that uses the shared object. It is important to restrict the number of such critical sections. The more places shared data can get updated, the more likely

- You will forget to protect one or more of those places—effectively breaking all code that modifies that shared data.
- There will be duplication of effort required to make sure everything is effectively guarded (a violation of DRY[3]).

[3] [Prag].

- It will be difficult to determine the source of failures, which are already hard enough to find.

Recommendation: Take data encapsulation to heart; severely limit the access of any data that may be shared.

> Future Uncle Bob: Consider using a functional language, or at least a functional style, when writing concurrent modules.

Corollary: Use Copies of Data

A good way to avoid shared data is to avoid sharing the data in the first place. In some situations, it is possible to copy objects and treat them as read-only. In other cases, it might be possible to copy objects, collect results from multiple threads in these copies, and then merge the results in a single thread.

If there is an easy way to avoid sharing objects, the resulting code will be far less likely to cause problems. You might be concerned about the cost of all the extra object creation. It is worth experimenting to find out if this is, in fact, a problem. However, if using copies of objects allows the code to avoid synchronizing, the savings in avoiding the intrinsic lock will likely make up for the additional creation and garbage collection overhead.

Corollary: Threads Should Be as Independent as Possible

Consider writing your threaded code such that each thread exists in its own world, sharing no data with any other thread. Each thread processes one client request, with all of its required data coming from an unshared source and stored as local variables. This makes each of those threads behave as if it were the only thread in the world and there were no synchronization requirements.

For example, classes that subclass from `HttpServlet` receive all of their information as parameters passed in to the `doGet` and `doPost` methods. This makes each `Servlet` act as if it has its own machine. So long as the code in the `Servlet` uses only local variables, there is no chance that the `Servlet` will cause synchronization problems. Of course, most applications using `Servlets` eventually run into shared resources such as database connections.

Recommendation: Attempt to partition data into independent subsets that can be operated on by independent threads, possibly in different processors.

Know Your Language and Library

Here are a few more tips.

- Use provided thread-safe collections.
- Use the threading frameworks (e.g., `Executor` in Java) for executing unrelated tasks.
- Use nonblocking solutions when possible.
- Know which library modules are and are not thread safe.

Thread-Safe Collections

When Java was young, Doug Lea wrote the seminal book[4] *Concurrent Programming in Java*. Along with the book, he developed several thread-safe collections, which later became part of the JDK in the `java.util.concurrent` package. The collections in that package are safe for multithreaded situations and they perform well. In fact, the `ConcurrentHashMap` implementation performs better than `HashMap` in nearly all situations. It also allows for simultaneous concurrent reads and writes, and it has methods supporting common composite operations that are otherwise not thread safe. If Java is the deployment environment, start with `ConcurrentHashMap`.

[4] [Lea99].

There are several other kinds of classes added to support advanced concurrency design. Here are a few examples in Java:

`ReentrantLock`	A lock that can be acquired in one method and released in another.
`Semaphore`	An implementation of the classic semaphore, a lock with a count.
`CountDownLatch`	A lock that waits for a number of events before releasing all threads waiting on it. This allows all threads to have a fair chance of starting at about the same time.

Recommendation: Review the classes available to you. In the case of Java, become familiar with `java.util.concurrent`, `java.util.concurrent.atomic`, and `java.util.concurrent.locks`.

Know Your Execution Models

There are several different ways to partition behavior in a concurrent application. To discuss them we need to understand some basic definitions.

Bound resources	Resources of a fixed size or number used in a concurrent environment. Examples include database connections and fixed-size read/write buffers.
Mutual exclusion	Only one thread can access shared data or a shared resource at a time.
Starvation	One thread or a group of threads is prohibited from proceeding for an excessively long time—or forever. For example, always letting fast-running threads through first could starve out longer-running threads if there is no end to the fast-running threads.
Deadlock	Two or more threads are waiting for each other to finish. Each thread has a resource that the other thread requires, and neither can finish until it gets the other resource.
Livelock	Threads are in lockstep, each trying to do work but finding another "in the way." Due to resonance, threads continue trying to make progress but are unable to for an excessively long time—or forever.

Given these definitions, we can now discuss the various execution models used in concurrent programming.

Producer-Consumer[5]

[5] http://en.wikipedia.org/wiki/Producer-consumer

One or more producer threads create some work and place it in a buffer or queue. One or more consumer threads acquire that work from the queue and complete it. The queue between the producers and consumers is a bound resource. This means producers must wait for free space in the queue before writing and consumers must wait until there is something in the queue to consume. Coordination between the producers and consumers via the queue involves producers and consumers signaling each other. The producers write to the queue and signal that the queue is no longer empty. Consumers read from the queue and signal that the queue is no longer full. Both potentially wait to be notified when they can continue.

Readers-Writers[6]

[6] http://en.wikipedia.org/wiki/Readers-writers_problem

When you have a shared resource that primarily serves as a source of information for readers, but which is occasionally updated by writers, throughput is an issue. Emphasizing throughput can cause starvation and the accumulation of stale information. Allowing updates can impact throughput. Coordinating readers so that they do not read something a writer is updating, and vice versa, is a tough

balancing act. Writers tend to block many readers for a long period of time, thus causing throughput issues.

The challenge is to balance the needs of both readers and writers to satisfy correct operation, provide reasonable throughput, and avoid starvation. A simple strategy makes writers wait until there are no readers before allowing the writer to perform an update. If there are continuous readers, however, the writers will be starved. On the other hand, if there are frequent writers and they are given priority, throughput will suffer. Finding that balance and avoiding concurrent update issues is what the problem addresses.

Dining Philosophers[7]

[7] http://en.wikipedia.org/wiki/Dining_philosophers_problem

Imagine a number of philosophers sitting around a circular table. A fork is placed to the left of each philosopher. There is a big bowl of spaghetti in the center of the table. The philosophers spend their time thinking until they get hungry. Once hungry, they pick up the forks on either side of them and eat. A philosopher cannot eat unless he is holding two forks. If the philosopher to his right or left is already using one of the forks he needs, he must wait until that philosopher finishes eating and puts the forks back down. Once a philosopher eats, he puts both his forks back down on the table and waits until he is hungry again.

Replace “philosophers” with “threads” and “forks” with “resources” and this problem is similar to many enterprise applications in which processes compete for resources. Unless carefully designed, systems that compete in this way can experience deadlock, livelock, throughput, and efficiency degradation.

Recommendation: Most concurrent problems you will likely encounter will be some variation of these three problems. Study these algorithms and write solutions using them on your own so that when you come across concurrent problems, you’ll be more prepared to solve them.

Beware Dependencies between Synchronized Methods

Dependencies between synchronized methods cause subtle bugs in concurrent code. The Java language has the notion of `synchronized`, which protects an individual method. However, if there is more than one `synchronized` method on the same shared class, then your system may be written incorrectly.

Recommendation: Avoid using more than one method on a shared object.

There will be times when you must use more than one method on a shared object. When this is the case, there are three ways to make the code correct.

- **Client-based locking:** Have the client lock the server before calling the first method, and make sure the lock’s extent includes code calling the last method.

- **Server-based locking:** Within the server, create a method that locks the server, calls all the methods, and then unlocks. Have the client call the new method.
- **Adapted server:** Create an intermediary that performs the locking. This is an example of server-based locking, where the original server cannot be changed.

Keep Synchronized Sections Small

The `synchronized` keyword introduces a lock. All sections of code guarded by the same lock are guaranteed to have only one thread executing through them at any given time. Locks are expensive because they create delays and add overhead. So we don't want to litter our code with `synchronized` statements. On the other hand, critical sections[8] must be guarded. So we want to design our code with as few critical sections as possible.

[8] A critical section is any section of code that must be protected from simultaneous use for the program to be correct.

Some naive programmers try to achieve this by making their critical sections very large. However, extending synchronization beyond the minimal critical section increases contention and degrades performance.

Recommendation: Keep your synchronized sections as small as possible.

Writing Correct Startup and Shutdown Code Is Hard

Writing a system that is meant to stay alive and run forever is different from writing something that works for a while and then shuts down gracefully.

Graceful shutdown can be hard to get correct. Common problems involve deadlock, with threads waiting for a signal to continue that never comes.

For example, imagine a system with a parent thread that spawns several child threads and then waits for them all to finish before it releases its resources and shuts down. What if one of the spawned threads is deadlocked? The parent will wait forever, and the system will never shut down.

Or consider a similar system that has been *instructed* to shut down. The parent tells all the spawned children to abandon their tasks and finish. But what if two of the children were operating as a producer/consumer pair. Suppose the producer receives the signal from the parent and quickly shuts down. The consumer might have been expecting a message from the producer and be blocked in a state where it cannot receive the shutdown signal. It could get stuck waiting for the producer and never finish, preventing the parent from finishing as well.

Situations like this are not at all uncommon and apply equally to startup scenarios. So, if you must write concurrent code that involves starting up and shutting down gracefully, expect to spend much of your time getting those scenarios to happen correctly.

Recommendation: Think about startup and shutdown early, and get it working early. It's going to take longer than you expect. Review existing algorithms, because this is probably harder than you think.

Testing Threaded Code

Proving that code is correct is impractical. Testing does not guarantee correctness. However, good testing can minimize risk. This is all true in a single-threaded solution. As soon as there are two or more threads using the same code and working with shared data, things get substantially more complex.

Recommendation: Write tests that have the potential to expose problems, and then run them frequently, with different programmatic configurations and system configurations and load. If tests ever fail, track down the failure. Don't ignore a failure just because the tests pass on a subsequent run.

That is a whole lot to take into consideration. Here are a few, more fine-grained recommendations.

- Treat spurious failures as candidate threading issues.
- Get your nonthreaded code working first.
- Make your threaded code pluggable.
- Make your threaded code tunable.
- Run with more threads than processors.
- Run on different platforms.
- Instrument your code to try and force failures.

Treat Spurious Failures as Candidate Threading Issues

Threaded code causes things to fail that "simply cannot fail." Most developers do not have an intuitive feel for how threading interacts with other code (authors included). Bugs in threaded code might exhibit their symptoms once in a thousand, or a million, executions. Attempts to repeat the systems can be frustrating. This often leads developers to write off the failure as a cosmic ray, a hardware glitch, or some other kind of "one-off." It is best to assume that one-offs do not exist. The longer these one-offs are ignored, the more code is built on top of a potentially faulty approach.

Recommendation: Do not ignore system failures as one-offs.

Get Your Nonthreaded Code Working First

This may seem obvious, but it doesn't hurt to reinforce it. Make sure code works outside of its use in threads. Generally, this means creating simple modules that are called by your threads. Since these modules are not thread aware, they can be

tested outside of the threaded environment. The more of your system you can place in such modules, the better.

Recommendation: Do not try to chase down nonthreading bugs and threading bugs at the same time. Make sure your nonthreaded code works outside of threads.

Make Your Threaded Code Pluggable

Write the concurrency-supporting code such that it can be run in several configurations:

- Write as one thread, several threads, or varied as it executes.
- Threaded code interacts with something that can be both real or a test double.
- Execute with test doubles that run quickly, slowly, variable.
- Execute with tests that run many iterations.

Recommendation: Make your thread-based code especially pluggable so that you can run it in various configurations.

Make Your Threaded Code Tunable

Getting the right balance of threads typically requires trial and error. Early on, find ways to time the performance of your system under different configurations. Allow the number of threads to be easily tuned. Consider allowing it to change while the system is running. Consider allowing self-tuning based on throughput and system utilization.

Run with More Threads Than Processors

Things happen when the system switches between tasks. To encourage task swapping, run with more threads than processors or cores. The more frequently your tasks swap, the more likely you'll encounter code that is missing a critical section or causes deadlock.

Run on Different Platforms

In the middle of 2007, we developed a course on concurrent programming. The course development ensued primarily under OS X. The class was presented using Windows XP running under a VM. Tests written to demonstrate failure conditions did not fail as frequently in an XP environment as they did running on OS X.

In all cases, the code under test was known to be incorrect. This just reinforced the fact that different operating systems have different threading policies, each of which impacts the code's execution. Multithreaded code behaves differently in different environments.[9] You should run your tests in every potential deployment environment.

[9] Did you know that the threading model in Java does not guarantee preemptive threading? Modern OSes support preemptive threading, so you get that "for free." Even so, it is not guaranteed by the JVM.

Recommendation: Run your threaded code on all target platforms early and often.

Instrument Your Code to Try and Force Failures

It is normal for flaws in concurrent code to hide. Simple tests often don't expose them. Indeed, they often hide during normal processing. They might show up once every few hours, or days, or weeks!

The reason that threading bugs can be infrequent, sporadic, and hard to repeat is that only a very few pathways out of the many thousands of possible pathways through a vulnerable section actually fail. So the probability that a failing pathway is taken can be startlingly low. This makes detection and debugging very difficult.

How might you increase your chances of catching such rare occurrences? You can instrument your code and force it to run in different orderings by adding calls to methods like `Object.wait()`, `Object.sleep()`, `Object.yield()`, and `Object.priority()`.

Each of these methods can affect the order of execution, thereby increasing the odds of detecting a flaw. It's better when broken code fails as early and as often as possible.

There are two options for code instrumentation:

- Hand-coded
- Automated

Hand-Coded

You can insert calls to `wait()`, `sleep()`, `yield()`, and `priority()` in your code by hand. It might be just the thing to do when you're testing a particularly thorny piece of code.

Here is an example of doing just that:

```
public synchronized String nextUrlOrNull() {
    if(hasNext()) {
        String url = urlGenerator.next();
        Thread.yield(); // inserted for testing.
        updateHasNext();
        return url;
    }
    return null;
}
```

The inserted call to `yield()` will change the execution pathways taken by the code and possibly cause the code to fail where it did not fail before. If the code does break, it was not because you added a call to `yield()`.[10] Rather, your code was broken and this simply made the failure evident.

[10] This is not strictly the case. Since the JVM does not guarantee preemptive threading, a particular algorithm might always work on an OS that does not preempt threads. The reverse is also possible, but for different reasons.

There are many problems with this approach.

- You have to manually find appropriate places to do this.
- How do you know where to put the call and what kind of call to use?
- Leaving such code in a production environment unnecessarily slows the code down.
- It's a shotgun approach. You may or may not find flaws. Indeed, the odds aren't with you.

What we need is a way to do this during testing but not in production. We also need to easily mix up configurations between different runs, which results in increased chances of finding errors in the aggregate.

Clearly, if we divide our system into simple modules that know nothing of threading and others that control the threading, it will be easier to find appropriate places to instrument the code. Moreover, we could create many different test jigs that invoke the modules under different regimens of calls to `sleep`, `yield`, and so on.

Automated

You could use tools like CGLIB or ASM to programmatically instrument your code with calls to a jiggler. For example, you could use a class with a single method:

```
public class ThreadJigglePoint {
    public static void jiggle() {
    }
}
```

You can add calls to this in various places within your code:

```
public synchronized String nextUrlOrNull() {
    if(hasNext()) {
        ThreadJigglePoint.jiggle();
        String url = urlGenerator.next();
        ThreadJigglePoint.jiggle();
        updateHasNext();
        ThreadJigglePoint.jiggle();
        return url;
```

```
    }
    return null;
}
```

Imagine that the `ThreadJigglePoint` class has two implementations. The first implements jiggle to do nothing and is used in production. The second generates a random number to choose between sleeping, yielding, or just falling through. If you run your tests a thousand times with random jiggling, you may root out some flaws. If the tests pass, at least you can say you've done due diligence. Though a bit simplistic, this could be a reasonable option in lieu of a more sophisticated tool.

The point is to jiggle the code so that threads run in different orderings at different times. The combination of well-written tests and jiggling can dramatically increase the chance of finding errors.

Recommendation: Use jiggling strategies to ferret out errors.

2025 Update and Report from the Field

Modern stacks handle multiple users well when those users do not share data. The following are some cases where knowledge of parallel/threading issues is still good to keep in mind.

A little background: Modern development stacks often purport to take care of many of the issues we used to hand-roll back when the first edition of *Clean Code* was written. However, many of the original problems and issues still exist in modern systems, even ones that handle "all the multiuser stuff" for you.

Here is a collection of examples that we've come across coincidentally just in the year before this second edition of *Clean Code* was released.

Data Integrity

In the past few years, we've come across a number of situations where a multiuser system had potential data integrity issues. All of these systems have multiple simultaneous users (parallel) doing something that updates shared data.

These examples come from two different multiuser systems based in React Native. They use AWS Lambda for a backend, and storage in some datastore. There's nothing inherently problematic with these kinds of systems. If two users do not share data, then there are likely no data integrity issues. However, what happens when multiple users do depend on shared data?

The examples are

- Joining a game
- Working on a shared design
- Players working in parallel to create a deck of cards
- Double-deleting of resources on rerender using a 2D library in React Native

- Loading icon resources

Joining a Game

Imagine a mobile games platform used for training workshops. Students log in to a game. One game had a limit of six players, so after the sixth player joins, subsequent attempts to join the game are rejected. Each player is assigned a role, numbered 1 to 6.

The backend system uses DynamoDB, which gives “eventually consistent” results.

For a number of months, we’ve had an occasional kerfuffle where it seems we were losing players. It turned out to be an issue with when we decided to assign player roles, as well as having players immediately join a game.

In the original implementation, we combined logging in with joining a game and assigning roles to players. Assigning players to games and roles, along with eventual consistency in DynamoDB, led to users getting the same roles, or games overfilling.

While it was possible to rewrite how we interacted with DynamoDB, there was another desired change that predated discovering this defect. That change would make the original problem disappear by separating logging in with joining a game, and subsequent role assignment.

The original issue was assigning roles to multiple users joining a game from multiple mobile clients, that is, in parallel. To fix this, we first expanded the domain. Rather than players joining a game at login time, players join a waiting room. Later, the administrator creates games and then issues a request to assign all waiting players to games.

This took something that was not safe without transaction isolation, both joining a game and role assignment, from being executed potentially in parallel to being executed later from a single thread/issued by a single user. The original implementation was not an essential part of how we wanted the game to work; it was one of the first ideas on how to handle multiple games going on at the same time.

One of the side effects of that approach was that the administrator would create games ahead of time, and then tell specific people to log in to specific games. Users would enter their name, a game name, and a game password. It was common for communication issues, causing people to join the wrong game.

After the change, all players join a waiting room, the name of which is the administrator’s email address, provided in a pull-down list. So, rather than game name, username, and password, players selected a waiting room name and provided their name. It made the UX simpler and less error prone.

This separated the multithread-safe part of the work—getting into the system—from the not multithread-safe part of the work—joining a specific game and assigning user roles.

Working on a Shared Design

Next up: a system allowing the shared creation of a design for the installation/updating of a commercial security system. For this system, we used a relational database as the data store.

Such a plan has dozens of images, and each image might have several elements that might be connected. How do you make sure that multiple people can work on the same design at the same time, without conflict?

For this particular domain, the number of users on a design is generally less than five, and the users tend to work on different parts of an overall design. So, while the possibility of conflict was there, given the typical use of the system, the likelihood of it happening was small.

Preexisting in the system was the need to select an element to edit it. This became a natural place to lock an element. We use a flag on the element for whether it is locked. Any attempt to lock when already locked fails, and informs the user that the element is locked. Upon deselection, the element's lock is released. We chose a flag on an element, rather than locking the element in the database, because the latter would require long-lived transactions.

With proper transaction isolation, this is a clean, safe solution to the particular problem.

Players Working in Parallel to Create a Deck of Cards

This is an example of an anti-case, that is, we chose to not solve the problem. Imagine a second mobile game, this one allowing small groups of players to compete in creating a deck of cards with certain characteristics.

In this case, we have up to four people creating a deck of cards. Those players might be remote, local, collocated, or not. What happens if two players create the same card?

Players attempt to create a deck with certain characteristics. If they happen to get duplicate cards, the instructor rejects their deck. However, when this happens, the team can clean up their deck and resubmit it.

Additionally, that experience also leads to good discussions about collaboration and communication, so trying to make sure it doesn't happen reduces the value of the experience.

In this case, there's a shared resource: the deck of cards. The addition of cards is done in parallel. There can be a "data integrity" issue, duplicated work, but the end result is easy to recover from, all done through human interaction.

Early on, we realized this was a potential issue. When we thought through solutions, we decided to leave the potential in the game. An instructor can run the game in different ways, with different rules. So, in this case, the solution was to leave it as is.

Double-Deleting of Resources on Rerender Using a 2D Library in React Native

We were using React Native, with React Native Web, to produce a web and mobile application for a customer. We additionally used a 2D graphics library called Skia.

Skia is a powerful library, and it has its own rendering engine. This means there are at least two rendering engines running at the same time—React Native and Skia—and those two rendering engines are not synchronized. We did not notice an issue until we added asynchronous loading of designs.

When a user opens a design, we load it asynchronously while moving into the design view of the application. When we started doing this, we found that Skia was frequently crashing.

What ultimately caused this was rendering the Skia view before we had finished loading. In React Native, this is normal. It rerenders the view. However, since Skia has its own rendering thread, when React Native rerendered the view at an unexpected time, Skia would start to delete, then redelete the same resources.

Deleting already-deleted memory killed Skia and required restarting the application. This wasn't an issue with Skia; it was how we were using it.

The solution for this was to not render the Skia view until loading was done. We had well-defined locations where we needed Skia to be rendering, and where it was not required. We make sure to

- Only render the Skia view when loading was complete; otherwise, we rendered an empty view.
- First clear a setting anytime we issued a change that caused a rerender of the Skia view. This would cause the Skia view to stop being rendered.
- Set a new value, causing everything to be rendered fresh.

The end solution resulted in cleaner code overall. The solution was to allow for "parallel" rendering threads only when a design was loaded. No design? No Skia rendering. Design? Skia rendering. Anytime we changed designs, we first set state to stop rendering Skia, loaded the design, and upon completion, began using Skia again.

Loading Icon Resources

We had an issue with the order in which icons were loaded. It was possible to load a design into the system before the code to load the icons had started executing. When this happened, we'd have to reload the design, or have several missing icons.

Since the resources were needed by other parts of the system, we moved them into a so-called provider in React Native. Then, any code that might cause a design to be loaded is always executed under that provider, that is, there was a parent–child relationship between the provider and anything using icons. The parent initializes first, guaranteeing the order of execution.

As with the previous example, this resulted in cleaner code. Things that were related moved closer together under a single place, and we removed duplication.

Sharing Data

In all of these examples, there's some issue with sharing data:

- Allowing players to join a specific game, and get a role assigned in parallel, with an eventually consistent database
- Allowing editing of a shared design, by multiple users in parallel
- Allowing players to create cards in parallel
- Multiple, dependent rendering loops that are not coordinated, with other asynchronous activity impacting them
- Not controlling the execution/loading of resources explicitly in an asynchronous environment

We took different approaches for each of these problems.

- Isolate the nonthread-safe code and execute it later in a single thread (isolate and separate).
- Lock objects with a flag, using a relational database.
- Do nothing. A human can choose to reject or accept based on the current rules of the game.
- Avoid using the second rendering thread until it's actually needed, and clear it before clearing everything else.
- Make the loading part of a parent–child relationship such that the child doesn't render until the parent is ready.

Once you've discovered an issue with shared data, there are many approaches you might take to address it. We've seen five examples and five different approaches to specific problems.

Conclusion

Concurrent code is difficult to get right. Code that is simple to follow can become nightmarish when multiple threads and shared data get into the mix. If you are faced with writing concurrent code, you need to write clean code with rigor or else face subtle and infrequent failures.

First and foremost, follow the Single Responsibility Principle. Break your system into simple modules that separate thread-aware code from thread-ignorant code. Make sure when you are testing your thread-aware code that you are only testing it and nothing else. This suggests that your thread-aware code should be small and focused.

Know the possible sources of concurrency issues: multiple threads operating on shared data, or using a common resource pool. Boundary cases, such as starting up and shutting down cleanly or finishing the iteration of a loop, can be especially thorny.

Learn your library and know the fundamental algorithms. Understand how some of the features offered by the library support solving problems similar to the

fundamental algorithms.

Learn how to find regions of code that must be locked, and lock them. Do not lock regions of code that do not need to be locked. Avoid calling one locked section from another. This requires a deep understanding of whether something is or is not shared. Keep the number of shared objects and the scope of the sharing as narrow as possible. Change designs of the objects with shared data to accommodate clients rather than forcing clients to manage shared state.

Issues will crop up. The ones that do not crop up early are often written off as a one-time occurrence. These so-called one-offs typically only happen under load or at seemingly random times. Therefore, you need to be able to run your thread-related code in many configurations on many platforms repeatedly and continuously. Testability, which comes naturally from following a good testing discipline, implies some level of plug-ability, which offers the support necessary to run code in a wider range of configurations.

You will greatly improve your chances of finding erroneous code if you take the time to instrument your code. You can either do so by hand or use some kind of automated technology. Invest in this early. You want to be running your thread-based code as long as possible before you put it into production.

If you take a clean approach, your chances of getting it right increase drastically.

Part III

Architecture

The architecture of a software system is the shape given to that system by those who build it. The form of that shape is in the division of that system into components, the arrangement of those components, and the ways in which those components communicate with each other.

The purpose of that shape is to facilitate the development, deployment, operation, and maintenance of the software system contained within it.

The strategy behind that facilitation is to leave as many options open as possible, for as long as possible.

Perhaps this statement has surprised you. Perhaps you thought that the goal of software architecture was to make the system work properly. Certainly we want the system to work properly; and certainly the architecture of the system must support that as one of its highest priorities.

However, the architecture of a system has very little bearing on whether that system works. There are many systems out there, with terrible architectures, that work just fine. Their troubles are not in their operation; they are in deployment, maintenance, and ongoing development.

This is not to say that architecture plays no role in supporting the proper behavior of the system. It certainly

does, and the role is critical. But the role is passive and cosmetic, not active or essential. There are few, if any, behavioral options that the architecture of a system can leave open.

The primary purpose of architecture is to support the lifecycle of the system. Good architecture makes the system easy to understand, easy to develop, easy to maintain, and easy to deploy. The ultimate goal is to minimize the lifetime cost of the system and maximize programmer productivity.

Note: This chapter is an overview. For a more complete treatment of the topic, see [Clean Arch].

Chapter 23. The Two Values of Software



There are two values to software. There is the value of its behavior, and there is the value of its structure. The second of these is the greater of the two because it is this value that makes software *soft*.

Software was invented because we needed a way to quickly and easily change the behavior of our machines. But that flexibility depends critically on the shape of the system, the arrangement of its components, and the way those components are interconnected.

Keeping Options Open

The way you keep software soft is to leave as many options open as possible, for as long as possible. What are the options that we need to leave open? *They are the details that don't matter.*

All software systems can be decomposed into two major elements: policy and detail. The policy element embodies all the business rules and procedures. The policy is where the true value of the system lives.

The details are those things that are necessary to enable humans, other systems, and programmers to communicate with the policy, but do not impact the behavior of the policy at all. These include IO devices, databases, Web systems, servers, frameworks, and communication protocols, among others.

The goal of architecture is to imbue the system with a shape that recognizes policy as the most essential element of the system while making the details irrelevant to that policy. This allows decisions about those details to be *delayed* and *deferred*.

For example:

- It is not necessary to choose a database system in the early days of development, because the high-level policy should not care what kind of database will be used. Indeed, if you are careful, the high-level policy

will not care if the database is relational, distributed, hierarchical, or just plain flat files.

- It is not necessary to choose a Web server early in development, because the high-level policy should not know that it is being delivered over the Web. If the high-level policy is unaware of HTML, CSS, AJAX, JSP, JSF, or any of the rest of the alphabet soup of web development, then you don't need to decide which Web system to use until much later in the project. Indeed, *you don't even have to decide if the system will be delivered over the Web*.

- It is not necessary to adopt REST early in development, because the high-level policy should be agnostic about the interface to the outside world. Nor is it necessary to adopt a microservices framework, or an SOA framework, or REACT . . . Again, the high-level policies should not care about these things.

- It is not necessary to adopt a dependency injection framework early in development, because the high-level policy should not care how dependencies are resolved.

I think you get the point. If you can develop the high-level policy without committing to the details that surround it, you can delay and defer decisions about those details for a long time. And the longer you wait to make those decisions, the more information you have with which to make them properly.

This also leaves you the option to try different experiments. If you have a portion of the high-level policy working and it is agnostic about the database, you could try connecting it to several different databases in order to check applicability and performance. The same is true with Web systems, Web frameworks, or even the Web itself.

The longer you leave options open, the more experiments you can run, the more things you can try, and the more information you will have when those decisions can no longer be deferred.

What if the decisions have already been made by someone else? What if your company has made a commitment to a certain database, or a certain Web server, or a certain framework? I suggest that you pretend that those decisions have *not* been made, and give the system a shape that allows those decisions to be deferred for as long as possible. You never know when decisions like that might suddenly change.

A good architecture maximizes the number of decisions not made.

Chapter 24. Independence



A good architecture must support

- The use cases of the system
- The operation of the system
- The development of the system
- The deployment of the system

Use Cases

The first bullet—use cases—means that the architecture of the system must support the intent of the system. If the system is a shopping cart application, then the architecture must support shopping cart use cases. Indeed, this is the first priority of the architecture. The architecture must support the use cases.

However, as we discussed previously, architecture does not wield much influence over the behavior of the system. There are very few behavioral options that the architecture can leave open. But influence isn't everything. The most important thing a good architecture can do to support behavior is to clarify and expose that behavior so that the intent of the system is visible at the architectural level.

A shopping cart application with a good architecture will *look* like a shopping cart application. The use cases of that system will be plainly visible within the structure of that system. Developers will not have to hunt for behaviors, because those behaviors will be first-class elements visible at the top level of the system. Those elements will be classes or functions or modules that have prominent positions within the architecture and will have names that clearly describe their function.

Operation

Architecture plays a more substantial, and less cosmetic, role in supporting the operation of the system. If the system must handle 100,000 transactions per second, the architecture must support that kind of throughput and response time for each use case that demands it. If the system must query big data cubes in milliseconds, then the

architecture must be structured to allow this kind of operation.

For some systems, this will mean arranging the processing elements of the system into an array of little services that can be run in parallel on many different servers. For other systems, it will mean a plethora of little lightweight threads sharing the address space of a single process within a single processor. Still other systems will need just a few processes running in isolated address spaces. And some systems can even survive as simple monolithic programs running in a single process.

As strange as it may seem, this decision is one of the options that a good architecture leaves open. A system that is written as a monolith and that depends upon that monolithic structure cannot easily be upgraded to multiple processes, multiple threads, or microservices should the need arise. On the other hand, an architecture that maintains the proper isolation of its components and does not assume the means of communication between those components will be much easier to transition through the spectrum of threads, processes, and services as the operational needs of the system change over time.

Development

Architecture plays a significant role in supporting the development environment. This is where Conway’s law and the Single Responsibility Principle (SRP) come into play. A system that must be developed by an organization with many teams and many concerns must have an architecture that facilitates independent actions by those teams so that the teams do not interfere with each other during development.

This is accomplished by properly partitioning the system into well-isolated, independently developable components. Those components can then be allocated to teams that can work independently of each other.

Deployment

Finally, the architecture plays a huge role in the ease with which the system is deployed. The goal is “immediate deployment.” A good architecture does not rely on dozens of little configuration scripts and property file tweaks. It does not require manual creation of directories or files that must be arranged just so. A good architecture helps the system be immediately deployable after each build.

Again, this is achieved through the proper partitioning and isolation of the components of the system, including those master components that tie the whole system together and ensure that each component is properly started, integrated, and supervised.

Leaving Options Open

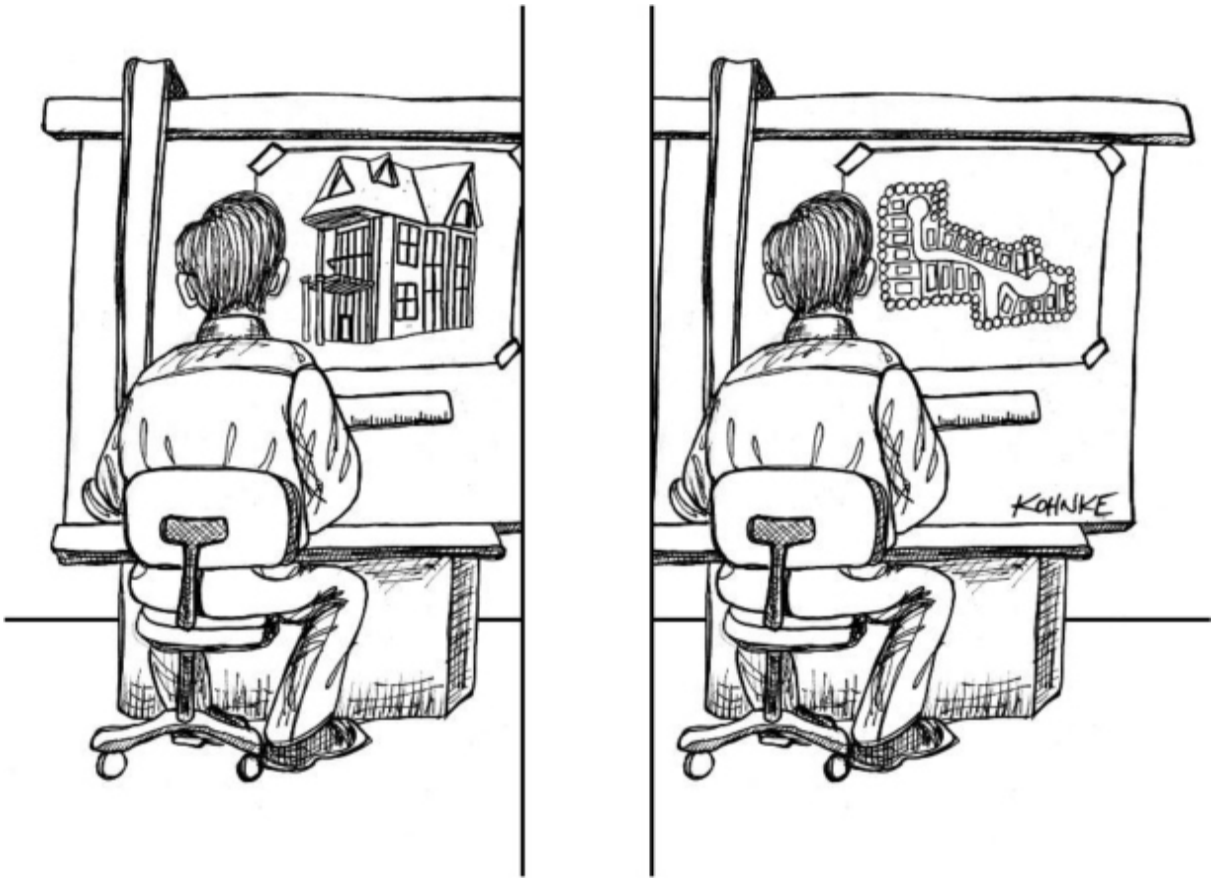
A good architecture balances all the above concerns with a component structure that mutually satisfies them all. Sounds easy, right? Well, it’s easy for me to write that.

The reality is that this is pretty hard. The problem is that most of the time we don’t know what all the use cases are, nor do we know the operational constraints, the team structure, or the deployment requirements. Worse, even if we *did* know them, they will all change as the system moves through its lifecycle. In short, the goals we must meet are indistinct and inconstant. Welcome to the real world.

But all is not lost. There are principles of architecture that are relatively inexpensive to implement and that can help balance those concerns, even when you don’t have a clear picture of the goals you have to hit. Those principles help us partition our systems into well-isolated components that allow us to leave as many options open as possible, for as long as possible.

A good architecture makes the system easy to change, in all the ways that it must change, by leaving options open.

Chapter 25. Architectural Boundaries



Software architecture is the art of drawing lines that I call *boundaries*. Those boundaries separate software elements from one another, and restrict those on one side from knowing about those on the other. Some of those lines are drawn very early in a project’s life—even before any code is

written. Others are drawn much later. Those that are drawn early are drawn for the purpose of deferring decisions for as long as possible, and of keeping those decisions from polluting the core business logic.

Remember that the goal of an architecture is to minimize the manpower required to build and maintain the required system. What is it that saps manpower? *Coupling*; and especially, coupling to premature decisions.

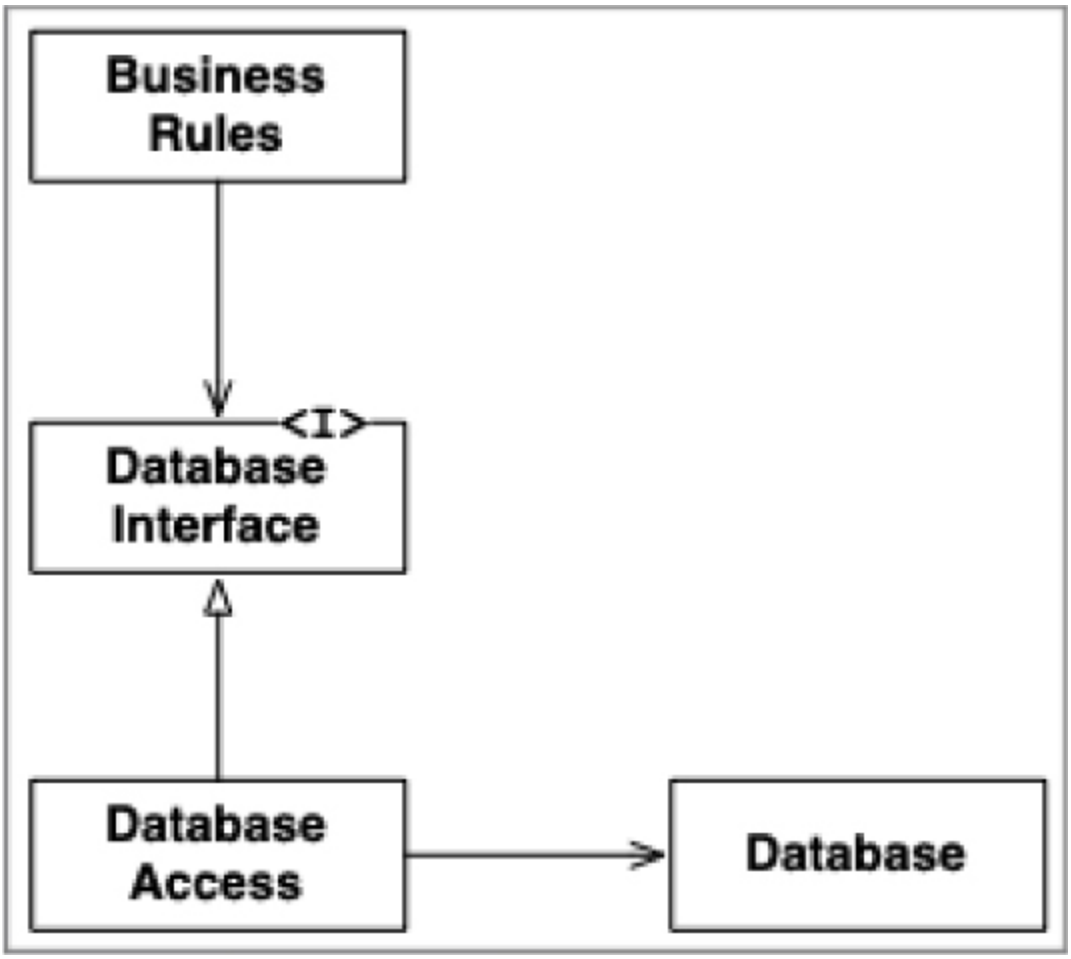
What kinds of decisions are premature? Decisions that have nothing to do with the business requirements—the use cases—of the system. These include decisions about frameworks, databases, Web servers, utility libraries, dependency injection, and the like. A good system architecture is one in which decisions like these are rendered ancillary and deferrable. A good system architecture does not depend on those decisions. A good system architecture allows those decisions to be made at the latest possible moment, without significant impact.

What Lines Do You Draw, and When?

You draw lines to separate things that matter from things that don’t. The GUI doesn’t matter to the business rules, so there should be a line between them. The database doesn’t matter to the GUI, so there should be a line between them. The database doesn’t matter to the business rules, so there should be a line between them.

Some of you may have rejected one or more of those statements; especially the part about the business rules not caring about the database. Many of us have been taught to believe that the database is inextricably connected to the business rules. Some of us have even been convinced that the database is the embodiment of the business rules.

But this idea is misguided. The database is a tool that the business rules can use indirectly. The business rules don't need to know about the schema, or the query language, or any of the other details about the database. All the business rules need to know is that there is a set of functions that can be used to fetch or save data. This allows us to put the database behind an interface.

