



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

FREE SAMPLE CHAPTER |



Praise for the First Edition of Clean Code

“*Clean Code* has changed the way I develop and maintain software in so many ways. It defines so well how code should be written, and I have not only learnt much from this book but have also put into practice many of its gems. I can honestly say that having gone on this journey, it is once again a pleasure to develop software and look at code that you are proud to have written.”

—Steve Roberts, head of IT

“Reading *Clean Code* validated my inner sense of how software should be written, while also improving the quality of my work by teaching me things I didn’t know. I’m grateful for this book and others by the same author as they remain fundamental prerequisites, even in the age of AI.”

—Milos Baic, developer and solution architect

“[*Clean Code*] was the single most influential thing that I encountered in my career. Provided I was in a team that adhered to it, it took a lot of stress away as the code was suddenly more understandable and thus easier to maintain. To design something with testability in mind eliminated a lot of complexity in design and maintenance. Above all, it made programming feel cozy and fun again. The feeling gets stronger when you move from poor to a cleaner code base.”

—Žarko Stojanović, staff software engineer

Robert C. Martin Series



THE ROBERT C. MARTIN SERIES is directed at software developers, team leaders, business analysts, and managers who want to increase their skills and proficiency to the level of a Master Craftsman.

Clean Code presents the principles, patterns, and practices of writing clean code and challenges programmers to closely read code, discovering what's right and what's wrong with it.

Clean Coder provides practical guidance on how to approach software development and work well and clean with a software team.

Clean Craftsmanship picks up where *Clean Code* leaves off, outlining additional ways to write quality and trusted code you can be proud of every day.

Clean Agile is a clear and concise guide to basic Agile values and principles. Perfect for those new to Agile methods and long-time developers who want to simplify approaches for the better.

Clean Architecture brings the methods and tactics of “clean coding” to system design.

Visit informit.com/martinseries for a complete list of available publications.

Clean Code

A HANDBOOK OF AGILE SOFTWARE CRAFTSMANSHIP

SECOND EDITION

Robert C. Martin

◆ Addison-Wesley

Hoboken, New Jersey

Cover image: Blueee77/Shutterstock

Page xxxi: Author photo courtesy of Robert C. Martin

Page 318: Photo of Harvard Mark I, Arnold Reinhold, licensed under CC BY-SA 3.0

Some third-party illustrations included in this book are used with permission from their original creators. These images are not authored or endorsed by Pearson Education and are included for educational and illustrative purposes only.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Please contact us with concerns about any potential bias at www.pearson.com/en-us/report-bias.html.

Author websites are not owned or managed by Pearson.

Visit us on the Web: informit.com

Library of Congress Control Number: 2025941348

Copyright © 2026 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/en-us/global-permission-granting.html.

ISBN-13: 978-0-13-539857-9

ISBN-10: 0-13-539857-6

\$PrintCode

CONTENTS

Foreword	xix
Introduction	xxv
Introduction (from Long Ago)	xxix
About the Author	xxxi
Chapter 1 Clean Code	1
There Will Be Code	2
Bad Code	3
Attitude	4
The Primal Conundrum	5
The Art of Clean Code	5
What Is Clean Code?	6
Putting All This Together	9
Why Should We Be Clean?	9
Productivity	10
Livability	13
We Read More Than We Write	13
The Boy Scout Rule	14
PART I Code	17
Chapter 2 Clean That Code!	19
The Cleaning Process	30
Conclusion	37
Postscript: Future Bob Playing with Grok3	37
Postscript Conclusion	40

Chapter 3	First Principles	41
	Everything Small, Well Named, Organized, and Ordered	42
	Functions	42
	A More Significant Example	44
	Independent Deployability	68
	A Final Thought	68
Chapter 4	Meaningful Names	71
	Use Intention-Revealing Names	72
	Build a System of Names	73
	Avoid Disinformation	74
	Make Meaningful Distinctions	75
	Use Pronounceable Names	76
	Use Searchable Names	77
	Use Names of Appropriate Length	78
	Avoid Encodings	80
	Use Appropriate Parts of Speech	81
	Consider Keyword Parameters	82
	Don't Be Cute	83
	Pick One Word per Concept	83
	Use Solution Domain Names	83
	Use Problem Domain Names	84
	Add Meaningful Context	84
	Don't Add Gratuitous Context	86
	Final Words	87
Chapter 5	Comments	89
	Compensating for Failure	90
	Hidden or Obscured Comments	90
	Lying Comments	91
	Comments That Are Too Intimate	92
	Comments Do Not Make Up for Bad Code	92
	Explain Your Intent in Code	92
	Good Comments	92
	Legal Comments	93
	Informative Comments	93
	Explanation of Intent	94
	Clarification	95
	Warning of Consequences	96

Amplification	96
Javadocs (and Their ilk) in Public APIs	97
Bad Comments	97
Mumbling	97
Redundant Comments	98
Misleading Comments	99
Redundancy and Imprecision	99
Mandated Comments	102
Journal Comments	102
Noise Comments	103
Scary Noise	105
TODO Comments	106
Use a Function or a Variable Instead of a Comment	106
Position Markers	107
Attributions and Bylines	107
Commented-Out Code	107
HTML Comments	108
Nonlocal Information	109
Too Much Information	109
Unobvious Connection	110
Function Headers	110
Javadocs in Nonpublic Code	111
Example	111
Conclusion	115
 Chapter 6	 Formatting
Chapter 6	117
The Purpose of Formatting	118
Vertical Formatting	118
Vertical Openness between Concepts	120
Vertical Density	121
Vertical Distance	122
Variable Declarations	122
Dependent Functions	125
Conceptual Affinity	126
Horizontal Formatting	127
Horizontal Openness and Density	128
Horizontal Alignment	129
Indentation	130
Breaking Indentation	132

	Team Rules	132
	Uncle Bob's Formatting Rules	133
Chapter 7	Clean Functions	137
	Small!	138
	Well-Written Prose	139
	One Level of Abstraction per Function	139
	Reading Code from Top to Bottom: The Stepdown Rule	140
	Entanglement	142
	Switch Statements	142
	Clean Functions: A Deeper Look	144
	Contextual	144
	Nameable	145
	Descriptive	145
	Convenient	147
	Insulated	148
	Homogenous	151
	Pure	153
	Partial Purity	155
	Conclusion	157
Chapter 8	Function Heuristics	159
	Function Arguments	159
	Variadic Arguments	161
	More Than Three?	161
	Keyword Arguments	161
	Flag Arguments	161
	Output Arguments	162
	Error Codes	163
	Command Query Separation	164
	Prefer Exceptions to Returning Error Codes	165
	Caveat Emptor	165
	Extract Try/Catch Blocks	167
	Error Handling Is One Thing	168
	The Dependency Magnet of Error Codes	168
	DRY: Don't Repeat Yourself	169
	Simple Repeated Code	169
	Similar Code	170
	Loop Duplication	174

Accidental versus Essential Duplication	177
Side Effects	177
We Are Bad at This	178
Functional Languages	178
Object-Oriented Languages	180
Structured Programming	181
Sequence	182
Selection	182
Iteration	182
This Is Too Much to Constantly Keep in Mind	183
Conclusion	184
Chapter 9 The Clean Method	185
Make It Right	186
Example	188
Considering the Design and Architecture	204
Conclusion	210
Chapter 10 One Thing	211
Extract Method Refactoring	212
This Shouldn't Be Controversial	214
1. Drowning	214
2. Small Functions Don't Obscure Intent	216
3. Performance	216
4. Bouncing Around	216
5. Entanglement	217
What Are Large Functions Anyway?	217
Extraction and Classes	235
Conclusion	239
Chapter 11 Be Polite	241
The Newspaper Metaphor	243
Be Polite	244
The Stepdown Rule: Once Again	245
The Abstraction Roller Coaster	246
This Is How We Write, but Not How We Want to Read	246
Chapter 12 Objects and Data Structures	249
What Is an Object?	250
Data Abstraction	251

Data/Object Antisymmetry	252
The Law of Demeter	255
Trainwrecks	256
Hybrids	256
Hiding Structure	257
Data Transfer Objects	258
The Object/Relational “Impedance Mismatch”	259
Using Objects and Data Structures	259
Switch Statements	260
The OO Solution	263
Not So Fast, Johnson	264
The OO/Procedural Trade-off	264
But What About Performance?	265
Conclusion	265
 Chapter 13 Clean Classes	 267
Classes and Modules versus Files	267
What Should a Class Contain?	268
Characterizing Class Design	269
Heuristics and Characteristics	270
When Is a Class Too Large?	271
Policies in Code	273
Where Reasons to Change Hide	274
Fixing the Problem	275
An Overly Open Implementation	277
Should We Do Anything Now?	278
What About Now?	278
Closed, Cohesive, Single-Responsibility Classes	282
When Policies Change	284
Is This Overengineering?	284
Simpler Testing	285
Enter AI	286
It Will Be Wrong	287
 Chapter 14 Testing Disciplines	 289
Discipline 1: Test-Driven Development (TDD)	291
The Three Laws of TDD	291
Discipline 2: Test && Commit Revert (TCR)	292

Discipline 3: Small Bundles	293
Design	293
Discipline	294
Tedious, Boring, and Slow	294
Debugging	294
Documentation	295
Reliability	295
Design	296
Reprise	296
The Angel and the Devil	297
Muzzling the Devil	297
Complications and Loopholes	298
Costs and Repercussions	299
Keeping Tests Clean	299
Tests Enable the -ilities	300
 Chapter 15 Clean Tests	 303
Domain-Specific Testing Language	307
Composed Assertions	307
Composed Test Results	307
Dual Standard	309
The Single Assert Rule	309
The Single Act Rule	310
F.I.R.S.T.	310
Fast	310
Isolated	310
Repeatable	310
Self-Validating	311
Timely	311
Test Design	311
Conclusion	311
 Chapter 16 Acceptance Testing	 313
The Acceptance Testing Discipline	314
The Discipline	315
The Continuous Build	316
Conclusion	316