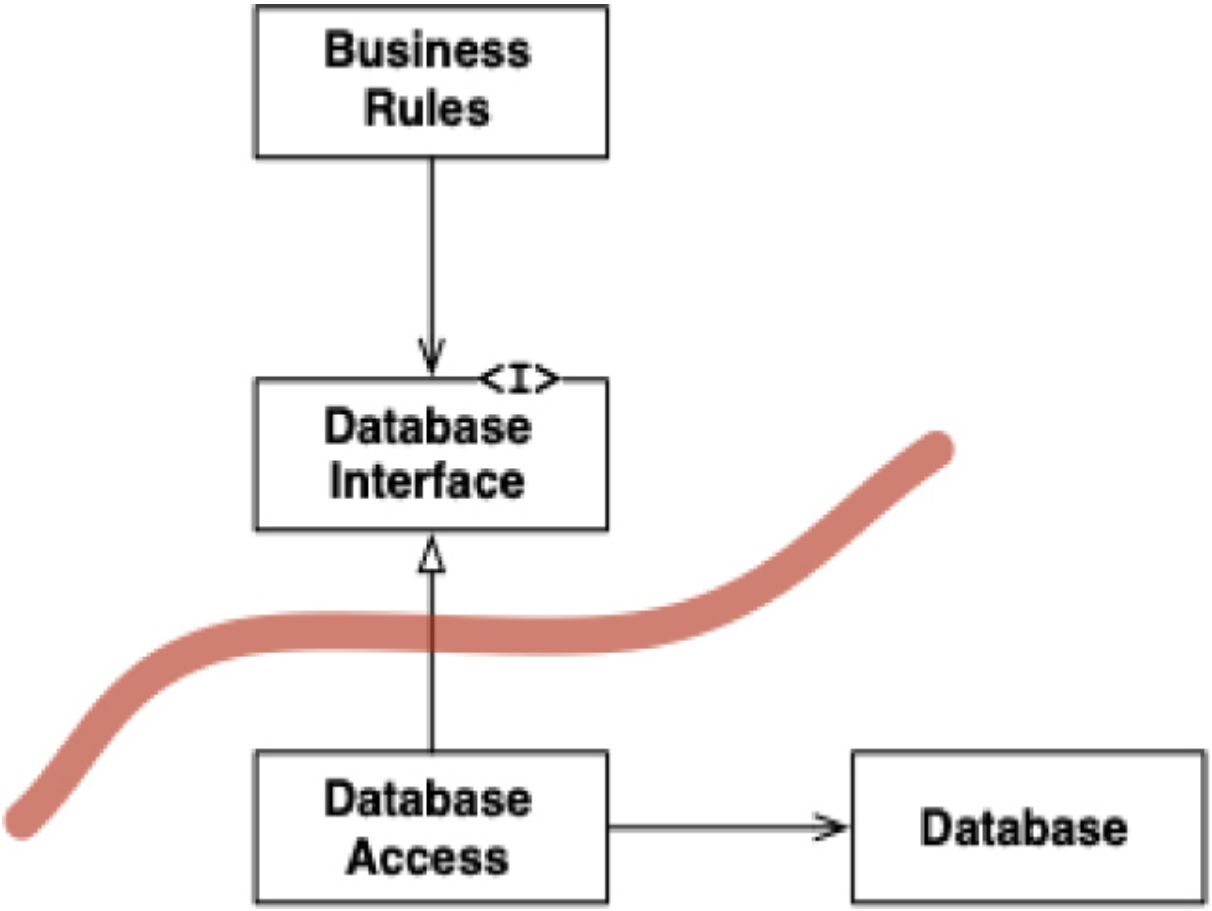


You can see this clearly in the diagram above. The `BusinessRules` use the `DatabaseInterface` to load and save data. The `DatabaseAccess` implements the interface and directs operation of the actual `Database`.

The classes and interfaces in this diagram are symbolic. In a real application, there would be many business rule classes, many database interface classes, and many database access

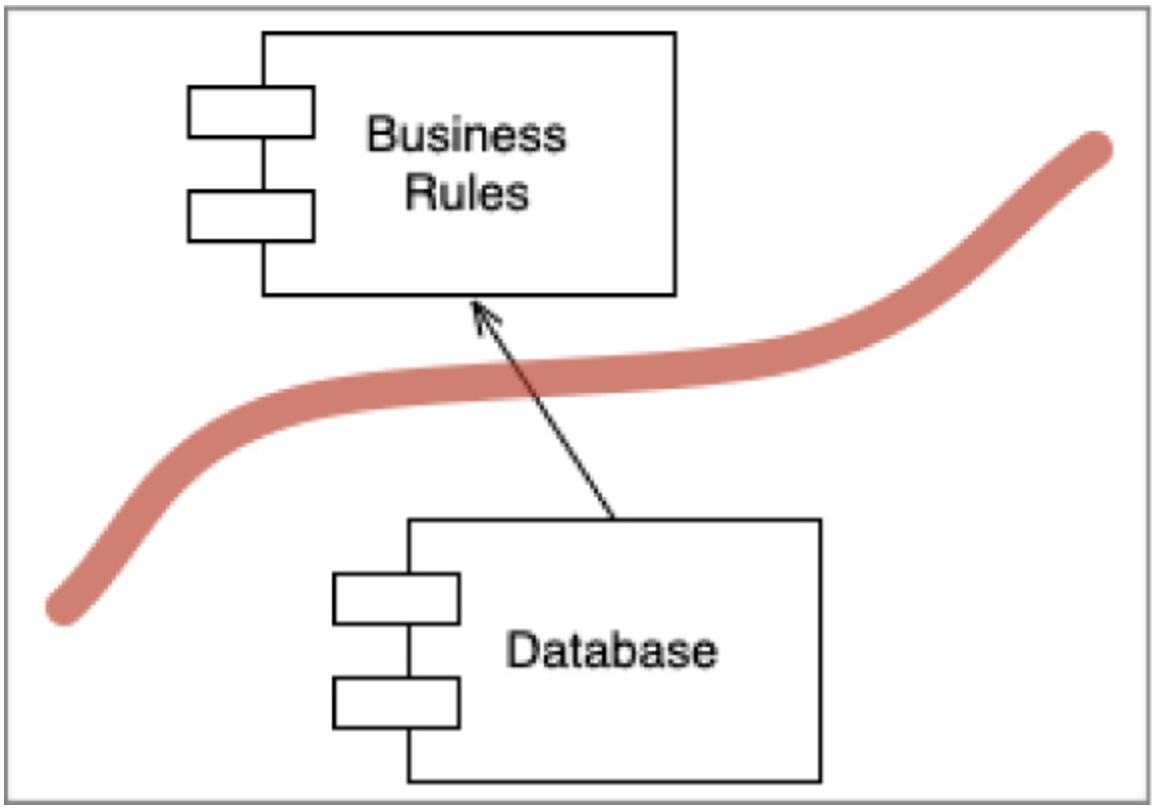
implementations. But they would all follow roughly the same pattern.

Where is the boundary line? The boundary is drawn across the inheritance/implements relationship, just below the `DatabaseInterface`, as shown in the following diagram.



Note the two arrows leaving the `DatabaseAccess` class. Those two arrows point away from the `DatabaseAccess` class. That means that none of these classes knows that the `DatabaseAccess` class exists. That's the line.

Now let's pull back a bit. We'll look at the component that contains many business rules, and the component that contains the database and all its access classes.



Note the direction of the arrow. The `Database` knows about the `BusinessRules`. The `BusinessRules` do not know about the `Database`. This implies that the `DatabaseInterface` classes live in the `BusinessRules` component, and the `DatabaseAccess` classes live in the `Database` component.

The direction of this arrow is important. It shows that the `Database` does not matter to the `BusinessRules`. But the `Database` cannot exist without the `BusinessRules`.

If that seems strange to you, just remember: That `Database` component contains the code that translates the calls made by the `BusinessRules` into the query language of the database. It is that translation code that knows about the `BusinessRules`.

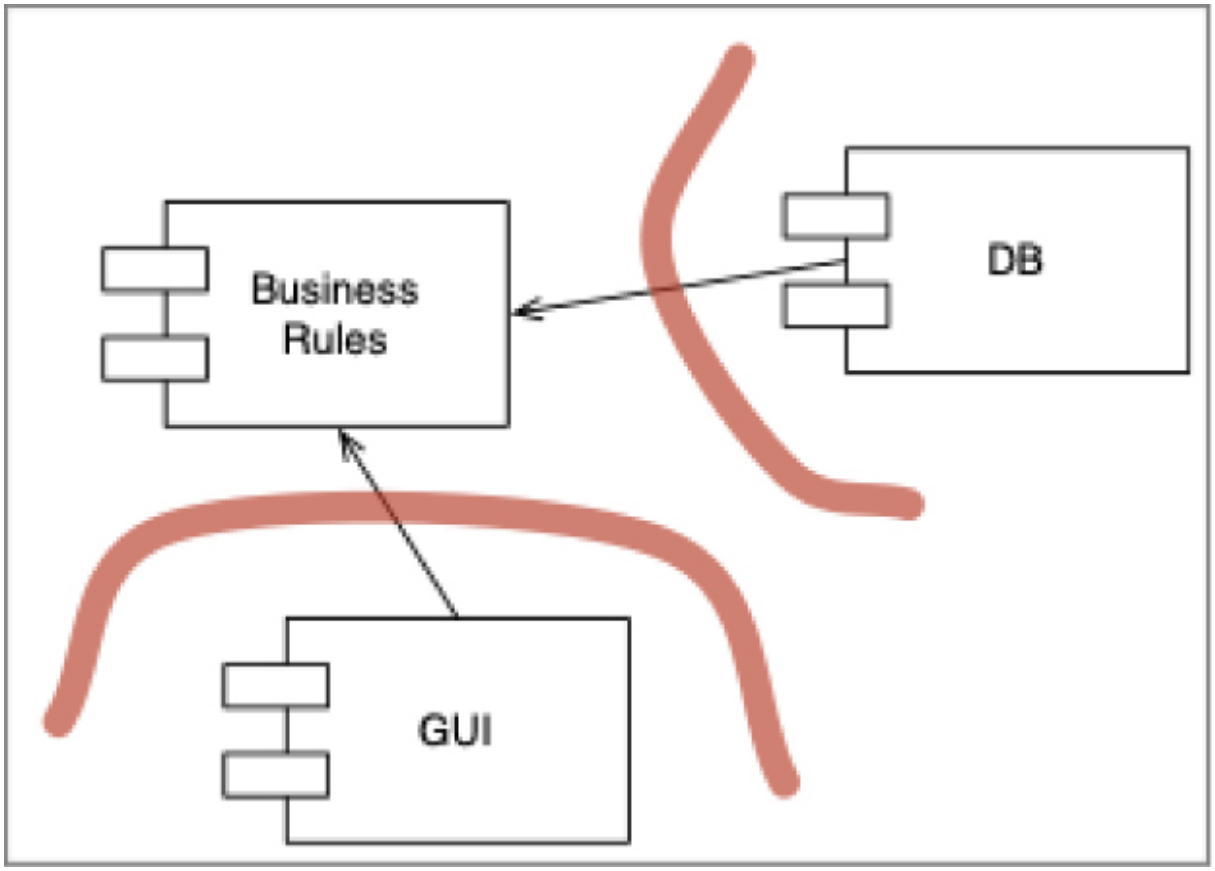
Having drawn this boundary line between the two components, and having set the direction of the arrow toward the `BusinessRules`, we can now see that the `BusinessRules` could use *any* kind of database at all. The

`Database` component could be replaced with many different implementations. The `BusinessRules` don't care.

The database could be implemented with Oracle, or MySQL, or Couch, or Datomic, or even flat files. The business rules don't care at all. And that means that the database decision can be deferred and you can focus on getting the business rules written and tested before you have to make the database decision.

Plug-in Architecture

Look at the two boundaries in the diagram below. They separate the `Database` and the `GUI` from the `BusinessRules`. The arrows point from the lower-level components to the high-level `BusinessRules` component.



The GUI and DB are plug-ins to the BusinessRules. This is a pattern that can be repeated for many different kinds of components. The core business rules can be kept separate from, and independent of, any components that either are optional or can be implemented in many different forms.

So, because the user interface in this design is considered to be a plug-in, we have made it possible to plug in many different kinds of user interfaces. They could be Web based, client/server based, SOA based, console based, or based on any other kind of user interface technology.

The same is true of the database. Since we have chosen to treat it as a plug-in, we can replace it with any of the various SQL databases, or a NoSQL database, or a file system–based database, or any other kind of database technology we might deem necessary in the future.

These replacements might not be trivial. If the initial deployment of our system was Web based, then writing the plug-in for a client/server UI could be challenging. It is likely that some of the communications between the BusinessRules and the new UI would have to be reworked. But by starting with the presumption of a plug-in structure, we have, at the very least, made such a change practical.

Case Study: FitNesse

My son, Micah, and I started work on FitNesse in 2001. The idea was to create a simple wiki that wrapped Ward Cunningham's FIT tool for writing acceptance tests. We wrote FitNesse in Java.

I was adamant that anything we produced should not require people to download more than one jar file. I called this rule *Download and Go*. This rule drove many of our decisions.

One of the first was to write our own Web server, specific to the needs of FitNesse. This might sound absurd. Even in 2001 there were plenty of open source Web servers that we could have used. Yet writing our own turned out to be a really good decision because a bare-bones Web server is a very simple piece of software to write, and it allowed us to postpone any Web framework decision until much later.[1]

[1] Many years later we were able to slip the Velocity framework into FitNesse.

Another early decision was to avoid thinking about a database. We had MySQL in the back of our minds, but we purposely delayed that decision by employing a design that made the decision irrelevant. That design was simply to put an interface between all data accesses and the data repository itself.

We put the data access methods into an interface named WikiPage. Those methods provided all the functionality we needed to find, fetch, and save pages. Of course, we didn't implement those methods at first; we simply stubbed them out while we worked on features that didn't involve fetching and saving the data.

Indeed, for three months we simply worked on translating wiki text into HTML. This didn't require any kind of data storage, so we created a class named MockWikiPage that simply left the data access methods stubbed.

Eventually, of course, those stubs were insufficient for the features we wanted to write. We needed real data access, not stubs. So we created a new derivative of WikiPage, named InMemoryPage. This derivative implemented the data access method to manage a hash table of wiki pages, which we kept in RAM.

This allowed us to write feature after feature for a full year. In fact, we got the whole first version of the FitNesse program

working this way. We could create pages, link to other pages, do all the fancy wiki formatting, and even run tests with FIT. What we couldn't do was save any of our work.

So, when it came time to implement persistence, we thought again about MySQL, but we decided that wasn't necessary in the short term because it was really easy to write the hash tables out to flat files. So we implemented FileSystemWikiPage, which just moved the functionality out to flat files, and then we continued developing more features.

Three months later we reached the conclusion that the flat-file solution was good enough, and we decided to abandon the idea of MySQL altogether. *We deferred that decision into nonexistence, and never looked back.*

That would be the end of the story if it weren't for a customer of ours who decided that he needed to put the wiki into MySQL for his own purposes. So we showed him the architecture of WikiPages that had allowed us to defer the decision. He came back a day later with the whole system working in MySQL. He simply wrote a MySqlWikiPage derivative and got it working.

We used to bundle that option with FitNesse, but nobody else ever used it, so eventually we dropped it. Even he eventually dropped it.

Early in the development of FitNesse, we drew a boundary line between business rules and databases. That line prevented the business rules from knowing anything at all about the database, other than the simple data access methods. That decision allowed us to defer the choice and implementation of the database for well over a year. It allowed us to try the file system option. And it allowed us to change direction when we saw a better solution. And yet it did not prevent, or even impede, the original direction, MySQL, when someone wanted it.

The fact that we did not have a database running for 18 months of development meant that, for 18 months, we did not have schema issues, query issues, database server issues, password issues, connection time issues, or all the other nasty issues you get when you fire up a database. It also meant that all our tests ran fast, because there was no database to slow them down.

In short, drawing the boundary lines that helped us delay and defer decisions saved us an enormous amount of manpower and headaches. And that’s what a good architecture should do.

Conclusion

This is how you draw boundary lines in a software architecture. You partition the system into components. Some of those components are core business rules. Some are plug-ins that contain necessary functions that are not directly related to the core business. Then, you arrange the code in those components such that the arrows between them point in one direction—toward the core business.

You should recognize this as an application of the Dependency Inversion Principle (DIP) and the Stable Abstractions Principle (SAP). Dependency arrows are arranged to point from lower-level details to higher-level abstractions.

And that is the Dependency Rule of Architecture.

Dependencies that cross architectural boundaries must always point toward the higher-level side.

Chapter 26. Clean Boundaries

by James Grenning



We seldom control all the software in our systems. Sometimes we buy third-party packages or use open source. Other times we depend on teams in our own company to produce components or subsystems for us. Somehow we must cleanly integrate this foreign code with our own. In this section, we look at practices and techniques to keep the boundaries of our software clean.

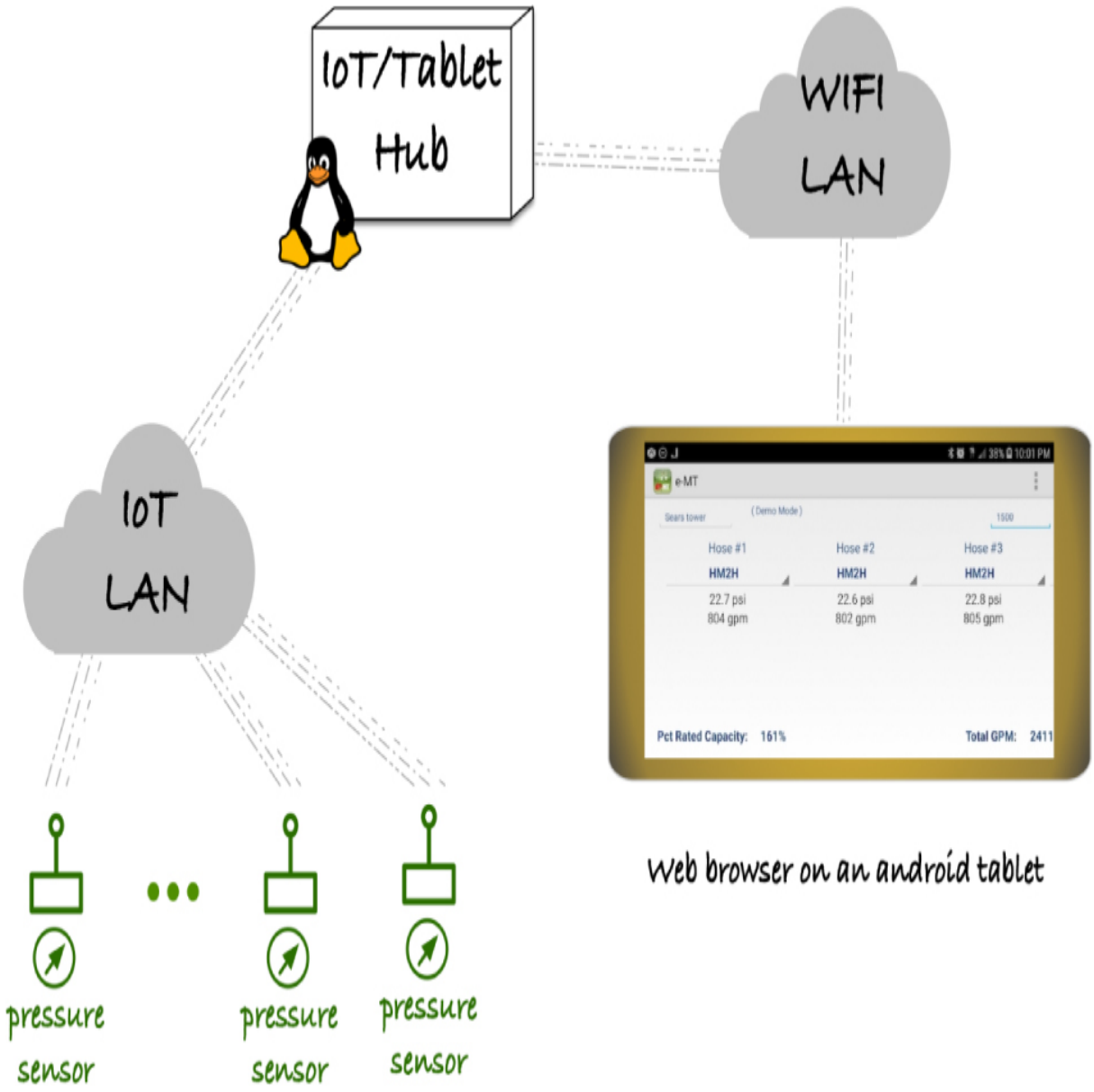
Third-Party IoT Framework: Lots o’ Boundaries

My brother had an idea for automating the manual process that users of his product perform. The product is used by a technician testing a high-capacity water pump. The pump is tested by flowing water and manually measuring and recording water pressure at several nozzles. Given the geometry of the nozzles, pressure is converted into flow rate.

The pump is permanently installed in a commercial or industrial building; water flows outside, onto the curb, and down a storm drain. Pump capacities are measured in thousands of gallons per minute. A manual test can take 20 to 30 minutes, and a lot of water is wasted in the process. Typically, pressure measurements are made with a physical gauge at several locations: inside the building next to the pump, and outside at the nozzles, with records kept manually on a clipboard.

The big idea was to measure pressure with an electronic gauge that could report its pressure once per second and have the readings shown on a tablet computer or smart phone. The first design used Bluetooth to communicate from the sensor to the tablet. It worked great in the lab, but in the field there was always a brick wall (or worse) in the way. Bluetooth could not penetrate the brick wall. So we dreamed up the following idea.

With an off-the-shelf Internet of Things (IoT) infrastructure, we could get the water pressure measurements through brick walls.



We selected an IoT vendor with infrastructure that let us build our application in Python. Their existing radio system, along with our production code in Python, would make it possible for us to get a proof-of-concept prototype developed quickly.

We set off to build a prototype, knowing we might need a different vendor once we went to market. Why did we doubt this vendor? At the prototype stage, we weren't that concerned with cost. That could change by the time we

went to market. Also, in the rapidly growing IoT market, this vendor may go out of business or get eaten by another IoT supplier. We wanted to preserve our business logic investment, so in our prototype, we needed clean code, separation of concerns, and a limited dependency on our vendor's infrastructure.

Nicely, our vendor provided a simple "hello world" IoT application to get started. Their "hello world" demo allowed us to send messages between a small Linux box (it was kind of like an industrial Raspberry Pi) and the little microcontroller-driven radios that would be married to our pressure sensors. We could morph their "hello world" example code into a layer that would contain all the dependencies on the vendor infrastructure.

Something I learned the hard way leads me to prefer starting with a working system and focusing on small changes to grow functionality and not break stuff. You know, it's easier to keep a system working than to fix it after you break it! Test-driven development (TDD) allowed me to see the value of small, verified changes.

The vendor had an elaborate application infrastructure we could take advantage of. It had a lot of capabilities. We just would not need everything it offered. We wanted to use the radios for some simple message passing, not marry the vendor (until death do us part). Consequently, we used very little of the vendor's very wide interface.

Let's start with the sensor and the hub getting to know each other. The vendor makes this very easy. The hub and radios interact using a remote procedure call (RPC) mechanism. The hub has a full Python environment while the radio is limited to MicroPython.

The project is code-named Hydra, after the many-headed monster. The class `HydraNetwork` encapsulates all

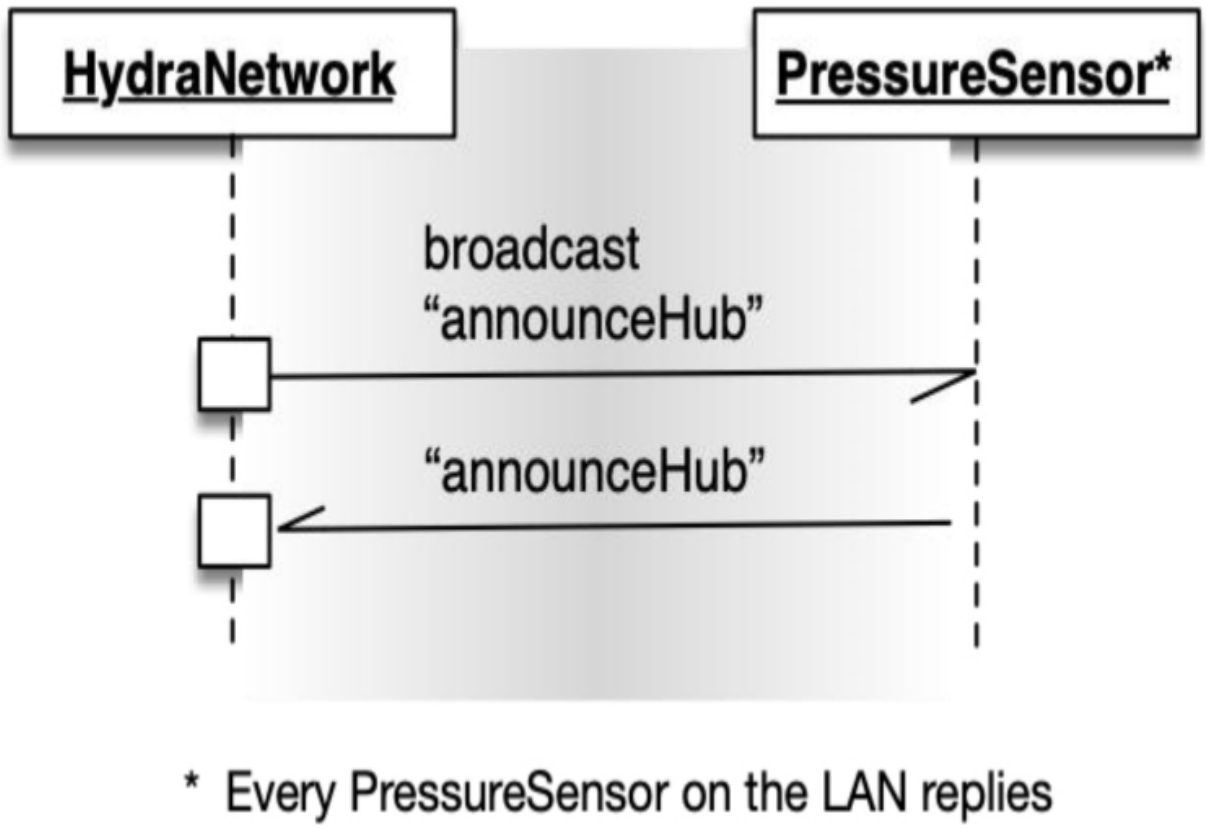
dependencies in the hub on the IoT vendor. The functions are low complexity and delegate to the communications infrastructure. Starting at the beginning, initialization, the hub announces itself to all the radios listening, then polls the framework for replies.

```
# In the hub
class HydraNetwork:
  …
  def announceHub(self):
    self.broadcast('announceHub', '')
    self.poll_a_second()
```

Through the magic of the infrastructure RPC mechanism, the `announceHub()` function is called in the radio. The radio remembers the hub's address and sends an acknowledgment to the hub.

```
# In the radio
def announceHub():
  global hub_addr
  hub_addr = rpcSourceAddr()
  info = nodeIdentity()
  sendAck(hub_addr, 'announceHub', info)
```

This sequence chart shows the asynchronous communications between the `HydraNetwork` hub and its `PressureSensors` on the IoT LAN.



Now, going back to the hub, the infrastructure RPC facility calls the hub’s `HydraNetwork.processAck()` function.

```
# In the hub
class HydraNetwork:
  …
  def processAck(self, command, payload):
    sensor_id = self.source_addr()
    self.ack_handler.handle_reply(sensor_id, command, payload)
```

The `ack_handler` is an injected dependency, which happens to be an instance of `HydraDataSource`. The method `handle_reply()` passes the received bookkeeping and payload to other parts of the Hydra system. As it turns out, there is nothing to do for the `'announceHub'` ACK from the radio.

The `HydraDataSource` initiates a network PSI[1] read when its `flow_update()` method is called. The `flow_update()` method is called once per second during a flow test. Each update has a `sync_token` so that asynchronous measurement replies can be grouped together by second. When a new second starts, a report is saved from the previous second. More on that later.

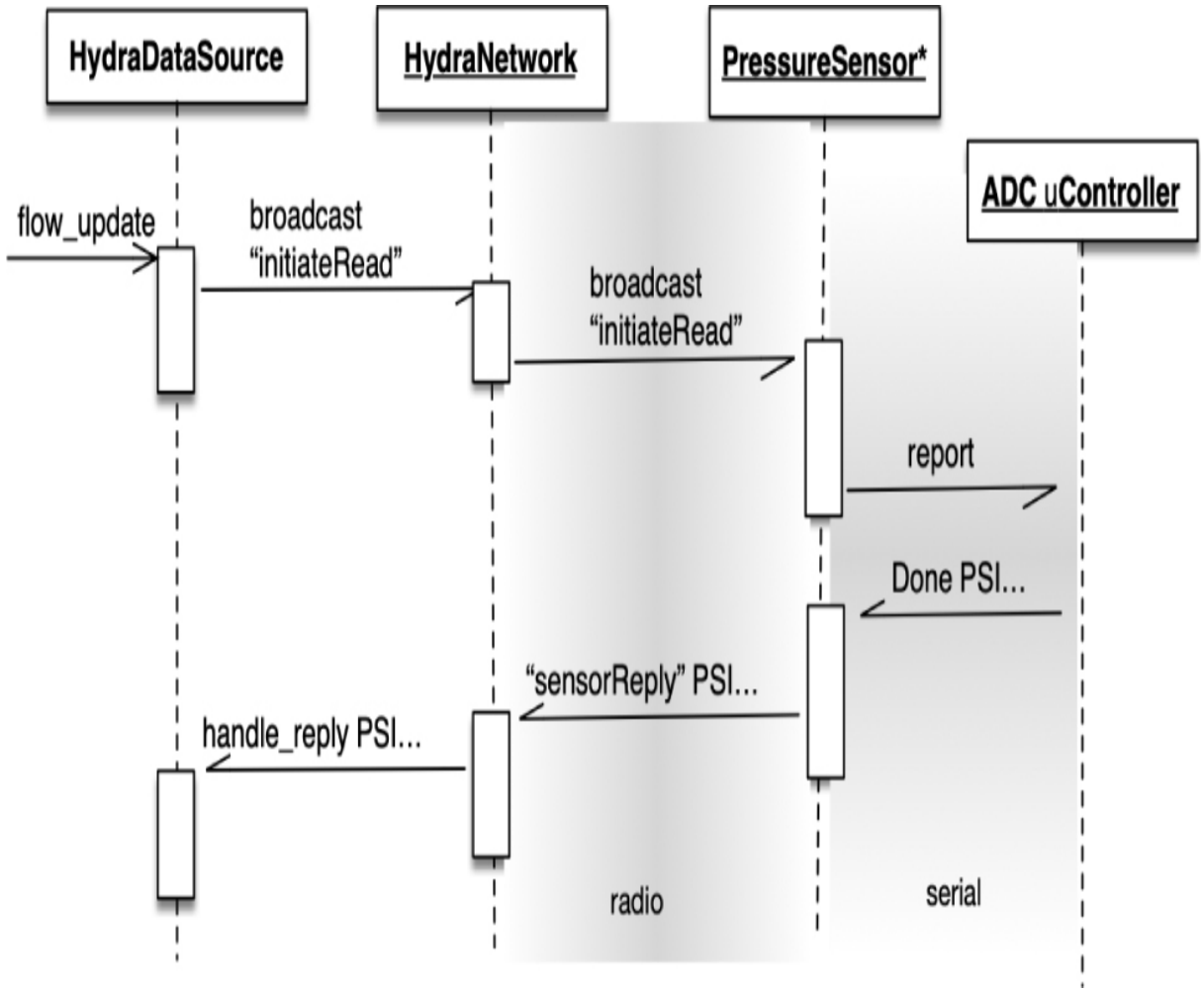
[1] For example, [PPP02], http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod, and https://en.wikipedia.org/wiki/SOLID_(object-oriented_design) (or just google "SOLID").

```
# In the hub
class HydraDataSource:
   …
   def flow_update(self):
        self.sync_token.next()
        self.network.broadcast(
                        'initiateRead',
                         self.sync_token.value())
        self.report = self.reply_dispatcher.report()
```

When `'initiateRead'` is broadcast to the sensor network, each radio replies asynchronously. Again, via the IoT framework, `initiateRead()` is called in the `PressureSensor` module.

First, `initiateRead()` responds with an ACK and then calls `reportMeasurement(sync_token)` to get the latest PSI value.

I neglected to tell you about another boundary. There is a separate microcontroller responsible for reading the analog-to-digital converter (ADC) and doing the needed math. The call to `reportMeasurement()` writes to a serial connection asking the ADC microcontroller for the next measurement.




```
# In the radio
def initiateRead(sync_token):
    global hub_addr
    hub_addr = rpcSourceAddr()
    sendAck(hub_addr, 'initiateRead', sync_token)
    reportMeasurement(sync_token)

def reportMeasurement(sync_token):
    print 'RA ' + sync_token

@setHook(HOOK_STDIN)
def measurementDone(reply):
```

```
    stdin_info = getInfo(STDIN_INFO)
    if stdin_info != STDIN_RECEIVED:
      reply = 'Error: Invalid STDIN_INFO'+
                               str(stdin_info)
    sendAck(hub_addr,'SensorReply',reply[:MAX_SIZE])
```

It takes a little time to get a new ADC value and convert it to PSI. When the measurement is ready, the ADC microcontroller sends back a CSV string using the serial connection. The `HOOK_STDIN` notifies `measurementDone()`, providing a reply. If there is no serial communication error, the reply includes the `sync_token` and sensor configuration. The reply is sent as a `SensorReply` ACK to the hub.

Meanwhile, back at the hub, the hub is collecting ACK messages. ACK messages are sent to `handle_reply()`. A `'SensorReply'` ACK is given to the `reply_dispatcher.sensor_update()`. The `reply_dispatcher` knows the message format, and where to send it once it is parsed. The payload is a simple set of CSVs. The `reply_dispatcher` and collaborators are test driven and free from IoT dependencies.

```
class HydraDataSource:
   …
  def handle_reply(self, sensor_id,
                   command, payload):
   sync_token = self.sync_token.value
    if command == 'SensorReply':
      result = self.reply_dispatcher.
                   sensor_update(sensor_id,
```

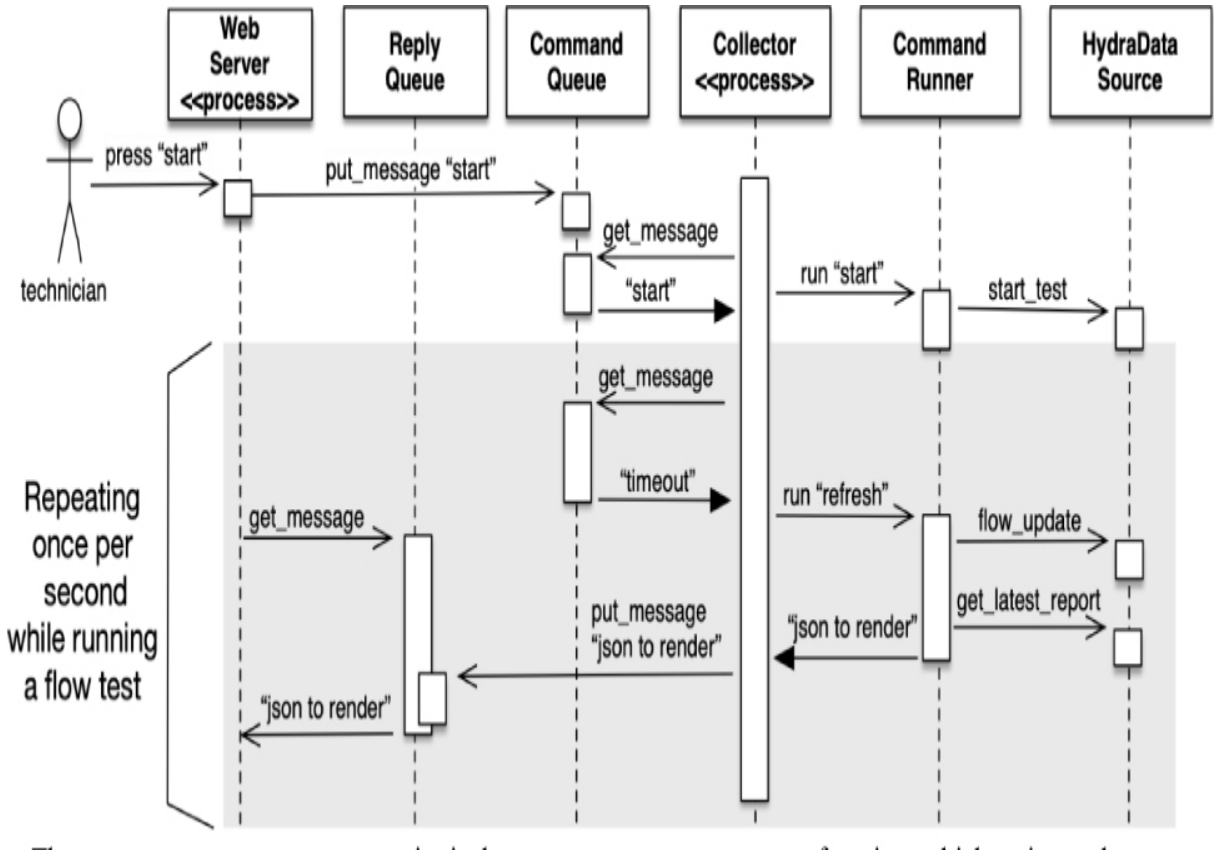
```
                                    sync_token,
                                    payload)
```

Because `HydraNetwork` encapsulated all IoT framework dependencies, we could substitute a different vendor's IoT infrastructure.

UI/Application Boundary

Next, let's look at the UI/application boundary. The user interface of the system is a Web browser on a tablet or smart phone. We used the popular Python/Flask Web server framework. The application runs on a small Linux box as two processes.

The `Collector` process collects measurements from the `HydraNetwork`. The `WebServer` process runs the customized Flask Web server in the main thread. The user has buttons to start and stop the flow test, which comes to the `Collector` in the `CommandQueue`. Measurement updates are sent asynchronously to the `ReplyQueue` once a synchronized group of measurements is collected.



The `Collector` process entry point is the `measurement_loop()` function, which waits on the `CommandQueue` with a one-second timeout. Each timeout initiates a `refresh/flow_update` of the measurement data all the way down to the ADC microcontroller (not shown in this diagram).

```
# Collector process entry function
def measurement_loop(command_q, reply_q,
                     command_runner, update_interval):
 command = None
  while command != HYDRA_EXIT:
    command = get_message(command_q, update_interval)
    if command == None:
      Command = HYDRA_REFRESH
    reply = command_runner.run(command)
    put_message(reply_q, reply)
```

There are a number of application objects not shown, but then they are not at the boundary. We'll look at a bigger picture soon.

Let's take a quick look at `hydra_run()`. This function is called from `main()` with its `data_source` injected. In production, a `HydraDataSource`, with its `HydraNetwork`, is injected. During development and testing, we have a simulated data source where a network is not needed.

The `hydra_run()` function puts everything in motion and is called from `main()`. The following code starts the `Collector` process running the Web server in the main thread of execution.

```
def hydra_run(port, data_source, polling_rate,
                                    logger):
  logger.info("Hydra start")
  command_runner = CommandRunner(data_source,
                                      logger)
  collector = Process(
                target=measurement_loop,
                args=(command_q, reply_q,

                      command_runner, polling_rate))
  collector.start()
  sleep(1)
  logger.info("Hydra collector started")
  run_hydra_web_server(port, command_q, reply_q)
  logger.info("Hydra web server exited")
  command_q.put(HYDRA_EXIT)
  collector.join()
```

```
  logger.info("Hydra shut down")
```

This is a little messy, as all the dependencies come together here. On the plus side, the code is not complex and probably won't change much.

Let's review some of the important attributes of our clean boundaries.

About the IoT isolation:

- The business logic is free from the radio system knowledge.
- IoT-dependent functions are low complexity.
- IoT-dependent functions are unlikely to change often or much.

About the UI isolation:

- Business logic is separate from the UI.
- Communications between the application to the Web server rely on a simple queuing mechanism.

About concurrency:

- None of the business logic is concerned with concurrency.
- Concurrency mechanisms are in low-complexity functions with a low risk of changes.

About object creation and binding:

- Object creation is separate from object usage.
- During initialization, the concrete objects are created and bound to processes and to each other.
- Binding is based on a need-to-know basis.

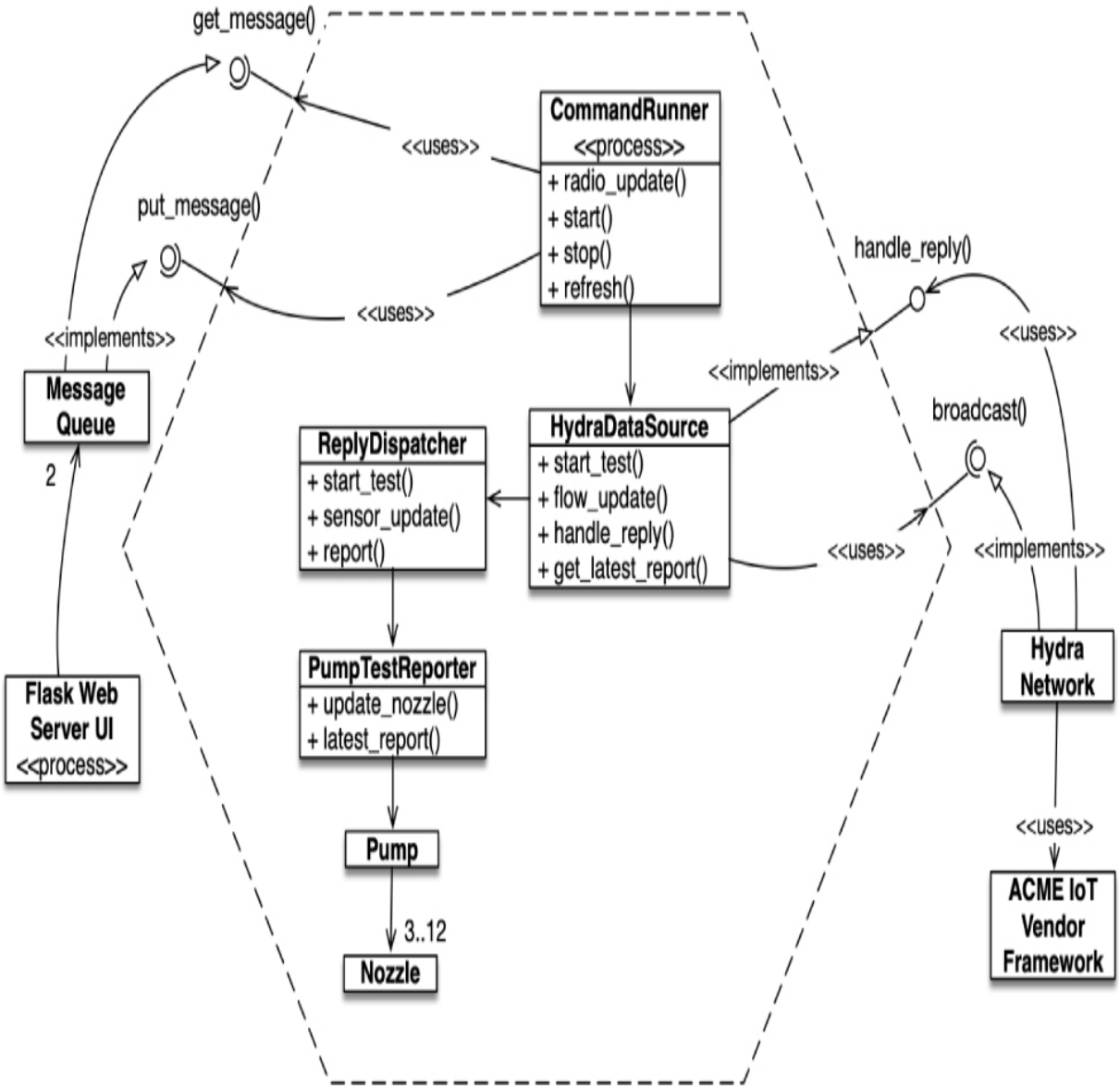
SOLID and Hexagonal Architecture

We've looked at the little pieces; now let's look at a bigger picture and the relationship between the classes in the design. Following the SOLID principles and isolating dependencies of the outside world naturally leads to something that Alistair Cockburn calls the *Hexagonal architecture*[2] (aka Ports and Adapters).

[2] [Hex].

Inside the hexagon is all the product's logic and behavior. Instead of a big ball of mud inside, responsibilities are allocated to separate classes, functions, and domain objects. Outside the hexagon is the execution environment. The application depends on service abstraction layers and the edge of the hexagon.

There are two `MessageQueue` instances: the `CommandQueue` and the `ReplyQueue`. The UI, through the `CommandQueue`, drives the system. The user starts a test with a button on the Web page, which writes the "start" command to the `CommandQueue`. The start command works its way to the sensors that start collecting measurements. The `CommandRunner` does a timed wait for new UI commands. Each one-second timeout causes the `CommandRunner` to initiate a new update from the sensors. The previous second's sensor readings are bundled into a report that is sent to the `ReplyQueue`. The following diagram shows how that message starts the flow test. We also saw the connection of the `CommandRunner` to the queues in the `measurement_loop()` in a code snippet earlier.



The `HydraNetwork` encapsulates the dependencies on the ACME IoT vendor framework. `HydraNetwork.broadcast()` tells all sensors in the network to report their latest measurements. As individual radio replies are received, `HydraNetwork` calls `handle_reply()`, feeding the latest `PressureSensor` messages into the Hydra application. Hydra knows nothing about the ACME IoT.

The interfaces of `broadcast()` and `handle_reply()` have more to do with what the application wants from the ACME IoT than what the ACME IoT offers. `HydraDataSource` knows it has a

network of sensors, but only needs it to broadcast the need for an update and to receive the updates. It’s important that the interfaces at the boundary of the hexagon are about the Hydra application’s needs and not about ACME’s offerings. The ACME IoT has facilities that are very close to our needs, so the adapter is very thin and has low complexity.

As our product, or the IoT vendor offerings, evolve, we may want to change vendors. If the time comes where we need to replace the ACME IoT or support multiple vendors, the `HydraNetwork` class would have to be evolved.

Imagine some new IoT vendor, BrandX, who could provide significant cost savings. Unfortunately, BrandX does not have a broadcast built in, as ACME does. Instead, each radio has to be addressed individually. The BrandX `HydraNetwork` implementation could implement `broadcast()`, but it would have knowledge of each sensor connection to do so. That layer might become nontrivial, but we could change the vendor without the risk of touching the Hydra core application.

Exploring and Learning Boundaries

Third-party code helps us get more functionality delivered in less time. Where do we start when we want to utilize some third-party package? It’s not our job to test the third-party code, but it may be in our best interest to write tests for the third-party code we use.

Suppose it is not clear how to use our third-party library. We might spend a day or two (or more) reading the documentation and deciding how we are going to use it. Then, we might write our code to use the third-party code and see whether it does what we think. We would not be surprised to find ourselves bogged down in long debugging

sessions trying to figure out whether the bugs we are experiencing are in our code or theirs.

Learning the third-party code is hard. Integrating the third-party code is hard too. Doing both at the same time is doubly hard.

What if we took a different approach? Instead of experimenting and trying out the new stuff in our production code, we could write some tests to explore our understanding of the third-party code. Jim Newkirk calls such tests *learning tests*.[3]

[3] [BeckTDD], pp. 136–137.

In learning tests, we call the third-party API, as we expect to use it in our application. We're essentially doing controlled experiments that check our understanding of that API. The tests focus on what we want out of the API.

Not having done concurrent programming in Python before, I thought it would be a good idea to learn how the concurrency and communications mechanisms work before using them. I could simply read the docs and integrate their usage into my production code, or I could experiment and learn from the comfort of test cases.

From looking at the documentation, I decided to test-drive two functions, `put_message()` and `get_message()`, that put and get messages from a queue.

Here are the tests that helped me know for sure that I understood how Python's `Queue` works.

```
class QueueTest(unittest.TestCase):

    def test_get_message_returns_message(self):

    q = Queue(maxsize=1)

    self.assertTrue(put_message(q, 'hey'))
```

```
    self.assertEqual('hey', get_message(q, 1.0))

  def test_get_message_None_when_timeout(self):
    q = Queue(maxsize=1)
    self.assertEqual(None, get_message(q, 0.01))

  def test_get_message_None_empty_no_wait(self):
    q = Queue(maxsize=1)
    self.assertEqual(None, get_message(q, 0.0))

  def test_put_message_True_for_success(self):
    q = Queue(maxsize=1)
    self.assertTrue(put_message(q, 'hey'))
    self.assertEqual('hey', get_message(q, 1.0))

  def test_put_message_False_when_full(self):
    q = Queue(maxsize=1)
    self.assertTrue(put_message(q, 'hey1'))
    self.assertFalse(put_message(q, 'hey2'))
    self.assertEqual('hey1', get_message(q, 1.0))
```

Now that we know how `Queue` functions work, let's use them to communicate with a `Process`. First, we define a function, `worker()`, that will be executed on a separate `Process`. It gets from `in_q`, puts to `out_q`, and then exits. At the time, we expected to use this approach to separate the Hydra core from the `WebServer`.

```
def worker(in_q, out_q):
  message = get_message(in_q, 0.05)
```

```
  if message:
    put_message(out_q, message)

class ProcessLearningTest(unittest.TestCase):

  def setUp(self):
    self.command_q = Queue(maxsize=1)
    self.reply_q = Queue(maxsize=1)
    self.process = Process(target=worker,
                       args=(self.command_q,
                       self.reply_q,))
    self.process.daemon = True

  def test_thread_q_message_echos(self):
    put_message(self.command_q, 'Do this')
    self.process.start()
    reply = self.reply_q.get(True, 0.1)
    self.process.join()
    self.assertEqual('Do this', reply)

  def test_thread_message_timeout(self):
    self.process.start()
    reply = get_message(self.reply_q, 0.1)
    self.process.join()
    self.assertEqual(None, reply))
```

These learning tests cost nothing. We have to learn mechanics anyway, and writing those tests was an easy and isolated way to get that knowledge. The learning tests were

precise experiments that helped increase our understanding.

Not only are learning tests free, they have a positive return on investment. When there are new releases of the third-party package, we run the learning tests to see whether there are behavioral differences.

Learning tests verify that the third-party packages we are using work the way we expect them to. Once integrated, there are no guarantees that the third-party code will stay compatible with our needs. The original authors will have pressures to change their code to meet new needs of their own. They will fix bugs and add new capabilities. With each release comes new risk. If the third-party package changes in some way that is incompatible with our tests, we will find out right away.

In this case, who expects Python to change? So, there is no future payback. Well, actually, Python did change. While converting the Hydra code from Python 2 to Python 3, the packaging changed; some edits were needed to compile and run the tests. Nicely, our learning tests showed us the behavior was preserved.

Regardless of whether you need the learning provided by the learning tests, a clean boundary should be supported by a set of boundary tests that exercise the interface the same way the production code does. Without these boundary tests to ease the migration, we might be tempted to stay with the old version longer than we should.

Using Code That Does Not Yet Exist

There is another kind of boundary, one that separates the known from the unknown. There are often places in the code where our knowledge seems to drop off the edge.

Sometimes what is on the other side of the boundary is unknowable (at least right now). Sometimes we choose to look no further than the boundary.

A number of years back I was part of a team developing software for a radio communications system. There was a subsystem, the "Transmitter," that we knew little about, and the people responsible for the subsystem had not gotten to the point of defining their interface. We did not want to be blocked, so we started our work away from the unknown part of the code.

We had a pretty good idea of where our world ended and the new world began. As we worked, we sometimes bumped up against this boundary. Though mists and clouds of ignorance obscured our view beyond the boundary, our work made us aware of what we *wanted* the boundary interface to be. We wanted to tell the transmitter something like this:

Key the transmitter on the provided frequency and emit an analog representation of the data coming from this stream.

We had no idea how that would be done, because the hardware and the API had not been designed yet. So we would have to work out the details later.

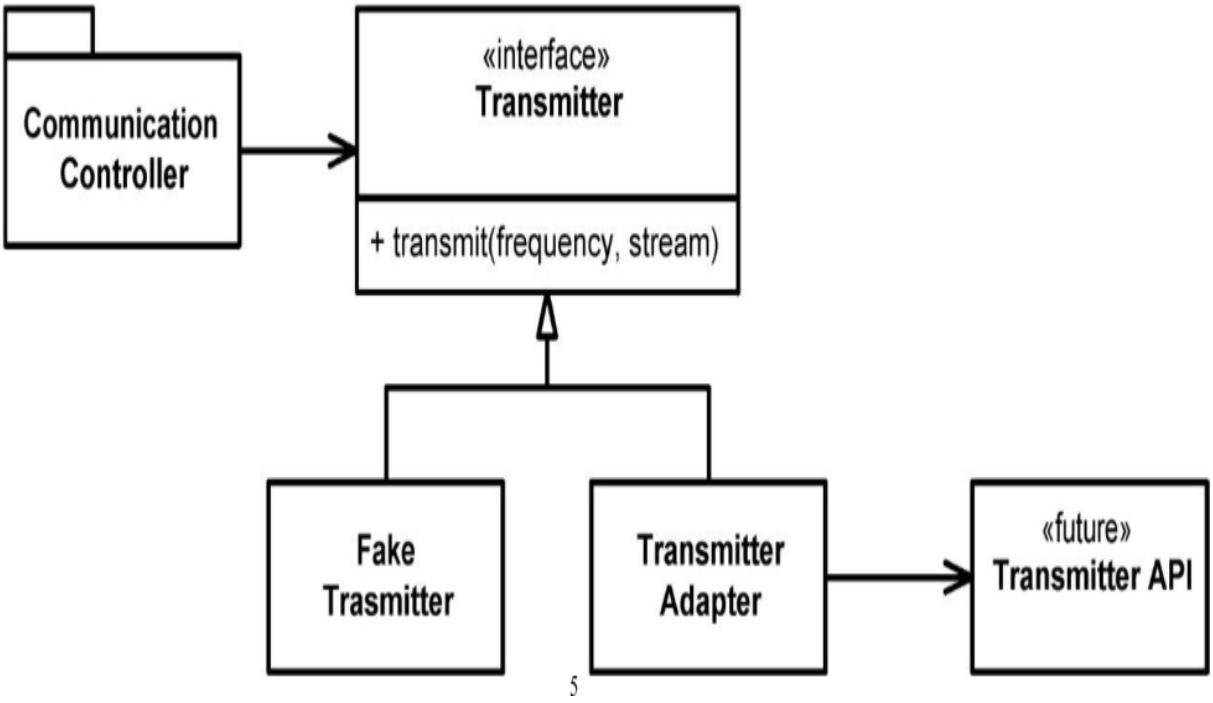
To keep from being blocked, we defined our own interface to the nonexistent hardware. We called it something catchy, like `Transmitter`. We gave it a method called `transmit` that took a frequency and a data stream. This was the interface we wished we had.

One good thing about writing the interface is that it's under our control. This helps keep client code more readable and focused on what it is trying to accomplish. On the client side of the boundary, we are not burdened by the many implementation details. If we had a hardware abstraction layer (HAL), we probably would have started depending on

that. HALs usually are not all that abstract from the application perspective. Not knowing the HAL, we had no way to depend on it. Instead of a liability, for now it was an advantage.

In the following diagram, you can see that we insulated the `CommunicationsController` classes from the transmitter API (which was out of our control and currently undefined). By using our own application-specific interface, we kept our `CommunicationsController` code clean and expressive. Once the transmitter API was defined, we wrote the `TransmitterAdapter` to bridge the gap. The Adapter[4] encapsulated the interaction with the API and provides a single place to change when the API evolves.

[4] See the Adapter pattern in [GOF95].



This design also gives us a very convenient seam[5] in the code for testing. Using a suitable `FakeTransmitter`, we can test the `CommunicationsController` classes. Once we have the `TransmitterAPI`, we can also create boundary tests that make sure we are using the API correctly.

[5] See more about seams in [WELC].

Clean Boundaries

Interesting things happen at boundaries. Change is one of those things. Good software designs accommodate change without huge investments and rework. When we use code that is out of our control, special care must be taken to protect our investment and make sure future change is not too costly.

Code at the boundaries needs clear separation and tests that define expectations. We should avoid letting too much of our code know about the third-party particulars. It’s better to depend on something *you* control than on something you don’t control, lest it end up controlling you.

We manage third-party boundaries by having very few places in the code that refer to them. We may wrap third-party dependencies in an ADAPTER to convert from our application-centered interface to the provided interface. Either way, our code speaks to us better, promotes internally consistent usage across the boundary, and has fewer maintenance points when the third-party code changes.

Part IV

Craftmanship



The following is an abridged and edited excerpt from my book Clean Craftsmanship*.*

The profession of software began, inauspiciously, in the summer of 1935 when Alan Turing began work on his paper.

His goal was to resolve a mathematical dilemma that had perplexed mathematicians for a decade or more: *the Entscheidungsproblem*. The decision problem.

In that goal, he was successful; but he had no idea, at the time, that his paper would spawn a globe-spanning industry upon which we would all depend and that now forms the lifeblood of our entire civilization.

In 1945, Turing wrote code for the Automatic Computing Engine (ACE). He wrote this code in binary machine language, using base 32 numbers. Fewer than a dozen people had ever written code like this before, so Turing had to invent and implement concepts like subroutines, stacks, and floating-point numbers.

“A Great Number”

After several months of inventing the basics and using them to solve mathematical problems, he wrote a report that stated the following conclusions.

“We shall need a great number of mathematicians of ability, because there will probably be a good deal of work of this kind to be done.”

“A great number.” How did he know? In reality, he had no idea just how prescient that statement was; he could not have envisioned the great number we have now.

But what was that other thing he said? “Mathematicians of ability.” Do you consider yourself to be a mathematician of ability? Turing did.

In the same report, he went on to write:

“One of our difficulties will be the maintenance of an appropriate discipline, so that we do not lose track of what we are doing.”

Discipline! Alan Turing looked forward through eight decades and saw that our problem would be discipline. And with that, he laid the first stone in the framework of software professionalism. He said that we should be mathematicians of ability who maintain an appropriate discipline.

Is that who we are? Is that who you are?

Eight Decades

One person’s lifetime. That’s how old our profession is as of this writing. And what has happened in those fourscore years?

In 1945, there were a handful of computers in the world, and an equally small number of programmers. In computer science terms, this is on the order of one: O(1). These numbers grew rapidly in those first years. But let’s use this as our origin.

1945. Computers: O(1). Programmers O(1).

In the decade that followed, the reliability, consistency, and power usage of vacuum tubes improved dramatically. This made it possible for larger and more powerful computers to be built.

By 1960, IBM had sold 140 of its 700 series computers. These were huge, expensive behemoths that could only be afforded by the military, the government, and very large corporations. They were, by our standards, slow, impoverished, and fragile.

It was in the ’50s that Grace Hopper invented the concept of a higher-level language and coined the term *compiler*. By 1960, her work led to COBOL.

In 1952, John Backus submitted the FORTRAN specification. This was followed rapidly by the development of ALGOL. By 1958, John McCarthy had developed LISP. So the proliferation of the language zoo had begun.

In those days, there were no operating systems, no frameworks, no function libraries. If something executed on your computer, it was because you wrote it. So, in those days, it took a staff of a dozen or more programmers just to keep one computer running.

By 1960, 15 years after Turing, there were O(100) computers in the world. The number of programmers was an order of magnitude greater: O(1000).

Who were these programmers? They were people like Grace Hopper, Edsger Dijkstra, John von Neumann, John Backus, and Jean Jennings. They were scientists and mathematicians and engineers. Most were people who already had careers and already understood the businesses and disciplines they were employed by. Many, if not most, were in their 30s, 40s, and 50s.

The 1960s was the decade of the transistor. Bit by bit these small, simple, inexpensive, and reliable devices replaced the vacuum tube. And the effect on computers was a game changer.

By 1965, IBM had produced over 10,000 transistor-based 1401 computers. They rented for about $2,500 per month, putting them within the reach of thousands of medium-sized businesses.

These machines were programmed in Assembler, Fortran, COBOL, and RPG. And all those companies who rented those machines needed staffs of programmers to write their applications.

IBM wasn’t the only company making computers at the time; so we’ll just say that by 1965, there were O(1E4) computers in the world. And if each computer needed ten programmers to keep it running, there must have been O(1E5) programmers.

Twenty years after Turing, there must have been several hundred thousand programmers in the world. Where did these programmers come from? There weren’t enough mathematicians, scientists, and engineers to cover the need. And there weren’t any computer science graduates coming out of the universities, because there weren’t any computer science degree programs—anywhere.

So companies drew from the best and brightest of their accountants, clerks, planners—anyone with some proven technical aptitude. And they found lots.

And, again, these were people who were already professionals in another field. They were in their 30s, 40s, and 50s. They already understood deadlines and commitments, what to leave in, what to leave out.[1] And, although these people were not mathematicians per se, they were disciplined professionals. Turing would likely have approved.

[1] Apologies to Bob Seeger.

But the crank kept on turning. By 1966, IBM was producing 1,000 System/360s every month. These computers were popping up everywhere. They were immensely powerful for the day. The model 30 could address 64K bytes of memory and execute 35,000 instructions per second.

It was during this period, in the mid-’60s, when Ole-Johan Dahl and Kristen Nygaard invented SIMULA 67, the first object-oriented language.

It was also during this period that Edsger Dijkstra invented structured programming.

And it was also during this time that Ken Thompson and Dennis Ritchie invented C and Unix.

And still the crank turned. In the early ’70s, the integrated circuit came into regular use. These little chips could hold dozens, hundreds, even thousands of transistors. They allowed electronic circuits to be massively miniaturized.

And so the minicomputer was born.

In the late ’60s and into the ’70s, Digital Equipment Corporation sold 50,000 PDP8 systems and hundreds of thousands of PDP11 systems.

And they weren’t alone! The minicomputer market exploded. By the mid-’70s, there were dozens and dozens of companies selling minicomputers.

And so by 1975, 30 years after Turing, there were O(1E6) computers in the world. And how many programmers were there? The ratio was starting to change. The number of computers was approaching the number of programmers. So, by 1975, there were O(1E6) programmers.

Where did these millions of programmers come from? Who were they?

They were me. Me and my buddies. Me and my cohort of young, energetic, geeky boys.

Tens of thousands of new EE and CS grads—we were all young. We were all smart. We, in the United States, were all concerned about the draft. And we were almost all male.

Oh, it’s not that women were leaving the field in any number—yet. That didn’t start until the mid-’80s. No, it’s

just that many more boys (and we were boys) were entering the field.

In my first job as a programmer, in 1969, there were a couple of dozen programmers. They were all in their 30s or 40s, and one-third to one-half were women.

Ten years later, I was working at a company with about fifty programmers, and perhaps three were women.

So, 30 years after Turing, the demographics of programming had shifted dramatically toward very young men. Hundreds of thousands of twenty-something males.

We were typically *not* what Turing would have described as disciplined mathematicians.

But businesses had to have programmers. The demand was through the roof. And what very young men lack in discipline, they make up for with energy.

And we were also cheap. Despite the high starting salaries of programmers today, back then companies could pick up programmers pretty inexpensively. My starting salary in 1969 was $7,200 per year.

And this has been the trend ever since. Young men and women have been pouring out of computer science programs every year; and industry seems to have an insatiable appetite for them.

In the 30 years between 1945 and 1975, the number of programmers grew by at least a factor of a million. In the 40 years since then, that growth rate has slowed a bit, but it is still very high.

How many programmers do you think are in the world this year, 2025? If you include the VBA programmers, I think the

number must be O(1E8). There must be hundreds of millions of programmers in the world today.

This is clearly exponential growth. Exponential growth curves have a doubling rate. You can do the math. Hey Albert, what's the doubling rate if we go from 1 to 1E8 in 80 years?

The log to the base 2 of one hundred million is approximately 27—divide that into 80 and you get about 2.96. So perhaps the number of programmers doubled roughly every three-ish years.

Actually, as we saw earlier, the rate was higher in the first decades and has slowed down a bit now. My guess is that the doubling rate is about five years. Every five years the number of programmers in the world doubles.

The implications of that hypothesis are staggering. If the number of programmers in the world doubles every five years, it means that half the programmers in the world have less than five years of experience; and this will always be true so long as that doubling rate continues. This leaves the programming industry in the precarious position of—perpetual inexperience.

Nerds and Saviors

Perpetual inexperience. Oh, don't worry, this doesn't mean that *you* are perpetually inexperienced. It just means that once you gain five years of experience, the number of programmers will have doubled. By the time you gain ten years of experience, the number of programmers will have quadrupled.

Perhaps you think AI will slow this rate. Don't be deluded. Every new software technology that was supposed to slow that rate has only increased it. Pick a new software technology. Fortran, C, Unix, OO, C++, Java, C#, Ruby,

Functional . . . Not one of them decreased the demand for new programmers. If anything, the demand accelerated because each of those technologies made the field accessible to larger and larger groups of people—mostly young.

People look at the number of young people in programming and conclude that it’s a young person’s profession. They ask, “Where are all the old people?”

We’re all still here! We haven’t gone anywhere. There just weren’t that many of us 50 years ago.

The problem is that there aren’t enough of us old guys to teach the new programmers coming in. For every programmer with 30 years of experience, there are ~63 programmers who need to learn something from her (or him); 32 of whom are brand new.

So, perpetual inexperience, with insufficient mentors to correct the problem. And so the same old mistakes get repeated over, and over, and over again.

Notoriety

But something else has happened in the last 80 years. Programmers have gained something that I am sure Alan Turing never anticipated: notoriety.

Back in the ’50s and ’60s, nobody knew what a programmer was. There weren’t enough of us to have a social impact. Programmers did not live next door to very many people.

That started to change in the ’70s. By then, fathers were advising their sons (and sometimes their daughters) to get degrees in computer science. There were enough programmers in the world so that everybody knew

somebody who knew one. And the image of the nerdy, Twinkie-eating geek was born.

Few people had seen a computer; but virtually everyone had heard about them. Computers showed up in TV shows, like *Star Trek*, and in movies, like *2001: A Space Odyssey* and *Colossus: The Forbin Project*. All too often in those shows, the computers themselves were cast as villains.

Notice, however, that in each of these cases, the programmer is not a significant character. Society didn't know what to make of programmers back then. They were shadowy, hidden, and somehow insignificant compared to the machines themselves.

I have fond memories of one television commercial from this era. A wife and her husband, a nerdy little guy with glasses, a pocket protector, and a calculator, were comparing prices at a grocery store. Mrs. Olsen described him as "a computer genius" and proceeded to school the wife and husband alike on the benefits of a particular brand of coffee.

The computer programmer in that commercial was naive, bookish, and inconsequential. Someone smart, but with no wisdom or common sense. Not someone you'd invite to parties.

Indeed, computer programmers were seen as the kind of people who got beaten up a lot at school.

By 1983, personal computers started to appear; and it was clear that teenagers were interested in them for lots of reasons. By this time, a rather large number of people knew at least one computer programmer. We were considered professionals; but still mysterious.

That year, the movie *War Games* depicted a young Matthew Broderick as a computer-savvy teenager and hacker. He

hacks into WOPR, the US weapons control system, thinking it’s a video game and starts the countdown to thermonuclear war. At the end of the movie, he saves the world by convincing the computer that the only winning move is not to play.

The computer and the programmer had switched roles. Now the computer was the childlike, naive character and the programmer the conduit, if not the source, of wisdom.

We saw something similar in 1986 in the movie *Short Circuit* in which the computerized robot known as #5 is childlike and innocent but learns wisdom with the help of its bumbling creator and his girlfriend.

By 1993, things had changed dramatically. In the film *Jurassic Park*, the programmer was the villain and the computer was not a character at all. It was just a tool.

Society was beginning to understand who we were and the role we played. We had graduated from nerd to teacher to villain in just 20 years.

But the vision changed again. In the 1999 film *The Matrix*, the main characters were both programmers and saviors. Indeed, their godlike powers came from their ability to read and understand “the code.”

Our roles were changing fast. Villain to savior in just a few years. Society at large was beginning to understand the power we have for both good and evil.

Role Models and Villains

Fifteen years later, in 2014, I visited the Mojang office in Stockholm to do give lectures on clean code and test-driven development (TDD).

Mojang, in case you didn’t know, is the company that produced the game *Minecraft*.

Afterward, since the weather was nice, the Mojang programmers and I sat outside at a beer garden, chatting.

All of a sudden a young boy, perhaps 12, ran up to the fence and called out to one of the programmers: “Are you Jeb?”

He was referring to Jens Bergensten, one of the lead programmers at Mojang.

The lad asked Jens for his autograph and peppered him with questions. He had eyes for no one else.

Programmers have become role models and idols for our children. They dream of growing up to be like Jeb, or Dinnerbone, or Notch.

Programmers, real-life programmers, are heroes.

But where there are heroes, there are also villains.

In October of 2015, Michael Horn, the CEO of Volkswagen North America, testified before the Congress of the United States regarding the software in their cars that was cheating the EPA testing devices.

When asked why the company did this, he blamed programmers. He said, “This was a couple of software engineers who put this in for whatever reasons.”

Of course he was lying about the “whatever reasons.” He knew perfectly well what the reasons were, and so did the Volkswagen company at large. His feeble attempt to shift blame onto the programmers was pretty transparent.

On the other hand, he was exactly right. It *was* some programmers who wrote that lying, cheating code.

And those programmers, whoever they were, gave us all a bad name. If we had a true professional organization, their recognition as programmers would, and should, be revoked. They betrayed us all. They besmirched the honor of our profession.

And so we've graduated. It's taken 80 years. But we've gone from nothing, to nerds, to role models, to heroes and villains in that time.

Society has begun—just begun—to understand who we are and the threats and promises that we represent. (For more on that, see the section "Why Should We Be Clean?" in Chapter 1, "Clean Code.")

We Rule the World

But society doesn't understand everything yet. Indeed, neither do we. As I told you in the introduction, you and I, we programmers; we rule the world.

That may seem like an exaggerated statement. But consider. There are more computers in the world right now than there are people. And the computers that outnumber us perform a myriad of essential tasks for us.

They keep track of our reminders. They manage our calendars. They deliver our Facebook messages and keep our photo albums. They connect our phone calls and deliver our text messages.

They control the engines in our cars, as well as the brakes, the accelerator, and sometimes even the steering wheels.

We can't cook without them. We can't wash clothes without them. They keep our houses warm in winter. They sweep and mop our floors. They entertain us when we're bored.

They keep track of our bank records and our credit cards. They help us pay our bills.

In fact, most people in the western world interact with some software system every waking minute of every day. Some even continue interacting while they sleep.

The point is that *nothing* happens in our society without software. No product gets bought or sold. No law gets enacted or enforced. No car drives. No Amazon products get delivered. No phone connects. No power comes out of outlets. No food gets delivered to stores. No water come out of faucets. No gas gets piped to furnaces. Without software monitoring and coordinating it all.

And *WE* write that software. And that makes *us* the rulers of the world.

Oh, other people think they make the rules—but then they hand those rules to us, and *WE* write the rules that execute in the machines that monitor and coordinate every aspect of our lives.

Society does not quite understand this yet. Not quite. Not yet. But the day is coming soon when our society will understand it all too well.

We, programmers, don’t quite understand this yet either. Not really. But again, the day is coming when it will be savagely driven home to us.

Catastrophes

We’ve seen plenty of software catastrophes over the years. Some have been pretty spectacular.

Between 2018 and 2019, two 737 MAX aircraft dove headlong into the ground at near the speed of sound, killing

346 people. The reason: a software weakness exposed by a hardware failure.

In 2016, we lost the Schiaparelli Mars lander and rover due to a software issue that caused the lander to believe it had already landed, when it was actually nearly 4 kilometers above the surface.

In 1999, we lost the Mars Climate Orbiter due to a ground-based software error that transmitted data to the orbiter using English units (pound-seconds) rather than metric units (newton-seconds). This error caused the orbiter to descend too far into the Martian atmosphere, where it was torn to pieces.

In 1996, the Ariane 5 launch vehicle and payload were destroyed 37 seconds after launch due to an integer overflow exception when a 64-bit floating-point number underwent an unchecked conversion to a 16-bit integer. The exception crashed the onboard computers, and the vehicle self-destructed.

Should we talk about the Therac-25 radiation therapy machine that, due to a race condition, killed three people and injured three others by blasting them with a high-powered electron beam?

Or maybe we should talk about Knight Capital, who lost $460 million in 45 minutes because they reused a flag that activated dead code inadvertently left in the system.

Or perhaps we should talk about the Toyota stack overflow bug that caused cars to accelerate out of control—killing perhaps as many as 89 people.

Or maybe we should talk about Healthcare.gov and the software failure that nearly overturned a new and controversial US healthcare law.

These disasters have cost billions of dollars and many lives. And they were caused by programmers.

We, programmers, through the code that we write, are killing people.

Now I know you didn’t get into this business in order to kill people. Probably you became a programmer because you wrote an infinite loop that printed your name once, and you experienced that joyous feeling of power.

But facts are facts. We are now in a position in our society where our actions can destroy fortunes, livelihoods, and lives.

One day, probably not to long from now, some poor programmer is going to do something just a little dumb, and tens of thousands of people will die.

This isn’t wild speculation—it’s just a matter of time.

And when this happens, the politicians of the world will demand an accounting. They will demand that we show them how we will prevent this in the future.

And if we show up without a statement of ethics; if we show up without any standards, or defined disciplines; if we show up and whine about how our bosses set unreasonable schedules and deadlines . . .

Then we will be found GUILTY.

The Oath

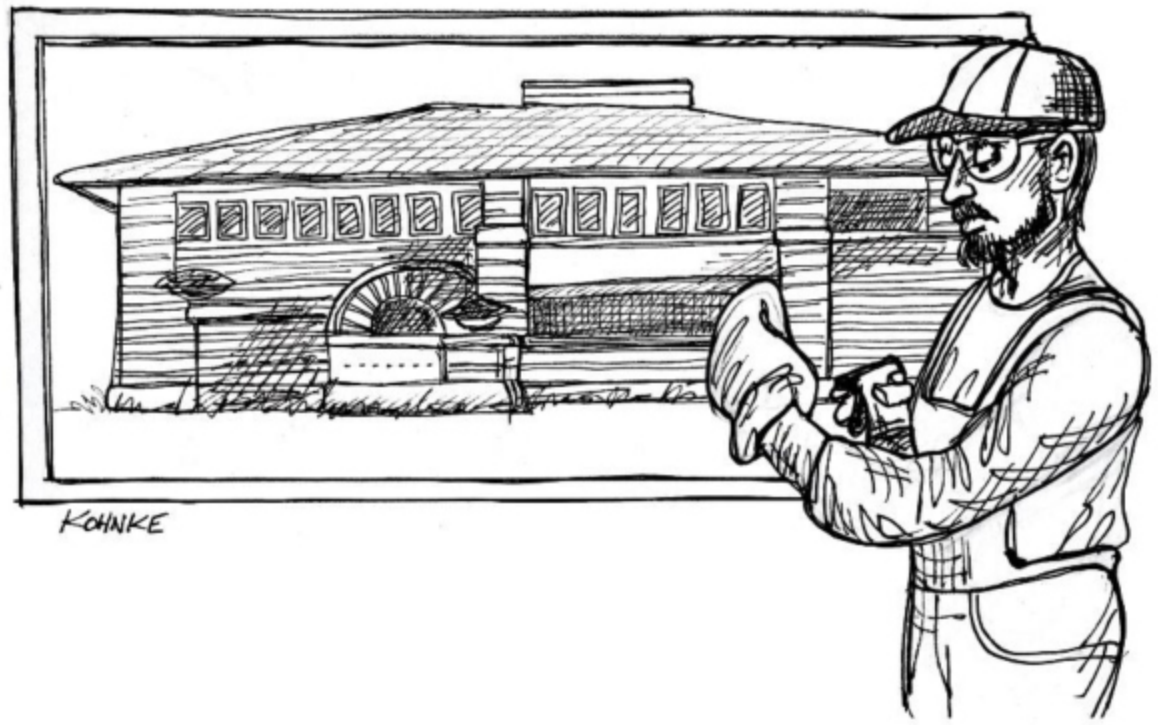
And so, to begin the definition of our ethics as software developers, I offer the following oath.

In order to defend and preserve the honor of the profession of computer programmers, I promise that, to the best of my

ability and judgment:

1. I will not produce harmful code.
2. The code that I produce will always be my best work. I will not knowingly allow code that is defective either in behavior or in structure to accumulate.
3. I will produce, with each release, a quick, sure, and repeatable demonstration that every element of the code works as it should.
4. I will make frequent, small releases so that I do not impede the progress of others.
5. I will fearlessly and relentlessly improve my creations at every opportunity. I will never degrade them.
6. I will do all that I can to keep the productivity of myself and others as high as possible. I will do nothing that decreases that productivity.
7. I will continuously ensure that others can cover for me and that I can cover for them.
8. I will produce estimates that are honest both in magnitude and precision. I will not make promises without reasonable certainty.
9. I will respect my fellow programmers for their ethics, standards, disciplines, and skill. No other attribute or characteristic will be a factor in my regard for my fellow programmers.
10. I will never stop learning and improving my craft.

Chapter 27. The Clean Architecture



Over the last several decades, we've seen a whole range of ideas regarding the architecture of systems. These include

- Hexagonal architecture (aka Ports and Adapters) by Alistair Cockburn, and adopted by Steve Freeman and Nat Pryce in their wonderful book *Growing Object-Oriented Software, Guided by Tests* (Addison-Wesley, 2009)

- Data, Context, and Interaction (DCI) architecture by James Coplien and Trygve Reenskaug
- Boundary–Control–Entity (BCE) architecture by Ivar Jacobson from his book *Object-Oriented Software Engineering: A Use-Case Driven Approach* (Addison-Wesley, 1992)

Though these architectures all vary somewhat in their details, they are very similar. They all have the same objective, which is the separation of concerns. They all achieve this separation by dividing the software into layers. Each has at least one layer for business rules and another for user and system interfaces.

Each of these architectures produce systems that are

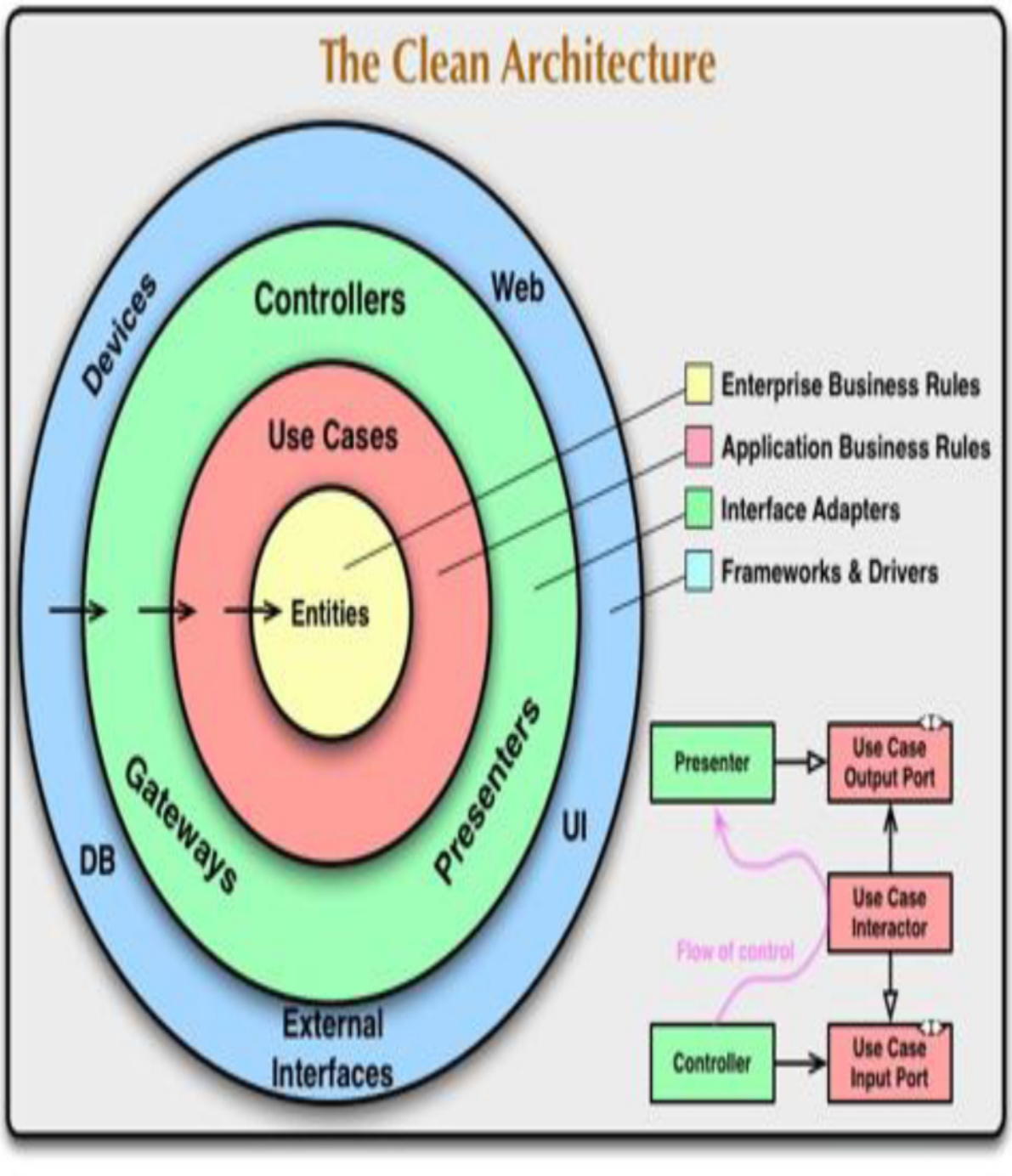
- Independent of frameworks. The architecture does not depend on the existence of some library of feature-laden software. This allows you to use such frameworks as tools, rather than forcing you to cram your system into their limited constraints.
- Testable. The business rules can be tested without the UI, database, Web server, or any other external element.
- Independent of UI. The UI can change easily, without changing the rest of the system. A Web UI could be replaced with a console UI, for example, without changing the business rules.
- Independent of database. You can swap out Oracle or SQL Server for Mongo, Bigtable, CouchDB, or something else. Your business rules are not bound to the database.
- Independent of any external agency. In fact, your business rules simply don’t know anything at all about

the interfaces to the outside world.