Chapter 17	Als, LLMs, and God Knows What	317
	Programming by Prompt	319
	Infancy	324
	A SWAG	324
	Conclusion	328
PART II	Design	329
Chapter 18	Simple Design	331
	YAGNI	333
	Covered by Tests	333
	An Asymptotic Goal	333
	Design?	334
	Maximize Expression	334
	The Underlying Abstraction	336
	Tests: The Other Half of the Problem	337
	Minimize Duplication	337
	Accidental Duplication	338
	Minimize Size	339
	Simple Design	339
Chapter 19	The SOLID Principles	341
	SRP: The Single Responsibility Principle	343
	Accidental Duplication	344
	Solutions	345
	Higher Levels	346
	OCP: The Open–Closed Principle	347
	A Thought Experiment	347
	Directional Control	350
	Information Hiding	350
	Conclusion	351
	LSP: The Liskov Substitution Principle	351
	LSP and Design	352
	Taxi Aggregator	353
	ISP: The Interface Segregation Principle	354
	ISP and Language	355
	ISP and Design	355
	DIP: The Dependency Inversion Principle	356
	Stable Abstractions	358

Factories	359
Concrete Components	360
Chapter 20 Component Principles	363
Components	364
A Brief History of Components	364
Relocatability	367
Linkers	367
Component Cohesion	368
The Reuse/Release Equivalence Principle (REP)	369
The Common Closure Principle (CCP)	370
The Common Reuse Principle (CRP)	371
The Tension Diagram for Component Cohesion	372
In Summary...	373
Component Coupling	373
The Acyclic Dependencies Principle (ADP)	374
The Stable Dependencies Principle (SDP)	379
The Stable Abstractions Principle (SAP)	383
Conclusion	387
Chapter 21 Continuous Design	389
Continuous Change	390
Continuous Design	391
Sailing on the Four Cs of Continuous Design	392
Clarity	393
Conciseness	398
Confirmability	406
Cohesion	416
When Else Do We Design?	420
Up-front Design	420
Readying for Work	421
Starting Work	421
Doing Work	422
Chapter 22 Concurrency	423
Why Concurrency?	424
Myths and Misconceptions	425
Challenges	426

Concurrency Defense Principles	427
Single Responsibility Principle	427
Corollary: Limit the Scope of Data	427
Corollary: Use Copies of Data	428
Corollary: Threads Should Be as Independent as Possible	428
Know Your Language and Library	428
Thread-Safe Collections	429
Know Your Execution Models	429
Producer-Consumer	430
Readers-Writers	430
Dining Philosophers	431
Beware Dependencies between Synchronized Methods	431
Keep Synchronized Sections Small	431
Writing Correct Startup and Shutdown Code Is Hard	432
Testing Threaded Code	432
Treat Spurious Failures as Candidate Threading Issues	433
Get Your Nonthreaded Code Working First	433
Make Your Threaded Code Pluggable	434
Make Your Threaded Code Tunable	434
Run with More Threads Than Processors	434
Run on Different Platforms	434
Instrument Your Code to Try and Force Failures	435
Hand-Coded	435
Automated	436
2025 Update and Report from the Field	437
Data Integrity	437
Conclusion	442
PART III Architecture	443
Chapter 23 The Two Values of Software	445
Keeping Options Open	446
Chapter 24 Independence	449
Use Cases	450
Operation	450
Development	451
Deployment	451
Leaving Options Open	451

Chapter 25	Architectural Boundaries	453
	What Lines Do You Draw, and When?	454
	Plug-in Architecture	456
	Case Study: FitNesse	457
	Conclusion	459
Chapter 26	Clean Boundaries	461
	Third-Party IoT Framework: Lots o' Boundaries	462
	UI/Application Boundary	466
	SOLID and Hexagonal Architecture	469
	Exploring and Learning Boundaries	470
	Using Code That Does Not Yet Exist	473
	Clean Boundaries	474
Chapter 27	The Clean Architecture	475
	The Dependency Rule	476
	Entities	477
	Use Cases	477
	Interface Adapters	478
	Frameworks and Drivers	478
	Only Four Circles?	478
	Crossing Boundaries	479
	What Data Crosses the Boundaries	479
	A Typical Scenario	479
	Conclusion	481
PART IV	Craftmanship	483
	“A Great Number”	484
	Eight Decades	484
	Nerds and Saviors	487
	Notoriety	488
	Role Models and Villains	489
	We Rule the World	490
	Catastrophes	491
	The Oath	492
Chapter 28	Harm	495
	No Harm to Society	496
	Harm to Function	497

	No Harm to Structure	499
	Soft	500
	Tests	501
Chapter 29	No Defect in Behavior or Structure	503
	Making It Right	504
	What Is Good Structure?	504
	Eisenhower's Matrix	506
	Programmers Are Stakeholders	507
	Do Your Best	508
Chapter 30	Repeatable Proof	511
	Dijkstra	511
	Proving Correctness	512
	Structured Programming	514
	Functional Decomposition	516
	Test-Driven Development et al.	517
Chapter 31	Small Cycles	519
	The History of Source Code Control	519
	Punch Cards	519
	Continuous Integration	523
	Short Cycles	524
	Continuous Integration	524
	Branches versus Toggles	525
	Continuous Deployment	527
	Continuous Build	528
Chapter 32	Relentless Improvement	529
	Test Coverage	529
	Mutation Testing	530
	Semantic Stability	530
	Cleaning	531
	Creations	531
Chapter 33	Maintain High Productivity	533
	Viscosity	533
	Building	534
	Testing	534

Debugging	535
Deploying	535
Managing Distractions	535
Meetings	536
Music	536
Mood	537
The Flow	537
Time Management	538
Chapter 34 Work as a Team	539
Collaborative Programming	539
Open/Virtual Office	540
Chapter 35 Estimate Honestly and Fairly	543
Lies	544
Honesty, Accuracy, Precision	544
Lessons from Me	545
Story #1: Vectors	545
Story #2: pCCU	546
The Lesson	547
Accuracy and Precision	547
Aggregation	548
Honesty	549
Pressure	550
Chapter 36 Respect for Fellow Programmers	553
Chapter 37 Never Stop Learning	555
Afterword	557
Appendix: The Clean Code Debate	561
Introductions	562
Method Length	563
Method Length Summary	575
Comments	576
Comments Summary	590
John's Rewrite of PrimeGenerator	591
A Tale of Two Programmers	596

Bob's Rewrite of PrimeGenerator2	599
Test-Driven Development	603
TDD Summary	612
Closing Remarks	613
Bibliography	615
Index	619

FOREWORD



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 because 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

This page intentionally left blank

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.

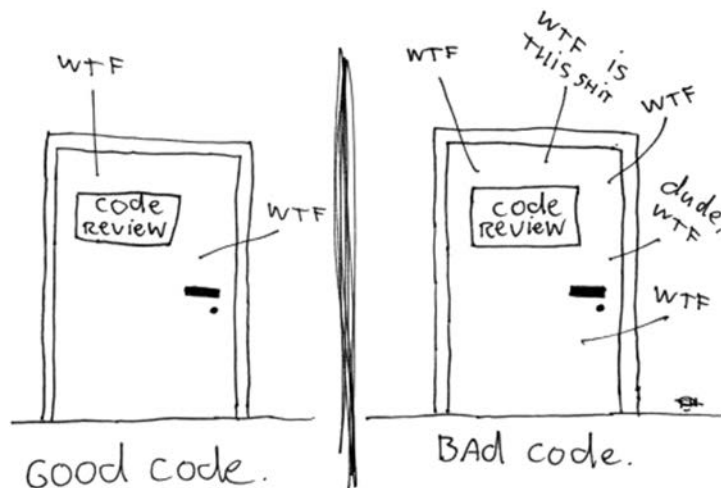
Register your copy of *Clean Code, Second Edition*, on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/cleancode2 and log in or create an account. The product ISBN (9780135398579) will already be populated. Click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

This page intentionally left blank

INTRODUCTION (FROM LONG AGO)

This is the Introduction to the first edition and has been edited and abridged.

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/minute



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.

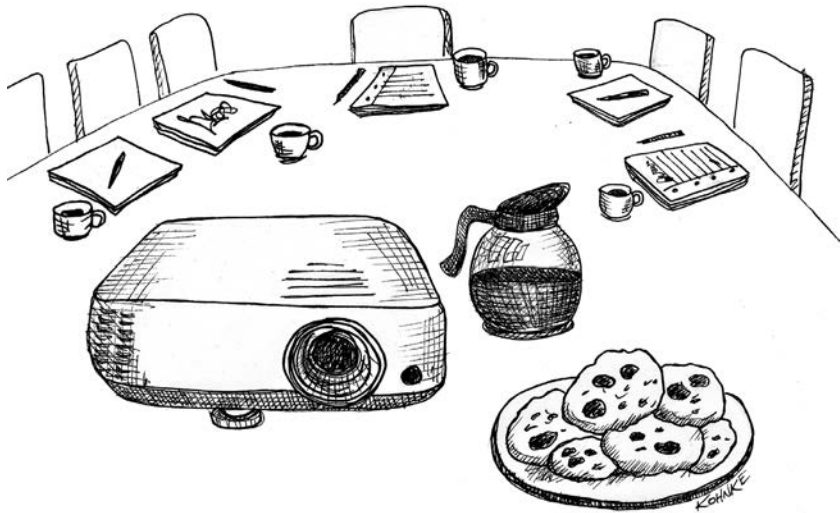
ABOUT THE AUTHOR



Robert C. Martin (Uncle Bob) has been a programmer since 1970. He is founder of Uncle Bob Consulting, LLC, and cofounder with his son Micah Martin of Clean Coders, LLC. Martin has published dozens of articles in various trade journals and is a regular speaker at international conferences and trade shows. He has authored and edited many books, including *Designing Object-Oriented C++ Applications Using the Booch Method*; *Pattern Languages of Program Design 3*; *More C++ Gems*; *Extreme Programming in Practice*; *Agile Software Development: Principles, Patterns, and Practices*; *UML for Java Programmers*; *Clean Code*; *The Clean Coder*; and *Functional Design: Principles, Patterns, and Practices*. A leader in the industry of software development, Martin served for three years as editor-in-chief of *The C++ Report*, and he served as the first chairman of the Agile Alliance.

This page intentionally left blank

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, more or less, guidelines. 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.

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."¹

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."  
        );  
    }  
}
```

1. Grady Booch, private email.