The Dependency Rule

The diagram that follows is an attempt at integrating all these architectures into a single actionable idea.

The concentric circles represent different areas of software. In general, the further in you go, the higher level the software becomes. The outer circles are mechanisms. The inner circles are policies.



The overriding rule that makes this architecture work is the Dependency Rule. This rule says:

Source code dependencies must only point inward, toward higher-level policies.

Nothing in an inner circle can know anything at all about something in an outer circle. In particular, the name of

something declared in an outer circle must not be mentioned by the code in an inner circle. That includes functions, classes, variables, or any other named software entity.

By the same token, data formats declared in an outer circle should not be used by an inner circle, especially if those formats are generated by a framework in an outer circle. We don’t want anything in an outer circle to impact the inner circles.

Entities

Entities encapsulate enterprise-wide critical business rules. An entity can be an object with methods, or it can be a set of data structures and functions. It doesn’t matter so long as the entities can be used by many different applications in the enterprise.

If you don’t have an enterprise and are just writing a single application, then these entities are the business objects of the application. They encapsulate the most general and high-level rules. They are the least likely to change when something external changes. For example, you would not expect these objects to be affected by a change to page navigation or security. No operational change to any particular application should affect the entity layer.

Use Cases

The software in this layer contains *application-specific* business rules. It encapsulates and implements all of the use cases of the system. These use cases orchestrate the flow of data to and from the entities, and direct those entities to use their critical business rules to achieve the goals of the use case.

We do not expect changes in this layer to affect the entities. We also do not expect this layer to be affected by changes to externalities such as the database, the UI, or any of the common frameworks. This layer is isolated from such concerns.

We do, however, expect that changes to the operation of the application will affect the use cases and therefore the software in this layer. If the details of a use case change, then some code in this layer will certainly be affected.

Interface Adapters

The software in this layer is a set of adapters that convert data from the format most convenient for the use cases and entities to the format most convenient for some external agency such as the database or the Web. It is this layer, for example, that will wholly contain the MVC architecture of a GUI. The presenters, views, and controllers all belong in here. The models are likely just data structures that are passed from the controllers to the use cases, and then back from the use cases to the presenters and views.

Similarly, data is converted in this layer from the form most convenient for entities and use cases into the form most convenient for whatever persistence framework is being used (i.e., the database). No code inward of this circle should know anything at all about the database. If the database is a SQL database, then all the SQL should be restricted to this layer, and in particular to the parts of this layer that have to do with the database.

Also in this layer is any other adapter necessary to convert data from some external form, such as an external service, to the internal form used by the use cases and entities.

Frameworks and Drivers

The outermost layer is generally composed of frameworks and tools such as the database and the Web framework. Generally, you don’t write much code in this layer other than glue code that communicates to the next circle inward.

This layer is where all the details go. The Web is a detail. The database is a detail. We keep these things on the outside, where they can do little harm.

Only Four Circles?

No, the circles are schematic. You may find that you need more than just these four. There’s no rule that says you must always have just these four. However, the Dependency Rule always applies. Source code dependencies always point inward. As you move inward, the level of abstraction and policy increases. The outermost circle is low-level concrete detail. As you move inward, the software grows more abstract and encapsulates higher-level policies. The innermost circle is the most general and highest level.

Crossing Boundaries

At the lower right of the diagram is an example of how we cross the circle boundaries. It shows the `Controllers` and `Presenters` communicating with the `Use Cases` in the next layer. Note the flow of control. It begins in the controller, moves through the use case, and then winds up executing in the presenter. Note also the source code dependencies. Each one of them points inward toward the use cases.

We usually resolve this apparent contradiction by using the Dependency Inversion Principle (DIP). In a language like Java, for example, we would arrange interfaces and

implements/inheritance relationships such that the source code dependencies oppose the flow of control at just the right points across the boundary.

For example, consider that the use case needs to call the presenter. However, this call must not be direct, because that would violate the Dependency Rule: No name in an outer circle can be mentioned by an inner circle. So we have the use case call an interface (shown here as `Use Case Output Port`) in the inner circle, and have the presenter in the outer circle implement it.

The same technique is used to cross all the boundaries in the architectures. We take advantage of dynamic polymorphism to create source code dependencies that oppose the flow of control so that we can conform to the Dependency Rule no matter in what direction the flow of control is going.

What Data Crosses the Boundaries

Typically, the data that crosses the boundaries are simple data structures. You can use basic structs or simple data transfer objects (DTOs) if you like. Or the data can simply be arguments in function calls. Or you can pack it into a hash map or construct it into an object. The important thing is that isolated, simple data structures are passed across the boundaries. We don’t want to cheat and pass entities or database rows. We don’t want the data structures to have any kind of dependency that violates the Dependency Rule.

For example, many database frameworks return a convenient data format in response to a query. We might call this a *row structure*. We don’t want to pass that row structure inward across a boundary. That would violate the Dependency Rule because it would force an inner circle to know something about an outer circle.

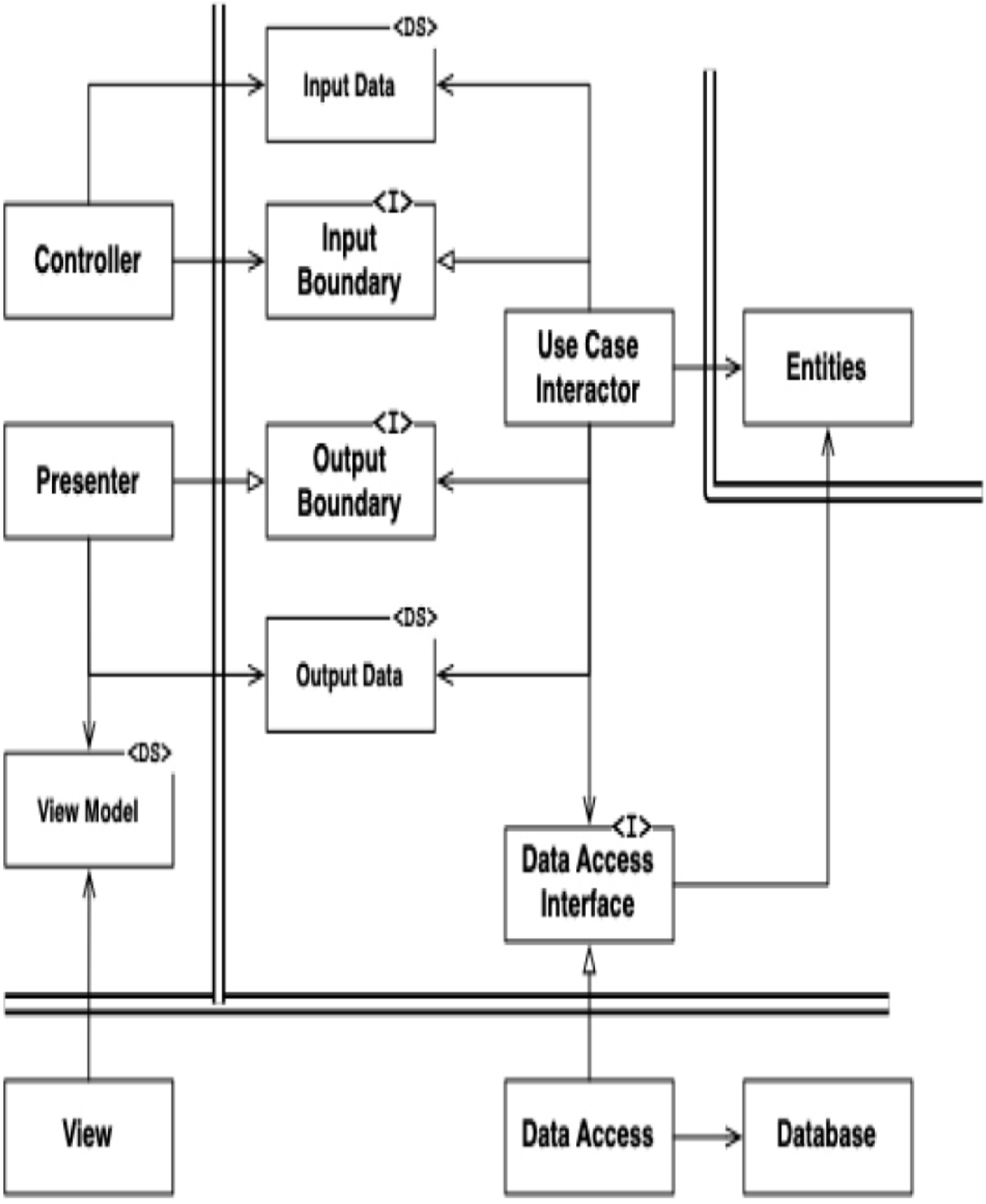
So, when we pass data across a boundary, it is always in the form that is most convenient for the inner circle.

A Typical Scenario

The following shows a typical scenario for a Web-based system using a database. The Web server gathers input data from the user and hands it to the `Controller` on the upper left. The `Controller` packages that data into a plain old object and passes it through the `InputBoundary` to the `UseCaseInteractor`. The `UseCaseInteractor` interprets that data and uses it to control the dance of the `Entities`. It also uses the `DataAccessInterface` to bring the data used by those `Entities` into memory from the `Database`. Upon completion, the `UseCaseInteractor` gathers data from the `Entities` and constructs the `OutputData` as another plain old object. The `OutputData` is then passed through the `OutputBoundary` interface to the `Presenter`.

The job of the `Presenter` is to repackage the `OutputData` into viewable form as the `ViewModel`, which is yet another plain old object. The `ViewModel` contains mostly `Strings` and flags that the `View` uses to display the data. Whereas the `OutputData` may contain `Date` objects, the `Presenter` will load the `ViewModel` with corresponding `Strings` already formatted properly for the user. The same is true of `Currency` objects, or any other business-related data. `Button` and `MenuItem` names are placed in the `ViewModel`. So are flags that tell the `View` whether those `Button`s and `MenuItem`s should be gray.

This leaves the `View` with almost nothing to do other than move the data from the `ViewModel` into the HTML page.



Note the directions of the dependencies. All dependencies cross the boundary lines pointing inward, following the Dependency Rule.

Conclusion

Conforming to these simple rules is not hard, and will save you a lot of headaches going forward. By separating the software into layers and conforming to the Dependency Rule, you will create a system that is intrinsically testable, with all the benefits that implies. When any of the external parts of the system become obsolete, like the database or the Web framework, you can replace those obsolete elements with a minimum of fuss.

Chapter 28. Harm

1. I will not produce harmful code.

The first promise of the software professional is: DO NO HARM! And that means that your code must not harm your users, your employers, your managers, or your fellow programmers.

You must know what your code does. You must know that it works. And you must know that it is clean.

Sometime back, it was discovered that some programmers at Volkswagen wrote some code that purposely thwarted EPA emissions tests. Those programmers wrote harmful code.

It was harmful because it was deceitful. That code fooled the EPA into allowing cars to be sold that emitted 20 times the amount of harmful nitrous oxides as the EPA deemed safe. Therefore, that code potentially harmed the health of everyone living where those cars were driven.

What should happen to those programmers? Did they know the purpose of that code? Should they have known?

I'd fire them and prosecute them. Because whether they knew or not, they should have known. Hiding behind requirements written by others is no excuse. It's our fingers on the keyboard. It's our code. We must know what it does!

That’s a tough one, isn’t it? We write the code that makes our machines work. And those machines are often in positions to do tremendous harm. Since we will be held responsible for any harm our code does, we must be responsible for knowing what our code will do.

Each programmer should be held accountable based on their level of experience and responsibility. As you advance in experience and position, your responsibility for your actions and the actions of those under your charge increases.

Clearly, we can’t hold junior programmers as responsible as team leads. We can’t hold team leads as responsible as senior developers. But those senior people ought to be held to a very high standard, and be ultimately responsible for those whom they direct.

That doesn’t mean that all the blame goes to the senior developers or the managers. Every programmer is responsible for knowing what the code does to the level of their maturity and understanding. Every programmer is responsible for the harm their code does.

No Harm to Society

First, you will do no harm to the society in which you live.

This is the rule that the VW programmers broke. Their software might have benefited their employer—Volkswagen. However, it harmed society in general. And we, programmers, must never do that.

But how do you know if you are harming society or not? For example, is building software that controls weapons systems harmful to society? What about gambling software? What

about violent or sexist video games? What about pornography?

If your software is within the law, might it still be harmful to society?

Frankly, that’s a matter for your own judgment. You’ll just have to make the best call you can. Your conscience is going to have to be your guide. Choose wisely!

Another example of harm to society was `Hhealthcare.gov`, the website that was mandated by the Affordable Care Act. Although in this case the harm was unintentional.

The Affordable Care Act was a bill passed by the Congress of the United States and signed into law by the president. Among its many directives was the demand that a website be created and activated on October 1, 2013.

Now, never mind the insanity of specifying, by law, a date by which a whole new massive software system must be activated. The real problem was that on October 1, 2013, they actually turned it on.

Do you think, maybe, there were some programmers hiding under their desks that day?

Oh man, I think they turned it on.

Yeah, yeah, they really shouldn’t be doing that.

Oh, my poor mother. Whatever will she do?

I won’t bore you with a laundry list of the problems that ensued. Suffice it to say that the website did not work—at all—for months. This is a case where a technical screwup put a huge new public policy at risk. The law was very nearly overturned because of these failures. And no matter what you think about the politics of the situation, that was harm to the society.

Who was responsible for that harm? Every programmer, team lead, manager, and director who knew that system wasn’t ready and who nevertheless remained silent.

That harm upon society was perpetrated by every software developer who maintained a passive-aggressive attitude toward their management. Everyone who said: “I’m just doing my job—it’s their problem.”

Every software developer who knew something was wrong, and yet did nothing to stop the deployment of that system, shares part of the blame.

Because here’s the thing. One of the prime reasons you were hired as a programmer was because you know when things are going wrong. You have the knowledge to identify trouble before it happens. And therefore, you have the responsibility to speak up before something terrible happens.

Harm to Function

You must KNOW that your code works. You must KNOW that the functioning of your code will not do harm to your company, your users, or your fellow programmers.

On August t, 2012, some technicians at Knight Capital loaded their servers with new software. Unfortunately, they loaded only seven of the eight servers, leaving the eighth running with the older version.

Why they made this mistake is anybody’s guess. Somebody got sloppy.

Knight Capital ran a trading system. They traded stocks on the NYSE. Part of their operation was to take large “parent” trades and break them up into many smaller “child” trades

in order to prevent other traders from seeing the size of the initial parent trade and adjusting prices accordingly.

Eight years earlier, a simple version of this parent–child algorithm, named Power-Peg, was disabled and replaced with something much better, called SMARS. Oddly, however, the old Power-Peg code was not removed from the system. It was simply disabled with a flag.

That flag had been used to regulate the parent–child process. When the flag was on, child trades were made. When enough child trades had been made to satisfy the parent trade, the flag was turned off.

They disabled the Power-Peg code by simply leaving the flag off.

Unfortunately, the new software update that made it onto only seven of the eight servers repurposed that flag. They turned it on. And that eighth server started making child trades in a high-speed infinite loop.

They knew something had gone wrong, but they didn't know exactly what. It took them 45 minutes to shut down that errant server; 45 minutes while it was making bad trades in an infinite loop.

The bottom line is that, in those first 45 minutes of trading, Knight Capital had unintentionally purchased over $7 billion worth of stock that it did not want, and had to sell it off at a $460 million loss. Worse, they only had $360M in cash. They were bankrupt.

Forty-five minutes; one dumb mistake; $460 million.

And what was that mistake? The mistake was that they did not KNOW what their system was going to do.

At this point, you may be worried that I'm demanding that programmers have perfect knowledge about the behavior of their code. Of course, perfect knowledge is not achievable. There will always be a knowledge deficit of some kind.

The issue isn't the perfection of knowledge. Rather, the issue is to KNOW that there will be no harm.

Those poor guys at Knight Capital had a knowledge deficit that was horribly harmful—and given what was at stake, it was one they should not have allowed to exist.

Or let's take the case of Toyota, and the software system that made cars accelerate uncontrollably.

As many as 89 people were killed by that software. And more were injured.

Imagine that you are driving in a crowded downtown business district. Imagine that your car suddenly begins to accelerate, and your brakes stop working. Within seconds, you are rocketing through stop lights and crosswalks, with no way to stop.

That's what investigators found that the Toyota software could do—and probably had done.

The software was killing people.

The programmers who wrote that code did not KNOW that their code would not kill. Notice the double negative. They did not KNOW that their code would NOT kill. And they should have known that. They should have known that their code would NOT kill.

Now again, this is all about risk. When the stakes are high, you want to drive your knowledge as near to perfection as you can get it. If lives are at stake, you have to KNOW that

your code won’t kill anyone. If fortunes are at stake, you have to KNOW that your code won’t lose them.

On the other hand, if you are writing a chat application or a simple shopping cart website, neither fortunes nor lives are at stake . . .

. . . or are they?

What if someone using your chat application starts having a medical emergency and types: “HELP ME. CALL 911” on your app? What if your app malfunctions and drops that message?

What if your website leaks personal information to hackers who use it to steal identities?

And what if the poor functioning of your code drives customers away from your employer and to a competitor?

The point is that it’s easy to underestimate the harm that can be done by software. It’s comforting to think that your software can’t harm anyone, because it’s not important enough.

But you forget that software is very expensive to write, and at the very least, the money spent on its development is at stake, not to mention whatever the users put at stake by depending on it. The bottom line is that there’s almost always more at stake than you think.

No Harm to Structure

You must not harm the structure of the code. You must keep the code clean and well organized.

Ask yourself why the programmers at Knight Capital did not KNOW that their code could be harmful.

I think the answer is pretty obvious. They had forgotten that the Power-Peg software was still in the system. They had forgotten that the flag they repurposed would activate it. And they assumed that all servers would always have the same software running.

The reason they did not know their system was harmful is because of the harm they had done to the structure of the system by leaving that dead code in place.

And that's one of the big reasons why code structure and cleanliness are so important. The more tangled the structure, the more difficulty in knowing what the code will do. The bigger the mess, the more the uncertainty.

Take the Toyota case, for example. Why didn't they know their software could kill people? Do you think the fact that they had thousands of global variables might have been a factor?

Making a mess in the software undermines your ability to know what the software does, and therefore your ability to prevent harm.

Messy software is harmful software.

Now, some of you might object by saying that sometimes a quick-and-dirty patch is necessary to fix a nasty production bug.

Sure. Of course. If you can fix a production crisis with a quick-and-dirty patch, then you should do it. No question.

A stupid idea that works is not a stupid idea.

However, you can't leave that quick-and-dirty patch in place without causing harm. The longer that patch remains in the code, the more harm it can do.

Remember that the Knight Capital debacle would not have happened if the old Power-Peg code had been removed from the codebase. It was that old, defunct code that actually made the bad trades.

So, what do we mean by “Harm to structure?”

Clearly, thousands of global variables is a structural flaw. So is dead code left in the codebase.

Structural harm is harm to the organization and content of the source code. It is anything that makes that source code hard to read, hard to understand, hard to change, or hard to reuse.

It is the responsibility of every professional software developer to know the disciplines and standards of good software structure. They should know how to refactor, how to write tests, how to recognize bad code, and how to decouple designs and create appropriate architectural boundaries. They should know and apply the principles of low- and high-level design. And it is the responsibility of every senior developer to make sure that younger developers learn these things, and satisfy them in the code that they write.

Soft

The first word is software is *SOFT*. Software is supposed to be SOFT. It’s supposed to be easy to change. If we didn’t want it to be easy to change, we’d have called it HARDware.

It’s important to remember why software exists at all. We invented it as a means to make the behavior of machines easy to change. To the extent that our software is hard to change, we have thwarted the very reason that software exists.

Remember that software has two values. There’s the value of its behavior; and then there’s the value of its “softness.” Our customers and users expect us to be able to change that behavior easily and without high cost.

Which of these two values is greater? Which value should we prioritize above the other? We can answer that question with a simple thought experiment.

Imagine two programs: One works perfectly but is impossible to change; the other does nothing correctly but is easy to change.

Which is more valuable?

I hate to be the one to tell you this, but in case you hadn’t noticed it, software requirements tend to change, and when they change, that first software will become useless—forever.

On the other hand, that second software can be made to work, because it’s easy to change. It may take some time and money to get it to work initially, but after that, it’ll continue to work forever with minimal effort.

And therefore, it is the second of the two values that should be given priority in all but the most urgent of situations.

And what do I mean by urgent? I mean a production disaster that’s losing the company $10 million per minute. That’s urgent.

I do not mean a software startup. A startup is not an urgent situation requiring you to create inflexible software. Indeed, the opposite is true. The one thing that is absolutely certain in a startup is that you are creating the wrong product.

No product survives contact with the user. As soon as you start to put the product into users’ hands, you will discover

that the product you've built is wrong in a hundred different ways. And if you can't change it without making a mess, you're doomed.

Indeed, this is one of the biggest problems with software startups. They believe they are in an urgent situation requiring them to throw out all the rules and dash to the finish line, leaving a huge mess in their wake.

Most of the time that huge mess starts to slow them down long before they do their first deployment. They'll go faster, and better, and survive a lot longer, if they keep the structure of the software from harm.

When it comes to software, it never pays to rush —Brian Marick

Tests

If you follow a comprehensive testing discipline, like TDD, test & commit || revert (TCR), or small bundles, you will KNOW, to a high degree of certainty, that the code works.

How can you prevent harm to the behavior of your code if you don't have the tests that demonstrate that it works?

How can you prevent harm to the structure of your code if you don't have the tests that allow you to clean it?

Is a testing discipline really a prerequisite to professionalism? Am I really suggesting that you can't be a professional software developer unless you practice such a discipline?

Yes, I think that's true. Or rather, it is becoming true. It is true for some of us; and with time it is becoming true for more and more of us. I think the time will come, and relatively soon, when the majority of programmers will agree that practicing a testing discipline is part of the

minimum set of disciplines and behaviors that mark a professional developer.

Why do I believe that?

Because, as I said earlier, we rule the world! We write the rules that make the whole world work.

In our society, nothing gets bought or sold without software. Nearly all correspondence is through software. Nearly all documents are written with software. Laws don’t get passed or enforced without it. There is virtually no activity of daily life that does not involve software.

Without software, our society does not function. Software has become the most important component in the infrastructure of our civilization.

Society does not understand this yet. We programmers don’t really understand it yet either. But the realization is dawning that the software we write is critical. The realization is dawning that too many lives and fortunes depend upon our software. And the realization is dawning that software is being written by people who do not profess a minimum level of discipline.

So, yes, I think TDD, or some discipline very much like it, will eventually be considered a minimum standard behavior for professional software developers. I think our customers and our users will insist upon it.

Chapter 29. No Defect in Behavior or Structure

2. The code that I produce will always be my best work. I will not knowingly allow code that is defective either in behavior or in structure to accumulate.

Remember, Kent Beck once said: “First, make it work. Then, make it right.” Getting the program to work is just the first and easiest step. The second and harder step is to clean the code.

Unfortunately, too many programmers think they are done once they get a program to work. Once it works, they move on to the next program, and then the next, and the next.

In their wake, they leave behind a history of tangled, unreadable code that slows down the whole development team.

They do this because they think their value is in speed. They know they are paid a lot; and so they feel that they must deliver a lot of functionality in a short amount of time.

But software is hard, and it takes a lot of time, and so they feel like they are going too slowly. They feel like they are failing. And this feeling of failure creates a pressure that causes them to try to go faster. It causes them to rush.

And so they rush to get the program working; and then they declare themselves to be done—because, in their minds, it’s

already taken them too long.

The constant tapping of the project manager's foot doesn't help, but that's not the real driver.

I teach lots of classes. In many of them, I give the programmers small projects to code. My goal is to give them a coding experience in which they can try out new techniques and new disciplines. I don't care if they actually finish the project.

Indeed, all the code is going to be thrown away.

And still I see people rushing. Some stay past the end of the class just hammering away at getting something utterly meaningless to work.

So, although the boss's pressure doesn't help, the real pressure comes from inside us. We consider speed of development to be a matter of our own self-worth.

Making It Right

As we saw in the last chapter, there are two values to software. There is the value of its behavior and the value of its structure. And in the last chapter, I also made the point that the value of structure is more important than the value of behavior. This is because, to have any long-term value, software systems must be able to react to changes in requirements.

Software that has a structure that makes it hard to change, makes it harder for that software to stay up-to-date with the requirements. Software with a bad structure can quickly get out-of-date.

So, in order to keep up with requirements, the structure of the software has to be clean enough to allow, and even

encourage, change. Software that is easy to change can keep up with changing requirements, allowing it to remain valuable with the least amount of effort. But if you've got a software system that's hard to change, then you're going to have a devil of a time keeping that system working when the requirements change.

When are requirements most likely to change? Requirements are most volatile at the start of a project; just after the users have seen the first few features work. That's because they're getting their first look at what the system actually does, as opposed to what they thought it was going to do.

This means that the structure of the system needs to be clean at the very start, if early development is to proceed quickly. If you make a mess at the start, then even the very first release will be slowed down by that mess.

Good structure enables good behavior and bad structure impedes good behavior. The better the structure, the better the behavior. The worse the structure, the worse the behavior. The value of the behavior depends critically on the structure. And therefore, the value of the structure is the more critical of the two values.

And that means that professional developers put a higher priority on the structure of the code than on the behavior.

Yes, first you make it work; but then, you make very sure that you continue to make it right. You keep the system structure as clean as possible throughout the lifetime of the project. From its very beginning to its very end, it must be clean.

What Is Good Structure?

Good structure makes the system easy to test, easy to change, and easy to reuse. Changes to one part of the code do not break other parts of the code. Changes to one module do not force massive recompiles and redeployments. High-level policies are kept separate and independent from low-level details.

Poor structure makes a system rigid, fragile, and immobile. These are the traditional design smells.

Rigidity is when relatively minor changes to the system cause large portions of the system to be recompiled, rebuilt, and redeployed. A system is rigid when the effort to integrate a change is much greater than the change itself.

Fragility is when minor changes to the behavior of a system force many corresponding changes in a large number of modules. This creates a high risk that a small change in behavior will silently break some other behaviors in the system that then get inadvertently deployed. When this happens, managers and customers will come to believe that the team has lost control over the software and don’t know what they are doing.

Immobility is when a module in an existing system contains a behavior you want to use in a new system but is so tangled up within the existing system that you can’t extract it for use in the new system.

These problems are all problems of structure and not behavior. The system may pass all its tests and meet all its functional requirements. And yet such a system may be close to worthless because it is too difficult to manipulate.

There’s a certain irony in the fact that so many systems that correctly implement valuable behaviors end up with a structure that is so poor that it negates that value and becomes worthless.

And *worthless* is not too strong a word. In Chapter 1, I told you the tale of the grand redesign in the sky. This is when developers tell management that the only way to make progress is to redesign the whole system from scratch; those developers have assessed the current system to be worthless.

When managers agree to let the developers redesign the system, it simply means that they have agreed with the developers' assessment that the current system is worthless.

What is it that causes these design smells that lead to worthless systems? Source code and data dependencies! And how do we fix those dependencies? Dependency management!

And how do we manage dependencies? We use principles of design, like the SOLID[1] principles, to keep the structure of our systems free of the design smells that would make it worthless.

[1] These principles are described in [Clean Arch] and [PPP02].

Since the value of structure is greater than the value of behavior, and since the value of structure depends upon good dependency management, and since good dependency management derives from good principles, it follows that the overall value of the system depends upon proper application of those principles.

Eisenhower's Matrix

General Dwight D. Eisenhower once said: "I have two kinds of problems, the urgent and the important. The urgent are not important, and the important are never urgent."

There is a deep truth to this statement. A deep truth about engineering. We might even call this "the engineer's motto":

The greater the urgency, the less the relevance.

Here is Eisenhower's decision matrix. Urgency is on the vertical axis, and importance is on the horizontal axis. The four possibilities are urgent and important, urgent and unimportant, important but not urgent, and neither important nor urgent.



Urgency is about time. Importance is not. Things that are important are long term. Things that are urgent are short term. Structure is long term; therefore, it is important. Behavior is short term; therefore, it is merely urgent.

So structure, the important stuff, comes first. Behavior is secondary.

Your boss might not agree with that priority; but that's because it's not your boss's job to worry about structure. It's yours. Your boss simply expects that you will keep the structure clean while you implement the urgent behaviors.

This seems to violate Beck's law: "First, make it work. Then, make it right." Now I'm saying that structure is higher priority than behavior. It's a chicken-and-egg dilemma, isn't it?

The reason we make it work first is because structure has to support behavior; so we implement behavior first, then we give it the right structure. But structure is more important than behavior. We give it a higher priority. We deal with structural issues before we deal with behavioral issues.

We can square this circle by breaking the problem down into tiny units. Let’s start with user stories. You get a story to work; and then you get its structure right. And you don’t work on the next story until that structure is right. The structure of the current story is higher priority than the behavior of the next story.

Except stories are too big. We need to go smaller. Not stories . . . tests. Tests are the perfect size.

First you get a test to pass, then you fix the structure of the code that passes that test, before you get the next test to pass.

This entire discussion has been the moral foundation for the Red-Green-Refactor cycle of a good testing discipline like test-driven development (TDD).

It is that cycle that helps us prevent harm to behavior and harm to structure. It is that cycle that allows us to prioritize structure over behavior. And that’s why we consider our testing discipline to be a design technique as opposed to a testing technique.

Programmers Are Stakeholders

Remember this. We have a stake in the success of the software. We, programmers, are stakeholders too.

Did you ever think about it that way? Did you ever view yourself as one of the stakeholders of the project?

But, of course, you are. The success of the project has a direct impact upon your career and reputation. So, yes, you are a stakeholder.

And as a stakeholder, you have a say in the way the system is developed and structured. I mean, it’s your butt on the line too.

But you are more than just a stakeholder. You are an engineer. You were hired because you know how to build software systems and how to structure those systems so that they last.

And with that knowledge comes the responsibility to produce the best product you can.

So, not only do you have the right as a stakeholder, but you have the duty as an engineer to make sure that the systems you produce do no harm either through bad behavior or through bad structure.

A lot of programmers don’t want that kind of responsibility. They’d rather just be told what to do. And that’s a travesty and a shame. It’s wholly unprofessional. Programmers who feel that way should be paid minimum wage, because that’s what their work output is worth.

If you don’t take responsibility for the structure of the system, who else will? Your boss?

Does your boss know the SOLID principles? How about design patterns? How about object-oriented design, and the practice of dependency inversion? Does your boss know the discipline of TDD? Do they know what a self-shunt is, or a test-specific subclass, or a humble object? Does your boss understand that things that change together should be grouped together and that things that change for different reasons should be separated?

Does your boss understand structure? Or is your boss's understanding limited to behavior?

Structure matters. If you aren't going to care for it, who will? If not you, then who?

What if your boss specifically tells you to ignore structure and focus entirely on behavior? You refuse. You are a stakeholder. You have rights too. And you are an engineer; you have responsibilities that your boss cannot override.

Perhaps you think that will get you fired. Probably not. Most managers expect to have to fight for things they need and believe in. And they respect those who are willing to do the same.

Oh, there will be a struggle; even a confrontation. And it won't be comfortable. But you are a stakeholder and an engineer. You can't just back down and acquiesce. That's not professional.

Most programmers do not enjoy confrontation. But dealing with confrontational managers is a skill we have to learn. We have to learn how to fight for what we know is right. Because taking responsibility for the things that matter, and fighting for those things, is how a professional behaves.

Do Your Best

This promise of The Oath is about doing your best.

Clearly, this is a perfectly reasonable promise for a programmer to make. Of course you are going to do your best, and of course you will not knowingly release code that is harmful.

And of course this is not black and white. There are times when structure must bend to schedule. For example, if, in

order to make a trade show, you have to put in some quick-and-dirty fix, then so be it.

The promise doesn't even prevent you from shipping code to customers with less-than-perfect structure. If the structure is close but not quite right, and customers are expecting the release tomorrow, then so be it.

On the other hand, the promise does mean that you will address those issues of behavior and structure before you add more behavior. You will not pile more and more behavior on top of known bad structure. You will not allow those defects to accumulate.

What if your boss tells you to anyway? Here's how that conversation should go.

Boss: I want this new feature added by tonight.

Programmer: I'm sorry, but I can't do that; I've got some structural cleanup to do before I can add that new feature.

Boss: Do the cleanup tomorrow. Get the feature done by tonight.

Programmer: That's what I did for the last feature, and now I have an even bigger mess to clean up. I really have to finish that cleanup before I start on anything new.

Boss: I don't think you understand. This is business. Either we have a business or we don't have a business. And if we can't get features done, we don't have a business. Now get the feature done.

Programmer: I understand. Really, I do. And I agree. We have to be able to get features done. But if I don't clean up the structural problems that have accumulated over the last few days, we're going to slow down and get even fewer features done.

Boss: Y’know, I used to like you. I used to say, that Danny, he’s pretty nice. But now I don’t think so. You’re not being nice at all. Maybe you shouldn’t be working with me. Maybe I should fire you.

Programmer: Well, sir, that’s your right. But I’m pretty sure you want features done quickly and done right. And I’m telling you that if I don’t do this cleanup tonight, then we’re going to start slowing down. And we’ll deliver fewer and fewer features.

Look, sir, I want to go fast, just like you do. You hired me because I know how to do that. You have to let me do my job. You have to let me do what I know is best.

Boss: You really think everything will slow down if you don’t do this cleanup tonight?

Programmer: I know it will. I’ve seen it before. And so have you.

Boss: And it has to be tonight?

Programmer: I don’t feel safe letting the mess get any worse.

Boss: You can give me the feature tomorrow?

Programmer: Yes, and it’ll be a lot easier to do once the structure is cleaned up.

Boss: OK. Tomorrow. No later. Now get to it.

Programmer: OK. I’ll get right on it.

Boss: (aside) I like that kid. He’s got guts. He’s got gumption. He didn’t back down even when I threatened to fire him. He’s gonna go far, trust me. But don’t tell him I said so.

Chapter 30. Repeatable Proof

3. I will produce, with each release, a quick, sure, and repeatable demonstration that every element of the code works as it should.

Does that sound unreasonable to you? Does it sound unreasonable to be expected to prove that the code you've written actually works? Allow me to introduce you to Edsger Wybe Dijkstra.

Dijkstra

Edsger Wybe Dijkstra was born in Rotterdam in 1930. He survived the bombing of Rotterdam and the German occupation of the Netherlands, and in 1948, he graduated high school with the highest possible marks in math, physics, chemistry, and biology.

In March of 1952, at the age of 21, and just nine months before I would be born, he took a job with the Mathematical Center of Amsterdam as the Netherlands' very first programmer.

In 1957, he married Maria Debets. In the Netherlands at the time, you had to state your profession as part of the marriage rites. The authorities were unwilling to accept "programmer" as his profession. They'd never heard of such a profession. So he settled for "theoretical physicist."

In 1955, having been a programmer for three years, and while still a student, he concluded that the intellectual challenge of programming was greater than the intellectual challenge of theoretical physics, and as a result chose programming as his long-term career.

In making this decision, he conferred with his boss, Adriaan van Wijngaarden. Dijkstra was concerned that no one had identified a discipline or science of programming, and that he would therefore not be taken seriously. His boss replied that Dijkstra might very well be one of the people who would make it a science.

In pursuit of that goal, Dijkstra was compelled by the idea that software was a formal system, a kind of mathematics. He reasoned that software could become a mathematical structure rather like Euclid's Elements—a system of postulates, proofs, theorems, and lemmas. And so he set about to create the language and discipline of software proofs.

Proving Correctness

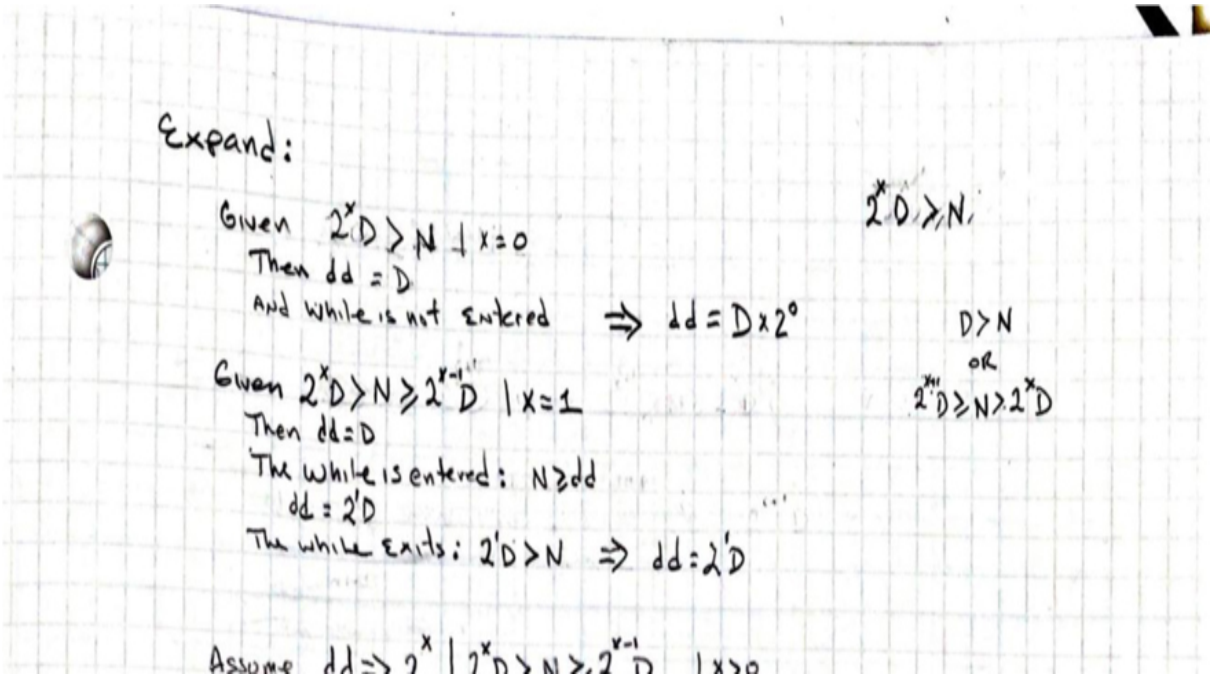
Dijkstra realized that there were only three techniques we could use to prove the correctness of an algorithm: enumeration, induction, and abstraction. Enumeration is used to prove that two statements in sequence, or two statements selected by a boolean expression, are correct. Induction is used to prove that a loop is correct. Abstraction is used to break groups of statements into smaller provable chunks.