```
}

// 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 those extractions 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 code.");
}

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() == 5);
    boolean coffeeFiveDays = items.stream().anyMatch(
        item -> item.type() == CatalogItem.COFFEE && item.days() == 5);
    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.²

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.³

—Statement.java—

...

```
public void rent(CatalogItem item, int days) {  
    int unitPrice = getUnitPrice(item);  
    int price = calculatePrice(item, days, unitPrice);  
    int thisTax = getTax(item, price);  
}
```

2. [Tidy].

3. The Single Responsibility Principle.

```
        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 unitPrice) {
        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)⁴ 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.

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 `CatalogItems` 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),
```

4. See Chapter 19, "The SOLID Principles."

```

    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 enums 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` enums, 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 probably⁵ `Statement.java` because it depends upon `CatalogItem.java`. Now ask yourself whether `Statement.java` *should have* to 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 `CatalogItems`, 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,
```

5. 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.

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

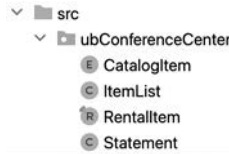
We'll have to come back to this module because that `getItems` method gives me the willies.⁶ It smells of an SRP violation. But for now, all that remains is to make a few adjustments to the `Statement.java` module.

```
...
public class Statement {
    ...
    private ItemList items = new ItemList();
    ...

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

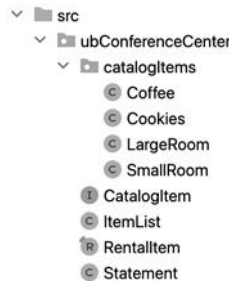
6. An archaic term that means nervous, uneasy, or creeped out. (According to Grok.)

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 unitPrice)
    {
        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";
    }
}
```

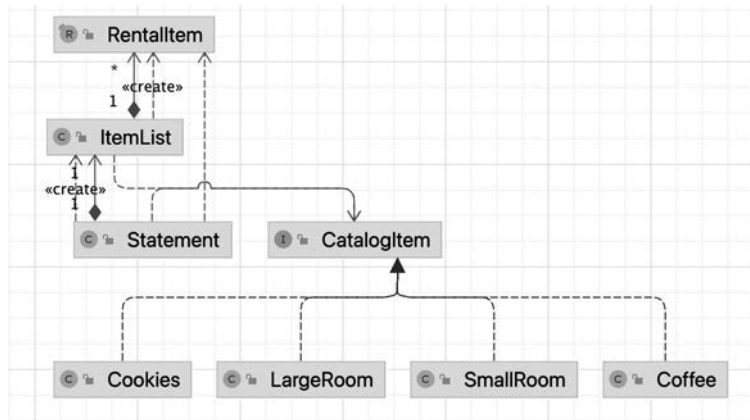
One of the people who commented on the first edition of this book looked at code like this and called it “*just dreadful*.” You might be feeling that now too. But remember, we are expecting business growth, and we have decided that it is necessary to make room in this code for that growth.

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.

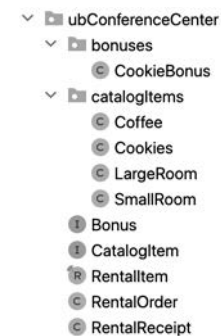
Indeed, the low-level details have become a *plug-in* to the high-level policy. Or, at least they are in a position to become such a plug-in.

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.

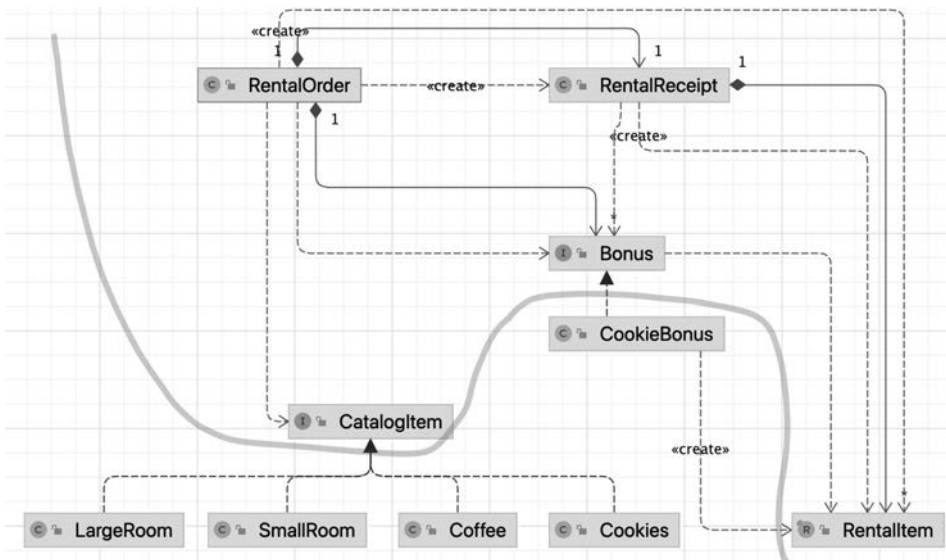
So with those changes in place, here's the directory structure:



The diagram is shown below. 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> bonusItems);
}
```

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 *Projectors*, 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 type,
                                              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.getReceipt();
    assertEquals(1, receipt.length);
    assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 1, 100, 100, 5);
    assertTotalAndTax(100, 5);
}

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

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

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

@Test
public void twoSmallRoomsForOneDay() throws Exception {
    order.rent(SMALL_ROOM, 1);
    order.rent(SMALL_ROOM, 1);
    RentalItem[] receipt = order.getReceipt();
    assertEquals(2, receipt.length);
    assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 1, 100, 100, 5);
    assertTypeDaysUnitTotalTax(receipt[1], "SMALL_ROOM", 1, 100, 100, 5);
}
```

```
        assertTotalAndTax(200, 10);
    }

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

    @Test
    public void oneSmallRoomCookiesCoffeeForFiveDays() throws Exception {
        order.rent(SMALL_ROOM, 5);
        order.rent(COFFEE, 5);
        order.rent(COOKIES, 5);
        RentalItem[] receipt = order.getReceipt();
        assertEquals(3, receipt.length);
        assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 5, 100, 450, 23);
        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 Exception {
        order.rent(LARGE_ROOM, 5);
        order.rent(COFFEE, 5);
        RentalItem[] receipt = order.getReceipt();
        assertEquals(3, receipt.length);
    }
```

```
    assertTypeDaysUnitTotalTax(receipt[0], "LARGE_ROOM", 5, 150, 675, 34);  
    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.

This page intentionally left blank

INDEX

NUMBERS

49FNGO, 22, 27, 34

A

Abstract Factory, 81, 143, 359, 360

abstraction

- algorithm proofs and, 512

- code expressivity and, 336

- comments and, 583–584, 587, 590

- data structures and, 251–252

- defined, 583

- function names and, 146, 242–243

- Hexagonal architecture and, 469

- for high-/low-level separation, 332

- homogenous functions and, 151–152

- importance of, 583

- reducing duplication via, 403

- stable, 358–359

- Stable Abstractions Principle (SAP), 383–387

- The Stepdown Rule for, 140–142, 243, 245

acceptance testing, 313–316, 457

accessors, 82, 252, 255, 256

accidental vs. essential duplication, 177, 338

accountability, programmer, 495–496

ActiveX, 368

Acyclic Dependencies Principle (ADP), 374–378

adapted servers, 431

aerospace catastrophes, 491

affinity, conceptual, 126–127

Affordable Care Act, 496

Agile development, 290, 291, 315, 540, 558

Agile Software Development, Principles, Patterns, and Practices, 111

AI (artificial intelligence)

- editing help from, 40, 210, 287

- fear of job theft by, 319

- how to train, 286–287

- limits of, 69, 287, 328

- need for programmers and, 488

- productivity acceleration via, 560

- prompt-based programming, 319–328

- requirements specification for, 2–3

- See also* prompts, AI

airline catastrophes, 491

ALGOL, 214, 485

algorithms

- comments to clarify, 587–589

- proving correctness of, 512

- understanding, 27, 571, 596, 597

Algorithms + Data Structures = Programs, 249

aliases, namespace, 79–80

alignment, horizontal, 129

analog-to-digital converters (ADCs), 465, 466

Ant, 119

Apache Commons, 110

A Philosophy of Software Design, 242, 562, 603, 604

APIs

- boundaries in undefined, 473–474

- documentation for public, 97, 111

- Swing API, 299

- testing API, 307

- third-party, 471

APOSD Google forum, 561

applications

- app info via names, 73–74
- dependency management in, 378
- UI/application boundary, 466–474
- use cases, 477–478

architectural boundaries

- change at, 474
- context and, 145
- Dependency Rule for, 208, 227, 459, 476–480
- DIP and, 356–361
- hardware boundaries, 298
- hiding side effects with, 181
- IDE horizontal limits, 128
- independent deployability via, 68
- known/unknown separation, 473–474
- OCP and, 234, 347–351
- overview of, 453–459
- protecting axes of change, 207, 346
- separating abstract from concrete, 360
- separating high-/low-level elements with, 58, 468
- third-party IoT software and, 462–466
- UI/application boundary, 466–474
- See also* design; structure

architecture

- architectural boundaries, 453–459
- axes of change in, 346
- clean, 475–481
- component cohesion, 370, 373
- DIP and, 356–361
- elements supported by, 449–452
- Hexagonal architecture, 469–470
- keeping options open in, 446–447, 451–452
- OCP and, 347
- plug-in, 456–457
- purpose of, 443, 459
- separation of concerns in clean, 476, 575
- SOLID principles of, 270, 271, 339, 341–361

A Relational Model of Data for Large Shared Databases, 250

arguments

- empty list of, 159–160
- first arguments, 160
- formatting separate, 128
- functions and, 148, 151, 153, 159–164
- name length and, 79
- overloaded constructors and, 82

Arrange/Act/Assert (AAA) pattern, 306, 310, 314

arrays

- cell classes vs., 74
- variable names and, 76

assembly language programming, 129

assignment statements

- formatting, 129–130
- pure functions as free of, 153–154

attributes, naming, 79

audience, code, 72, 87

authorship statements, 93

Automatic Computing Engine (ACE), 484

automation

- concurrency support via, 436
- consistent formatting via, 118

B

Backus, John, 485

bad comments, 97–114

banners, 107

BASIC, 80

beans, 258

Beck, Kent

- code expressivity for, 337
- design principles of, 3, 270, 271, 331, 332, 339
- “First, make it work, Then, make it right”, 6, 20, 186, 247, 503, 506
- refactoring example by, 111
- testing disciplines of, 138, 291, 292, 337
- YAGNI and, 333

behavior-driven development (BDD), 314, 324

behavior of software, 445, 503–509

binary function libraries, 355, 367

BitKeeper, 523

blank lines/space, 120, 128

blocks

- duplicate loop reduction via, 175
- indentation of, 130–131
- managing memory, 178
- small functions in, 139

bonus code, 58

Booch, Grady, 7, 42

Boolean expressions, 162, 405

boundaries. *See* architectural boundaries

Boundary–Control–Entity (BCE)

architecture, 475

bound resources, 429
 Boyce, Raymond, 250
 Boy Scout Rule, 529
 Boy Scouts of America, 15, 529
 branches vs. toggles, 525–526
 browsers, component caching by, 68
 bugs. *See* debugging
 build, continuous, 316, 528
 bundling test approach, 293, 604, 605, 607, 610, 611–612
 business analysts (BA), 313–316, 527
 business rules, 60, 234, 357, 454–456, 468, 476, 477

C

C#

argument prefixes, 161
 constructors, 82
 evolution of, 250, 318
 ISP and, 355
 keyword parameters, 82
 naming properties in, 79
 private variable placement, 123–124
 this argument, 160

C++ programming language

constructors, 82
 CQS and, 165
 evolution of, 250, 318
 exceptions and, 165
 inventor of, 6
 scissors rule, 123
 TemplateMethod pattern, 175
 this argument, 160
 variable-naming rule and, 79

calling subroutines, 138

care, taking, 7–8

catalogItem type, 52, 54

catch block, 77, 98, 104, 105, 167

change(s)

architecture to reduce need for, 347
 boundaries as location of, 474
 class design and, 269–274, 284
 code fragility and, 48
 constants, 199
 continuous, 390–391
 data structures, 62, 176, 207
 goal changes, 451
 inevitable code, 91, 104, 118
 keeping options open for, 446–447

learning tests and anticipation of, 472–473

OO/procedural trade-off, 260–261, 264–266

Open–Closed Principle for, 48–49

passing tests and, 297–298

recompilation and, 58

rigidity and fragility from, 262, 505

software flexibility for, 445, 500–501

SOLID and toleration of, 342

stability and resistance to, 379

startups and frequency of, 504

state changes, 177

TDD and small, verified, 463

tunable threaded code, 434

See also growth capacity

checkForIllegalPrefixCombinations

function, 32

checkForImproperRepetitions functions, 32

chopped-up functions, 26, 52

clarity

cohesion via, 419

comments for, 95

formatting for, 118

four Cs of design and, 392–397, 418

names for, 73–74, 76, 83

classes

concurrency support via, 429

creating fields in, 160

creating new, 273

design best practices, 269–285

enumerators to, 52

extraction and, 235–239

vs. files, 267–268

formatting declarations of, 130

function multiplication vs., 215

instance variable placement for, 123

large functions with, 219, 225, 234

methods of derived, 146

naming, 79, 81–82, 83, 84

organization of, 207, 268

placing functions into, 192–193

relationships between, 204

size of, 271–273, 284–285

SOLID principles and, 341

spelling inconsistencies for, 75

Clean Agile, 543

Clean Architecture, 208, 329, 332, 341, 363, 443, 505

clean code

- care taken via, 7–8
- clean architecture, 475–481
- clean boundaries, 474
- clean classes, 270–285
- clean structure, 504–506
- clean tests, 299–300, 303–312
- consistent standards for, 28
- continuous improvement for, 15
- defined, 1, 6–9
- ease of reading, 8, 14, 572
- ethical oath for, 492–556
- importance of, 2–3, 10, 13
- meaningful context in, 85
- naming tips for, 72–86
- one thing rule, 6, 42, 211, 212, 271
- perfect code vs., 13
- platform-independent, 17
- productivity boost via, 10–11, 13
- relentless improvement in, 529–531
- size of elements, 42, 339
- small functions in, 138
- vs. unclean, 269
- workability and then, 6, 37, 142, 184

Clean Code, first edition, 437, 558, 559, 562, 566, 576, 593

Clean Craftsmanship, 292, 298, 311, 329, 331, 483

cleaning code

- AI support for, 37, 40
- bug detection via, 24
- comments vs., 104–105
- continuous improvement via, 531
- example of, 188–204
- fear of, 297, 407
- historic process of, 186
- how to, 187
- large function division in, 234
- number conversion example, 20–40
- order of operations, 32
- perspective shifts for, 24
- principles of, 41–69
- speculative cleanup, 278
- testing and, 187, 188–204, 210, 222, 297–298

client-based locking, 431

Clojure

- clean code in, 17
- extraction and classes in, 238–239

function calls in, 216

inverted order in, 244

keyword parameters, 82

side effects and, 179

clutter, comments as, 101, 102

COBOL, 214, 325, 485

Cockburn, Alistair, 469, 475

Codd, Edgar, 250

code

Beck's law for, 6, 20, 184, 186, 247, 503, 506

comments to describe, 107–108, 587–589

concurrent, 424

consistent formatting of, 118

context provided by, 73

do no harm with, 495–502

English language vs., 578–579

executable vs. structure of, 48

failures of, 90, 92, 97, 576

fragility of, 48

increasing complexity of, 45–46

inevitable change of, 91, 104, 118

instrumentation of, 435, 442

intimacy with details of, 92, 582, 588, 596, 598

maintenance vs. fixes to, 12

moving comments into, 580

nonthreaded, 433

one thing rule for, 6, 42, 211, 212, 271, 343

ongoing need for, 2–3

precision of, 102

reliable information via, 91, 115

repeatable proof for, 511–518

requirements-based fixes, 12

size of elements, 42, 271

standard of cleanliness via, 28

standards document, 28, 133

startup/shutdown, 432

test coverage for all, 333–334

tests for flexible/reusable, 300–301

verified via comments, 586, 590

written for your audience, 72, 87

YAGNI principle for, 56–57

codebase community, 72, 87

code smells

defined, 270

design perspective of, 271

- feature envy, 276, 277
- Primitive Obsession, 84
- Code That Fits in Your Head*, 8
- cognitive load
 - complex structure and, 68
 - deep methods and, 563
 - pronunciation and, 76
 - unreasonable, 574
- cohesion, design, 401, 416–420
- collaborative programming, 37, 539–540, 553
- collections, thread-safe, 429
- colloquialisms, 83
- colors
 - comment display, 90
 - syntactic element clarity via, 75
- COLT (Central Office Line Tester), 545
- Command pattern, 174
- commenting-out code, 107–108
- comments
 - abstraction and, 583–584, 587, 590
 - assessing helpfulness of, 39, 576–591
 - bad, 97–114
 - comprehensibility of, 92, 110
 - editing, 111–114, 593–596
 - good, 92–97
 - hidden/obscured, 90–91, 93, 582, 587
 - historical, 109–110
 - inaccurate, 91, 94, 95, 99, 579–580, 586
 - interface comments, 584, 586–587, 590
 - language for, 578–579, 590
 - limits of, 89, 275
 - long names vs., 580–582
 - low-level technique of, 37
 - missing vs. incorrect, 579–580
 - necessary evil of, 90, 577
 - revision vs., 89, 395
 - usefulness of, 89, 114, 578, 597, 614
 - verification of, 586
- Common Closure Principle (CCP), 346, 369, 370–371, 372, 373
- Common Reuse Principle (CRP), 356, 369, 371–372
- compilers/compiling
 - context provision for, 84
 - encoding types and, 80
 - errors and recompilation, 168
 - history of, 365, 485
 - linking loaders for, 367–368
 - name checking by, 54
 - spelling inconsistencies and, 75–76
 - testing and, 187
- completion, automatic code, 75
- complexity
 - comments to explain, 590, 593, 596, 601
 - function, 148
 - name length and, 580–582
 - software design, 562, 569
 - splitting methods and, 572
 - structure and added, 68
- component caching, browser, 68
- components
 - abstract, 383
 - boundary lines between, 454–456, 459
 - Common Closure Principle for, 370
 - component cohesion, 368–373
 - coupling, 373–387
 - defined, 364
 - history of, 364–368
 - main sequence and position of, 387
 - stability metrics for, 380–381
 - See also* modules
- composed assertions, 201, 223, 307
- composed test results, 307
- comprehensibility
 - blank lines for, 120
 - code changes and, 118
 - comment, 91, 92
 - CQS (Command Query Separation) for, 164
 - understanding algorithms, 27, 571, 596, 597
- computers, history of, 484–486
- concepts
 - conceptual affinity and formatting density, 126–127, 128
 - consistent names for, 83–84
 - editing, 234
 - grouped into single constructs, 267
 - separating problem/solution, 84
 - vertical openness between, 120–121
- conciseness, design, 398–406, 419
- concrete components, 358–361
- concurrency
 - C++ and, 165
 - challenges of, 424, 425–426
 - decoupling via, 424
 - IoT system, 468, 471
 - myths about, 425
 - principles for efficient, 427–437
 - reasons for, 424–425

- temporal coupling and, 178
- terminology, 429–430
- Concurrent Programming in Java*, 429
- confirmability, design, 406–416, 419
- conjoined functions, 563
- constants
 - appropriate level for, 126
 - changing, 199
 - coupling type constants, 230
- constructors, 82
- containing class, 94
- context
 - contexts as nouns, 184
 - functions in, 144–145, 151
 - providing, 73, 84–86
- continuous build, 316, 528
- continuous deployment, 527
- continuous design, 277, 316, 389–422
- continuous integration, 523–525
- control flow, 183, 276, 278
- convenient functions, 147–148
- convert function, 34
- convertLettersToNumbers function, 27
- Copilot, 20, 24, 210
- Coplien, James O., 423, 475
- copyrights, 93
- CountDownLatch, 429
- couplings
 - arguments as, 160
 - components, 373–387
 - functions, 148–151
 - premature decisions and, 453
 - temporal, 177
 - type field/type constant, 230
 - See also* decoupling
- C programming language
 - clean code in, 17
 - contexts in, 145
 - first arguments, 160
 - functions in, 214
 - history of, 486
 - output arguments in, 162
- CQS (Command Query Separation), 164–165
- critical sections, 427, 432, 434, 565
- culture, names and shared, 87
- Cunningham, Ward, 8, 147, 241, 457, 558, 559
- CVS (Concurrent Versions System), 522, 523, 524, 525

D

- Dahl, Ole-Johan, 144, 249, 318, 486
- Data, Context, and Interaction (DCI)
 - architecture, 475
- databases
 - architectural boundaries in, 454–456, 458, 479–480
 - boundary crossing in, 479
 - hierarchical elements in, 446–447
 - object-oriented, 259
 - relational databases, 250–251
 - static, 260
- data integrity, 437, 439
- data structures
 - abstraction and, 251–252
 - architectural drawing of, 349
 - changes to, 62, 176, 207
 - complex, 175
 - contexts and, 145
 - DTOs (data transfer objects), 258
 - hiding internal, 207
 - higher-level technique of, 37
 - hybrid object/data structure, 256
 - Law of Demeter and, 255–258
 - loop duplication and, 174
 - objects vs., 252–255, 258–266
 - one thing rule for, 343
 - Open–Closed Principle and, 48
 - privacy concerns, 256, 259
 - relational database, 250–251
 - simple, concrete, 61
 - switch statements and, 48
- Date, Chris, 250
- deadlines, making, 4–5, 11
- deadlock, 430, 431
- debugging
 - AI prompts, 328
 - cleaning to detect bugs, 24
 - comments, 108, 598
 - requirements-based fixes, 12
 - speed of, 535
 - TDD and need for, 197, 605, 606
 - temporal coupling and, 178
 - testing and reduced, 294
 - threaded code, 433, 435
- decisions, deferred, 453, 458
- declaring variables, 122–124
- decomposition, 563–564, 566–567, 575, 576

- decoupling
 - concurrency for, 424
 - dependencies, 374–387
 - locks/critical sections, 565
 - testable code as decoupled code, 334, 605
 - tests from production code, 191, 204
 - See also* couplings
- deep methods, 563
- defaults, locating comments on, 109
- delete function, 167
- deletions, double-, 440
- dependencies
 - architectural, 459
 - component, 371
 - concrete vs. volatile, 358–359, 379
 - decoupling, 374–387
 - dependency diagrams, 57, 58
 - dependency inversion, 61, 69, 360
 - Dependency Rule for, 476–480
 - designing for appropriate, 207–208, 468, 470, 476, 505
 - eliminating dependency cycles, 374–378
 - error codes and, 168–169
 - isolated tests, 310
 - pure functions as free of, 153
 - source code, 348, 349–350, 355, 357–358, 360
 - synchronized method, 431
 - transitive, 351
- Dependency Inversion Principle (DIP)
 - architecture and, 459
 - boundary crossing and, 479
 - conformity to, 64, 264
 - designing for, 207
 - eliminating dependency cycles and, 376
 - high-/low-level separation and, 332
 - overview of, 343, 356–361
 - performance and, 265
 - violations of, 50, 262
- Dependency Rule, 208, 216, 361, 459, 476–480
- dependent functions, vertical placement of, 125–126
- deployability, independent, 68, 364, 546
- deployment, program, 313–316, 398, 449, 451, 527, 535
- derivatives
 - enumerations as, 52
 - instantiated, 64
 - low-level techniques in, 54, 64
 - switch statements to create, 144
- descriptive functions, 145–147
- design
 - abstraction in, 583
 - axes of change in, 207
 - build speed and, 534
 - class, 204–207, 269–271
 - continuous, 277, 316, 389–422
 - ease of comprehension and, 572
 - eliminating untestable code in, 298–299
 - evaluating ideas for, 562
 - focus on priorities in, 601
 - four Cs of, 392–420
 - growth capacity in, 58
 - ISP and, 355–356
 - LSP and, 352–354
 - modular design, 563
 - OCP as driver of, 351
 - OO/procedural trade-off, 258–265
 - OO vs. data structure, 252–255
 - OO vs. relational database, 250–251
 - perspective shifts for, 605
 - principles of simple, 331–339, 392
 - project development process, 420–422
 - simple, 331–339
 - TDD and inspection of, 604, 605, 607–610, 613
 - test design, 189, 191, 204, 311
 - testing disciplines and, 293–294, 296
 - time-honored patterns, 287
 - See also* architectural boundaries; structure
- Design Patterns*, 8, 176
- details, software, 446
- development process
 - architecture in, 449, 451
 - change frequency for startups, 504
 - design in, 420–422
 - honest/fair estimates, 420–421, 543–551
 - overly tactical, 607–608, 611
- digital switching, 546
- digraph processing, 32, 34
- Dijkstra, Edsger, 181–182, 485, 486, 511–514, 517
- dining philosophers problem, 431
- directional control, 350
- directory structures
 - architectural boundaries in, 58, 68
 - clear organization in, 52

- dependencies and, 57, 58
- enum to interface changes in, 52
- disinformation, spelling variations as, 74–75
- dispatch, polymorphic, 231–233, 254
- distractions, managing, 535–538
- documents
 - code standards document, 28, 133
 - comment, 97, 579
- domain-specific test language, 307–310
- drivers, frameworks and, 478
- DRY (Don't Repeat Yourself), 258
- DTOs (data transfer objects), 258–264, 479
- duplication
 - accidental vs. essential, 177, 338–339, 344–345
 - copy-paste programming, 270
 - DRY (Don't Repeat Yourself), 169–170
 - loop duplication, 174–176
 - principle of minimizing, 337–339
 - reducing, 37, 402–404
 - similar code repetition, 170–174
 - See also* redundancy

E

- editing code
 - AI help with, 40
 - annoyances in, 27
 - clarity via, 395
 - comments vs., 89
 - concurrency and multiuser editing, 439
 - editing tests, 188–204, 222–223
 - fear of, 407, 607
 - functionality vs. improvement, 20
 - inevitability of, 183–184
 - manual, 521–522
 - member distinctions and, 81
 - name changes in, 234
 - number conversion example, 20–40
 - PrimeGenerator edits, 591–596, 599–603
 - readability via, 26–27
 - reading vs. writing times and, 13–14
 - time allocated for, 24
 - writing as including, 87
- Eisenhower, General Dwight D., 506–507, 610