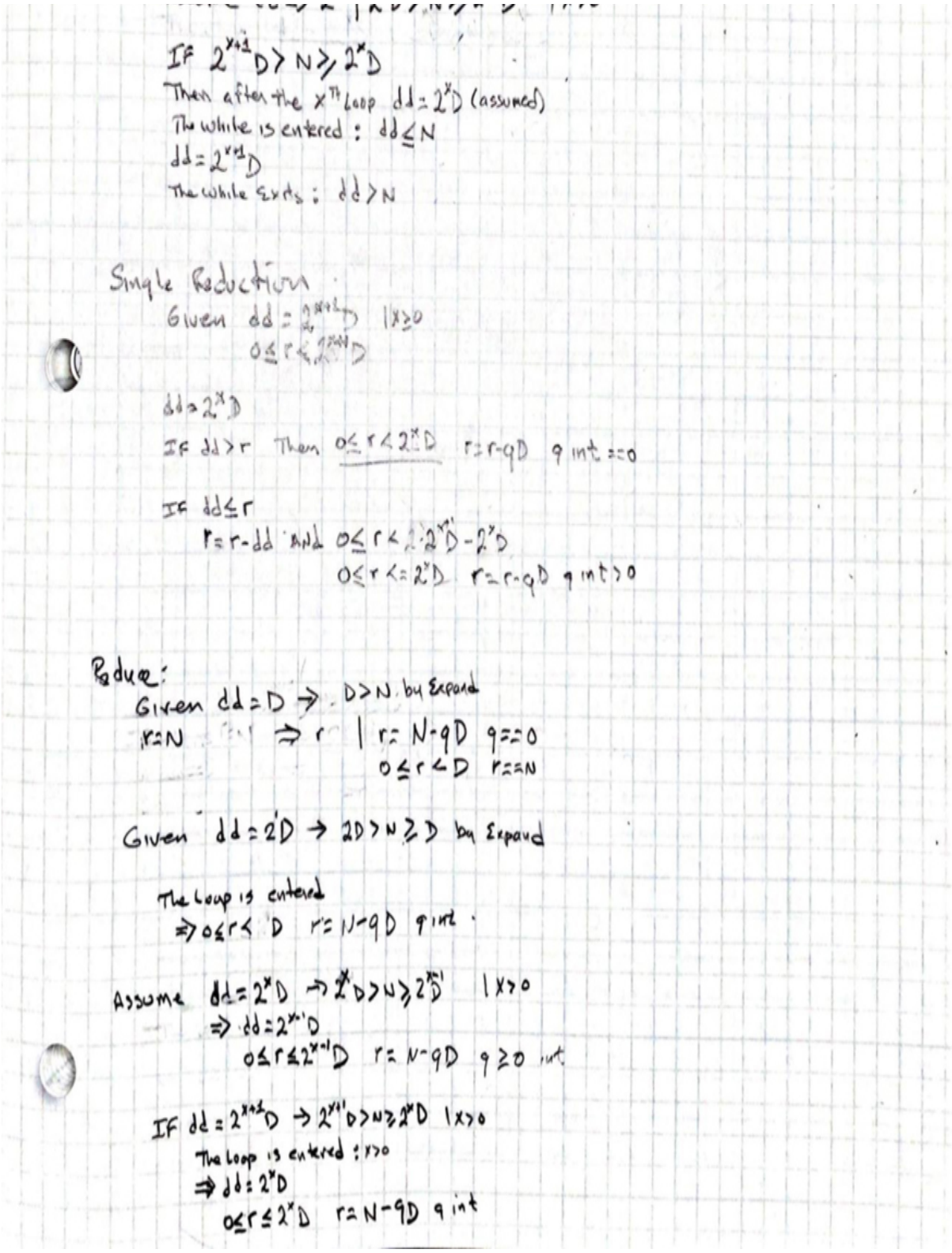
If this sound hard, it is. As an example of just how hard this is, I have included a simple Java program for calculating the remainder of an integer, and my handwritten proof of that algorithm below it.[1]

[1] This is a translation into Java of a demonstration from Dijkstra's work.

```
public static int remainder(int numerator, int denominator) {
  assert(numerator >= 0 && denominator > 0);
  int r = numerator;
  int dd = denominator;
  while (dd<=r)
    dd *= 2;
  while (dd != denominator) {
    dd /= 2;
    if (dd <= r)
      r -= dd;
  }
  return r;
}
```

Don't worry. I don't expect you to read my proof. I just want you to get a feel for what such a proof entails.





I think you can see the problem with this approach. Indeed, this is something that Dijkstra complained bitterly about:

“Of course I would not dare to suggest (at least at present!) that it is the programmer’s duty to supply such a proof whenever he writes a simple loop in his program. If so, he could never write a program of any size at all.”[2]

[2] [SP72].

Dijkstra’s hope was that such proofs would become more practical through the creation of a library of theorems, again similar to Euclid’s Elements.

But Dijkstra did not understand just how prevalent and pervasive software would become. He did not foresee in those early days that computers would outnumber people and that vast quantities of software would be running in the walls of our homes, in our pockets, and on our wrists. Had he known, he would have realized that the library of theorems he envisioned would be far too vast for any mere human to grasp.

And so Dijkstra’s dream of explicit mathematical proofs for programs has faded into near oblivion. Oh, there are some holdouts who hope against hope for a resurgence of formal proofs, but their vision has not penetrated very far into the software industry at large.

But while the dream may have passed, it drew something deeply profound in its wake. Something that we use today, almost without thinking about it.

Structured Programming

In the early days of programming, the 1950s and 1960s, we used languages like Fortran. Have you ever seen Fortran? Here, let me show you what it was like.

```
        WRITE(4,99)
99      FORMAT(" NUMERATOR:")
```

```
        READ(4,100)NN
        WRITE(4,98)
98      FORMAT(" DENOMINATOR:")
        READ(4,100)ND
100     FORMAT(I6)
        NR=NN
        NDD=ND
1       IF(NDD-NR)2,2,3
2       NDD=NDD*2
        GOTO 1

3       IF(NDD-ND)4,10,4
4       NDD=NDD/2
        IF(NDD-NR)5,5,6
5       NR=NR-NDD
6       GOTO 3

10      WRITE(4,20)NR
20      FORMAT(" REMAINDER:",I6)
        END
```

This little Fortran program implements the same remainder algorithm as the earlier Java program.

Now, I'd like to draw your attention to those `GOTO` statements. You probably haven't seen statements like that very often.

The reason you haven't seen statements like that very often is that nowadays we look upon them with disfavor. In fact, most modern languages don't even have `GOTO` statements like that anymore.

Why don’t we favor GOTO statements? Why don’t our languages support them anymore? Because in 1968, Edsger Dijkstra wrote an article for the *Communications of the ACM*. The editor, Niklaus Wirth, thought it was so profound that he bypassed the normal review process and published it as a letter to the editor in the March issue. He gave it the title “*Go To Statement Considered Harmful*.”

Why did Dijkstra consider the GOTO statement to be harmful? It all comes back to the three strategies for proving a function correct: enumeration, induction, and abstraction.

Enumeration depends upon the fact that each statement in sequence can be analyzed independently, and that the result of one statement feeds into the next.

It should be clear to you that in order for enumeration to be an effective technique for proving the correctness of a function, every statement that is enumerated must have a single entry point and a single exit point. Otherwise, we could not be sure of either the inputs or the outputs of a statement.

What’s more, induction is simply a special form of enumeration, where we assume the enumerated statement is true for some x, and then prove by enumeration that it is true for $x + 1$.

Thus, the body of a loop must be enumerable. It must have a single entry and a single exit.

GOTO is considered harmful because a GOTO statement can jump into or out of the middle of an enumerated sequence. GOTOs make enumeration intractable, making it impractical to prove an algorithm correct by enumeration or induction. So, in order to keep code provable, Dijkstra recommended that it be constructed out of three standard building blocks.

- **Sequence:** Depicted as two or more statements ordered in time. This represents nonbranching lines of code.
- **Selection:** Depicted as two or more statements selected by a predicate. This represents `if/else` and `switch/case` statements.
- **Iteration:** Depicted as a statement repeated under the control of a predicate. This represents a `while` or `for` loop.

Dijkstra knew[3] that any program, no matter how complicated, can be composed of nothing more than these three structures, and that programs structured in that manner are provable.

[3] Because of the work of Böhm and Jacopini. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM* 9, no. 5 (May 1966): 366–371.

He called the technique *structured programming*.

Why is this important if we aren't going to write those proofs? If something is provable, it means you can reason about it. If something is unprovable, it means you cannot reason about it.

And if you can't reason about it, you can't properly test it.

Functional Decomposition

In 1968, Dijkstra's ideas were not immediately popular. Most of us were using languages that depended upon `GOTO`. So the idea of abandoning `GOTO` or imposing discipline on `GOTO` was abhorrent.

The debate over Dijkstra's ideas raged for several years. We didn't have an internet in those days, so we didn't use

Facebook memes or flame wars. But we did write letters to the editors of the major software journals of the day.

And those letters raged. Some claimed Dijkstra to be a god. Others claimed him to be a fool. Just like social media today; except slower.

But in time, the debate slowed, and Dijkstra's position gained more and more support. Until nowadays, most of the languages we use simply don't have a `GOTO`.

Nowadays, we are all structured programmers, because our languages don't give us a choice.

We all build our programs out of sequence, selection, and iteration. And very few of us make regular use of unconstrained `GOTO` statements.

An unintended side effect of composing programs from those three structures was a technique called *functional decomposition*.

Functional decomposition is the process whereby you start at the top level of your program and recursively break it down into smaller and smaller provable units. Indeed, this is the reasoning process behind structured programming. Structured programmers reason from the top down through this recursive decomposition into smaller and smaller provable functions.

This connection between structured programming and functional decomposition was the basis for the structured revolution that took place in the '70s and '80s.

People like Ed Yourdon, Larry Constantine, Tom DeMarco, and Meilir Page-Jones popularized the techniques of structured analysis and structured design during those decades.

Test-Driven Development et al.

Test-driven development (TDD), the red → green → refactor cycle, is functional decomposition. After all, you have to write tests against small bits of the problem. That means that you must functionally decompose the problem into testable elements.

That means that every system built with TDD is built from functionally decomposed elements that conform to structured programming. And that means that the system they compose is provable.

And the tests are the proof. Or, rather, the tests are the *theory*.

The tests created by TDD are not a formal mathematical proof, like Dijkstra wanted. In fact, Dijkstra is famous for saying that tests can only prove a program wrong; they can never prove a program right.

This is where Dijkstra missed it, in my opinion. Dijkstra thought of software as a kind of mathematics. He wanted us to build up a superstructure of postulates, theorems, corollaries, and lemmas.

Instead, what we have realized is that software is a kind of science. We validate that science with experiments. We build up a superstructure of theories based upon passing tests. Just like all other sciences do.

Have we proven the theory of evolution, or the theory of relativity, or the Big Bang theory, or any of the major theories of science? No. We can’t prove them in any mathematical sense.

But we believe them, within limits, nonetheless. Indeed, every time you get into a car or an airplane, you are betting

your life that Newton's laws of motion are correct. Every time you use a GPS you are betting that Einstein's theory of relativity is correct.

The fact that we have not mathematically proven these theories correct does not mean that we don't have sufficient proof to depend upon them, even with our lives.

That's the kind of proof that TDD gives us. Not formal mathematical proof; but experimental empirical proof. The kind of proof we depend upon every day.

And that brings us back to the third promise in the Oath:

I will produce, with each release, a quick, sure, and repeatable demonstration that every element of the code works as it should.

Quick, sure, and repeatable. *Quick* means that the test suite should run in a very short amount of time. Minutes instead of hours.[4]

[4] James T. Kirk, *Star Trek II: The Wrath of Khan.*

Sure means that when the test suite passes, you know you can ship.

Repeatable means that those tests can be run by anybody at any time to ensure that the system is working properly. Indeed, we want the tests run many times per day.

Some may think that it is too much to ask that programmers supply this level of proof. Some may think that programmers should not be held to this high a standard. I, on the other hand, can imagine no other standard that makes any sense.

When a customer pays us to develop software for them, aren't we honor bound to prove, to the best of our ability, that the software we've created does what that customer has paid us for?

Of course we are.

We owe this promise to our customers, our employers, and our teammates. We owe it to our business analysts, our testers, and our project managers.

But mostly we owe this promise to ourselves. For how can we consider ourselves professionals if we cannot prove that the work we have done is the work we have been paid to do?

What you owe, when you make that promise, is not the formal mathematical proof that Dijkstra dreamed of; rather, it is the scientific suite of tests that covers all the required behavior, runs in seconds or minutes, and produces the same clear pass/fail result every time it is run.

Chapter 31. Small Cycles

4. I will make frequent, small releases so that I do not impede the progress of others.

Making small releases just means changing a small amount of code for each release. The system may be large, but the change to that system is small.

The History of Source Code Control

The following is a look back at the history of source code control.

Punch Cards

Let’s go back to the 1960s for a moment. What is your source code control system, when your source code is punched on a deck of cards?



The source code is not stored on disk. It’s not “in the computer.” Your source code is, literally, in your hand.

What is the source code control system? It is your desk drawer.

When you literally possess the source code, there is no need to “control” it. Nobody else can touch it.

And this was the situation throughout much of the ’50s and ’60s. Nobody even dreamed of something like a source code control system. You simply kept the source code under control by putting it in a drawer or a cabinet.

If anyone wanted to “check out” the source code, they simply went to the cabinet and took it. When they were done, they put it back.

We certainly didn’t have merge problems. It was physically impossible for two programmers to be making changes to the same modules at the same time.

But things started to change in the ’70s. The idea of storing your source code on magnetic tape, or even on disk, was becoming attractive.

We wrote line editing programs that allowed us to add, replace, and delete lines in source files on tape.

These weren’t screen editors. We punched our add, change, and delete directives on cards. The editor would read the source tape, apply the changes from the edit deck, and write the new source tape.

You may think this sounds awful. Looking back on it, it was. But it was better than trying to manage programs on cards! I mean, 6,000 lines of code on cards weighs 30 pounds. If you dropped it…

If you drop a tape, ..you can just pick it up again.

Anyway, notice what happened. We start with one source tape, and in the editing process, we wind up with a second, new source tape. But the old tape still exists. If you put that old tape back on the rack, someone else might inadvertently

apply their own changes to it, leaving you with a merge problem.

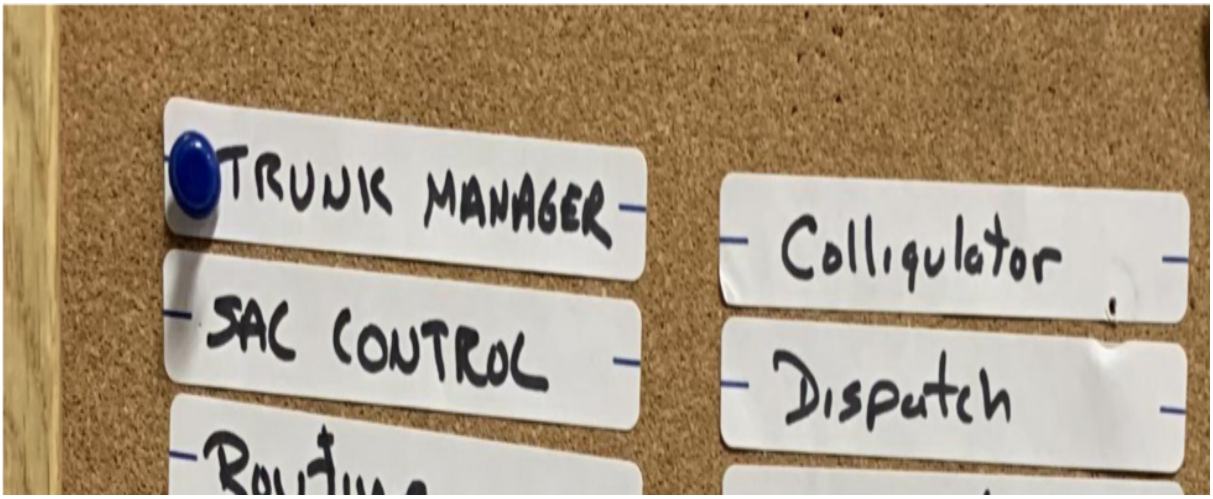
To prevent that, we simply kept the master source tape in our possession until we were done with our edits and our tests. Then, we put a new master source tape back on the rack. We controlled the source code by keeping possession of the tape.

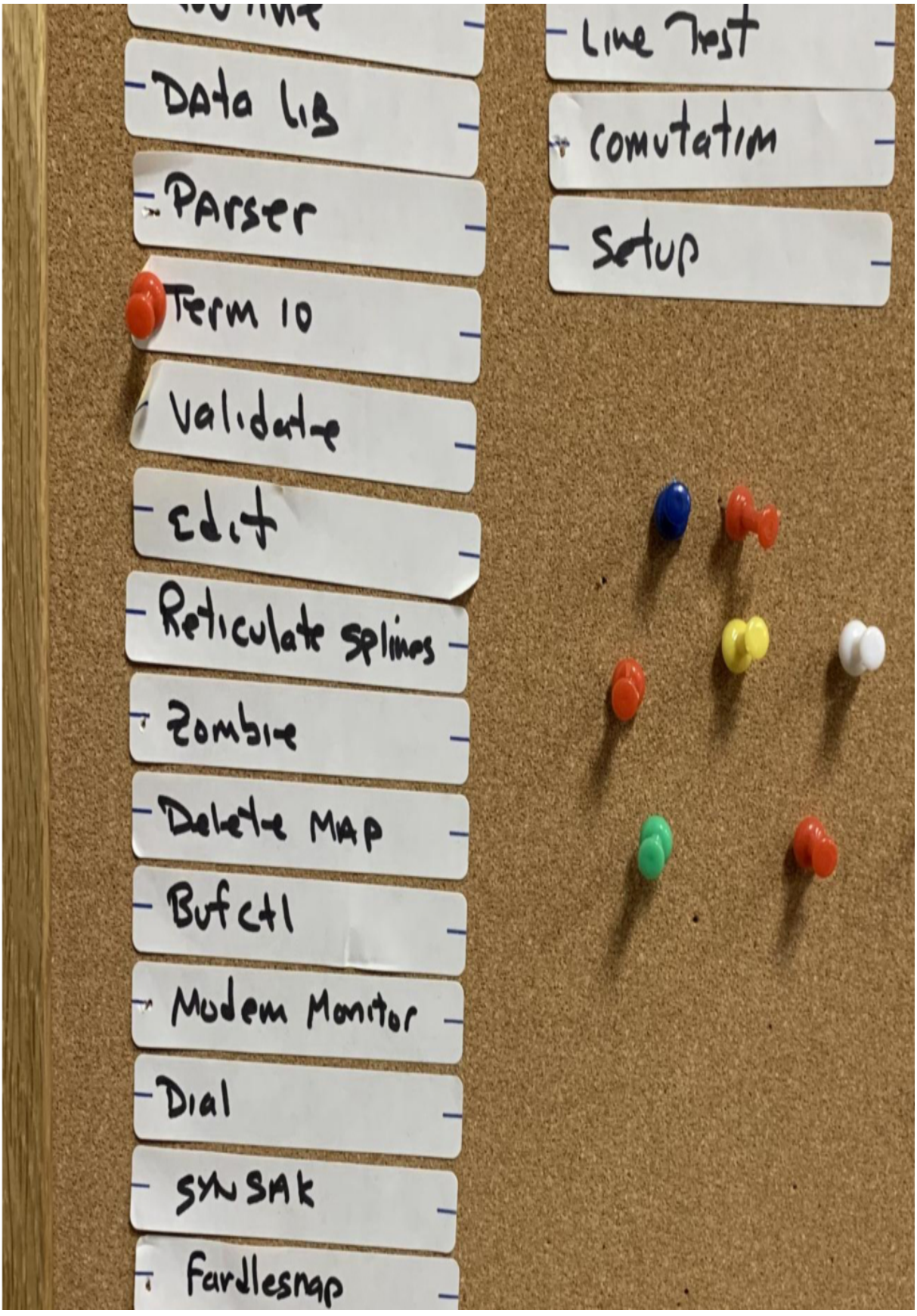
But now this had to be done by process and convention. A true source code control process had to be used. No software yet, just human rules. But still, the concept of source code control had become separate from the source code itself.

As systems got bigger and bigger, they needed more and more programmers working on the same code at the same time. Grabbing the master tape and holding it became a real nuisance for everyone else. I mean, you could keep the master tape out of circulation for a couple of days or more.

So then, we decided to extract modules from the master tape. The whole idea of modular programming was pretty new at the time. The notion that a program could be made up of many different source files was revolutionary.

So we started to use bulletin boards, like this:




Line Test
comutation
Setup
Data lib
Parser
Term IO
Validate
edit
Reticulate splines
Zombie
Delete MAP
Buf ctl
Modem Monitor
Dial
SYN SAK
Fardlesnap



The bulletin board had labels on it for each of the modules in the system. Each programmer had their own color thumbtack. I was blue. Ken was red. My buddy CK was yellow, and so on.

If I wanted to edit the “Trunk Manager” module, I’d look on the bulletin board to see if there was a pin in that module. If not, I put a blue pin in it.

Then, I took the master tape from the rack and copied it onto a separate tape.

I would edit the Trunk Manager module, and only the Trunk Manager module, generating a new tape with my changes.

I’d compile, test, wash, and repeat until I got my changes to work.

Then, I’d go get the master tape from the rack, and I would create a new master by copying the current master, but replacing the Trunk Manager module with my changes.

I’d put the new master back in the rack.

Finally, I’d remove my blue pin from the bulletin board.

This worked; but it only worked because we all knew each other; we all worked in the same office together; we all knew what each other was doing. And we *talked* to each other all the time.

I’d holler across the lab: “Ken, I’m going to change the Trunk Manager module.” He’d say, “Put a pin in it.” I’d say, “I already did.”

The pins were just reminders. We all knew the status of the code and who was working on what. And that's why the system worked.

Indeed, it worked very well. Knowing what each other was doing meant that we could help each other. We could make suggestions. We could warn each other about problems we'd recently encountered. And we could avoid merges.

Back then, merges were *not* fun.

Then, in the '80s, came the disks. Disks got big, and they got permanent.

By big, I mean hundreds of megabytes. By permanent, I mean that they were permanently mounted—always online.

The other thing that happened is that we got machines like PDP11s and VAXes. We got screen editors, real operating systems, and multiple terminals. More than one person could be editing at exactly the same time.

The era of thumbtacks and bulletin boards had to come to an end.

First of all, by then, we had 20 or 30 programmers. There weren't enough thumbtack colors. Second of all, we had hundreds and hundreds of modules. We didn't have enough space on the bulletin board.

Fortunately, there was a solution.

In 1972, Marc Rochkind wrote the first source code control program. It was called SCCS, and he wrote it in SNOBOL.[1]

[1] A lovely little string processing language from the '60s that had many of the pattern matching facilities of our modern languages.

Later he rewrote it in C, and it became part of the Unix distribution on PDP11s. SCCS only worked on one file at a

time, but it allowed you to lock that file so that someone else couldn’t edit it until you were done. It was a lifesaver.

In 1982 RCS, the Revision Control System, was created by Walter Tichy. It too was file based and knew nothing of projects, but it was considered an improvement on SCCS and rapidly became the standard source code control system of the day.

Then, in 1986, CVS, the Concurrent Versions System, came along. It extended RCS to deal with whole projects, not just individual files. It also introduced the concept of optimistic locking.

Up to this time, most source code control systems worked like my thumbtacks. If you had checked out the module, nobody else could edit it. This is called *pessimistic locking*.

CVS used *optimistic locking*. Two programmers could check out and change the same file at the same time. CVS would try to merge any nonconflicting changes, and would alert you if it couldn’t figure out how to do the merge.

After that, source code control systems exploded and even became commercial products. There were literally hundreds of them. Some used optimistic locking, others used pessimistic locking. Indeed, the locking strategy became a kind of religious divide in the industry.

Then, in 2000, Subversion was created. It vastly improved upon CVS and was instrumental in driving the industry away from pessimistic locking once and for all.

Subversion was also the first source code control system to be used in the “cloud.” Does anybody remember Source Forge?

Up to this point, all source code control systems were based on the master tape idea that I used back in my bulletin

board days. The source code was maintained in a single central master repository. Source code was checked out from that master repository, and commits were made back into that master repository.

But all that was about to change.

Continuous Integration

The year is 2005. Multigigabyte disks are in our laptops. Network speeds are fast, and getting faster. Processor clock rates have plateaued at 2.6GHz.

We are very, very far away from my old bulletin board control system for source code. But we are still using the master tape concept. We still have a central repository that everybody has to check in and out of. Every commit; every reversion; every merge requires network connectivity to the master.

...and then came git.

Well, actually, git was presaged by BitKeeper and Monotone, but it was git that caught the attention of the programming world and changed everything.

Because, you see, git eliminates the master tape.

Oh, you still need a final authoritative version of the source code. But git does not automatically provide this location for you. Git simply doesn’t care about that. Where you decide to put your authoritative version is entirely up to you. Git has nothing whatsoever to do with it.

Git keeps the entire history of the source code on your local machine. Your laptop, if that’s what you use. On your machine, you can commit changes, create branches, check out old versions. And generally do anything you could do

with a centralized system like Subversion; except you don’t need to be connected to some central server.

Anytime you like, you can connect to another user and push any of the changes you’ve made to that user. Or you can pull any changes they have made into your local repository. Neither is the master. Both are equal. That’s why they call it *peer to peer*.

And the final authoritative location that you use for making production releases is just another peer that people can push to or pull from, anytime they like.

The end result of this is that you are free to make as many small commits as you like before you push your changes somewhere else. You can commit every 30 seconds if you like. You could make a commit every time you get a unit test to pass.

And that brings us to the point of this whole historical discussion.

If you stand back and look at the trajectory of the evolution of source code control systems, you can see that they have been driven, perhaps unconsciously, by a single underlying imperative.

Short Cycles

Consider, again, how we began. How long was the cycle when source code was controlled by physically possessing a deck of cards?

You checked the source code out by taking those cards out of the cabinet. You held on to those cards until you were done with your project. Then, you committed your changes by putting the changed deck of cards back in the cabinet.

The cycle time was the whole project.

Then, when we used thumbtacks on the bulletin board, the same rule applied. You kept your thumbtacks in the modules you were changing until you were done with the project you were working on.

Even in the late ’70s and into the ’80s, when we were using SCCS and RCS, we continued to use this pessimistic locking strategy, keeping others from touching the modules we were changing until we were done.

But CVS changed things—at least for some of us. Optimistic locking meant that one programmer could not lock others out of a module. We still only committed when we were done with a project; but others could be working concurrently on the same modules. So the average time between commits on the project shrank drastically. The cost of this, of course, was merges.

And how we hated doing merges.

Merges are awful. Especially without unit tests! They are tedious, time-consuming, and dangerous.

Indeed, our distaste for merges drove us to a new strategy.

Continuous Integration

By 2000, even while we were using tools like Subversion, we had begun teaching the discipline of committing every few minutes.

The rationale was simple. The more frequently you commit, the less likely you are to face a merge. And if you do have to merge, the merge will be trivial.

We called this discipline *continuous integration*.

Of course, this depends critically upon having a very reliable suite of unit tests. I mean, without good unit tests, it would be easy to make a merge error and break someone else's code. So continuous integration goes hand in hand with a good testing discipline.

And now, with tools like git, there is almost no limit to how small we can shrink the cycle.

And that begs the question: Why are we so concerned about shrinking the cycle?

Because long cycles impede the progress of the team.

The longer the time between commits, the greater the chance that someone else on the team—perhaps the whole team—will have to wait for you. And that violates the promise.

Perhaps you think this is only about production releases. No, it's actually about every other cycle. It's about iterations and sprints. It's about the edit/compile/test cycle. It's about the time between commits. It's about everything.

And remember the rationale: "So that you do not impede the progress of others."

Branches versus Toggles

I used to be a branch dogmatist. Back when I was using CVS and Subversion, I refused to allow members on my teams to branch the code. I wanted all changes returned to the main line as frequently as possible.

My rationale was simple. A branch is simply a long-term checkout. And as we've seen, long-term checkouts impede the progress of others by prolonging the time between integrations.

But then I switched to git—and everything changed overnight.

At the time, I was managing the open source FitNesse project, which had a dozen or so people working on it. I had just moved the FitNesse repository from Subversion (Source Forge) to git (GitHub). Suddenly, branches started appearing all over the place.

For the first few days, these crazy branches in git had me confused. Should I abandon my branch dogmatism stance? Should I abandon continuous integration and simply allow everyone to make branches willy-nilly, forgetting about the cycle time issue?

But then, it occurred to me that these “branches” I was seeing were not true named branches. Instead, they were just the stream of commits made by a developer between pushes.

In fact, all git had really done was to record the actions of the developer between continuous integration cycles.

So I resolved to continue my rule restricting branches. It’s just that now it’s not commits that return immediately to the main line. It’s pushes. Continuous integration was preserved.

If we are following continuous integration and pushing to the main line every hour or so, then we’ll certainly have a bunch of half-written features on the main line. There are typically two strategies for dealing with that: branches and toggles.

The branching strategy is simple. You create a new branch of the source code in which to develop the feature. You merge it back when the feature is done.

If you keep the branch out of the main line for days or weeks, then you are likely facing a big merge; and you'll certainly be impeding the team.

However, there are cases where the new feature is so isolated from the rest of the code that branching it is not likely to cause a big merge. In those cases, it might be better to let the developers work in peace on the new feature without continuously integrating.

In fact, we had a case like this with FitNesse a few years ago. We completely rewrote the parser. This was a big project. It took a few person-weeks. And there was no way to do it incrementally. I mean, the parser is the parser.

So we created a branch and kept that branch isolated from the rest of the system until the parser was ready.

There was a merge to do at the end, but it wasn't too bad. The parser was isolated well enough from the rest of the system. And fortunately, we had a very comprehensive set of unit and acceptance tests.

Most of the time, however, I think it's better to keep new feature development on the main line, and use toggles to turn those features off until ready.

Sometimes we use flags for those toggles, but more often we use the Command pattern, the Decorator pattern, and special versions of the Factory pattern to make sure the partially written features cannot be executed in a production environment.

And most of the time, we simply don't give the user the option to use the new feature. I mean, if the button isn't on the Web page, you can't execute that feature...

In many cases, of course, new features will be completed as part of the current iteration—or at least before the next

production release. So there’s no real need for any kind of toggle.

You only need a toggle if you are going to be releasing to production while some of the features are unfinished. So how often should that be?

Continuous Deployment

What if we could eliminate the delays between production releases? What if we could release to production several times per day? After all, delaying production releases impedes others.

I want you to be able to release your code to production several times per day. I want you to feel comfortable enough with your work that you could release your code to production on every push.

This, of course, depends upon automated testing. Automated tests are written by programmers to cover every line of code. Automated tests are written by business analysts and Q/A testers to cover every desired behavior.

Remember the preceding chapter? Those tests are the scientific proof that everything works as it is supposed to. And if everything works as it is supposed to, the next step is to deploy to production.

And by the way, that’s how you know if your tests are good enough. Your tests are good enough if, when they pass, you feel comfortable deploying. If passing tests don’t allow you to deploy, your tests are deficient.

Perhaps you think that deploying every day, or even several times per day, would lead to chaos. However, the fact that *you* are ready to deploy does not mean that the business is

ready to deploy. As a development team, your standard is to always be ready.

What’s more, we want to help the business remove all the impediments to deployment so that the business deployment cycle can be shortened as much as possible.

After all, the more ceremony and ritual the business uses for deployment, the more expensive deployment becomes. Any business would like to shed that expense.

The ultimate goal for any business is continuous, safe, and ceremony-free deployment. Deployment should be as close to a nonevent as possible.

And since deployment is often a lot of work, with servers to configure and databases to load, you need to automate the deployment procedure.

And since deployment scripts are part of the system, you write tests for them.

For many of you, the idea of continuous deployment may be so far away from your current process that it’s inconceivable. But that doesn’t mean there aren’t things you can do to shorten the cycle.

And who knows? If you keep on shortening the cycle, month after month, year after year, perhaps one day you’ll find that you are deploying continuously.

Continuous Build

Clearly, if you are going to deploy in short cycles, you have to be able to build in short cycles. If you are going to deploy continuously, you have to be able to build continuously.

Perhaps some of you have slow build times. If you do, speed them up.

Seriously, with the memory and speed of modern systems, there is no excuse for slow builds. None. Speed them up. Consider it a design challenge.

And then, get yourself a continuous build tool and use it. Make sure that you kick off a build at every push. And do what it takes to ensure that the build never fails.

A build failure is a red alert. It’s an emergency. If the build fails, I want emails and text messages sent to every team member. I want sirens going off. I want a flashing red light on the CEO’s desk. I want everybody to stop whatever they are doing and deal with the emergency.

Keeping the build from failing is not rocket science. You simply run the build in your local environment, along with all the tests, before you push. You only push the code when all the tests pass.

If the build fails after that, you’ve uncovered some environmental issue that needs to be resolved posthaste.

You never allow the build to go on failing, because if you allow the build to fail, you’ll get used to it failing. And if you get used to the failures, you’ll start ignoring them.

The more you ignore those failures, the more annoying the failure alerts become. And the more tempted you are to turn the failing tests off until you can fix them—later. You know. Later?

And that’s when the tests become lies.

With the failing tests removed, the build passes again. And that makes everyone feel good again. But it’s a lie.

So build continuously. And never let the build fail.

Chapter 32. Relentless Improvement

5. I will fearlessly and relentlessly improve my creations at every opportunity. I will never degrade them.

Robert Baden Powell, the father of the Boy Scouts of America, left a posthumous message exhorting the scouts to leave the world a better place than they found it. It was from this statement that the “Boy Scout Rule” was derived.

Check the code in cleaner than you checked it out.

How? By performing random acts of kindness upon the code every time you check it in. One of those random acts of kindness is to increase test coverage.

Test Coverage

Do you measure how much of your code is covered by your tests? Do you know what percentage of lines are covered? Do you know what percentage of branches are covered? If not, why not?

There are plenty of tools that can measure coverage for you. For most of us, those tools come as part of our IDE and are trivial to run. So there’s usually no excuse for not knowing what your coverage numbers are.

What should you do with those numbers? First, let me tell you what not to do. Don’t turn them into management metrics. Don’t fail the build if your test coverage is too low. Test coverage is a complicated concept that should not be used so naively.

Such naive usage sets up perverse incentives to cheat. And it is very easy to cheat test coverage. Remember that coverage tools only measure the amount of code that was executed, not the code that was actually tested. This means that you can drive the coverage number very high by pulling the assertions out of your failing tests. And, of course, that makes the metric useless.

The best policy is to use the coverage numbers as a developer tool to help you improve the code. You should work to meaningfully drive the coverage toward 100% by writing actual tests.

One hundred percent test coverage is always the goal, but it is also an asymptotic goal. Most systems never reach 100%, but that should not deter you from constantly trying to attain it.

That’s what you use the coverage numbers for. You use them as a measurement to help you improve, not as a bludgeon with which to punish the team and fail the build.

Mutation Testing

One hundred percent test coverage implies that any semantic change to the code should cause a test to fail. TDD, TCR, and to a lesser extent, Small Bundles, are good disciplines to approximate that goal because, if you follow those disciplines ruthlessly, every line of code, every condition, and every branch is tested.

Such ruthlessness, however, is often impractical. Programmers are human, and disciplines are always subject to pragmatics. So the reality is that even the most assiduous test-driven developer will leave gaps in the test coverage.

Mutations testing is a way to find those gaps; and there are mutation testing tools that can help.

A mutation tester runs your test suite and measures the coverage. Then, it goes into a loop, mutating your code in some semantic way and then running the test suite with coverage again. The semantic changes are things like changing `>` to `<` or `==` to `!=` or `x=<something>` to `x=null`. Each such semantic change is called a *mutation*.

The tool expects each mutation to fail the tests. Mutations that do not fail the tests are called *surviving mutations*. Clearly, the goal is to ensure that there are no surviving mutations.

Running a mutation test can be a big investment of time. Even relatively small systems can require hours of runtime. So these kinds of tests are best run over weekends, or at month's end.

I have frequently been impressed by the kinds of subtle problems these tools have uncovered. So it is definitely worth the occasional effort.

Semantic Stability

The goal of test coverage and mutation testing is to create a test suite that ensures semantic stability. The semantics of a system are the required behaviors of that system. A test suite that ensures semantic stability is one that fails whenever a required behavior is broken. We use such test suites to eliminate the fear of refactoring and cleaning the

code. Without a semantically stable test suite, the fear of change is often too great.

Following one of the testing disciplines gives us a good start on a semantically stable test suite; but none of those disciplines is sufficient for full semantic stability. Coverage, mutation testing, and acceptance testing should be used to improve the semantic stability toward completeness.

Cleaning

Perhaps the most effective of the random acts of kindness that will improve the code is simple cleaning—refactoring with the goal of improvement.

What kinds of improvements can be made? There is, of course, the obvious elimination of code smells. But I will often clean code even when it isn’t smelly.

I make tiny little improvements in the names, in the structure, in the organization. These changes might not be noticed by anyone else. Some folks might even think they make the code less clean. But my goal is not simply the state of the code. By doing minor little cleanings, I learn about the code. I become more familiar and more comfortable with it. Perhaps my cleaning did not actually improve the code in any objective sense; but it improved my understanding of and facility with that code. The cleaning improved me as a developer of that code.

The cleaning also did something else that should not be understated. By cleaning the code, even in minor ways, I am flexing that code. And one of the best ways to ensure that code stays flexible is to regularly flex it. Every little bit of cleaning I do is actually a test of the code’s flexibility. If I find a small cleanup to be a bit difficult, I have detected an area of inflexibility that I can now correct.

Remember, software is supposed to be soft. How do you know that it is soft? By testing that softness on a regular basis. By doing little cleanups and little improvements and feeling how easy or difficult those changes are to make.

Creations

The word used in the promise above is *creations*. In this chapter I have focused mostly on code; but code is not the only thing that programmers create. We create designs, and documents, and schedules, and plans. All these artifacts are creations that should be continuously improved.

We are human beings. Human beings make things better with time. We constantly improve everything we work on.

Chapter 33. Maintain High Productivity

6. I will do all that I can to keep the productivity of myself and others as high as possible. I will do nothing that decreases that productivity.

Productivity. That's quite a topic, isn't it. How often do you feel that it's the only thing that matters at your job? If you think about it, productivity is what this book, and what all my books on software, is about.

They're about how to go fast.

And what we've learned over the last eight decades of software is that the way you go fast is to go well.

The *only* way to go fast is to go well.

So you keep your code clean. You keep your designs clean. You write semantically stable tests and keep your coverage high. You know and use appropriate design patterns. You keep your methods small and your names precise.

But those are all *indirect* methods of achieving productivity. In this chapter, we're going to talk about much more direct ways to keep productivity high:

- Viscosity: Keeping your development environment efficient

- Managing distractions: Dealing with everyday business and personal life
- Time management: Effectively separating productive time from all the other junk you have to do

Viscosity

Programmers are often very myopic when it comes to productivity. They view the primary component to productivity as their ability to write code quickly.

But the writing of code is a very small part of the overall process. If you made the writing of code *infinitely* fast, it would increase overall productivity by only a small amount.

That's because there's a lot more to the software process than just writing code. There's at least

- Building
- Testing
- Debugging
- Deploying

And that doesn't count the requirements, the analysis, the design, the meetings, the research, the infrastructure, the tooling, and all the other stuff that goes into a software project.

So, although it is important to be able to write code efficiently, it's not even close to the biggest part of the problem. Let's tackle some of the other issues one at a time.

Building

If it takes you 30 minutes to build after a 5-minute edit, then you can’t be very productive, can you?

There is no reason, in the third and subsequent decades of the twenty-first century, that builds should take more than a minute or two.

Before you object to this, think about it. How could you speed up the build? Are you utterly certain, in this age of cloud computing, that there is no way to dramatically speed up your build? Find whatever is causing the build to be slow, and fix it. Consider it a design challenge.

Testing

Is it your tests that are slowing the build? Same answer: Speed up your tests. Here, look at it this way. My poor little laptop has four cores running at a clock rate of 2.8GHz. That means it can execute around *10 billion instructions per second.*

Do you even have 10 billion instructions in your whole system? If not, then you should be able to test your whole system in less than a second.

Unless, of course, you are executing some of those instructions more than once.

For example, how many times do you have to test login to know that it works? Generally speaking, once should be sufficient.

So how many of your tests go through the login process? Any more than one would be a waste!

If login is required before each test, then, when executing tests, you should short-circuit the login process. Use one of

the mocking patterns. Or, if you must, remove the login process from systems built for testing.

The point is, don’t tolerate repetition like that in your tests. It can make them horrifically slow.

As another example, how many times do your tests walk through the navigation and menu structure of the user interface? How many tests start at the top and then walk through a long chain of links to finally get the system into the state where the test can be run?

Any more than once per navigation pathway would be a waste! So build a special testing API that allows the tests to quickly force the system into the state you need, without logging in, and without navigating.