- Eisenhower's decision matrix, 506–507
- else statements
 - extracting, 213
 - small functions in, 139
 - unnecessary, 405–406
- Emacs, 13
- emotional stress, 537
- encapsulation of functions, 148, 181, 225
- enclosing scope, 160
- encoding
 - encoded names, 80
 - language support for, 80
- English language, 578–579
- entanglement, 142, 217, 332, 563, 571, 574, 576
- entities, 477
- enumerations
 - abandoning, 54
 - algorithm proofs and, 512
 - checking for, 52
 - enumerators as instances, 49
 - enumerators to classes, 52
 - enum to interface changes, 52
 - goto statements and, 515
 - replacing old, 68
- errors
 - AI, 321–322
 - catastrophic software, 491–492
 - code instrumentation to check for, 435–436
 - error codes, 163–164
 - error handling as higher-level, 37
 - errors as dependency magnets, 168–169
 - exceptions vs. error codes, 165–167
 - formalized language to avoid AI, 324, 328
 - jiggling code to find, 436–437
 - repetitive error handling, 402–403
 - social effects of software, 496
 - SRP for error handling, 168
 - try/catch block error name, 77
- essential duplicates, 177
- estimates, project, 420–421, 543–551
- example code, 606
- exceptions, 165–167
- exclusion, zones of, 385–386
- executable code, 48
- execution models, 429
- expressivity of code, 334–337
- extracted functions, 43, 46, 170, 172, 212–217, 225, 227, 235–239, 242, 246, 394, 395, 401, 564
- extracted methods, 275–276
- extracting implementation detail, 277
- Extract Method refactoring, 46, 212–217
- Extract Till You Drop, 37, 213, 225, 239, 275
- Extreme Programming, 8, 290, 291, 333

F

- Facade pattern, 345
- Fan-in/Fan-out metrics, 380–381
- Feathers, Michael, 7, 13
- fields, 230
- files
 - classes and modules vs., 267–268
 - source, 241, 268
 - vertical formatting by file size, 119
- finalizing bonus code, 58
- financial considerations
 - cost of object creation vs. locks, 428
 - cost per line, 186
 - honest/fair estimates, 543–551
 - programmer salaries, 487
 - project development and, 420–421
- finite state machine, 148–149
- Fit, 8
- FitNesse, 93, 95, 97, 119, 125, 133, 304, 314, 457–459, 525, 526
- FIT tool, 457–459
- flag arguments, 161–162
- flow state, 537
- fonts, syntactic clarity via, 75
- footers, 227, 228
- fopen function, 155–156
- for loops, 94, 213, 598
- formatting
 - communication via, 118
 - consistent rules for, 118
 - horizontal, 127–132
 - low-level technique of, 37
 - openness/density in, 120, 126, 128
 - purpose of, 118
 - separating business rules from, 234
 - team rules, 132–133
 - Uncle Bob’s rules on, 133–135
 - vertical, 118–127
- FORTRAN, 80, 138, 214, 318, 485, 514, 578
- Fowler, Martin, 37, 187, 219, 270, 271
- frameworks and drivers, 478
- fromRoman function, 20–40, 47
- functional decomposition, 142, 516–517
- Functional Design*, 153, 180
- functionality
 - harnessing design functionality, 563
 - increasing complexity, 45–46
 - one thing rule, 6, 42–43
 - prioritizing, 19–20, 37
 - pure function, 153
 - reliability vs., 295–296
 - workable code, 6, 37, 142, 184, 186, 506
- functional languages, 155, 177, 178–180, 267, 428
- functions
 - accessor, 82, 252, 255, 256
 - chopped-up, 26, 52, 566
 - classes in large, 219, 225, 234
 - comments describing, 109
 - CQS (Command Query Separation) for, 164–165
 - encapsulation of, 148, 181, 218, 225
 - entanglement of, 142, 217
 - extracted, 43, 46, 170, 172, 212–217, 227, 235–239, 242, 246, 394, 395, 401, 564
 - extracting try/catch blocks into, 105, 167
 - function arguments, 159–164
 - function call speeds, 216
 - function libraries, 365
 - functions as verbs, 184
 - headers for, 110
 - helper, 34, 394
 - history of, 138, 214
 - increasing complexity of, 45–46
 - indentation for short, 132
 - inlining, 216, 239
 - naming, 26, 27, 42, 76, 79, 145–148, 215, 242–243
 - one thing rule for, 42–43, 343
 - ordering, 26
 - organizing extracted, 216
 - placed into classes, 192–193
 - placement of dependent, 125–126
 - principles of clean, 144–157
 - private, 148, 242, 250
 - public, 242, 250
 - “pure”, 28
 - replacing comments with, 106–107
 - scope size and, 147–148
 - side effects of, 177–180
 - size of, 42, 213, 214, 215, 394, 563, 566
 - small, well-named, single-concept, 138–140, 184
 - SOLID principles and, 341
 - split to provide context, 85
 - The Stepdown Rule for, 140–142, 245–246

- switch statements and, 142–144, 264
- variable declarations and, 122

`f(x)` call, 160

G

garbage collection, 178, 428

gi (graphic interpreter), 6, 211, 215, 217

GitHub, 20, 235, 523, 525

Given-When-Then style, 315, 324

global functions, 148

global variables, name length for, 79

Go

- class/function design in, 228
- clean code in, 17
- errors in, 169
- evolution of, 250
- exceptions and, 165
- function arguments in, 160, 163
- try/catch block error name, 77

Golang code, 20, 139, 227

Goldstine, Herman, 269

good comments, 92–97

“Go To Statement Considered Harmful”, 182, 515

goto statements, 182, 183, 214, 515, 516

Grenning, James, 356, 461

Grok3

- code production by, 560
- editing help from, 37, 39, 40
- implementation improvement by, 47
- limits of, 69, 319–321, 328

growth capacity

- axes of change and, 207
- code duplication and, 169–170
- concurrency and, 425
- dependency structure, 377
- extractions for, 235–236, 237, 238–239
- objects, data structures and, 259–260
- structuring in, 46–48, 50, 52, 56
- See also* change(s)

guards, 166, 198, 611

H

hand coding, 435–436

hardware abstraction layers (HALs), 473

hardware boundaries, 298

harm, do no, 495–502

Harvard Mark 1, 144, 318

headers

- clean code and, 228
- function, 110, 142
- interface documentation and, 585

Healthcare.gov, 10, 492, 496, 544

“hello world” example, 463, 544

helper functions, 34, 394

Hexagonal architecture, 469, 475

hidden/obscured comments, 90–91

hierarchical elements

- classes/namespaces, 215
- class hierarchies, 94
- function call hierarchy, 142
- indentation to show, 130
- policies and details in, 446
- type hierarchy, 231

higher-level policies

- context to describe, 145
- dependencies for, 57, 58, 208
- Dependency Inversion Principle and, 50
- Dependency Rule for, 476–480
- dividing line for, 58
- element hierarchy and, 446
- extracted functions and, 216, 227
- function SRP and, 139–140
- isolating low-level details from, 54, 332, 357, 360
- module layout and, 243–245
- overview of, 37
- Stable Abstractions Principle and, 384
- The Stepdown Rule and, 243–244, 246

The Hitchhiker’s Guide to the Galaxy, 557, 558

Hollerith line limit, 128

homogenous functions, 151–153

honest/fair estimates, 420–421, 543–551

Hopper, Grace, 144, 317, 318, 485

horizontal formatting, 127–132

The Humble Object Pattern, 298

Hungarian notation (HN), 80

I

IBM, 485, 486

IDE (Integrated Development Environment)

- collapsing functions, 83
- comment hiding by, 90–91, 93, 582, 587
- dependency diagrams in, 57
- Refactor menu in, 212
- type encoding and, 80

ideas, eliminating poor, 31

if/else chains, 143, 213, 260–265, 336, 405, 516

if statements

- checking for enumerators with, 52
- code changes and, 261
- CQS in, 164
- extracting, 213, 217–218
- indentation for, 132
- refactoring, 107
- small functions in, 139, 564

implementation details, 274, 275, 277, 284, 286, 332, 360, 395, 396, 402, 403, 427, 473, 583, 599

Implementation Patterns, 3

importance vs. urgency, 506–507

improvement, continuous, 529–531, 555–556

indentation, 130–132, 139, 215

independent deployability, 68, 364, 546

induction, 512

informative comments, 93–94, 578

initialization, 566, 567

inlining functions, 216, 239, 567

inputs

- acceptance testing, 314
- arguments as, 162
- function, as couplings, 148

instance variables

- denoting close association for, 121–122
- enumerators as, 49
- function purity and, 28
- internal objects with, 34
- methods communicating via, 193
- name length for, 78
- placing declaration of, 123–124
- promoting local to, 218
- static vs., 238
- well-named, 26

instantiating derivatives, 64

instrumentation of code, 435–436, 442

insulated functions, 148–151

integration, continuous, 523–525

IntelliJ, 210

intention

- comments to explain, 92, 94, 275, 596
- context to describe, 145
- declarations for, 396
- function extraction to reveal, 227
- maximizing expression of, 334–337
- names to reveal, 72

organization of elements and, 216

testing, and AI assistance, 287

tests to reveal, 337

interfaces

- application-specific, 473
- architectural drawing of, 349
- changes from enums to, 52
- changes to abstract, 358–359
- deep method, 563
- Hexagonal architecture, 470
- interface adapters, 478
- interface comments, 584, 586–587, 590
- known/unknown boundary, 473
- LSP and, 352
- module, 242
- polymorphic, 357
- prefixing, 81
- Strategy pattern and, 208

Interface Segregation Principle (ISP), 342, 354–356

invalid tests, 30

IOException, 98

IoT (Internet of Things) software, 9, 462–466, 468, 470

ItemList, 57–58

ItemList.java, 53

iterations, 182, 573

J

Jacobson, Ivar, 475

jar files, 68, 71, 264, 368, 457

Java

- clean code in, 17
- components, 364
- constructors, 82
- context in, 145
- evolution of, 250, 318
- expressivity of, 335
- extraction and classes in, 238
- ISP and, 355
- Javadocs for, 97
- private variable placement, 123–124
- this argument, 160
- thread-safe collections, 429

JavaBeans, 82

Javadocs

- noisy, 105–106
- nonpublic code with, 111
- redundant, 99–102

- required, 102
- useful documentation via, 97
- JavaScript, 17, 50, 68, 418, 419
- JCommon library, 42
- JDepend, 119
- Jennings, Jean, 485
- jiggling code, 436–437
- journal/log comments, 102–103
- JUnit 4, 96, 119, 124, 126
- JVM startup times, 210
- K**
- Kay, Alan, 249, 250
- Kerievsky, Josh, 87
- Kernighan, Brian W., 89
- keywords
 - keyword arguments, 161
 - keyword parameters, 82–83
- Knight Capital, 10, 491, 497–498, 499
- Knuth, Don, 570, 587, 593
- L**
- lambdas, 174–176, 397
- Langr, Jeff, 1, 89, 267, 389
- languages
 - classes/modules vs. files in, 267
 - code as truth-telling, 115
 - components by language, 364
 - documentation systems for, 97
 - encoding type systems, 80
 - expressivity of, 335
 - failures in, 90
 - formalized AI, 324, 328
 - functional language, 178–180
 - history of, 214–215, 249–250, 318–319, 485
 - human vs. programming, 578–579, 590
 - ISP and, 355
 - learning multiple, 555
 - nouns and verbs in programming, 184
 - static and dynamic typed, 355
- Laszczak, Robert, 235
- Law of Demeter, 255–258
- Law of Least Astonishment, 163
- learning, continuous, 555–556
- learning tests, 470–473
- legal comments, 93
- libraries, 365, 442, 470
- licensing, 93
- lifecycle, system, 443
- lines
 - cost per, 186
 - readability and blank, 120–121
 - vertical density of, 121
 - width of, 127
- linking loaders, 367
- Liskov, Barbara, 342, 351
- Liskov Substitution Principle (LSP), 342, 351–354
- list, use of term, 74
- “Literate Programming”, 570
- livelock, 430, 431
- LLMs (large language models), 322, 324, 328
 - See also* AI (artificial intelligence)
- loading order, 441
- local variables
 - name length for, 78, 79
 - promoted to instance variables, 218
- locks
 - adapted server locks, 431
 - atomic pop function via, 165
 - client-based locking, 431
 - element selection and, 439
 - finding locations for, 442
 - object creation cost vs., 428
 - ReentrantLock, 429
 - SCCS, 522
 - server-based locking, 431
- logs, comments as, 102–103
- loops
 - comments on limits for, 114
 - control variable placement for, 123
 - duplication of, 174–176
 - extracting for, 97
 - indentation for short, 132
 - writing and refactoring, 185–186, 187–188
- low coupling of functions, 148
- lower-level techniques
 - business rules in, 62–64
 - context to describe, 145
 - dependencies for, 57, 58, 208
 - Dependency Inversion Principle and, 50, 54, 64
 - Dependency Rule for, 476–480
 - extracted functions and, 216, 227
 - function SRP and, 139–140
 - growth capacity in, 64

- hierarchical placement of, 446
 - isolating high-level from, 54, 332, 357, 360
 - module layout and, 244, 245
 - overview of, 37
 - plug-ins for high-level, 57
 - The Stepdown Rule and, 244, 246
- M**
- main sequence, 386
 - maintenance
 - architecture and ease of, 443
 - clean code and ease of, 195
 - comment, 91, 582
 - cost of, 153
 - dependency structure and, 378
 - file size and, 119
 - readable code and ease of, 118
 - makeStatement function, 227, 242, 244
 - management
 - programmer tussles with, 508–509
 - speaking truth to, 4–5
 - test, 299
 - time pressures from, 11, 503, 549–551
 - Marick, Brian, 501
 - marking positions, 107
 - Martin, Justin Michael, 557
 - Martin, Robert (Uncle Bob), 71, 117, 133–135, 561
 - max function, 198
 - meaning, comments and clarity of, 97–98, 99
 - meaningful name distinctions, 75–76, 84–86
 - meetings, 536
 - memory
 - historic speeds for, 366
 - meaning, comments and clarity of, 309
 - side effects, 178
 - speed/clarity balance, 27–28
 - trading speed for, 153
 - memory items, 294
 - merges, 522, 524
 - methods
 - formatting declarations of, 130–131
 - instant variables to communicate, 193
 - Law of Demeter and, 255
 - length of, 563–576
 - naming, 82, 93, 581, 585
 - side effects in private methods, 181
 - subdividing, 592
 - synchronized, 431
 - Meyer, Bertrand, 347
 - microservices, 424
 - mid-level components, 342
 - Minecraft*, 368, 489
 - Minesweeper*, 73
 - mocking, 606, 607
 - modern environments
 - automatic code completion in, 75
 - concurrency support in, 437
 - exceptions in, 165
 - modular design, 144, 214, 563, 575
 - modules
 - vs. files, 267–268
 - history of, 144
 - how to write a clean, 188–204
 - interface of, 242
 - Law of Demeter and, 255
 - length of, and namespace names, 80
 - naming, 83, 243–244
 - one thing rule for, 343
 - recompilation of, 50
 - size of, 214
 - SOLID principles and, 342
 - structure of, 37
 - See also* components
 - Mojang, 489
 - Monotone, 523
 - mood and stress, 537
 - multithreaded environments, 238
 - music, writing with, 536
 - mutation testing, 530
 - mutators, 82
 - mutual exclusion, 429
 - MySQL, 457, 458
- N**
- names
 - algorithm comprehension and, 571
 - arguments in function calls, 161
 - avoiding colloquial/slang, 83
 - checking for correct, 54
 - comments vs. long, 580, 585
 - consistency auditing for, 74, 83
 - documentary value of, 139
 - editing and changes in, 234
 - encoding in, 80
 - function, 26, 27, 42, 76, 79, 145–148, 215, 239, 242–243
 - importance of good, 58, 71–72

- instance variables, 26
- intention revealed by, 72
- length of, 73, 580–582
- letters/numbers in, 75
- low-level technique of, 37
- meaningful context for, 84–86
- naming tips by code element, 78–83
- noise words in, 76
- order of elements and, 58
- problem/solution domain, 83–84
- pronounceable, 76–77
- renaming, 87, 147, 236
- scope size and, 78–79, 147
- searchable, 77–78
- Singleton pattern, 93
- subjectivity of, 72
- syntax for, 81
- test, 190, 223
- well-named elements, 42
- namespaces, 79–80, 196, 215, 267
- Nassi-Schneiderman flowchart, 183
- nesting
 - duplicate loops, 175
 - escaping from nested loops, 602
 - functions holding, 139
- .NET, 364
- networks
 - browser component caching on, 68
 - historic network speeds, 523
- Newkirk, Jim, 470
- noise comments, 103–105
- noise words, 76
- number conversion code, 20–40
- numbers
 - letter/number name combinations, 75
 - number-series naming, 75
 - searchable names with, 77
- numbers.add, 27
- Nygaard, Kristen, 144, 249, 318, 486
- O**
- oath for software ethics
 - continuous learning, 493, 555–556
 - defect-free behavior/software, 49, 503–509
 - do no harm, 492, 495–502
 - high productivity, 492, 533–538
 - honest/fair estimates, 493, 543–551
 - relentless improvement, 492, 529–531
 - repeatable proof, 492, 511–518
 - respect for fellow programmers, 493, 553
 - small cycles/releases, 492, 519–528
 - teamwork, 493, 539–541
- Objective-C, 250
- Object Mentor Inc., 71
- object-oriented (OO) programming
 - Abstract Factory in, 360
 - arguments in, 160
 - classes vs. files in, 267
 - contexts in, 145
 - data structures vs., 253
 - DTOs and, 263
 - history of, 249–250
 - LSP and, 352
 - OO/procedural trade-off, 264–266
 - performance and, 265
 - side effects in, 180–181
- objects
 - Abstract Factories and, 359, 360
 - creating, 27, 428
 - data structures vs., 252–255, 258–266
 - higher-level technique of, 37
 - hybrid object/data structure, 256
 - instance variables and, 34
 - Law of Demeter and, 255–258
 - object creation in IoT system, 468
 - passing arguments into, 160
- one thing rule
 - clean code and, 6, 211
 - decomposition and, 564–565, 566, 576
 - defining, 211, 212
 - error handling and, 168
 - flag arguments vs., 161
 - functions and, 42, 46, 139, 143, 157, 213
 - going too far with, 271
 - names and, 110, 147
 - switch statements and, 142
- Open–Closed Principle (OCP)
 - changes to old modules and, 49
 - class design and, 272, 278, 284
 - example of, 234, 264
 - extreme expression of, 284, 285
 - overview of, 342, 347–351
 - switch statements and, 48, 265
 - system design and, 287, 339
 - violations of, 143, 261
- open/virtual offices, 540–541
- operating systems, thread policies of, 434

operation, supporting system, 449, 450, 451
 operators, accentuating precedence of, 128
 optimistic locking, 523
 options, kept open, 446–447, 451–452
 organization of elements
 argument order, 160, 161
 classes for, 207
 containing class hierarchy, 94
 dependent function placement, 125
 directory structures and, 52
 entanglement and, 571, 574
 extracted function by call order, 216
 functions in, 138
 growth capacity and, 46–48, 50, 52
 intent revealed by, 216
 key principle of, 42
 message-passing concept and, 250, 258
 naming and, 58
 organization of classes, 268
 politeness and, 243–244
 SOLID principles and, 341
 temporal coupling and, 178
 ORMs (object-relational mappers), 259
 Ottinger, Tim, 71
 Ousterhout, Dr. John Kenneth, 27, 142, 217, 242, 271, 293, 561
 outputs
 acceptance testing, 314
 arguments as, 162–163
 function, as couplings, 148

P

parametric (table driven) tests, 203
 parent-child relationships, 441
 Parnas, David, 145
 pass/fail tests, 311
 Pauling, Linus, 31
 PDP11s, 522
 peer to peer systems, 524
 PEP 8 standard, 189
 performance. *See* speed
 perspective shifts, 24
 pessimistic locking, 522
 pipelines, functional, 396, 397, 398
 PL/1, 138, 163
 Plaugher, P. J., 89
 pluggable threaded code, 434
 plug-ins, 57, 456–457
 policies and details, software, 446
 polite code, 228, 241, 243–244
 polymorphic methods
 clean switch statements via, 143, 144
 enumerator checking via, 53
 for high-/low-level separation, 332
 OO vs. data structures and, 253
 optimization time, 68–69
 polymorphic dispatch, 208, 231–233
 polymorphic interfaces, 357
 Pomodoro Technique, 538
 pop function, 165
 Ports and Adapters, 469, 475
 position marking, 107
 Powell, Robert Baden, 529
 precedence of operators, 128
 predicates, 82, 275–276
 prefactoring, 278
 PrimeGenerator function, 110, 111, 114, 568, 569, 570, 572, 574, 576, 580, 584, 587, 591–596, 599–603, 614
 Primitive Obsession, 84
 principles
 architectural separation of concerns, 476
 cleaning code principles, 41–69
 component cohesion, 368–373
 component coupling principles, 373–387
 concurrency, 427–437
 Dependency Rule, 208, 361, 459, 476–480
 Extract Till You Drop, 37, 213, 225, 239, 275
 “First, make it work. Then, make it right”, 6, 20, 184, 186, 247, 503, 506
 function, 144–157
 idea evaluation principles, 562–563
 indentation, 130–132
 living and breathing the, 558–559
 one thing rule, 6, 42, 211, 212, 271, 564
 OO programming, 258
 politeness, 228, 241, 243–244
 small, well-named, organized, 42–44
 SOLID, 270, 271, 341–361
 The Stepdown Rule, 140–142, 151, 243, 245–246
 YAGNI (You Aren’t Going to Need It), 56–57, 333
 See also SOLID design principles
 privacy
 constructor, 82

- OO vs. data structure function privacy, 256, 259
 - private functions, 148, 242, 250
 - private variable placement, 123–124
 - side effects in private methods, 181
 - probability distributions, 548, 549
 - problem domain names, 83, 84
 - procedural code, 255, 264–266
 - producer-consumer threads, 430
 - production code
 - clean tests for, 300
 - decoupling tests from, 68, 191, 204
 - increasingly generic, 197
 - table-driven, 203
 - test timing and, 311
 - writing tests before, 291
 - productivity
 - clean code to increase, 10–11, 13
 - comments and lost, 590
 - principles driving, 559–560
 - programmer oath for, 492, 533–538
 - remote work and, 540
 - programmers
 - AI and changes for, 488
 - care taken by, 7–8
 - catastrophic errors by, 492
 - code authoring by, 72
 - continuous learning by, 555–556
 - deadline pressures on, 4–5, 503
 - distraction management for, 535–537
 - errors by, 10
 - ethical oath for, 492–556
 - financial compensation for, 487
 - functional, 28
 - gender demographics of, 486–487
 - global dependence on, 9, 490–491