How many times do you have to execute a query to know that it works? Once! So mock out your databases for the majority of your tests. Don’t allow the same queries to be executed over and over and over again.

Peripheral devices are slow. Disks are slow. Web sockets are slow. UI screens are slow. Even SSDs can be slow. Don’t let slow things slow down your tests. Mock them out. Bypass them. Get them out of the critical path of your tests.

Don’t tolerate slow tests. Keep your tests running fast!

Debugging

Does it take a long time to debug things? Why? Why is debugging slow?

You are using a testing discipline and writing unit tests, aren’t you? You are writing acceptance tests too, right? And you are measuring test coverage with a good coverage analysis tool, right? And you are periodically proving that

your tests are semantically stable by using a mutation tester, right?

If you are doing all those things, or even just *some* of those things, your debug time can be reduced to insignificance.

Deploying

Does deployment take forever? Why? I mean, you *are* using deployment scripts, right? You aren't deploying manually, are you?

Remember, you are programmers. Deployment is a procedure; automate it! And write tests for that procedure too! You should be able to deploy your system, every time, with a single click.

Managing Distractions

One of the most pernicious destroyers of productivity is distraction from the job. There are many different kind of distractions. It is important for you to know how to recognize them and defend against them.

Meetings

Are you slowed down by meetings? I have a very simple rule for dealing with meetings. It goes like this:

When the meeting gets boring, leave.

You should be polite about it. Wait a few minutes for a lull in the conversation, and then tell the participants that you believe your input is no longer required and ask them if they would mind if you returned to the rather large amount of work you have to do.

Never be afraid to leave a meeting. If you don’t figure out how to leave, then some meetings will keep you forever.

You would also be wise to decline most meeting invitations. The best way to avoid getting caught in long, boring meetings is to politely refuse the invitation in the first place.

Don’t be seduced by the fear of missing out. If you are truly needed, they’ll come get you.

When someone invites you to a meeting, make sure they’ve convinced you that you really need to go. Make sure they understand that you can only afford a few minutes and that you are likely to leave before the meeting is over.

And make sure you sit close to the door.

And if you are a group leader or a manager, remember that one of your primary duties is to defend your team’s productivity by *keeping them out of meetings*.

Music

I used to code to music; long, long ago. But I found that listening to music impedes my concentration. Over time, I realized that listening to music only *feels* like it helps me concentrate, when actually it divides my attention.

One day, while looking over some year-old code, I realized that my code was suffering under the lash of the music. There, scattered through the code in a series of comments, were the lyrics to the song I had been listening to.

Since then, I’ve stopped listening to music while I code, and I’ve found I am much happier with the code I write and with the attention to detail that I can give it.

Programming is the act of arranging elements of procedure through sequence, selection, and iteration.

Music is composed of tonal and rhythmic elements arranged through sequence, selection, and iteration.

Could it be that listening to music uses the same parts of your brain that programming uses, thereby consuming part of your programming ability? That's my theory, and I'm sticking to it.

You will have to work this out for yourselves. Maybe the music really does help you. But maybe it doesn't. I'd advise you to try coding without music for a week, and see if you don't end up producing more and better code.

Mood

It's important to realize that being productive requires that you become skilled at managing your emotional state. Emotional stress can kill your ability to code. It can break your concentration and keep you in a perpetually distracted state of mind.

For example, have you ever noticed that you can't code after a huge fight with your significant other?[1] Oh, maybe you type a few random characters in your IDE, but they don't amount to much. Perhaps you pretend to be productive by hanging out in some boring meeting that you don't have to pay much attention to.

[1] One of the wisest attitudes I've encountered regarding relationships came from the movie *The Comancheros* (1961). John Wayne's character said: "Me and my wife used to fight like a couple of wildcats with only one tree between us. But you soon find out that it doesn't make a tinker's damn who's got the upper hand. A few years roll by and you kinda settle down to being at ease with each other. Then life gets worth living."

Here's what I've found works best to restore me to productivity.

Act. Act on the root of the emotion. Don’t try to code. Don’t try to cover the feelings with music or meetings. It won’t work. Act to resolve the emotion.

If you find yourself at work too sad or depressed to code because of a fight with your spouse, then call them to try to resolve the issue. Even if you don’t actually get the issue resolved, you’ll find that the action to attempt a resolution will sometimes clear you mind well enough to code.

You don’t actually have to solve the problem. All you have to do is convince yourself that you’ve taken enough appropriate action. I usually find that’s enough to let me redirect my thoughts to the code I have to write.

The Flow

There’s an altered state of mind that many programmers enjoy. It’s that hyper-focused, tunnel vision state where the code seems to pour out of every orifice of your body. It can make you feel superhuman.

Despite the euphoric sensation, I’ve found over the years that the code I produce in that altered state tends to be pretty bad. The code is not nearly as well considered as code I write in a normal state of attention and focus.

So nowadays, I resist getting into the flow.

Pairing is a very good way to stay out of the flow. The very fact that you must communicate and collaborate with someone else seems to interfere with the flow.

Avoiding music also helps me stay out of the flow, because it allows the actual environment to keep me grounded in the real world.

If I find that I’m starting to hyper focus, I’ll break away and do something else for a while.

Time Management

One of the most important ways to manage distraction is to employ a time management discipline. The one I like best is *the Pomodoro Technique.*[2]

[2] [Pomodoro].

Pomodoro is Italian for tomato. Indeed, English-speaking teams tend to use the word *tomato* instead. But you’ll have better luck with Google if you search for “Pomodoro Technique.”

The aim of the technique is to help you manage your time and focus during a regular workday. It doesn’t concern itself with anything beyond that.

At its core, the idea is quite simple. Before you begin to work, you set a timer (traditionally a kitchen timer in the shape of a tomato) for 25 minutes.

Next, you work. And you work until the timer rings.

Then, you break for five minutes, clearing your mind and body.

Then, you start again. Set the timer for 25 minutes, work until the timer rings, and then break for 5. And you do this over and over.

There’s nothing magical about 25 minutes. I’d say anything between 15 and 45 minutes is reasonable. But once you choose a time, stick with that time for a while.

Of course, if I were 30 seconds away from getting a test to pass when the timer rang, I’d finish the test. On the other

hand, maintaining the discipline is important. I wouldn’t go more than a minute beyond.

So far, this sounds mundane, but handling interruptions, like phone calls, is where this technique shines. The rule is to *Defend the Tomato!*

Tell whoever is trying to interrupt you that you’ll get back to them within 25 minutes—or whatever the length of your tomato is. Dispatch the interruption as quickly as possible, and then return to work.

Then, after your break, handle the interruption.

This means that the time between tomatoes will sometimes get pretty long, because people who interrupt you often require a lot of time.

Again, that’s the beauty of this technique. Because, at the end of the day, you count the number of tomatoes you completed. And that gives you a measure of your productivity.

Once you’ve gotten good at breaking your day up into tomatoes like this, and defending the tomatoes from interruptions, then you can start planning your day by allocating tomatoes to it. You may even begin estimating your tasks in terms of tomatoes and planning your meetings and lunches around them.

Chapter 34. Work as a Team

7. I will continuously ensure that others can cover for me and that I can cover for them.

Imagine a team of soccer players moving the ball down the field toward the goal. Imagine that, during a play, one of the players trips and falls down. What do his teammates do? They step into the breech, cover the hole, and *keep the ball moving down the field*.

The sailors on a navy vessel each have a job to do, but they also know how to do someone else's job. Why? Because in the heat of a battle, a sailor might go down. Another sailor *must* be able to step into the breech and *keep the ship in the battle*.

It is no different on a software team. Indeed, that's why we call it a *team*. If one of the members of the team goes down for any reason, others on the team must be able to step into the breech, fill that hole, and keep the project on track.

Collaborative Programming

Segregation of knowledge into silos is extremely detrimental to a team and an organization. The loss of an individual can mean the loss of an entire segment of knowledge. It can paralyze the team and the organization. It also means that the individuals on the team don't have sufficient context to

understand each other. Often, they wind up talking past each other.

The cure for that is to spread knowledge through the team. Make sure each team member knows a lot about the work that other team members are performing.

And the best way to spread that knowledge is to *work together*—to *pair*, or *mob*.

The truth is that there's hardly any better way to improve the productivity of a team than collaborative programming. Because a team that knows the deep connections among the work that's being done can't help but be much more productive than a group of silos.

Open/Virtual Office

It is also important that the members of the team see and interact with each other very frequently. The best way to achieve this is to put them into a room together.

In the early 2000s I owned a company that helped organizations adopt Agile development. We would send a group of instructors and coaches to those companies and guide them through the change. Before each engagement began, we told the managers to rearrange the office space so that the teams we would be coaching worked together in their own team rooms.

It happened more than once that, before we arrived to begin coaching, the managers told us that the teams were already much more productive, just because they were working together in the same room.

I wrote the first version of this chapter in the first quarter of 2021. The COVID-19 pandemic was beginning to wane and we were all looking forward to a return to normal life. But

the pandemic has left, in its wake, a large number of software teams that work remotely.

Working remotely can never be quite as productive as working together in the same room. Even with the best electronic help, seeing each other on screens is just not as good as seeing each other in person. Still, the electronic systems for collaboration are very good nowadays. So, if you are working remotely, *use them*.

Create a virtual team room. Keep everyone's face in view. Keep everyone's audio channel as open as feasible. Your goal is to create the illusion of a team room, with everyone in it working together.

Pairing and mobbing enjoy a lot of electronic support nowadays. It is relatively easy to share screens and program together over distances. While you do that, keep the faces and the audio up and running. You want to be able to *see* each other while you are collaborating on code.

Remote teams should try, as hard as they can, to maintain the same working hours. This is very difficult when there is a huge East–West distribution of programmers. So try to keep the number of time zones in each team as small as possible, and try very hard to have at least six contiguous hours per day when everyone can be together in the virtual team room.

Have you ever noticed how easy it is to yell at another driver while driving your car? This is the windscreen effect. When you are sitting behind a windscreen, it's easy to see other people as fools, imbeciles, and even enemies. It is easy to dehumanize them. This effect happens, to a lesser degree, behind computer screens.

So teams should get together in the same physical room several times per year. I recommend one week each quarter.

This will help the team congeal and maintain itself as a team. It is very hard to fall into the windscreen trap with someone you ate lunch with, and physically collaborated with, two weeks ago.

Chapter 35. Estimate Honestly and Fairly

8. I will produce estimates that are honest both in magnitude and precision. I will not make promises without reasonable certainty.

In this chapter, we're going to talk about estimating projects and large tasks that take many days or weeks to accomplish. This is not the same as the Agile practice of estimating small tasks and stories, which I describe in the book *Clean Agile.*[1]

[1] [Clean Agile].

Knowing how to estimate is an essential skill for every software developer; and one that most of us are very, very bad at.

The skill is essential because every business needs to know, roughly, how much something is going to cost before they commit resources to it.

Unfortunately, our failure to understand what estimates actually are, and how to create them, has led to an almost catastrophic loss of trust between programmers and businesses.

The landscape is littered with billions of dollars in software failures. Often, those failures are due to poor estimation. It is not uncommon for estimates to be off by a factor of 2, 3, or

even 4 or 5. But why? Why are estimates so hard to get right?

Mostly it’s because we don’t understand what estimates actually are, and how to create them.

You see, in order for estimates to be useful, they must be honest. They must be honestly accurate, and they must be honestly precise. But most estimates are neither. Indeed, most estimates are lies.

Lies

Most estimates are lies because most estimates are constructed backward from a known end date.

Consider H`ealthcare.gov`, for example. The president of the United State signed a bill into law that mandated a specific date when that software system was to be turned on.

The depth of that illogic is nausea inducing. I mean, how absurd. Nobody was asked to estimate the end date; they were just told what the end date had to be—by law!

So, of course, all estimates associated with that mandated date were lies. How could they be anything else?

It reminds me of a team I was consulting for about twenty years ago. I remember being in the project room with them, when the project manager walked in.

He was a young fellow; perhaps 25. He’d just returned from a meeting with his boss. He was visibly agitated.

He told the team how important the end date was. He said: “We really have to make that date. I mean, we *really* have to make that date.”

Of course, the rest of the team just rolled their eyes and shook their heads. The need to make the date was not a solution for making the date. The young manager offered no solution.

Of course, estimates in an environment like that are just lies that support the plan.

And that reminds me of another client of mine who had a huge software production plan on the wall—full of circles and arrows and labels and tasks.

The programmers referred to it as *The Laugh Track*.

In this chapter, we're going to talk about real, worthwhile, honest, accurate, and precise estimates. The kinds of estimates that professionals create.

Honesty, Accuracy, Precision

The most important aspect of an estimate is honesty. Estimates don't do anybody any good unless they are honest.

The most honest estimate you can give is: "I don't know." But that estimate is not particularly accurate or precise. After all, you do know *something* about the estimate. So the challenge is to quantify what you do *and don't* know.

First, your estimate must be accurate. That doesn't mean you give a firm date—you must not dare be that precise. It just means that you name a *range* of dates that you feel confident about.

So, for example, "Sometime between now and 10 years from now" is a pretty accurate estimate for how long it would take you to write a hello world program. But it lacks precision.

On the other hand, “Yesterday at 2:15 a.m.” is a very precise estimate, but probably not very accurate if you haven’t started yet.

Do you see the difference? When you give an estimate, you want it to be honest both in accuracy and in precision. To be accurate, you name a range of dates within which you are confident. To be precise, you narrow that range down to the level of your confidence.

And for both of these operations, brutal honesty is the only option.

Lessons from Me

To be honest about these things, you have to have some idea of how *wrong* you might be. So let me tell you two stories about how wrong I once was.

Story #1: Vectors

The year was 1978. I was working at a company named Teradyne, in Deerfield, Illinois. We built automated test equipment for the telephone company.

I was a young programmer, 26 years old. And I was working on the firmware for an embedded measurement device that bolted into racks in telephone central offices. This device was called a *COLT*—Central Office Line Tester.

The processor in the COLT was an Intel 8085—an early 8-bit microprocessor. We had 32K of solid-state RAM and another 32K of ROM. The ROM was based on the Intel 2708 chip, which stored 1K × 8. So we used 32 of those chips.

Those chips were plugged into sockets on our memory boards. Each board could hold 12 chip,; so we used three

boards.

The software was written in 8085 assembler. The source code was held in a set of source files that were compiled as a single unit. The output of the compiler was a single binary file somewhat less than 32K in length.

We took that file and cut it up into thirty-two 1K chunks. Each 1K chunk was then burned onto one of the ROM chips, which were then inserted into the sockets on the ROM boards.

As you can imagine, you had to get the right chip into the right socket on the right board. So we were very careful to label them.

We sold hundreds of these devices. And they were installed in telephone central offices all over the country.

So what do you think happened when we changed that program? Just a one-line change?

If we added or removed a line, then all the addresses of all the subroutines after that point changed. And since those subroutines were called by other routines earlier in the code, every chip was affected. We had to reburn all 32 chips for a one-line change!

This was a nightmare. We had to burn hundreds of sets of chips and ship them to all the field service reps, all around the world. Then, those reps would have to drive hundreds of miles to get to all the central offices in their district. They'd have to open our units, pull out all the memory boards, remove all 32 of the old chips, insert the 32 new chips, and reinsert the boards.

Now, I don't know if you know this, but the act of removing and inserting a chip into a socket is not entirely reliable. The little pins on the chips tend to bend and break in

frustratingly silent ways. So the poor field service folks had to have lots of spares of each of the 32 chips; and suffer through the inevitable debugging by removing and reinserting chips until they could get a unit to work.

So my boss came to me one day and told me that we had to solve this problem by making each chip *independently deployable*. He didn't use those words, of course, but that was the intent. Each chip needed to be turned into an independently compilable and deployable unit. This would allow us to make changes to the program without forcing all 32 chips to be reburned. Indeed, in most cases, we could simply redeploy a single chip—the chip that was changed.

I won't bore you with the details of the implementation. Suffice it to say that it involved vector tables, indirect calls, and the partitioning of the program into independent chunks of less than 1K each.[2]

[2] In other words, each chip was turned into a polymorphic object.

My boss and I talked through the strategy, and then he asked me how long it would take me to get this done.

I told him two weeks.

But I wasn't done in two weeks. I wasn't done in four weeks. Nor was I done in six, eight, or ten weeks.

The job took me 12 weeks to complete. It was a lot more complicated than I thought.

So I was off by a factor of six. Six!

Fortunately, my boss didn't get mad. He saw me working on it every day. He got regular status updates from me. He understood the complexities I was dealing with.

But still. Six? How could I have been so wrong?

Story #2: pCCU

Then there was that time, in the early ’80s, that I had to work a miracle. You see, we had promised a new product to our customer. It was called *CCU-CMU*.

Copper is a precious metal. It’s rare and expensive. The phone company decided to harvest the huge network of copper wires that it had installed all over the country during the last century. They replaced it with a much cheaper, high-bandwidth network of coaxial cable and fiber carrying digital signals. This was known as *digital switching*.

The CCU-CMU was a complete rearchitecture of our measurement technology that fit within the new digital switching architecture of the phone company.

Now, we had promised the CCU-CMU to the phone company a couple of years before. We knew it was going to take us one person-year or so to build the software. But then, we just never quite got around to building it.

You know how it goes. The phone company delayed their deployment, so we delayed our development. There were always lots of other, more urgent issues to deal with.

So one day, my boss calls me into his office and says that they had forgotten about one small customer who had already installed an early digital switch. That customer was now expecting a CCU/CMU within the next month—as promised.

So now I had to create a person-year of software in less than a month.

So I told my boss that this was impossible. There was no way that I could get a fully functioning CCU/CMU built in one month.

Then, he looked at me with a sneaky grin and said that there was a way to cheat.

You see, this was a very small customer. Their installation was literally the smallest possible configuration for a digital switch. What's more, the configuration of their equipment just happened—just happened—to eliminate virtually all of the complexity that the CCU/CMU solved.

Long story short: I got a special-purpose, one-of-a-kind unit up and running for the customer in two weeks. We called it the *pCCU*.

The Lesson

Those two stories are examples of the huge range that estimates can have. On the one hand, I underestimated the vectoring of the chips by a factor of six. On the other, we found a solution to the CCU/CMU in one twentieth the expected time.

This is where honesty comes in. Because, honestly, when things go wrong, they can go very, very wrong. And when things go right, they can sometimes go very, very right.

And this makes estimating one hell of a challenge.

Accuracy and Precision

It should be clear by now that an estimate for a project *cannot* be a date. A single date is far too precise for a process that can be off by as much as a factor of six, or even twenty.

That variation means that every estimate is likely wrong. That's why we call them *estimates*. But even though an estimate is wrong, it may not be very wrong. So part of the

job of estimation is to estimate how wrong the estimate probably is.

A single date is too precise to be honest, so estimates are ranges; they must be *probability distributions*. Probability distributions have a mean, and a width—sometimes called the *standard deviation* or the *sigma*. We need to be able to express our estimates with both a mean and a sigma.

My favorite technique for estimating the probability distribution of an estimate is to estimate three numbers: the best case, the worst case, and the normal case.

The normal case is how long you think the task would take you if the average number of things go wrong—if things go the way they *usually* do. Think of it as the date that you've got a 50% chance of making.

The worst-case estimate is the Murphy's law estimate. It assumes that anything that can go wrong will go wrong. It is deeply pessimistic. Think of it as the date that you've got a 95% chance of making.

The best-case estimate is when everything goes right. You eat the right breakfast cereal every morning. When you get into work, your coworkers are all polite and friendly. There are no disasters in the field, no meetings, no telephone calls, no distractions.

Your chances of hitting the best-case estimate are 5%: one in twenty.

OK, so now we have three numbers. The best case which has a 5% chance of success. The nominal case which has a 50% chance of success. And the worst case, which has a 95% chance of success, represents a normal curve —a probability distribution. It is this probability distribution that is your actual estimate.

Notice that this is not a date. We don’t know the date. We don’t know when we are really going to be done. All we really have is a crude idea of the probabilities.

Without certain knowledge, probabilities are the only logical way to estimate.

If your estimate is a date, you are making a commitment, not an estimate. And if you make a commitment, you *must* succeed.

Sometimes you have to make commitments. But the thing about commitments is that you absolutely must succeed. You must never promise to make a date that you aren’t sure you can make. To do so would be deeply dishonest.

So, if you don’t know—and I mean *know*—that you can make a certain date, then you don’t offer that date as an estimate. You offer a range of dates instead. Offering a range of dates with probabilities is much more honest.

Aggregation

OK, so now let’s say that you’ve got a whole project full of tasks that have been described in terms of Best- (B), Normal- (N), and Worst- (W) case estimates. How do you aggregate them all into a single estimate for the whole project?

You simply represent the probability of each task, and then accumulate those probabilities using standard statistical methods.

The first thing we want to do is represent each task in terms of its expected completion time and the standard deviation.

Now, remember six standard deviations (three on each side of the mean) corresponds to a probability of better than

99%. So we're going to set our standard deviation, our sigma, to Worse minus Best over 6.

The expected completion time (Mu) is a bit trickier. Notice that N is probably not equal to (W − B), the midpoint. Indeed, the midpoint is probably well past N. This is because it is much more likely for a project to go long than to go short. So it's probably best to use a weighted average, like this: Mu = (2N + (B + W) ÷ 2) ÷ 3.

Now you have calculated the Mu and sigma for a set of tasks. So the expected completion time for the whole project is just the sum of all the Mus. The sigma for the project is the square root of the sum of the squares of all the sigmas.

This is just basic statistical mathematics.

What I've just described is the estimation procedure invented back in the late 1950s to manage the Polaris Fleet Ballistic Missile program. It has been used successfully on many thousands of projects since.

It is called *PERT*—Program Evaluation and Review Technique.

Honesty

So we started with honesty. Then, we talked about accuracy and precision. Now it's time to come back to honesty.

The kind of estimating we are talking about here is intrinsically honest. It is a way of communicating, to those who need to know, the level of your uncertainty.

This is honest because you truly are uncertain. And those with the responsibility to manage the project must be aware of the risks they are taking so that they can manage those risks.

But uncertainty is something that people don’t like. Your customers and managers will almost certainly press you to be more certain.

We’ve already talked about the cost of increasing certainty. The only way to truly increase certainty is to do parts of the project. You can only get perfect certainty if you do the entire project.

So part of what you have to tell your customers and managers is the cost of increasing certainty.

Sometimes, however, your superiors may ask you to increase certainty using a different tactic. They may ask you to commit.

You need to recognize this for what it is. They are trying to manage their risk, by putting it onto you. By asking you to commit, they are asking you to take on the risk that it is their job to manage.

Now, there’s nothing wrong with this. Managers have a perfect right to do this. And there are many situations in which you should comply. But—and I stress this—only if you are reasonably certain you *can* comply.

If your boss comes to you and asks if you can get something done by Friday, you should think very hard about whether that is reasonable. And if it is reasonable and probable, then by all means, say yes!

But under no circumstances should you say yes if you are not sure.

If you are not sure, then you *must* say no, and then describe your uncertainty as we’ve described. It is perfectly OK to say: “I can’t promise Friday. It might take as long as the following Wednesday.”

In fact, it’s absolutely critical that you say no to commitments that you are not sure of. Because if you say yes, you set up a long domino chain of failures for you, your boss, and many others. They’ll be counting on you, and you’ll let them down.

So, when you are asked to commit, and you are sure you can comply, then say yes. But if you aren’t sure, then say *no*, and describe your uncertainty.

And be willing to discuss options and workarounds. Be willing to hunt for ways to say yes. Never be eager to say *no*. But also, never be afraid to say no.

You see, you were hired for your ability to say no. Anybody can say yes. But only people with skill and knowledge know when and how to say no.

One of the prime values you bring to the organization is your ability to know when the answer must be no. By saying no at those times, you will save your company untold grief and money.

Pressure

One last thing: Often a manager will try to cajole you into committing—into saying yes. Watch out for this.

They might tell you that you aren’t being a team player, or that other people have more commitment than you do. Don’t be fooled by those games.

Be willing to work with them to find solutions, but don’t let them bully you into saying yes when you know you shouldn’t.

And be very careful with the word *try*. Your boss might say something reasonable, like, “Well, will you at least try?”

The answer to this is:

NO! I am already trying. How dare you suggest that I am not? I am trying as hard as I can; and there is no way I can try harder. There are no magic beans in my pocket with which I can work miracles.

You might not want to use those exact words, but that's exactly what you should be thinking.

And remember this: If you say "Yes, I'll try," then you are lying. Because you have no idea how you are going to succeed. You don't have any plan to change your behavior. You said yes just to get rid of them. And that is the most dishonest thing of all.

Chapter 36. Respect for Fellow Programmers

9. I will respect my fellow programmers for their ethics, standards, disciplines, and skill. No other attribute or characteristic will be a factor in my regard for my fellow programmers.

We, software professionals, accept the weighty burden of our craft. We brave folks are men and women, straight and gay, black, brown, yellow, and white, republicans, democrats, religious believers, and atheists. We are humans, in all the many forms and varieties that humans come in. We are a community—a brotherhood—of mutual respect.

The only qualifications for entry into our brotherhood, and for receiving the acceptance and the respect of each and every member of that brotherhood, are the skills, disciplines, standards, and ethics of our profession. No other human attribute is worthy of consideration. No discrimination on any other basis can be tolerated.

Within our profession, we are programmers first.

Chapter 37. Never Stop Learning

10. I will never stop learning and improving my craft.

A programmer never stops learning.

I'm sure you've heard it said that you should learn a new language every year. Well, you should. A good programmer should know a dozen languages or so.

And not just a dozen varieties of the same language. Not just C, C++, Java, and C#. Rather, you should know languages from many different families.

You should know a statically typed language like Java or C#. You should know a procedural language like C or Pascal. You should know a logic language like Prolog. You should know a stack language like Forth. You should know a dynamically typed language like Ruby. You should know a functional language like Clojure or Haskell.

You should also know several different frameworks. And several different design methodologies, and several different development processes.

I don't mean to say you should be an expert in all these things; but you should make a point to expose yourself to them at significantly more than a cursory level.

Indeed, the list of things you should similarly expose yourself to is virtually endless. Our industry has experienced rapid change over the decades, and that change is likely to continue for some time. You have to keep up with it.

And that means you have to keep on learning. Keep reading books and blogs. Keep watching videos. Keep going to conferences and user groups. Keep going to training courses. Keep learning.

Pay attention to the treasured works of the past. The books written in the ’60s and ’70s and ’80s are wonderful sources of insight an information. Don’t fall into the trap of thinking that all that old stuff is out of date. There is not much in our industry that actually goes out of date. Respect the effort and accomplishments of those who came before you, and study their advice and conclusions.

And don’t fall into the trap of thinking that it is your employer’s job to train you. This is *your* career; you have to take responsibility for it. It is your job to learn. It is your job to figure out what you should be learning.

If you are lucky enough to work for a company that will buy you books and send you to conferences and training classes, then take full advantage of that. If not, then pay for those books, conferences, and courses yourself.

And plan to spend some time at this. Time every week.

You owe your employer 35 to 40 hours per week. You owe your career another 10 to 20.

That’s what a professional does. A professional puts in the time to groom and maintain their career. And that means you should be working 50 to 60 hours per week total. Mostly at work, but a lot at home too.

Afterword



Justin Michael Martin

“When you’re a carpenter making a beautiful chest of drawers, you’re not going to use a piece of plywood on the back, even though it faces the wall and nobody will ever see it. You’ll know it’s there, so you’re going to use a beautiful piece of wood on the back. For you to sleep well at night, the aesthetic, the quality, has to be carried all the way through.”

—Steve Jobs

Aspiring to avoid a colleague quoting *The Hitchhiker’s Guide to the Galaxy*’s “...probably the oddest thing in the universe”

when talking about your code is likely a noble pursuit. This was, of course, in reference to the legendary Babel Fish, which could translate any alien language directly into your ear.

I am often reminded of *The Hitchhiker's Guide to the Galaxy* when thinking of or reading *Clean Code*—not just because my father loves the book series and encouraged all of his children to read it, not only because he references it in *Clean Code* along with his other books, and not only because *Clean Code* inspires deep thought. I am reminded of *The Hitchhiker's Guide to the Galaxy* because *Clean Code* is the must-have companion guide to navigating the ever-changing universe of software development.

This is not to say that this book will automagically allow you to understand any code. On the contrary, *Clean Code* is the inverted Babel Fish that you place at your fingertips so that other developers don't need a Babel Fish to understand your code!

When I was in seventh grade, my dad had me code one of my first games: *Hunt the Wumpus*. It is a simple text-based adventure game whereby you traverse a cave to find and terminate the wumpus while avoiding traps, pits, and whatever else a clever developer can code to challenge you.

Years later, I was volunteering at the Agile Conference in 2007. A workshop room had been set up for attendees to sit down and pair-program on a *Hunt the Wumpus* game built in Java. After hearing some great talks, I was eager to do some coding, so I took a sticky note off a community-managed Kanban-style whiteboard and sat down to start adding a new "fire arrow" feature. Soon after, none other than Ward Cunningham sat down beside me to start pairing. Being the young tyke that I was, I didn't recognize I was now coding with the man who built the first wiki.

Unabashed, I charged forward and explained the feature to my new pair, and my plan to create an inheritance structure that would allow us to add not only “fire arrows” but also “ice arrows” or “arcane arrows” and so on. He nodded his head with grandiose solemnity and encouraged me to proceed.

I started by writing a test for a new `Arrow` class, which would later be the parent for our `FireArrow` class. I had already written this game, so I totally knew what I was doing. Once I had my failing test, I passed Ward the keyboard. He dutifully made the test pass, but then paused with a meaningful “hrmmpph” and proceeded to query whether we really ought to introduce such complexity so early. He posited that rather than jumping to our end-state architecture, we might try to add the functionality gradually and allow the test-driven process to reveal an appropriate shape. Frustrated because I *knew* how it ought to be architected, I begrudgingly agreed to follow his lead. I was here to learn, after all.

His next move shocked me. He opened up the class with the code that launched the arrows and proceeded to comment a bunch of it out! Aghast, I righteously protested, thinking we were going backward rather than forward! In response, he showed how “everything is still green,” all the tests were passing, and that we might be able to get away with a little refactoring. The next 30 minutes passed in a series of moments of awe.

First, we did a number of subtle refactorings, adding a few tests and renaming/extracting some functions. I remember being struck by the fluidity in which he moved through the phases of red, green, refactor. It became abundantly clear that this wasn’t just a process he used while coding. It was a principle to him—a way of thinking, problem-solving, and producing clean and clear results so ingrained into his

workflow that the tests and the code were really one and the same.

Second, we began adding the new functionality for a “fire arrow.” Now that everything was neatly laid out, this was simple. He wrote the failing test. I made it pass in what couldn’t have been more than a line or two of code. I had finished making it pass before even realizing we had actually completed the feature. When it dawned on me, he nodded appreciatively.

Next, he prompted me to grab the next feature. I seem to recall grabbing “ice arrow” but then being instructed toward greater ambition, and grabbed “crossbow bolt” instead, as it was going to be more challenging. However, after writing the failing test and passing Ward the keyboard, I knew my assumption was wrong, and it was remarkably simple to add this totally new weaponization against the wumpus.

Finally, after crushing the new crossbow bolt feature, he asked me what the code looked like. I think he might have used verbiage like “what does it smell like?” but I can’t be certain. After pondering for a moment, thinking about “ice arrows” and “arcane bolts,” the code spoke to me. We had already outlined an interface for Projectile, which could encompass a whole range of new weaponry for the would-be wumpus hunter. I proceeded to extract out the new Projectile interface, and all the tests were still a beautiful shimmering green. He nodded satisfactorily, stood, said it was fun, and thanked me for pairing before leaving the workspace. It was a long while before I escaped my rumination and casually added in several new projectiles.

Shortly thereafter, my dad walked over and asked me how I enjoyed pairing with Ward Cunningham, the creator of the wiki. Suddenly, everything made sense. I had been coding with a legend, and legends don’t frolic with principles—they

live and breathe them. Within about a year, my dad codified many of these same principles in the first edition of *Clean Code*.

I tell this story to highlight the profound impact that can be made when the disciplines, strategies, and principles within this book are taken to heart.

Over the years, I have founded or worked at over a dozen early-stage startups. Oftentimes, I'd join a small team of developers who already had a product out the door serving a vast user base. More often than not, the driving principle was to go fast at all costs. The website/app is too slow? Throw more hardware at it. Product development is going slow? Hire more developers and product managers to speed things up. Move fast and break things.

Gradually, however, as the industry matured, teams started to focus more and more on having a robust test base, continuous deployment, code reviews, and, critically, clean code. In almost every case, the teams that followed the disciplines and principles laid out in this book were the ones with the highest product velocity. At first, I served as a steady reminder for everyone to write tests, use clear naming, and clean up or refactor the code. These days, it is often a given that the team will already be practicing these principles. Why? Because they work. Much of the startup world has already adopted these lessons because it is a winning strategy. The goal is still to go as fast as possible, but as my dad is fond of saying, "The only way to go fast is to go well."

The world of software development is changing fast. Advanced AI tools like Claude, Cursor, GPT-o, and Grok have made it easier than ever to produce working code. Developers are already using them to kick off projects, debug tricky errors, or even whip up fully tested versions of

Hunt the Wumpus. This trend isn't slowing down—it's accelerating.

It is for this reason that *Clean Code* principles are more crucial now than ever. AI can churn out code like a factory, but it's often like raw dough—tasty maybe, but not a fully baked pie. *Clean Code* is the recipe that turns it into something polished, whether you're refactoring by hand or tweaking AI prompts to guide the process. It's about crafting software that doesn't just work, but lasts.

Whether you are using AI to write code or using AI as a personal Babel Fish to understand someone else's code, if the intent is not inherent in the code, it is likely to be lost. Picture this: Grok spits out some code, you clean it up with Cursor and Claude, and it runs fine—until you realize it's a tangled mess. So you ask GPT-o to sort it out, only to end up with a knot of interdependencies. Without *Clean Code* as a compass, AI's output can quickly become a maintenance nightmare. The craft of writing clear, readable code remains essential.

As AI keeps evolving, it might one day churn out better code on its own—but don't bet on it yet. Developers who stick to *Clean Code* principles will be the ones delivering high-quality, maintainable software no matter how smart AI gets. *Clean Code* is the guide that ensures you or your team's work isn't just functional—it's something to be proud of.

Appendix. The Clean Code Debate



(Uncle) Bob Martin



Dr. John Kenneth Ousterhout

This document is the result of a series of discussions, some online and some in person, held between Robert “Uncle Bob” Martin and John Ousterhout between September 2024 and February 2025. If you would like to comment on anything in this discussion, we recommend that you do so on the Google group associated with APOSD.[1]

[1] [APOSD].

Introductions

JOHN:

Hi (Uncle) Bob! You and I have each written books on software design. We agree on some things, but there are some pretty big differences of opinion between my recent book, *A Philosophy of Software Design* (hereafter “APOSD”),

and your classic book *Clean Code*. Thanks for agreeing to discuss those differences here.

UB:

My pleasure, John. Before we begin, let me say that I've carefully read through your book, and I found it very enjoyable and full of valuable insights. There are some things I disagree with you on, such as TDD and abstraction-first incrementalism, but overall I enjoyed it a lot.

JOHN:

I'd like to discuss three topics with you: method length, comments, and test-driven development. But before getting into these, let's start by comparing overall philosophies. When you hear about a new idea related to software design, how do you decide whether or not to endorse that idea?

I'll go first. For me, the fundamental goal of software design is to make it easy to understand and modify the system. I use the term *complexity* to refer to things that make it hard to understand and modify a system. The most important contributors to complexity relate to information.

- How much information must a developer have in their head in order to carry out a task?
- How accessible and obvious is the information that the developer needs?

The more information a developer needs to have, the harder it will be for them to work on the system. Things get even worse if the required information isn't obvious. The worst case is when there is a crucial piece of information hidden in some faraway piece of code that the developer has never heard of.

When I’m evaluating an idea related to software design, I ask whether it will reduce complexity. This usually means either reducing the amount of information a developer has to know, or making the required information more obvious.

Now over to you: Are there general principles that you use when deciding which ideas to endorse?

UB:

I agree with your approach. A discipline or technique should make the job of programmers easier. I would add that the programmer we want to help most is not the author. The programmer whose job we want to make easier is the programmer who must read and understand the code written by others (or by themself a week later). Programmers spend far more hours reading code than writing code, so the activity we want to ease is that of reading.

Method Length

JOHN:

Our first area of disagreement is method length. On page 34 of *Clean Code*, you say: “The first rule of functions is that they should be small. The second rule of functions is that *they should be smaller than that*.” Later on, you say, “Functions should hardly ever be 20 lines long” and suggest that functions should be “just two, three, or four lines long.” On page 35, you say: “Blocks within `if` statements, `else` statements, `while` statements, and so on should be one line long. Probably that line should be a function call.” I couldn’t find anything in *Clean Code* to suggest that a function could ever be too short.

I agree that dividing up code into relatively small units (*modular design*) is one of the most important ways to reduce the amount of information a programmer has to keep in their mind at once. The idea, of course, is to take a complex chunk of functionality and encapsulate it in a separate method with a simple interface. Developers can then harness the functionality of the method (or read code that invokes the method) without learning the details of how the method is implemented; they only need to learn its interface. The best methods are those that provide a lot of functionality but have a very simple interface: They replace a large cognitive load (reading the detailed implementation) with a much smaller cognitive load (learning the interface). I call these methods *deep*.

However, like most ideas in software design, decomposition can be taken too far. As methods get smaller and smaller, there is less and less benefit to further subdivision. The amount of functionality hidden behind each interface drops, while the interfaces often become more complex. I call these interfaces *shallow*: They don't help much in terms of reducing what the programmer needs to know. Eventually, the point is reached where someone using the method needs to understand every aspect of its implementation. Such methods are usually pointless.

Another problem with decomposing too far is that it tends to result in *entanglement*. Two methods are entangled (or *conjoined* in APOSD terminology) if, in order to understand how one of them works internally, you also need to read the code of the other. If you've ever found yourself flipping back and forth between the implementations of two methods as you read code, that's a red flag that the methods might be entangled. Entangled methods are hard to read because the information you need to have in your head at once isn't all in the same place. Entangled methods can usually be

improved by combining them so that all the code is in one place.

The advice in *Clean Code* on method length is so extreme that it encourages programmers to create teeny-tiny methods that suffer from both shallow interfaces and entanglement. Setting arbitrary numerical limits such as two to four lines in a method and a single line in the body of an `if` or `while` statement exacerbates this problem.

UB:

While I do strongly recommend very short functions, I don't think it's fair to say that the book sets arbitrary numerical limits. The two-to-four-line functions that you referred to on page 34 were part of the *Sparkle* applet that Kent Beck and I wrote together in 1999 as an exercise for learning TDD. I thought it was remarkable that most of the functions in that applet were two to four lines long because it was a Swing program; and Swing programs tend to have very long methods.

As for setting limits, on page 13 I make clear that although the recommendations in the book have worked well for me and the other authors, they might not work for everyone. I claimed no final authority, nor even any absolute "rightness." They are offered for consideration.

JOHN:

I think these problems will be easiest to understand if we look at specific code examples. But before we do that, let me ask you, Bob: Do you believe that it's possible for code to be over-decomposed, or is smaller always better? And, if you believe that over-decomposition is possible, how do you recognize when it has occurred?

UB:

It is certainly possible to over-decompose code. Here's an example:

```
void doSomething() {doTheThing()} // over-decomposed.
```

The strategy that I use for deciding how far to take decomposition is the old rule that a method should do *One Thing*. If I can *meaningfully* extract one method from another, then the original method did more than one thing. "Meaningfully" means that the extracted functionality can be given a descriptive name, and that it does less than the original method.

JOHN:

Unfortunately, the One Thing approach will lead to over-decomposition:

The term *one thing* is vague and easy to abuse. For example, if a method has two lines of code, isn't it doing two things?

You haven't provided any useful guardrails to prevent over-decomposition. The example you gave is too extreme to be useful, and the "can it be named" qualification doesn't help: Anything can be named.

The One Thing approach is simply wrong in many cases. If two things are closely related, it might well make sense to implement them in a single method. For example, any thread-safe method will first have to acquire a lock, then carry out its function. These are two "things," but they belong in the same method.

UB:

Let me tackle the last thing first. You suggested that locking the thread and preforming a critical section should be together in the same method. However, I would be tempted to separate the locking from the critical section.

```
void concurrentOperation() {
    lock()
    criticalSection();
    unlock()
}
```

This decouples the critical section from the lock and allows it to be called at times when locking isn't necessary (e.g., in single-thread mode) or when the lock has already been set by someone else.

Now, on to the "ease of abuse" argument. I don't consider that to be a significant concern. `If` statements are easy to abuse. `Switch` statements are easy to abuse. Assignment statements are easy to abuse. The fact that something is easy to abuse does not mean that it should be avoided or suppressed. It simply means people should take appropriate care. There will always be this thing called *judgment*.

So, when faced with this snippet of code in a larger method:

```
...
amountOwed=0;
totalPoints=0;
...
```

it would be poor judgment to extract them as follows, because the extraction is not meaningful. The implementation is not more deeply detailed than the interface.

```
void clearAmountOwed() {
  amountOwed=0;
}
```

```
void clearTotalPoints() {
  totalPoints=0;
}
```

However, it may be good judgment to extract them as follows because the interface is abstract, and the implementation has deeper detail.

```
void clearTotals() {
  amountOwed=0;
  totalPoints=0;
}
```

The latter has a nice descriptive name that is abstract enough to be meaningful without being redundant. And the two lines together are strongly related so as to qualify for doing *one thing*: initialization.

JOHN:

Of course, anything can be abused. But the best approaches to design encourage people to do things the right way and discourage abuse. Unfortunately, the One Thing rule encourages abuse for the reasons I gave above.

And of course, software designers will need to use judgment: It isn’t possible to provide precise recipes for software design. But good judgment requires principles and guidance. The *Clean Code* arguments about decomposition, including the One Thing rule, are one-sided. They give strong, concrete, quantitative advice about when to chop things up, with virtually no guidance for how to tell you’ve gone too far. All I could find is a two-sentence example on

page 36 about Listing 3-3 (which is pretty trivial), buried in the middle of exhortations to “chop, chop, chop.”

One of the reasons I use the deep/shallow characterization is that it captures both sides of the trade-off; it will tell you when a decomposition is good and also when decomposition makes things worse.

UB:

You make a good point that I don’t talk much in the book about how to make the judgment call. Back in 2008, my concern was breaking the habit of the very large functions that were common in those early days of the Web. I have been more balanced in the second edition.

Still, if I must err, I’d rather err on the side of decomposition. There is value in considering and visualizing decompositions. They can always be inlined if we judge them to have gone too far.

JOHN:

Coming back to your `clearTotals` example:

- The `clearTotals` method seems to contradict the One Thing rule: The variables `amountOwed` and `totalPoints` don’t seem particularly related, so initializing them both is doing two things, no? You say that both statements are performing initialization, which makes it just one thing (initialization). Does that mean it would also be OK to have a single method that initializes two completely independent objects with nothing in common? I suspect not. It feels like you are struggling to create a clean framework for applying the One Thing rule; that makes me think it isn’t a good rule.
- Without seeing more context, I’m skeptical that the `clearTotals` method makes sense.

UB:

I hope you agree that between these two examples, the former is a bit better.

```
public  String makeStatement() {
  clearTotals();
  return makeHeader() + makeRentalDetails() + makeFooter();
}


public String makeStatement() {
  amountOwed=0;
  totalPoints=0;
  return makeHeader() + makeRentalDetails() + makeFooter();
}
```

JOHN:

Well, actually, no. The second example is completely clear and obvious: I don’t see anything to be gained by splitting it up.

SPOCK (a.k.a. UB):

Fascinating.

JOHN:

I think it will be easier to clarify our differences if we consider a nontrivial code example. Let’s look at the `PrimeGenerator` class from *Clean Code*, which is Listing 10-8 on pages 145–146. This Java class generates the first *N* prime numbers:

```
package literatePrimes;
```

```
import java.util.ArrayList;

public class PrimeGenerator {
    private static int[] primes;
    private static ArrayList<Integer> multiplesOfPrimeFactors;

    protected static int[] generate(int n) {
        primes = new int[n];
        multiplesOfPrimeFactors = new ArrayList<Integer>();
        set2AsFirstPrime();
        checkOddNumbersForSubsequentPrimes();
        return primes;
    }

    private static void set2AsFirstPrime() {
        primes[0] = 2;
        multiplesOfPrimeFactors.add(2);
    }

    private static void checkOddNumbersForSubsequentPrimes() {
        int primeIndex = 1;
        for (int candidate = 3;
             primeIndex < primes.length;
             candidate += 2) {
            if (isPrime(candidate))
                primes[primeIndex++] = candidate;
        }
    }
```

```
private static boolean isPrime(int candidate) {
    if (isLeastRelevantMultipleOfLargerPrimeFactor(candidate))
        multiplesOfPrimeFactors.add(candidate);
        return false;
    }
    return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
}

private static boolean
isLeastRelevantMultipleOfLargerPrimeFactor(int candidate) {
    int nextLargerPrimeFactor =
      primes[multiplesOfPrimeFactors.size()];
    int leastRelevantMultiple =
      nextLargerPrimeFactor * nextLargerPrimeFactor;
    return candidate == leastRelevantMultiple;
}

private static boolean
isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
    for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
        if (isMultipleOfNthPrimeFactor(candidate, n))
            return false;
    }
    return true;
}

private static boolean
```

```
    isMultipleOfNthPrimeFactor(int candidate, int n) {
        return
          candidate ==
          smallestOddNthMultipleNotLessThanCandidate(candidate, n)
    }

    private static int
    smallestOddNthMultipleNotLessThanCandidate(int candidate, int
    {
        int multiple = multiplesOfPrimeFactors.get(n);
        while (multiple < candidate)
            multiple += 2 * primes[n];
        multiplesOfPrimeFactors.set(n, multiple);
        return multiple;
    }
}
```

Before we dive into this code, I'd encourage everyone reading this article to take time to read over the code and draw your own conclusions about it. Did you find the code easy to understand? If so, why? If not, what makes it complex?

Also, Bob, can you confirm that you stand by this code (i.e., the code properly exemplifies the design philosophy of *Clean Code* and this is the way you believe the code should appear if it were used in production)?

UB:

Ah, yes. The `PrimeGenerator`. This code comes from the 1982 paper "Literate Programming"[2] written by Donald Knuth.

The program was originally written in Pascal, and was automatically generated by Knuth's WEB system into a single, very large method, which I translated into Java.

[2] https://www.cs.tufts.edu/~nr/cs257/archive/literate-programming/01-knuth-lp.pdf

Of course, this code was never meant for production. Both Knuth and I used it as a pedagogical example. In *Clean Code*, it appears in a chapter named "Classes." The lesson of the chapter is that a very large method will often contain many different sections of code that are better decomposed into independent classes.

In the chapter, I extracted three classes from that function: `PrimePrinter`, `RowColumnPagePrinter`, and `PrimeGenerator`.

One of those extracted classes was the `PrimeGenerator`. It had the following code (which I did not publish in the book). The variable names and the overall structure are Knuth's.

```
public class PrimeGenerator {
    protected static int[] generate(int n) {

        int[] p = new int[n];
        ArrayList<Integer> mult = new ArrayList<Integer>();
        p[0] = 2;
        mult.add(2);
        int k = 1;
        for (int j = 3; k < p.length; j += 2) {
            boolean jprime = false;
            int ord = mult.size();
            int square = p[ord] * p[ord];
            if (j == square) {
                mult.add(j);
```

```
            } else {
                jprime=true;
                for (int mi = 1; mi < ord; mi++) {
                    int m = mult.get(mi);
                    while (m < j)
                        m += 2 * p[mi];
                    mult.set(mi, m);
                    if (j == m) {
                        jprime = false;
                        break;
                    }
                }
            }
            if (jprime)
                p[k++] = j;
        }
        return p;

    }
}
```

Even though I was done with the lesson of the chapter, I didn’t want to leave that method looking so outdated. So I cleaned it up a bit as an afterthought. My goal was not to describe how to generate prime numbers. I wanted my readers to see how large methods, that violate the Single Responsibility Principle, can be broken down into a few smaller, well-named classes containing a few smaller, well-named methods.

JOHN:

Thanks for the background. Even though the details of that code weren't the main point of the chapter, presumably the code represents what you think is the "right" and "cleanest" way to do things, given the algorithm at hand. And that's where I disagree.

There are many design problems with `PrimeGenerator`, but for now I'll focus on method length. The code is chopped up so much (eight teeny-tiny methods) that it's difficult to read. For starters, consider the `isNotMultipleOfAnyPreviousPrimeFactor` method. This method invokes `isMultipleOfNthPrimeFactor`, which invokes `smallestOddNthMultipleNotLessThanCandidate`. These methods are shallow and entangled: In order to understand `isNot...`, you have to read the other two methods and load all of that code into your mind at once. For example, `isNot...` has side effects (it modifies `multiplesOfPrimeFactors`), but you can't see that unless you read all three methods.

UB:

I think you have a point. Eighteen years ago, when I was in the throes of this refactoring, the names and structure made perfect sense to me. They make sense to me now, too—but that's because I once again understand the algorithm. When I returned to the algorithm for the first time a few days ago, I struggled with the names and structure. Once I understood the algorithm, the names and structure made perfect sense.

JOHN:

Those names are problematic even for someone who understands the algorithm; we'll talk about them a bit later, when discussing comments. And if code no longer makes sense to the writer when the writer returns to the code later, that means the code is problematic. The fact that code can eventually be understood (with great pain and suffering) does not excuse its entanglement.

UB:

Would that we had such a crystal ball that we could help our future selves avoid such “*great pain and suffering*.” ;-)

JOHN:

There is no need for a crystal ball. The problems with `PrimeGenerator` are pretty obvious, such as the entanglement and interface complexity; maybe you were surprised that it is hard to understand, but I am not. Said another way, if you are unable to predict whether your code will be easy to understand, there are problems with your design methodology.

UB:

Fair enough. I will say, however, that I had equal “*pain and suffering*” interpreting your rewrite (below). So, apparently, neither of our methodologies were sufficient to rescue our readers from such struggles.

JOHN:

Going back to my introductory remarks about complexity, splitting up `isNot...` into three methods doesn’t reduce the amount of information you have to keep in your mind. It just spreads it out, so it isn’t as obvious that you need to read all three methods together. And it’s harder to see the overall structure of the code because it’s split up: Readers have to flip back and forth between the methods, effectively reconstructing a monolithic version in their minds. Because the pieces are all related, this code will be easiest to understand if it’s all together in one place.

UB:

I disagree. Here is `isNotMultipleOfAnyPreviousPrimeFactor`.

```
private static boolean
```

```
isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
   for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
        if (isMultipleOfNthPrimeFactor(candidate, n))
            return false;

    }
    return true;
}
```

If you trust the `isMultipleOfNthPrimeFactor` method, then this method stands alone quite nicely. I mean, we loop through all *N* previous primes and see if the candidate is a multiple. That's pretty straightforward.

Now, it would be fair to ask the question how we determine whether the candidate is a multiple, and in that case, you'd want to inspect the `isMultiple...` method.

JOHN:

This code does appear to be simple and obvious. Unfortunately, this appearance is deceiving. If a reader trusts the name `isMultipleOfNthPrimeFactor` (which suggests a predicate with no side effects) and doesn't bother to read its code, they will not realize that it has side effects, and that the side effects create a constraint on the candidate argument to `isNot...` (it must be monotonically non-decreasing from invocation to invocation). To understand these behaviors, you have to read both `isMultiple...` and `smallestOdd...`. The current decomposition hides this important information from the reader.

If there is one thing more likely to result in bugs than not understanding code, it's thinking you understand it when you don't.

UB:

That's a valid concern. However, it is tempered by the fact that the functions are presented in the order they are called. Thus, we can expect that the reader has already seen the main loop and understands that candidate increases by two each iteration.

The side effect buried down in `smallestOddNth...` is a bit more problematic. Now that you've pointed it out, I don't like it much. Still, that side effect should not confound the basic understanding of `isNot...`.

In general, if you trust the names of the methods being called, then understanding the caller does not require understanding the callee. For example:

```
for (Employee e : employees)
    if (e.shouldPayToday())
        e.pay();
```

This would not be made more understandable if we replaced those two method calls with their implementations. Such a replacement would simply obscure the intent.

JOHN:

This example works because the called methods are relatively independent of the parent. Unfortunately, that is not the case for `isNot...`.

In fact, `isNot...` is not only entangled with the methods it calls, it's also entangled with its callers. `isNot...` only works if it is invoked in a loop where `candidate` increases monotonically. To convince yourself that it works, you have to find the code that invokes `isNot...` and make sure that `candidate` never decreases from one call to the next.

Separating `isNot...` from the loop that invokes it makes it harder for readers to convince themselves that it works.

UB:

Which, as I said before, is why the methods are ordered the way they are. I expect that by the time you get to `isNot...`, you've already read `checkOddNumbersForSubsequentPrimes` and know that `candidate` increases by twos.

JOHN:

Let's discuss this briefly, because it's another area where I disagree with *Clean Code*. If methods are entangled, there is no clever ordering of the method definitions that will fix the problem.

In this particular situation, two other methods intervene between the loop in `checkOdd...` and `isNot...`, so readers will have forgotten the loop context before they get to `isNot...`. Furthermore, the actual code that creates a dependency on the loop isn't in `isNot...`: it's in `smallestOdd...`, which is even farther away from `checkOdd...`.

UB:

I sincerely doubt anyone is going to forget that `candidate` is being increased by twos. It's a pretty obvious way to avoid waste.

JOHN:

In my opening remarks, I talked about how it's important to reduce the amount of information people have to keep in their minds at once. In this situation, readers have to remember that loop while they read four intervening methods that are mostly unrelated to the loop. You apparently think this will be easy and natural (I disagree). But it's even worse than that. There is no indication which

parts of `checkOdd...` will be important later on, so the only safe approach is to remember *everything*, from *every* method, until you have encountered every other method that could possibly descend from it. And to make the connection between the pieces, readers must also reconstruct the call graph to notice that, even through four layers of method call, the code in `smallestOdd...` places constraints on the loop in `checkOdd...`. This is an unreasonable cognitive burden to place on readers.

If two pieces of code are tightly related, the solution is to bring them together. Separating the pieces, even in physically adjacent methods, makes the code harder to understand.

To me, all of the methods in `PrimeGenerator` are entangled: In order to understand the class, I had to load all of them into my mind at once. I was constantly flipping back and forth between the methods as I read the code. This is a red flag indicating that the code has been over-decomposed.

Bob, can you help me understand why you divided the code into such tiny methods? Is there some benefit to having so many methods that I have missed?

UB:

I think you and I are just going to disagree on this. In general, I believe in the principle of small, well-named methods and the separation of concerns. Generally speaking, if you can break a large method into several well-named smaller methods with different concerns, and by doing so expose their interfaces and the high-level functional decomposition, then that’s a good thing.

- Looping over the odd numbers is one concern.
- Determining primality is another.

- Marking off the multiples of primes is yet another.

It seems to me that separating and naming those concerns helps to expose the way the algorithm works—even at the expense of some entanglement.

In your solution, which we are soon to see below, you break the algorithm up in a similar way. However, instead of separating the concerns into functions, you separate them into sections with comments above them.

You mentioned that, in my solution, readers will have to keep the loop context in mind while reading the other functions. I suggest that, in your solution, readers will have to keep the loop context in mind while reading your explanatory comments. They may have to "flip back and forth" between the sections in order to establish their understanding.

Now, perhaps you are concerned that, in my solution, the "flipping" is a longer distance (in lines) than in yours. I'm not sure that's a significant point, since they all fit on the same screen (at least they do on my screen) and the landmarks are pretty obvious.

Method Length Summary

JOHN:

It sounds like it's time to wrap up this section. Is this a reasonable summary of where we agree and disagree?

- We agree that modular design is a good thing.
- We agree that it is possible to over-decompose, and that *Clean Code, First Edition* doesn't provide much guidance on how to recognize over-decomposition.

- We disagree on how far to decompose: You recommend decomposing code into much smaller units than I do. You believe that the additional decomposition you recommend makes code easier to understand; I believe that it goes too far and actually makes code more difficult to understand.
- You believe that the One Thing rule, applied with judgment, will lead to appropriate decompositions. I believe it lacks guardrails and will lead to over-decomposition.
- We agree that the internal decomposition of `PrimeGenerator` into methods is problematic. You point out that your main goal in writing `PrimeGenerator` was to show how to decompose into classes, not so much how to decompose a class internally into methods.
- Entanglement between methods in a class doesn't bother you as much as it bothers me. You believe that the benefits of decomposing methods can compensate for problems caused by entanglement. I believe they can't: When decomposed methods are entangled, they are harder to read than if they were not decomposed, and this defeats the whole purpose of decomposition.
- You believe that ordering the methods in a class can help to compensate for entanglement between methods; I don't.

UB:

I think this is a fair assessment of our agreements and disagreements. We both value decomposition, and we both avoid entanglement; but we disagree on the relative weighting of those two values.

Comments

JOHN:

Let’s move on to the second area of disagreement: comments. In my opinion, the *Clean Code* approach to commenting results in code with inadequate documentation, which increases the cost of software development. I’m sure you disagree, so let’s discuss.

Here is what *Clean Code* says about comments (page 54):

“The proper use of comments is to compensate for our failure to express ourselves in code. Note that I use the word *failure*. I meant it. Comments are always failures. We must have them because we cannot always figure out how to express ourselves without them, but their use is not a cause for celebration… Every time you write a comment you should grimace and feel the failure of your ability of expression.”

I have to be honest: I was horrified when I first read this text, and it still makes me cringe. This stigmatizes writing comments. Junior developers will think “if I write comments, people may think I’ve failed, so the safest thing is to write no comments.”

UB:

That chapter begins with these words:

“Nothing can be quite so helpful as a well-placed comment.”

It goes on to say that comments are a *necessary* evil.

The only way a reader could infer that they should write no comments is if they hadn’t actually read the chapter. The chapter walks through a series of comments, some bad, some good.

JOHN:

Clean Code focuses a lot more on the “evil” aspects of comments than the “necessary” aspects. The sentence you quoted above is followed by two sentences criticizing comments. Chapter 4 spends 4 pages talking about good comments, followed by 15 pages talking about bad comments. There are snubs like “the only truly good comment is the comment you found a way not to write.” And “Comments are always failures” is so catchy that it’s the one thing readers are most likely to remember from the chapter.

UB:

The difference in page count is because there are just a few ways to write good comments and so many more ways to write bad ones.

JOHN:

I disagree; this illustrates your bias against comments. If you look at Chapter 13 of APOSD, it finds a lot more constructive ways to use comments than *Clean Code*. And if you compare the tone of Chapter 13 of APOSD with Chapter 4 of *Clean Code*, the hostility of *Clean Code* towards comments becomes pretty clear.

UB:

I’ll leave you to balance that last comment with the initial statement, and the final example, in the “Comments” chapter. They do not communicate “hostility.”

I’m not hostile to comments in general. I *am* very hostile to gratuitous comments.

You and I likely both survived through a time when comments were absolutely necessary. In the ’70s and ’80s, I

was an assembly language programmer. I also wrote a bit of FORTRAN. Programs in those languages that had no comments were impenetrable.

As a result, it became conventional wisdom to write comments by default. And indeed, computer science students were taught to write comments uncritically. Comments became *pure good*.

In *Clean Code*, I decided to fight that mindset. Comments can be *really bad* as well as good.

JOHN:

I don't agree that comments are less necessary today than they were 40 years ago. Comments are crucially important and add enormous value to software. The problem is that there is a lot of important information that simply cannot be expressed in code. By adding comments to fill in this missing information, developers can make code dramatically easier to read. This is not a "failure of their ability to express themselves," as you put it.

UB:

It's very true that there is important information that is not, or cannot be, expressed in code. That's a failure. A failure of our languages, or of our ability to use them to express ourselves. In every case, a comment is a failure of our ability to use our languages to express our intent.

And we fail at that very frequently, and so comments are a necessary evil—or, if you prefer, *an unfortunate necessity*. If we had the perfect programming language,™ we would never write another comment.

JOHN:

I don’t agree that a perfect programming language would eliminate the need for comments. Comments and code serve very different purposes, so it’s not obvious to me that we should use the same language for both. In my experience, English works quite well as a language for comments. Why do you feel that information about a program should be expressed entirely in code, rather than using a combination of code and English?

UB:

I bemoan the fact that we must sometimes use a human language instead of a programming language. Human languages are imprecise and full of ambiguities. Using a human language to describe something as precise as a program is very hard, and fraught with many opportunities for error and inadvertent misinformation.

JOHN:

I agree that English isn’t always as precise as code, but it can still be used in precise ways, and comments typically don’t need the same degree of precision as code. Comments often contain qualitative information such as *why* something is being done, or the overall idea of something. English works better for these than code because it is a more expressive language.

UB:

I have no argument with that statement.

JOHN:

Are you concerned that comments will be incorrect or misleading and that this will slow down software development? I often hear people complain about stale comments (usually as an excuse for writing no comments at all), but I have not found them to be a significant problem

over my career. Incorrect comments do happen, but I don't encounter them very often, and when I do, they rarely cost me much time. In contrast, I waste *enormous* amounts of time because of inadequate documentation; it's not unusual for me to spend 50% to 80% of my development time wading through code to figure out things that would be obvious if the code was properly commented.

UB:

You and I have had some very different experiences. I have certainly been helped by well-placed comments. I have also, just as certainly (and within this very document), been distracted and confused by a comment that was incorrect, misplaced, gratuitous, or otherwise just plain bad.

JOHN:

I invite everyone reading this to ask yourself the following questions.

- How much does your software development speed suffer because of incorrect comments?
- How much does your software development speed suffer because of missing comments?

For me, the cost of missing comments is easily 10 to 100 times the cost of incorrect comments. That is why I cringe when I see things in *Clean Code* that discourage people from writing comments. Let's consider the `PrimeGenerator` class. There is not a single comment in that code; does this seem appropriate to you?

UB:

I think it was appropriate for the purpose for which I wrote it. It was an adjunct to the lesson that very large methods can be broken down into smaller classes containing smaller

methods. Adding lots of explanatory comments would have detracted from that point.

In general, however, the commenting style I used in Listing 4-8 is more appropriate. That listing, at the very end of the "Comments" chapter, describes yet another `PrimeGenerator` with a slightly different algorithm and a better set of comments.

JOHN:

I disagree that adding comments would have distracted from your point, and I think Listing 4-8 is also woefully under-commented. But let's not argue about either of those issues. Instead, let's discuss what comments the `PrimeGenerator` code *should* have if it were used in production. I will make some suggestions, and you can agree or disagree.

For starters, let's discuss your use of megasyllabic names like `isLeastRelevantMultipleOfLargerPrimeFactor`. My understanding is that you advocate using names like this instead of using shorter names augmented with descriptive comments: You're effectively moving the comments into code. To me, this approach is problematic.

- Long names are awkward. Developers effectively have to retype the documentation for a method every time they invoke it, and the long names waste horizontal space and trigger line wraps in the code. The names are also awkward to read: My mind wants to parse every syllable every time I read it, which slows me down. Notice that both you and I resorted to abbreviating names in this discussion: That's an indication that the long names are awkward and unhelpful.

- The names are hard to parse and don't convey information as effectively as a comment. When students read `PrimeGenerator`, one of the first things they complain about is the long names (students can't make sense of them). For example, the name above is vague and cryptic: What does "least relevant" mean, and what is a "larger prime factor"? Even with a complete understanding of the code in the method, it's hard for me to make sense of the name. If this name is going to eliminate the need for a comment, it needs to be even longer.

In my opinion, the traditional approach of using shorter names with descriptive comments is more convenient and conveys the required information more effectively. What advantage is there in the approach you advocate?

UB:

"Megasyllabic": great word!

I like my method names to be sentence fragments that fit nicely with keywords and assignment statements. It makes the code a bit more natural to read.

```
if (isTooHot)
    cooler.turnOn();
```

I also follow a simple rule about the length of names. The larger the scope of a method, the shorter its name should be, and vice versa: The shorter the scope, the longer the name. The private methods I extracted in this case live in very small scopes, and so have longish names. Methods like this are typically called from only one place, so there is no burden on the programmer to remember a long name for another call.

JOHN:

Names like `isTooHot` are totally fine by me. My concern is about names like `isLeastRelevantMultipleOfLargerPrimeFactor`.

It's interesting that, as methods get smaller and narrower, you recommend longer names. What this says to me is that the interfaces for those functions are more complex, so it takes more words to describe them. This provides supporting evidence for my assertion a while back that the more you split up a method, the shallower the resulting methods will be.

UB:

It's not the functions that get smaller. It's the scope that gets smaller. A private function has a smaller scope than the public function that calls it. A function called by that private function has an even smaller scope. As we descend in scope, we also descend in situational detail. Describing such detail often requires a long name, or a long comment. I prefer to use a name.

As for long names being hard to parse, that's a matter of practice. Code is full of things that take practice to get used to.

JOHN:

I don't accept this. Code may be full of things that take practice to get used to, but that doesn't excuse it. Approaches that require more practice are worse than those that require less. If it's going to take a lot of work to get comfortable with the long names, then there had better be some compensating benefit; so far, I'm not seeing any. And I don't see any reason to believe that practice will make those names easier to digest.

In addition, your comment above violates one of my fundamental rules, which is "complexity is in the eye of the

reader.” If you write code that someone else thinks is complicated, then you must accept that the code is probably complicated (unless you think the reader is completely incompetent). It is not OK to make excuses or suggest that it is really the reader’s problem (“you just don’t have enough practice”). I’m going to have to live by this same rule a bit later in our discussion.

UB:

Fair enough. As for the meaning of `leastRelevant`, that’s a much larger problem that you and I will encounter shortly. It has to do with the intimacy that the author has with the solution, and the reader’s lack of that intimacy.

JOHN:

You still haven’t answered my question: Why is it better to use super-long names rather than shorter names augmented with descriptive comments?

UB:

It’s a matter of preference for me. I prefer long names to comments. I don’t trust comments to be maintained, nor do I trust that they will be read. Have you ever noticed that many IDEs paint comments in light gray so that they can be easily ignored? It’s harder to ignore a name than a comment.

(BTW, I have my IDE paint comments in bright fire-engine red.)

JOHN:

I don’t see why a monster name is more likely to be “maintained” than a comment, and I don’t agree that IDEs encourage people to ignore comments (this is your bias coming out again). My current IDE (VSCode) doesn’t use a

lighter color for comments. My previous one (NetBeans) did, but the color scheme didn't hide the comments; it distinguished them from the code in a way that made both code and comments easier to read.

Now that we've discussed the specific issue of comments versus long method names, let's talk about comments in general. I think there are two major reasons why comments are needed. The first reason for comments is abstraction. Simply put, without comments, there is no way to have abstraction or modularity.

Abstraction is one of the most important components of good software design. I define an abstraction as "a simplified way of thinking about something that omits unimportant details." The most obvious example of an abstraction is a method. It should be possible to use a method without reading its code. The way we achieve this is by writing a header comment that describes the method's interface (all the information someone needs in order to invoke the method). If the method is well designed, the interface will be much simpler than the code of the method (it omits implementation details), so the comments reduce the amount of information people must have in their heads.

UB:

Long ago, in a 1995 book, I defined abstraction as "The amplification of the essential and the elimination of the irrelevant."

I certainly agree that abstraction is of importance to good software design. I also agree that well-placed comments can enhance the ability of readers to understand the abstractions we are attempting to employ. I disagree that comments are the *only*, or even the *best*, way to understand those abstractions. But sometimes they are the only option.

But consider:

```
addSongToLibrary(String title, String[] authors, int durationInSec
```

This seems like a very nice abstraction to me, and I cannot imagine how a comment might improve it.

JOHN:

Our definitions of abstraction are very similar; that's good to see. However, the `addSongToLibrary` declaration is not (yet) a good abstraction, because it omits information that is essential. In order to use `addSongToLibrary`, developers need answers to the following questions.

- Is there any expected format for an author string, such as "LastName, FirstName"?
- Are the authors expected to be in alphabetical order? If not, is the order significant in some other way?
- What happens if there is already a song in the library with the given title but different authors? Is it replaced with the new one, or will the library keep multiple songs with the same title?
- How is the library stored (e.g., is it entirely in memory? saved on disk?)? If this information is documented somewhere else, such as the overall class documentation, then it need not be repeated here.

Thus, `addSongToLibrary` needs quite a few comments. Sometimes the signature of a method (names and types of the method, its arguments, and its return value) contains all the information needed to use it, but this is pretty rare. Just skim through the documentation for your favorite library package: In how many cases could you understand how to use a method with only its signature?

UB:

Yes, there are times when the signature of a method is an incomplete abstraction and a comment is required. This is especially true when the interface is part of a public API, or an API intended for use by a separate team of developers. Within a single development team, however, long descriptive comments on interfaces are often more of an impediment than a help. The team has intimate knowledge of the internals of the system, and will generally be able to understand an interface simply from its signature.

JOHN:

In one of our in-person discussions, you argued that interface comments are unnecessary because, when a group of developers is working on a body of code, they can collectively keep the entire code “loaded” in their minds, so comments are unnecessary: If you have a question, just ask the person who is familiar with that code. This creates a huge cognitive load to keep all that code mentally loaded, and it’s hard for me to imagine that it would actually work. Maybe your memory is better than mine, but I find that I quickly forget code that I wrote just a few weeks ago. In a project of any size, I think your approach would result in developers spending large amounts of time reading code to re-derive the interfaces, and probably making mistakes along the way. Spending a few minutes to document the interfaces would save time, reduce cognitive load, and reduce bugs.

UB:

I think that certain interfaces need comments, even if they are private to the team. But I think it is more often the case that the team is familiar enough with the system that well-named methods and arguments are sufficient.

JOHN:

Let's consider a specific example from `PrimeGenerator`: the `isMultipleOfNthPrimeFactor` method. When someone reading the code encounters the call to `isMultiple...` in `isNot...`, they need to understand enough about how `isMultiple...` works in order to see how it fits into the code of `isNot...`. The method name does not fully document the interface, so if there is no header comment, then readers will have to read the code of `isMultiple`. This will force readers to load more information into their heads, which makes it harder to work in the code.

Here is my first attempt at a header comment for `isMultiple`:

```
/**
 * Returns true if candidate is a multiple of primes[n], false oth
 * May modify multiplesOfPrimeFactors[n].
 * @param candidate
 *      Number being tested for primality; must be at least as
 *      large as any value passed to this method in the past.
 * @param n
 *      Selects a prime number to test against; must be
 *      <= multiplesOfPrimeFactors.size().
 */
```

What do you think of this?

UB:

I think it's accurate. I wouldn't delete it if I encountered it. I don't think it should be a Javadoc.

The first sentence is redundant with the name `isMultipleOfNthPrimeFactor` and so could be deleted. The warning of the side effect is useful.

JOHN:

I agree that the first sentence is largely redundant with the name, and I debated with myself about whether to keep it. I decided to keep it because I think it is a bit more precise than the name; it's also easier to read. You propose to eliminate the redundancy between the comment and the method name by dropping the comment; I would eliminate the redundancy by shortening the method name.

By the way, you complained earlier about comments being less precise than code, but in this case, the comment is *more* precise (the method name can't include text like `primes[n]`).

UB:

Fair enough. There are times when precision is better expressed in a comment.

Continuing with my critique of your comment above: The name `candidate` is synonymous with "Number being tested for primality."

In the end, however, all the words in a comment are just going to have to sit in my brain until I understand why they are there. I'm also going to have to worry if they are accurate. So I'm going to have to read the code to understand and validate the comment.

JOHN:

Whoa. That loud sound you just heard was my jaw hitting the floor. Help me understand this a bit better: Approximately what fraction of comments that you encounter in practice are you willing to trust without reading the code to verify them?

UB:

I look at every comment as potential misinformation. At best, they are a way to crosscheck the author's intent against the code. The amount of credence I give to a comment depends a lot on how easy they make that crosscheck. When I read a comment that does not cause me to crosscheck, then I consider it to be of no value. When I see a comment that causes me to crosscheck, and when that crosscheck turns out to be valuable, then that's a really good comment.

Another way to say this is that the best comments tell me something surprising and verifiable about the code. The worst are those that waste my time telling me something obvious or incorrect.

JOHN:

It sounds like your answer is 0%: You don't trust any comment unless it has been verified against the code. This makes no sense to me. As I said above, the vast majority of comments are correct. It's not hard to write comments; the students in my software design class are doing this pretty well within a few weeks. It's also not hard to keep comments up-to-date as code evolves. Your refusal to trust comments is another sign of your irrational bias against comments.

Refusing to trust comments incurs a very high cost. In order to understand how to invoke a method, you will have to read all of the code of that method; if the method invokes other methods, you will also have to read them, and the methods they invoke, recursively. This is an enormous amount of work in comparison to reading (and trusting) a simple interface comment like the one I wrote above.

If you choose not to write an interface comment for methods, then you leave the interface of that method undefined. Even if someone reads the code of the method, they won't be able to tell which parts of the implementation

are expected to remain the same and which parts may change (there is no way to specify this “contract” in code). This will result in misunderstanding and more bugs.

UB:

Well, I guess I’ve just been burned more than you have. I’ve gone down too many false comment–induced rabbit holes and wasted too much time on worthless word salads.

Of course, my trust in comments is not a binary thing. I read them if they are there; but I don’t implicitly trust them. The more gratuitous I feel the author was, or the less adept at English the author is, the less I trust the comments.

As I said above, our IDEs tend to paint comments in an ignorable color. I have my IDE paint comments in bright fire-engine red because, when I write a comment, I intend for it to be read.

By the same token, I use long names as a substitute for comments because I intend for those long names to be read; and it is very hard for a programmer to ignore names.

JOHN:

I mentioned earlier that there are two general reasons why comments are needed. So far, we’ve been discussing the first reason (abstraction). The second general reason for comments is for important information that is not obvious from the code. The algorithm in `PrimeGenerator` is very nonobvious, so quite a few comments are needed to help readers understand what is going on and why. Most of the algorithm’s complexity arises because it is designed to compute primes efficiently.

- The algorithm goes out of its way to avoid divisions, which were quite expensive when Knuth wrote his original version (they aren’t that expensive nowadays).

- The first multiple for each new prime number is computed by squaring the prime, rather than multiplying it by 3. This is mysterious: Why is it safe to skip the intervening odd multiples? Furthermore, it might seem that this optimization only has a small impact on performance, but in fact, it makes an *enormous* difference (orders of magnitude). Using the square has the side-effect that when testing a candidate, only primes up to the square root of the candidate are tested. If 3x were used as the initial multiple, primes within a factor of 3 of the candidate would be tested; that's *a lot* more tests. This implication of using the square is so nonobvious that I only realized it while preparing material for this discussion; it never occurred to me in the many times I have discussed the code with students.

Neither of these issues is obvious from the code; without comments, readers are left to figure them out on their own. The students in my class are generally unable to figure out either of them in the 30 minutes I give them, but I think that comments would have allowed them to understand in a few minutes. Going back to my introductory remarks, this is an example where information is important, so it needs to be made available.

Do you agree that there should be comments to explain each of these two issues?

UB:

I agree that the algorithm is subtle. Setting the first prime multiple as the square of the prime was deeply mysterious at first. I had to go on an hourlong bike ride to understand it.

Would a comment help? Perhaps. However, my guess is that no one who has been reading our conversation has been helped by it, because you and I are now too intimate with

the solution. You and I can talk about that solution using words that fit into that intimacy; but our readers likely do not yet enjoy that fit.

One solution is to paint a picture—being worth a thousand words. Here’s my attempt.

```
                                                                 X
                                                     1111111111111111111111111
        11111222223333344444555556666677777888889999900000111112222233333444444
    35791357913579135791357913579135791357913579135791357913579135791357913579
    !!! !! !! !  !!  ! !! !  !  !!  ! !!  ! !  !   ! !! !! !
  3 |||-||-||-||-||-||-||-||-||-||-||-||-||-||-||-||-||-||-||-||-
  5 |||||||||||-||||-||||-||||-||||-||||-||||-||||-||||-||||-||||-
  7 |||||||||||||||||||||||-||||||-||||||-||||||-||||||-||||||-||||||-
 11 |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||-||||||||||-
 13 ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
...
113||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

I expect that our readers will have to stare at this for some time, and also look at the code. But then there will be a *click* in their brains and they’ll say “Ohhh! Yes! I see it now!”

JOHN:

I found this diagram very hard to understand. It begs for supplemental English text to explain the ideas being presented. Even the syntax is nonobvious: What does `1111111111111111111111111` mean?

Maybe we have a fundamental difference of philosophy here. I get the sense that you are happy to give readers a few clues and leave it to them to put the clues together. Perhaps you don’t mind if people have to stare at something for a while to figure it out. I don’t agree with this approach: It results in wasted time, misunderstandings, and bugs. I think software should be totally *obvious*, where readers don’t need to be clever or “stare at this for some time” to figure things out. Suffering followed by catharsis is great for

Greek tragedies, but not for reading code. Every question a reader might have should be naturally answered, either in the code or in comments. Key ideas and important conclusions should be stated explicitly, not left for the reader to deduce. Ideally, even if a reader is in a hurry and doesn't read the code very carefully, their first guesses about how things work (and why) should be correct. To me, that's clean code.

UB:

I don't disagree with your sentiment. Good, clean code should be as easy as possible to understand. I want to give my readers as many clues as possible so that the code is intuitive to read.

That's the goal. As we are about to see, that can be a tough goal to achieve.

JOHN:

In that case, do you still stand by the "picture" you painted above? It doesn't seem consistent with what you just said. And if you really wanted to give your readers as many clues as possible, you'd include a lot more comments.

UB:

I stand by the picture as far as its accuracy is concerned. And I think it makes a good crosscheck. I have no illusions that it is easy to understand.

This algorithm is challenging and will require work to comprehend. I finally understood it when I drew this picture in my mind while on that bike ride. When I got home, I drew it for real and presented it in hopes that it might help someone willing to do the work to understand it.

Comments Summary

JOHN:

Let’s wrap up this section of the discussion. Here is my summary of where we agree and disagree.

- Our overall views of comments are fundamentally different. I see more value in comments than you do, and I believe that they play a fundamental and irreplaceable role in system design. You agree that there are places where comments are necessary, but that comments don’t always make it easier to understand code, so you see far fewer places where comments are needed.
- I would probably write 5 to 10 times more lines of comments for a given piece of code than you would.
- I believe that missing comments are a much greater cause of lost productivity than erroneous or unhelpful comments; you believe that comments are a net negative, as generally practiced: Bad comments cost more time than good comments save.
- You view it as problematic that comments are written in English rather than a programming language. I don’t see this as particularly problematic and think that, in many cases, English works better.
- You recommend that developers should take information that I would represent as comments and recast it into code if at all possible. One example of this is super-long method names. I believe that super-long names are awkward and hard to understand, and that it would be better to use shorter names supplemented with comments.

- I believe that it is not possible to define interfaces and create abstractions without a lot of comments. You agree for public APIs, but see little need to comment interfaces that are internal to the team.
- You are unwilling to trust comments until you have read code to verify them. I generally trust comments; by doing so, I don't need to read as much code as you do. You think this exposes me to too much risk.
- We agree that implementation code only needs comments when the code is nonobvious. Although neither of us argues for a large number of implementation comments, I'm more likely to see value in them than you do.

Overall, we struggled to find areas of agreement on this topic.

UB:

This is a fair assessment of our individual positions; which I assume are based on our different individual experiences. Over the years, I have found the vast majority of comments, as generally practiced in the industry, to be unhelpful. You seem to have found more help in the comments you have encountered.

John's Rewrite of `PrimeGenerator`

JOHN:

I mentioned that I ask the students in my software design class to rewrite `PrimeGenerator` to fix all of its design problems. Here is my rewrite (note: This was written before we began our discussion; given what I have learned during the discussion, I would now change several of the comments, but I have left this in its original form):

```
package literatePrimes;
import java.util.ArrayList;
public class PrimeGenerator2 {

    /**
     * Computes the first prime numbers; the return value contains
     * computed primes, in increasing order of size.
     * @param n
     *      How many prime numbers to compute.
     */
    public static int[] generate(int n) {
        int[] primes = new int[n];

        // Used to test efficiently (without division) whether a c
        // is a multiple of a previously-encountered prime number.
        // Each entry here contains an odd multiple of the corresp
        // entry in primes. Entries increase monotonically.
        int[] multiples = new int[n];

        // Index of the last value in multiples that we need to co
        // when testing candidates (all elements after this are gr
        // than our current candidate, so they don't need to be
        // considered).
        int lastMultiple = 0;

        // Number of valid entries in primes.
        int primesFound = 1;
```

```
primes[0] = 2;
multiples[0] = 4;

// Each iteration through this loop considers one candidat
// the even numbers, since they can't be prime.
candidates:
for (int candidate = 3; primesFound < n; candidate += 2) {
    if (candidate >= multiples[lastMultiple]) {
        lastMultiple++;
    }

    // Each iteration of this loop tests the candidate aga
    // potential prime factor. Skip the first factor (2) s
    // only consider odd candidates.
    for (int i = 1; i <= lastMultiple; i++) {
        while (multiples[i] < candidate) {
            multiples[i] += 2*primes[i];
        }
        if (multiples[i] == candidate) {
            continue candidates;
        }
    }
    primes[primesFound] = candidate;

    // Start with the prime's square here, rather than 3x
    // prime. This saves time and is safe because all of t
    // intervening multiples will be detected by smaller p
    // numbers. As an example, consider the prime 7: the v
```

```
            // in multiples will start at 49; 21 will be ruled out
            // multiple of 3, and 35 will be ruled out as a multip
            // so 49 is the first multiple that won't be ruled out
            // smaller prime.
            multiples[primesFound] = candidate*candidate;
            primesFound++;
        }
        return primes;
    }
}
```

Everyone can read this and decide for themselves whether they think it is easier to understand than the original. I'd like to mention a couple of overall things.

- There is only one method. I didn't subdivide it, because I felt the method already divides naturally into pieces that are distinct and understandable. It didn't seem to me that pulling out methods would improve readability significantly. When students rewrite the code, they typically have two or three methods, and those are usually OK too.
- There are *a lot* of comments. It's extremely rare for me to write code with this density of comments. Most methods I write have no comments in the body, just a header comment describing the interface. But this code is subtle and tricky, so it needs a lot of comments to make the subtleties clear to readers. The long length of some of the comments is a red flag indicating that I struggled to find a clear and simple explanation for the code. Even with all the additional explanatory material,

this version is a bit shorter than the original (65 lines versus 70).

UB:

I presume this is a complete rewrite. My guess is that you worked to understand the algorithm from *Clean Code* and then wrote this from scratch. If that's so, then fair enough.

In *Clean Code*, I *refactored* Knuth's algorithm in order to give it a little structure. That's not the same as a complete rewrite.

Having said that, your version is much better than either Knuth's or mine.

I wrote that chapter 18 years ago, so it's been a long time since I saw and understood this algorithm. When I first saw your challenge, I thought: "Oh, I can figure out my own code!" But no. I could see all the moving parts, but I could not figure out why those moving parts generated a list of prime numbers.

So then, I looked at your code. I had the same problem. I could see all the moving parts, all with comments, but I still could not figure out why those moving parts generated a list of prime numbers.

Figuring that out required a lot of staring at the ceiling, closing my eyes, visualizing, and riding my bike.

Among the problems I had were the comments you wrote. Let's take them one at a time.

```
/**
 * Computes the first prime numbers; the return value contains the
 * computed primes, in increasing order of size.
 * @param n
 *      How many prime numbers to compute
```

```
        How many prime numbers to compute.
 */
public static int[] generate(int n) {
```

It seems to me that this would be better as:

```
public static int[] generateNPrimeNumbers(int n) {
```

or, if you must:

```
//Return the first n prime numbers
public static int[] generate(int n) {
```

I’m not opposed to Javadocs as a rule; but I write them only when absolutely necessary. I also have an aversion for descriptions and `@param` statements that are perfectly obvious from the method signature.

The next comment cost me a good 20 minutes of puzzling things out.

```
// Used to test efficiently (without division) whether a candidate
// is a multiple of a previously-encountered prime number. Each en
// here contains an odd multiple of the corresponding entry in
// primes. Entries increase monotonically.
```

First of all, I’m not sure why the “division” statement is necessary. I’m old-school, so I expect that everyone knows to avoid division in inner loops if it can be avoided. But maybe I’m wrong about that…

Also, the *Sieve of Eratosthenes* does not do division, and is a lot easier to understand *and explain* than this algorithm. So why this particular algorithm? I think Knuth was trying to

save *memory*—and in 1982, saving memory was important. This algorithm uses a lot less memory than the sieve.

Then came the phrase: `Each entry here contains an odd multiple...`. I looked at that, and then at the code, and I saw `multiples[0] = 4;`.

“That’s not odd,” I said to myself. “So maybe he meant even.”

So then, I looked down and saw `multiples[i] += 2*primes[i];`.

“That’s adding an even number!” I said to myself. “I’m pretty sure he meant to say ‘even’ instead of ‘odd’.”

I hadn’t yet worked out what the `multiples` array was. So I thought it was perfectly reasonable that it would have even numbers in it, and that your comment was simply an understandable word transposition. After all, there’s no compiler for comments, so they suffer from the kinds of mistakes that humans often make with words.

It was only when I got to `multiples[primesFound] = candidate*candidate;` that I started to question things. If the `candidate` is prime, shouldn’t `prime*prime` be odd in every case beyond 2? I had to do the math in my head to prove that: (2n + 1)(2n + 1) = 4n^2 + 4n + 1 … Yeah, that’s odd.

OK, so the `multiples` array is full of odd multiples, except for the first element, since it will be multiples of 2.

So perhaps that comment should be:

```
// multiples of corresponding prime.
```

Or perhaps we should change the name of the array to something like `primeMultiples` and drop the comment altogether.

Moving on to the next comment:

```
// Each iteration of this loop tests the candidate against one
// potential prime factor. Skip the first factor (2) since we
// only consider odd candidates.
```

That doesn't make a lot of sense. The code it's talking about is:

```
for (int i = 1; i <= lastMultiple; i++) {
    while (multiples[i] < candidate) {
```

The `multiples` array, as we have now learned, is an array of *multiples* of prime numbers. This loop is not testing the candidate against prime *factors*; it's testing it against the current prime *multiples*.

Fortunately for me, the third or fourth time I read this comment I realized that you really meant to use the word *multiples*. But the only way for me to know that was to understand the algorithm. And when I understand the algorithm, why do I need the comment?

That left me with one final question: what the deuce was the reason behind:

```
multiples[primesFound] = candidate*candidate;
```

Why the square? That makes no sense. So I changed it to:

```
multiples[primesFound] = candidate;
```

And it worked just fine. So this must be an optimization of some kind.

Your comment to explain this is:

```
// Start with the prime's square here, rather than 3x the prime.
// This saves time and is safe because all of the intervening
```

```
// multiples will be detected by smaller prime numbers. As an
// example, consider the prime 7: the value in multiples will
// start at 49; 21 will be ruled out as a multiple of 3, and
// 35 will be ruled out as a multiple of 5, so 49 is the first
// multiple that won't be ruled out by a smaller prime.
```

The first few times I read this it made no sense to me at all. It was just a jumble of numbers.

I stared at the ceiling, and closed my eyes to visualize. I couldn’t see it. So I went on a long, contemplative bike ride, during which I realized that the prime multiples of 2 will at one point contain 2*3 and then 2*5. So the `multiples` array will at some point contain multiples of primes *larger* than the prime they represent. *And it clicked!*

Suddenly it all made sense. I realized that the `multiples` array was the equivalent of the array of booleans we use in the *Sieve of Eratosthenes*—but with a really interesting twist. If you were to do the sieve on a whiteboard, you *could* erase every number less than the candidate, and only cross out the numbers that were the next multiples of all the previous primes.

That explanation makes perfect sense to me now, but I’d be willing to bet that those who are reading it are puzzling over it. The idea is just hard to explain.

Finally, I went back to your comment and could see what you were saying.

A Tale of Two Programmers

UB:

The bottom line here is that you and I both fell into the same trap. I refactored that old algorithm 18 years ago, and I thought all those method and variable names would make my intent clear—*because I understood that algorithm.*

You wrote that code awhile back and decorated it with comments that you thought would explain your intent—*because you understood that algorithm.*

But my names didn't help me 18 years later. They didn't help you or your students either. And your comments didn't help me.

We were inside the box trying to communicate to those who stood outside and could not see what we saw.

The bottom line is that it is very difficult to explain something to someone who is not intimate with the details you are trying to explain. Often, our explanations make sense only after the reader has worked out the details for themself.

JOHN:

There's a lot of stuff in your discussion above, but I think it all boils down to one thing: You don't like the comments that I wrote. As I mentioned earlier, complexity is in the eye of the reader: If you say that my comments were confusing or didn't help you to understand the code, then I have to take that seriously.

At the same time, you have made it clear that you don't see much value in comments in general. Your preference is to have essentially no comments for this code (or any code). You argue above that there is simply nothing that comments can do to make the code easier to understand; the only way to understand the code is to read the code. That is a cop-out.

UB:

Sorry to interrupt you, but I think you are overstating my position. I certainly never said that comments can never be helpful. Sometimes, of course, they are. What I said was that I only trust them if the code validates them. Sometimes a comment will make that validation a lot easier.

JOHN:

You keep saying that you sometimes find use for comments, but the reality is that “sometimes” almost never occurs in your code. We’ll see this when we look at your revision of my code.

Now back to my point. In order to write our various versions of the code, you and I had to accumulate a lot of knowledge about the algorithm, such as why it’s OK for the first multiple of a prime to be its square. Unfortunately, not all of that knowledge can be represented in the code. It is our professional responsibility to do the best we can to convey that knowledge in comments so that readers do not have to reconstruct it over and over. Even if the resulting comments are imperfect, they will make the code easier to understand.

If a situation like this occurred in real life, I would work with you and others to improve my comments. For example, I would ask you questions to get a better sense of why the “squared prime” comment didn’t seem to help you.

- Are there things in the comment that are misleading or confusing?
- Is there some important piece of information you acquired on your bike ride that suddenly made things clear?

I would also show the comment to a few other people to get their takes on it. Then, I would rework the comment to

improve it.

Given your fundamental disbelief in comments, I think it's likely that you would still see no value in the comment, even after my reworking. In this case, I would show the comment to other people, particularly those who have a more positive view of comments in general, and get their input. As long as the comment is not misleading and at least a few people found it helpful, I would retain it.

Now let me discuss two few specific comments that you objected to. The first comment was the one for the `multiples` variable:

```
// Used to test efficiently (without division) whether a candidate
// is a multiple of a previously-encountered prime number. Each en
// here contains an odd multiple of the corresponding entry in
// primes. Entries increase monotonically.
```

There is a bug in this comment that you exposed (the first entry is not odd); good catch! You then argued that most of the information in the comment is unnecessary and proposed this as an alternative:

```
// multiples of corresponding prime.
```

You have left out too much useful information here. For example, I don't think it is safe to assume that readers will figure out that the motivation is avoiding divisions. It's always better to state these assumptions and motivations clearly so that there will be no confusion. And I think it's helpful for readers to know that these entries never decrease. I would simply fix the bug, leaving all of the information intact:

```
// Used to test efficiently (without division) whether a candidate
// is a multiple of a previously-encountered prime number. Each en
// (except the first, which is never used) contains an odd multipl
// the corresponding entry in primes. Entries increase monotonical
```

The second comment was this one, for the `for` loop:

```
// Each iteration of this loop tests the candidate against one
// potential prime factor. Skip the first factor (2) since we
// only consider odd candidates.
```

You objected to this comment because the code of the loop doesn't actually test the candidate against the prime factor; it tests it against a multiple. When I write implementation comments like this, my goal is not to restate the code; comments like that don't usually provide much value. The goal here was to say *what* the code is doing in a logical sense, not *how* it does it. In that sense, the comment is correct.

However, if a comment causes confusion in the reader, then it is not a good comment. Thus, I would rewrite this comment to make it clear that it describes the abstract function of the code, not its precise behavior:

```
// Each iteration of this loop considers one existing prime, rulin
// out the candidate if it is a multiple of that prime. Skip the
// first prime (2) since we only consider odd candidates.
```

To conclude, I agree with your assertion "it is very difficult to explain something to someone who is not intimate with the details you are trying to explain." And yet, it is our responsibility as programmers to do exactly that.

UB:

I'm glad we agree. We also agree about getting others to review the code and make recommendations on the code *and* the comments.

Bob's Rewrite of `PrimeGenerator2`

UB:

When I saw your solution, and after I gained a good understanding of it, I refactored it just a bit. I loaded it into my IDE, wrote some simple tests, and extracted a few simple methods.

I also got rid of that *awful* labeled `continue` statement. And I added 3 to the primes list so that I could mark the first element as *irrelevant* and give it a value of −1. (I think I was still reeling from the even/odd confusion.)

I like this because the implementation of the `generateFirstNPrimes` method describes the moving parts in a way that hints at what is going on. It's easy to read that implementation and get a glimpse of the mechanism. I'm not at all sure that the comment helps.

I think it is just the reality of this algorithm that the effort required to properly explain it, and the effort required for anyone else to read and understand that explanation, is roughly equivalent to the effort needed to read the code and go on a bike ride.

```
package literatePrimes;


public class PrimeGenerator3 {

    private static int[] primes;
```

```
private static int[] primeMultiples;
private static int lastRelevantMultiple;
private static int primesFound;
private static int candidate;

// Lovely little algorithm that finds primes by predicting
// the next composite number and skipping over it. That predic
// consists of a set of prime multiples that are continuously
// increased to keep pace with the candidate.

public static int[] generateFirstNPrimes(int n) {
    initializeTheGenerator(n);

    for (candidate = 5; primesFound < n; candidate += 2) {
        increaseEachPrimeMultipleToOrBeyondCandidate();
        if (candidateIsNotOneOfThePrimeMultiples()) {
            registerTheCandidateAsPrime();
        }
    }
    return primes;
}

private static void initializeTheGenerator(int n) {
    primes = new int[n];
    primeMultiples = new int[n];
    lastRelevantMultiple = 1;

    // prime the pump. (Sorry, couldn't resist.)
```

```
    primesFound = 2;
    primes[0] = 2;
    primes[1] = 3;

    primeMultiples[0] = -1;// irrelevant
    primeMultiples[1] = 9;
}

private static void increaseEachPrimeMultipleToOrBeyondCandida
    if (candidate >= primeMultiples[lastRelevantMultiple])
        lastRelevantMultiple++;

    for (int i = 1; i <= lastRelevantMultiple; i++)
        while (primeMultiples[i] < candidate)
            primeMultiples[i] += 2 * primes[i];
}

private static boolean candidateIsNotOneOfThePrimeMultiples()
    for (int i = 1; i <= lastRelevantMultiple; i++)
        if (primeMultiples[i] == candidate)
            return false;
    return true;
}

private static void registerTheCandidateAsPrime() {
    primes[primesFound] = candidate;
    primeMultiples[primesFound] = candidate * candidate;
    primesFound++;
```

```
    }
}
```

JOHN:

This version is a considerable improvement over the version in *Clean Code*. Reducing the number of methods made the code easier to read and resulted in cleaner interfaces. If it were properly commented, I think this version would be about as easy to read as my version (the additional methods you created didn't particularly help, but they didn't hurt either). I suspect that if we polled readers, some would like your version better and some would prefer mine.

Unfortunately, this revision of the code creates a serious performance regression: I measured a factor of three to four times slowdown compared to either of the earlier revisions. The problem is that you changed the processing of a particular candidate from a single loop to two loops (the `increaseEach...` and `candidateIsNot...` methods). In the loop from earlier revisions, and in the `candidateIsNot` method, the loop aborts once the candidate is disqualified (and most candidates are quickly eliminated). However, `increaseEach...` must examine every entry in `primeMultiples`. This results in 5 to 10 times as many loop iterations and a three-to-four-times overall slowdown.

Given that the whole reason for the current algorithm (and its complexity) is to maximize performance, this slowdown is unacceptable. The two methods must be combined.

I think what happened here is that you were so focused on something that isn't actually all that important (creating the tiniest possible methods) that you dropped the ball on other issues that really are important. We have now seen this twice. In the original version of `PrimeGenerator`, you were so

determined to make tiny methods that you didn’t notice that the code was becoming incomprehensible. In this version, you were so eager to chop up my single method that you didn’t notice that you were blowing up the performance.

I don’t think this was just an unfortunate combination of oversights. One of the most important things in software design is to identify what is important and focus on that; if you focus on things that are unimportant, you’re likely to mess up the things that are important.

The code in your revision is still under-commented. You believe that there is no meaningful way for comments to assist the reader in understanding the code. I think this stems from your general disbelief in the value of comments; you are quick to throw in the towel. This algorithm is unusually difficult to explain, but I still believe that comments can help. For example, I believe you must make some attempt to help readers understand why the first multiple for a prime is the square of the prime. You have taken a lot of time to develop your understanding of this; surely there must be some way to convey that understanding to others? If you had included that information in your original version of the code, you could have saved yourself that long bike ride. Giving up on this is an abdication of professional responsibility.

The few comments that you included in your revision are of little value. The first comment is too cryptic to provide much help: I can’t make any sense of the phrase “predicting the next composite number and skipping over it,” even though I completely understand the code it purports to explain. One of the comments is just a joke; I was surprised to see this, given your opposition to extraneous comments.

Clearly, you and I live in different universes when it comes to comments.

Finally, I don't understand why you are offended by the labeled `continue` statement in my code. This is a clean and elegant solution to the problem of escaping from nested loops. I wish more languages had this feature; the alternative is awkward code where you set a variable, then exit one level of loop, then check the variable and exit the next level.

UB:

Good catch! I would have caught that too had I thought to profile the solution. You are right that separating the two loops added some unnecessary iteration. I found a nice way to solve that problem without using the horrible `continue`. My updated version is now faster than yours! A million primes in 440ms as opposed to yours, which takes 561ms. ;-) Below are just the changes.

```
public static int[] generateFirstNPrimes(int n) {
    initializeTheGenerator(n);

    for (candidate = 5; primesFound < n; candidate += 2)
        if (candidateIsPrime())
            registerTheCandidateAsPrime();

    return primes;
}

private static boolean candidateIsPrime() {
```

```
    if (candidate >= primeMultiples[lastRelevantMultiple])
        lastRelevantMultiple++;

    for (int i = 1; i <= lastRelevantMultiple; i++) {
        while (primeMultiples[i] < candidate)
            primeMultiples[i] += 2 * primes[i];
        if (primeMultiples[i] == candidate)
            return false;
    }
    return true;
}
```

JOHN:

Yep, that fixes the problem. I note that you are now down to four methods, from eight in the *Clean Code* version.

Test-Driven Development

JOHN:

Let’s move on to our third area of disagreement, which is test-driven development. I am a huge fan of unit testing. I believe that unit tests are an indispensable part of the software development process and pay for themselves over and over. I think we agree on this.

However, I am not a fan of test-driven development (TDD), which dictates that tests must be written before code and that code must be written and tested in tiny increments. This approach has serious problems, without any compensating advantages that I have been able to identify.

UB:

As I said at the start, I have carefully read *A Philosophy of Software Design*. I found it to be full of worthwhile insights, and I strongly agree with most of the points you make.

So I was surprised to find, on page 157, that you wrote a very short, dismissive, pejorative, and inaccurate section on test-driven development. Sorry for all the adjectives, but I think that's a fair characterization. So my goal here is to correct the misconceptions that led you to write the following:

"Test-driven development is an approach to software development where programmers write unit tests before they write code. When creating a new class, the developer first writes unit tests for the class, based on its expected behavior. None of these tests pass, since there is no code for the class. Then the developer works through the tests one at a time, writing enough code for that test to pass. When all of the tests pass, the class is finished."

This is just wrong. TDD is quite considerably different from what you describe. I describe it using three laws.

1. You are not allowed to write any production code until you have first written a unit test that fails because that code does not exist.
2. You are not allowed to write more of a unit test than is sufficient to fail, and failing to compile is failing.
3. You are not allowed to write more production code than is sufficient to make the currently failing test pass.

A little thought will convince you that these three laws will lock you into a cycle that is just a few seconds long. You'll write a line or two of a test that will fail, you'll write a line or two of production code that will pass, around and around every few seconds.

A second layer of TDD is the Red-Green-Refactor loop. This loop is several minutes long. It is comprised of a few cycles of the three laws, followed by a period of reflection and refactoring. During that reflection, we pull back from the intimacy of the quick cycle and look at the design of the code we've just written. Is it clean? Is it well structured? Is there a better approach? Does it match the design we are pursuing? If not, should it?

JOHN:

Oops! I plead "guilty as charged" to inaccurately describing TDD. I will fix this in the next revision of APOSD. That said, your definition of TDD does not change my concerns.

Let's discuss the potential advantages and disadvantages of TDD; then readers can decide for themselves whether they think TDD is a good idea overall.

Before we start that discussion, let me clarify the approach I prefer as an alternative to TDD. In your online videos, you describe the alternative to TDD as one where a developer writes the code, gets it fully working (presumably with manual tests), then goes back and writes the unit tests. You argue that this approach would be terrible: Developers lose interest once they think code is working, so they wouldn't actually write the tests. I agree with you completely. However, this isn't the only alternative to TDD.

The approach I prefer is one where the developer works in somewhat larger units than in TDD, perhaps a few methods or a class. The developer first writes some code (anywhere from a few tens of lines to a few hundred lines), then writes unit tests for that code. As with TDD, the code isn't considered to be "working" until it has comprehensive unit tests.

UB:

How about if we call this technique “bundling” for purposes of this document? This is the term I use in *Clean Code, Second Edition.*

JOHN:

Fine by me. The reason for working in larger units is to encourage design thinking so that a developer can think about a collection of related tasks and do a bit of planning to come up with a good overall design where the pieces fit together well. Of course, the initial design ideas will have flaws and refactoring will still be necessary, but the goal is to center the development process around design, not tests.

To start our discussion, can you make a list of the advantages you think that TDD provides over the approach I just described?

UB:

The advantages I usually attribute to TDD are

- Very little need for debugging. After all, if you just saw everything working a minute or two ago, there’s not much to debug.
- A stream of reliable, low-level documentation, in the form of very small and isolated unit tests. Those tests describe the low-level structure and operation of every facet of the system. If you want to know how to do something in the system, there are tests that will show you how.
- A less-coupled design, which results from the fact that every small part of the system must be designed to be testable, and testability requires decoupling.
- A suite of tests that you trust with your life, and therefore supports fearless refactoring.

However, you asked me which of these advantages TDD might have over *your* preferred method. That depends on how big you make those larger units you described. The important thing to me is to keep the cycle time short, and to prevent entanglements that block testability.

It seems to me that working in small units, and then immediately writing after-the-fact tests, can give you all the above advantages, so long as you are very careful to test every aspect of the code you just wrote. I think a disciplined programmer could effectively work that way. Indeed, I think such a programmer would produce code that I could not distinguish from code written by another programmer following TDD.

Above you suggested that bundling is to encourage design. I think encouraging design is a very good thing. My question for you is: Why do you think that TDD does not encourage design? My own experience is that design comes from strategic thought, which is independent of the tactical behavior of either TDD or bundling. Design is taking one step back from the code and envisioning structures that address a larger set of constraints and needs.

Once you have that vision in your head, it seems to me bundling and TDD will yield similar results.

JOHN:

First, let me address the four advantages you listed for TDD.

- Very little need for debugging? I think any form of unit testing can reduce debugging work, but not for the reason you suggested. The benefit comes because unit tests expose bugs earlier and in an environment where they are easier to track down. A relatively simple bug to fix in development can be very painful to track down in production. I’m not convinced by your argument

that there's less debugging because "you just saw everything working a minute ago": It's easy to make a tiny change that exposes a really gnarly bug that has existed for a long time but hasn't yet been triggered. Hard-to-debug problems arise from the accumulated complexity of the system, not from the size of the code increments.

UB:

True. However, when the cycles are very short, then the causes of even the gnarliest of bugs have the best chance of being tracked down. The shorter the cycles, the better the chances.

JOHN:

This is only true up to a point. I think you believe that making units smaller and smaller continues to provide benefits, with almost no limit to how small they can get. I think that there is a point of diminishing returns, where making things even smaller no longer helps and actually starts to hurt. We saw this disagreement over method length, and I think we're seeing it again here.

- Low-level documentation? I disagree: Unit tests are a poor form of documentation. Comments are a much more effective form of documentation, and you can put them right next to the relevant code. Trying to learn a method's interface by reading a bunch of unit tests seems much more difficult than just reading a couple of sentences of English text.

UB:

Nowadays, it's very easy to find the tests for a function by using the "where-used" feature of the IDE. As for comments being better, if that were true, then no one would publish example code. A less-coupled design? Possibly, but I haven't

experienced this myself. It’s not clear to me that designing for testability will produce the best design. Generally the decoupling arises because the test requires a mock of some kind. Mocks tend to force abstractions that might otherwise not exist.

JOHN:

In my experience, mocking virtually never changes interfaces; it just provides replacements for existing (typically immovable) interfaces.

UB:

Our experiences differ.

- Enabling fearless refactoring? Bingo! This is the where almost all of the benefits from unit testing come from, and it is a really, really big deal.

UB:

Agreed.

JOHN:

I agree with your conclusion that TDD and bundling are about the same in terms of providing these benefits. Now let me explain why I think TDD is likely to result in bad designs. The fundamental problem with TDD is that it forces developers to work too tactically, in units of development that are too small; it discourages design thinking. With TDD, the basic unit of development is one test: First the test is written, then the code to make that test pass. However, the natural units for design are larger than this: a class or method, for example. These units correspond to multiple test cases. If a developer thinks only about the next test, they are only considering part of a design problem at any given time. It’s hard to design something well if you don’t

think about the whole design problem at once. TDD explicitly prohibits developers from writing more code than is needed to pass the current test; this discourages the kind of strategic thinking needed for good design.

TDD does not provide adequate guidance to encourage design. You mentioned the Red-Green-Refactor loop, which recommends refactoring after each step, but there's almost no guidance for refactoring. How should developers decide when and what to refactor? This seems to be left purely to their own judgment. For example, if I am writing a method that requires multiple iterations of the TDD loop, should I refactor after every iteration (which sounds pretty tedious) or wait until after several iterations so that I can look at a bigger chunk of code when refactoring and hence be more strategic? Without guidance, it will be tempting for developers to keep putting off refactoring.

TDD is similar to the One Thing rule we discussed earlier in that it is biased: It provides very strong and clear instructions pushing developers in one direction (in this case, acting tactically) with only vague guidance in the other direction (designing more strategically). As a result, developers are likely to err on the side of being too tactical.

TDD guarantees that developers will initially write bad code. If you start writing code without thinking about the whole design problem, the first code you write will almost certainly be wrong. Design only happens after a bunch of bad code has accumulated. I watched your video on TDD, and you repeatedly wrote the wrong code, then fixed it later. If the developer refactors conscientiously (as you did), they can still end up with good code, but this works against human nature. With TDD, that bad code will actually work (there are tests to prove it!), and it's human nature not to want to change something that works. If the code I'm developing is nontrivial, I will probably have to accumulate a lot of bad

code with TDD before I have enough code in front of me to understand what the design should have been. It will be very difficult for me to force myself to throw away all that work.

It’s easy for a developer to believe they are doing TDD correctly while working entirely tactically, layering on hack after hack with an occasional minor refactor, without ever thinking about the overall design.

I believe that the bundling approach is superior to TDD because it focuses the development process around design: Design first, then code, then write unit tests. Of course, refactoring will still be required: It’s almost never possible to get the design right the first time. But starting with design will reduce the amount of bad code you write and get you to a good design sooner. It is possible to produce equally good designs with TDD; it’s just harder and requires a lot more discipline.

UB:

I’ll address your points one at a time.

- I haven’t found that the scale of TDD is so tactical that it discourages thinking. Every programmer, regardless of their testing discipline, writes code one line at a time. That’s immensely tactical and yet does not discourage design. So why would one test at a time discourage it?

- The literature on TDD strongly discourages delaying refactoring. While thinking about design is strongly encouraged, both are integral parts of the discipline..

- We all write bad code at the start. The discipline of TDD gives us the opportunity, and the safety, to continuously clean it. Design insights arise from those kinds of cleaning activities. The discipline of

refactoring allows bad designs to be transformed, one step at a time, into better designs.

- It’s not clear to me why the act of writing tests late is a better design choice. There’s nothing in TDD that prevents me from thinking through a design long before I write the very first tested code.

JOHN:

You say there is nothing about TDD that stops developers from thinking ahead about design. This is only partly true. Under TDD, I can think ahead, but I can’t actually write my ideas down in the form of code, since that would violate TDD Rule 1. This is a significant discouragement.

You claim that “thinking about design is strongly encouraged” in TDD, but I haven’t seen this in your discussions of TDD. I watched your video example of using TDD for computing bowling scores, and design is never even mentioned after the first minute or two (ironically, one of the conclusions of this example is that the brief initial design turned out to be useless). There is no suggestion of thinking ahead in the video; it’s all about cleaning up messes after the fact. In all of the TDD materials you have shown me, I have not seen any warnings about the dangers of becoming so tactical with TDD that design never occurs (perhaps you don’t even view this as a serious risk?).

UB:

I usually use an abbreviated form of UML to capture my early design decisions. I have no objection to capturing them in pseudocode, or even real code. However, I would not commit any such prewritten code. I would likely hold it in a text file, and consult it while following the TDD cycle. I might feel safe enough to copy and paste from the text file into my IDE in order to make a failing test pass.

The Bowling game is an example of how wildly our initial design decisions can deviate from our eventual solutions. It's true that introductory videos often do not expose the depth of a discipline.

JOHN:

As I was watching your TDD video for the second time, you said something that jumped out at me:

"Humans consider things that come first to be important and things that come at the end to be less important and somehow optional; that's why they are at the end, so we can leave them out if we have to."

This captures perfectly my concern about TDD. TDD insists that tests must come first, and design, if it happens at all, comes at the end, after code is working. I believe that good design is the most important thing, so it must be the top priority. I don't consider tests optional, but delaying them is safer than delaying design. Writing tests isn't particularly difficult; the most important thing is having the discipline to do it. Getting a good design is really hard, even if you are very disciplined; that's why it needs to be the center of attention.

UB:

TDD is a coding discipline. Of course, design comes before coding—I don't know anyone who thinks otherwise. Even the Bowling Game video made that point. But as we saw in the Bowling Game video, sometimes the code will take you in a very different direction.

That difference doesn't imply that the design shouldn't have been done. It just implies that designs are speculative and may not aways survive reality.

As Eisenhower once said:

“In preparing for battle, I have always found that plans are useless, but planning is indispensable.”

JOHN:

You ask why writing tests later is a better design choice. It isn’t. The benefit of the bundled approach doesn’t come from writing tests later; it comes from doing design sooner. Writing tests (a bit) later is a consequence of this choice. The tests are still written pretty early on with the bundled approach, so I don’t think the delay causes significant problems.

UB:

I think we simply disagree that TDD discourages design. The practice of TDD does not discourage me from design, because I value design. I would suggest that those who do not value design will not design, no matter what discipline they practice.

JOHN:

You claim that the problems I worry about with TDD simply don’t happen in practice. Unfortunately, I have heard contrary claims from senior developers that I trust. They complain about horrible code produced by TDD-based teams, and they believe that the problems were caused by TDD. Of course, horrible code can be produced with any design approach. And maybe those teams didn’t implement TDD properly, or maybe those cases were outliers. But the problems reported to me line up exactly with what I would expect to happen, given the tactical nature of TDD.

UB:

My experience differs. I’ve worked on many projects where TDD has been used effectively and profitably. I’m sure the senior developers that you trust are telling you the truth

about their experience. Having never seen TDD lead to such bad outcomes myself, I sincerely doubt that the blame can be traced to TDD.

JOHN:

You ask me to trust your extensive experience with TDD, and I admit that I have no personal experience with TDD. On the other hand, I have a lot of experience with tactical programming, and I know that it rarely ends well. TDD is one of the most extreme forms of tactical programming I've encountered. In general, if "making it work" is the #1 priority instead of "develop a clean design," code turns to spaghetti. I don't see enough safeguards in your approach to TDD to prevent the disaster scenarios; I don't even see a clear recognition of the risk.

Overall, TDD is in a bad place on the risk–reward spectrum. In comparison to the bundling approach, the downside risks for poor code quality in TDD are huge, and I don't see enough upside reward (if any) to compensate.

UB:

All I can say to that is that your opinion is based on a number of false impressions and speculations, and not upon direct experience.

JOHN:

Now let me ask you a couple of questions. First, at a microscopic level, why on earth does TDD prohibit developers from writing more code than needed to pass the current test? How does enforcing myopia make systems better?

UB:

The goal of the discipline is to make sure that everything is tested. One good way to do that is to refuse to write any code unless it is to make a failing test pass. Also, working in such short cycles provides insights into the way the code is working. Those insights often lead to better design decisions.

JOHN:

I agree that seeing code (partially) working can provide insights. But surely that benefit can be had without such a severe restriction on how developers think?

Second, at a broader level, do you think TDD is likely to produce better designs than approaches that are more design centric, such as the bundling approach I described? If so, can you explain why?

UB:

My guess is that someone adept at bundling and someone adept at TDD would produce very similar designs, with very similar test coverage. I would also venture to guess that the TDDer would be somewhat more productive than the bundler if for no reason other than that the TDDer finds and fixes problems earlier than the bundler.

JOHN:

I think that the bundling approach will result in a better design because it actually focuses on design, rather than focusing on tests and hoping that a good design will magically emerge. I think it’s really hard to argue that the best way to achieve one thing is to focus your attention on something else. And the bundling approach will make progress faster because the early thinking about design will reduce the amount of bad code you end up having to throw away under TDD. Overall, I’d argue that the best-case

outcomes for the two approaches will be about the same, but average and (especially) worst-case outcomes will be far worse for TDD.

JOHN:

I don't think we're going to resolve our disagreements on TDD. To do that, we'd need empirical data about the frequency of good and bad outcomes from TDD. Unfortunately, I'm not aware of any such data. Thus, readers will have to decide for themselves whether the potential benefits of TDD outweigh the risks.

For anyone who chooses to use TDD, I urge you to do so with extreme caution. Your primary goal must not be just working code, but rather, a clean design that will allow you to develop quickly in the future. TDD will not lead you naturally to the best design, so you will need to do significant and continuous refactoring to avoid spaghetti code. Ask yourself repeatedly: "Suppose that I knew everything I know now when I first started on this project; would I have chosen the current structure for the code?" When the answer is no (which will happen frequently), stop and refactor. Recognize that TDD will cause you to write more bad code than you may be used to, so you must be prepared to throw out and rewrite more than you are used to. Take time to plan ahead and think about the overall design, rather than just making the next test work. If you do all of these things diligently, I think it is possible to mitigate the risks of TDD and produce well-designed code.

UB:

Let's just say that I agree with all that advice, but disagree with your assertion that TDD might be the cause of bad code.

TDD Summary

JOHN:

Here is my attempt to summarize our thoughts on test-driven development: We agree that unit tests are an essential element in software development. They allow developers to make significant changes to a system without fear of breaking something. We agree that it is possible to use TDD to produce systems with good designs.

- I believe that TDD discourages good design and can easily lead to very bad code. You do not believe that TDD discourages good design and don’t see much of a risk of bad code.
- I believe that there are better approaches than TDD for producing good unit test suites, such as the bundling approach discussed above. You agree that bundling can produce outcomes just as good as TDD but think it may lead to somewhat less test coverage.
- I believe that TDD and bundling have similar best-case outcomes, but that the average and worst-case outcomes will be much worse for TDD. You disagree and believe that, if anything, TDD may produce marginally better outcomes than bundling. You also think that preference and personality are larger factors in making the choice between the two.

UB:

This is a fair summary of our discussion. We seem to disagree over the best application of discipline. I prefer a disciplined approach to keep the code covered by tests written first in very short cycles. You prefer a disciplined approach of writing relatively longer bundles of code and

then writing tests for those bundles. We disagree on the risks and rewards of these two disciplines.

Closing Remarks

JOHN:

First, I'd like to thank you for tolerating (and responding to) the arguments I have made about some of the key ideas in *Clean Code*. I hope this discussion will provide food for thought for readers.

We have covered a lot of topics and subtopics in this discussion, but I think that most of my concerns result from two general errors made by *Clean Code*: failure to focus on what is important, and failure to balance design trade-offs.

In software design (and probably in any design environment), it is essential to identify the things that really matter and focus on those. If you focus your attention on things that are unimportant, you are unlikely to achieve the things that really are important. Unfortunately, *Clean Code* repeatedly focuses on things that don't really matter, such as:

- Dividing 10-line methods into 5-line methods and dividing 5-line methods into 2- or 3-line methods
- Eliminating the use of comments written in English
- Writing tests before code and making the basic unit of development a test rather than an abstraction

None of these provides significant value, and we have seen how they distract from producing the best possible designs.

Conversely, *Clean Code* fundamentally undervalues comments, which are essential and irreplaceable. This comes at a huge cost. Without interface comments, the

specifications for interfaces are incomplete. This is guaranteed to result in confusion and bugs. Without implementation comments, readers are forced to re-derive knowledge and intentions that were in the mind of the original developer. This wastes time and leads to more bugs.

In my opening remarks, I said that systems become complex when important information is not accessible and obvious to developers. By refusing to write comments, you are hiding important information that you have and that others need.

The second general error in *Clean Code* has to do with balance. Design represents a balance between competing concerns. Almost any design idea becomes a bad thing if taken to the extreme. However, *Clean Code* repeatedly gives very strong advice in one direction without correspondingly strong advice in the other direction or any meaningful guidance about how to recognize when you have gone too far. For example, making methods shorter is often a good thing, but the *Clean Code* position is so one-sided and extreme that readers are likely to chop things up too much. We saw in the `PrimeGenerator` example how this resulted in code that was nearly incomprehensible. Similarly, the *Clean Code* position on TDD is one-sided, failing to recognize any possible weakness and encouraging readers to take this to a tactical extreme where design is completely squeezed out of the development process.

UB:

John, I'd like to thank you for participating in this project. This was a lot of fun for me. I love disagreement and debate with smart people. I also think that we share far more values than separate us.

For my part, I'll just say that I have given due consideration to the points you've made, and while I disagree with your closing remarks, I have integrated several of your better

ideas, as well as this entire document, into the second edition of *Clean Code*.

Thanks again, and give my best to your students.

Bibliography

[A+DS=P]

Niklaus Wirth, *Algorithms + Data Structures = Programs*, Prentice Hall, 1976.

[APOSD]

John K. Ousterhout, *A Philosophy of Software Design*, Yaknyam Press, 2018.

[BeckTDD]

Kent Beck, *Test Driven Development*, Addison-Wesley, 2003.

[Clean Agile]

Robert C. Martin, *Clean Agile*, Pearson Education Inc., 2020.

[Clean Arch]

Robert C. Martin, *Clean Architecture*, Pearson Education Inc., 2018.

[Clean Craftsmanship]

Robert C. Martin, *Clean Craftsmanship: Disciplines, Standards, and Ethics*, Pearson Education Inc., 2022.

[DDD]

Eric Evans, *Domain Driven Design*, Addison-Wesley, 2004.

[DOET]

Don Norman, *The Design of Everyday Things*, Basic Books, 2002.

[Functional Design]

Robert C. Martin, *Functional Design*, Pearson Education Inc., 2024.

[GOF95]

Gamma, Helm, Johnson, Vlissides, *Design Patterns*, Addison-Wesley, 1995.

[Hex]

Alistair Cockburn and Juan Manuel Garrido de Paz, *Hexagonal Architecture Explained*, Humans and Technology Inc., 2024.

[IMP]

Kent Beck, *Implementation Patterns*, Addison-Wesley, 2007.

[KP78]

Kernighan and Plaugher, *The Elements of Programming Style, Second Edition*, McGraw-Hill, 1978.

[Lea99]

Doug Lea, *Concurrent Programming in Java: Design Principles and Patterns*, *Second Edition*, Prentice Hall, 1999.

[OOSC]

Bertrand Meyer, *Object Oriented Software Construction*, Prentice Hall, 1988.

[Pomodoro]

Francesco Cirillo, *The Pomodoro Technique*, Crown Publishing Group, 2006.

[PPEDC]

M. V. Wilkes, D. J. Wheeler, and S. Gill, *The Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley, 1951.

[PPP02]

Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2002.

[Prag]

Andrew Hunt and Dave Thomas, *The Pragmatic Programmer*, Addison-Wesley, 2000.

[Refactoring]

Martin Fowler, *Refactoring, Second Edition*, Pearson Education Inc., 2019.

[RSpec]

Aslak Hellesøy and David Chelimsky, *RSpec: Behavior Driven Development for Ruby Programmers*, Pragmatic Bookshelf, 2008.

[Singularity]

Ray Kurzweil, *The Singularity is Nearer*, Viking, 2024.

[SP72]

O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, 1972.

[Tidy]

Kent Beck, *Tidy First?*, O’Reilly, 2023.

[We Programmers]

Robert C. Martin, *We, Programmers*, Pearson Education Inc., 2025.

[WELC]

Michael Feathers, *Working Effectively with Legacy Code*, Pearson Education Inc., 2004.