 - good, 2
 - history of, 485–487
 - pop culture notoriety of, 488–490
 - respect for fellow, 553
 - responsibilities of, 9–10, 40
 - skills necessary for, 40
 - unethical, 490, 495
 - programming
 - acceptance testing in, 313–316
 - AI and prompt-based, 319–328
 - catastrophes, 491–492
 - collaborative, 539–541
 - comments and failures of, 90, 97
 - concurrent, 424
 - continuous evolution of, 487
 - defined, 2
 - ethical oath for, 492–556
 - functional language, 178–180
 - hand coding, 435–436
 - history of, 317–318, 483–491, 519–524
 - increasing complexity of, 214–215
 - modular, 214
 - OOPs. *see* object-oriented (OO)
 - programming
 - OO vs. data structures in, 252–255
 - precision in, 102
 - “purity” of functional, 155
 - software development process, 420–422
 - Structured Programming, 181–183
 - TDD style of, 290–292
 - technical names in, 84
 - unethical, 495
 - programs. *See* software
 - project development, 420–422
 - prompts, AI
 - code as basis for, 2–3
 - debugging, 328
 - formalized language for, 324, 328
 - prompt-based programming, 319–328
 - pronounceable names, 76–77
 - properties, naming, 79
 - public functions, 242, 250
 - punch cards, 519–523
 - purity of functions, 153–156
 - Python
 - abstract components in, 383
 - clean code in, 17
 - concurrent programming in, 471
 - dynamic typing in, 208
 - evolution of, 250, 319
 - function arguments in, 160, 163
 - IoT (Internet of Things) system using, 463
 - ISP and, 355
 - keyword parameters, 82
 - meaningful context for, 85
 - plug-in support, 210
 - try/catch block error name, 77
- Q**
- quadratic formula solution, 20
 - queue function, 471

R

- race conditions, 95, 165
- RAII (Resource Acquisition Is Initialization), 178
- random delays, 95
- RCS (Revision Control System), 522, 524
- React Native, 437, 438, 440, 441
- readers-writers, throughput and, 430
- reading code
 - blank lines for, 120
 - comment verification via, 586
 - CQS (Command Query Separation) for, 164
 - denoting close associations for, 121–122
 - direction of, 120
 - ease of, 14, 42, 87, 228, 243–244, 247, 393, 572
 - editing and, 13–14
 - encoding types and, 80
 - extracted functions and, 216
 - first vs. edited drafts, 26–27
 - indentation for, 130–131
 - readable code as maintainable, 118
 - readable tests, 303, 305–306, 308
 - reading/writing differences, 246–247
 - The Stepdown Rule, 140
 - understanding via, 597
 - well-written prose, 7, 139, 157, 214, 335
- recompilation
 - changes and complete, 58
 - DIP and, 50
 - eliminating need for, 64
 - growth and forced, 62
 - module, 50
 - reducing burden of, 52, 54
- recursion, 182, 183
- redeployment
 - changes and complete, 58
 - eliminating need for, 64
 - growth and forced, 62
 - JavaScript, 50
 - reducing burden of, 50, 52, 54
- Red-Green-Refactor cycle, 507, 517, 559, 604, 607
- redundancy
 - comment, 98–102, 104, 105, 109, 585
 - DRY (Don't Repeat Yourself), 169–174
 - name, 76, 86–87
- Reenskaug, Trygve, 475
- ReentrantLock, 429
- Reeves, Jack W., 389
- Refactoring, 85, 187, 219, 256, 270
- refactoring
 - defined, 187
 - delayed, 607
 - example of, 111–114
 - Extract Method refactoring, 46, 212–217
 - function call hierarchy and, 142
 - loop, 187–188
 - passed tests and, 68
 - unit testing and, 607
 - See also* testing
- Refactoring*, 37
- reformatting, 129
- regular expressions, 93–94
- related concepts, denoting, 121–122
- relational databases, 250–251, 259, 438
- relationships
 - inheritance/implements, 455
 - OCP and unidirectional, 349–350
 - parent-child, 441
 - proximity and close, 574
 - reciprocal, 204
 - theory and feeling of, 388
- remote procedure call (RPC), 463, 464
- remote work, 540
- renaming elements, 87
- repeatable tests, 310
- requirements specification, 2–3, 12
- respect for fellow programmers, 553
- responsibilities
 - isolating categories of, 48
 - programmer, 9–10, 40, 491–492, 495–496, 498
- REST interface, 352, 353
- Reuse/Release Equivalence Principle (REP), 369–370, 372, 373
- revision. *See* editing code
- Rochkind, Marc, 522
- Roman numeral conversion code, 20–40
- row structures, 479
- Ruby
 - abstract components in, 383
 - clean code in, 17
 - components, 364
 - evolution of, 250, 319
 - function arguments in, 161, 163
 - ISP and, 355
 - keyword parameters, 82
- r-values, formatting, 129–130

S

SCCS, 522, 524

Schuchert, Brett L., 423

scissors rule, 123

scopes

concurrency and limited, 427, 442

enclosing scope, 160

encoded into names, 80

hierarchy of, 130

indentation of, 130, 132

 name length and scope size, 78–79,
 147, 581

size of, 581

searchable names, 77–78

Seeman, Mark, 8

selections, 182

self argument, 160, 193

self-validating tests, 311

semantic stability, 530

semaphores, 95, 165, 178, 181, 429

Sammelweis, Ignaz, 5

sequences, 182

server-based locking, 431

services, 424

servlet model, 424

shared data

avoiding via copies, 428

integrity of, 437, 439

problems/solutions with, 437–441

updates, 427

See also concurrency

Shotgun surgery, 270

shutdown code, 432

side effects, 177–180

Sieve of Eratosthenes, 594, 596

sigmas, 154–155, 548

SimpleDateFormat, 94, 96

simplicity, 331–339

Simula language, 249, 318, 486

Single Act Rule for tests, 309–310

single-letter names, 77–78

Single Responsibility Principle (SRP)

accidental vs. essential duplication and, 177

architectural support for, 451

class design and, 271, 272, 282–284, 285

Common Closure Principle and, 370

concurrency and, 427, 442

designing for, 207

DRY (Don't Repeat Yourself) and, 169–174

error handling and, 168

examples of, 48

functions following, 139, 147

module layout and, 244

overview of, 342, 343–346

recompilation and, 50

refactoring and, 46

simple designs with, 339

 violations of, 51, 143, 261, 273, 344–345,
 382, 419, 571

single-threaded systems, 424–425

Singleton design pattern, 93, 281

size of code elements, 42

Skia, 440

slang names, 83

slogging/wading in code, 4

Smalltalk, 8, 250, 268, 285

SNOBOL, 522

software

acceptance testing of, 313–316

architecture, 443

behavior and structure in, 445, 504

building blocks, 391

catastrophes, 491–492

changeability of, 500–501, 531

components, 364–368

development process, 420–422

effects of erroneous, 9–10

growth capacity of, 50

increasing complexity of, 45–46

iterative naming for, 87

mathematical structure of, 484, 512

policies and details in, 446

third-party, 462–466

ubiquitous nature of, 9, 490–491

unethical, 495

See also programming*Software Development* magazine, 111

SOLID design principles

class quality and, 270

dependencies eliminated by, 505

 Dependency Inversion Principle (DIP),
 343, 356–361

design perspective of, 271, 339

Hexagonal architecture and, 469–470

 Interface Segregation Principle (ISP), 342,
 354–356 Liskov Substitution Principle (LSP), 342,
 351–354

- Open–Closed Principle (OCP), 342, 347–351
- Single Responsibility Principle (SRP), 342, 343–346
 - See also individual principle by name*
- solution domain names, 83–84
- source code
 - attributions/bylines in, 107
 - consistency in, 133
 - continuous build and, 316
 - control systems, 103, 108, 519–524
 - dependencies, 348, 349–350, 355, 357–358, 360
 - editors, 338
 - indentation and hierarchy in, 130
 - vertical formatting of, 118
- Source Forge, 523, 525
- speed
 - complex structure and slowed, 68–69
 - concurrency and, 425
 - debugging, 535
 - development, 503–504
 - function call, 216
 - historic network, 523
 - improving build, 534
 - OO designs and, 265
 - speed/clarity balance, 27–28
 - speed of functions, 216
 - test, 210, 310, 534–535
 - trading speed for memory, 153
- spelling
 - clarity via consistent, 74–75
 - compilation errors and, 75
- SQL, 332, 457, 476, 478
- Stable Abstractions Principle (SAP), 383–387, 459
- Stable Dependencies Principle (SDP), 379–383
- stakeholders
 - isolating responsibilities to, 48
 - programmers as, 507–508
- standard deviations, 548, 549
- standards document, 28, 133
- startup code, 432
- startups, software, 501
- starvation, 429, 430
- state changes, 177
- state machine strategy, 31
- Statement class, 45
- Statement.java, 53
- Statement module, 49
- statements
 - comments as unnecessary for, 400
 - following The Stepdown Rule, 142
- State pattern, 284
- static convert function, 28
- static databases, 260
- static initializers, 96
- static typing, 54, 355
- static variables, 238
- statistical analyses, 117
- status values, 73
- The Stepdown Rule, 140–142, 151, 243, 245–246
- Strategy pattern, 174, 208, 278, 280, 281, 338
- stress, 537
- String.format function, 161
- strings, static type safety vs., 54
- Stroustrup, Bjarne, 6, 211, 249
- structure
 - axes of change in, 207
 - comments vs. improved, 105
 - complexity added via, 68
 - concurrency and, 424
 - elements of good, 504–506
 - function arguments, 159–164
 - growth capacity and, 47–48, 50, 52, 58
 - harm to, 499–500
 - horizontal formatting, 127–132
 - intent revealed by, 216
 - limiting functions with identical, 143
 - loop duplication in, 174
 - names to clarify, 58, 76, 84
 - no defects in, 503–509
 - showing hierarchical, 130
 - Singleton design pattern, 93
 - software value of, 445
 - vertical formatting, 118–127
 - See also architectural boundaries; design*
- Structured Programming, 142, 181–183, 486, 514–517
- styles, syntactic element clarity via, 75
- subjectivity, name, 72, 87
- subroutines
 - changes to, 545
 - history of, 138, 144, 269, 337–338
 - inline encoding of, 137
- subtypes, 351

Subversion, 523, 524, 525

surprise, 163

surviving mutations, 530

Swing, 299

Swing programs, 564

switch statements

checking for enumerators with, 52

OCP violations and, 48

optimization time, 68–69

tips for clean, 142–144

type code and, 231

when to use/not use, 260–265

synchronized keyword, 427, 431–432

system of names, 73–74

system requirements, 313–314

T

table-driven tests, 203

tables, naming, 76

tactical programming, 607–608, 611

tasks

conceptual affinity between, 126

identified via function names, 242–243

task swapping, 434

TODO comments in backlog of, 106

TCR (test && commit || revert), 289,

292–293, 501

TDD

alternatives to, 604

author's introduction to, 138

benefits of, 605

certainty via following, 501

debugging and, 197

design choices and, 607–609, 613

failure/passing order in, 189

higher-level policy of, 37

history of, 289

living and breathing, 558–559

repeatable proof via, 517–518

small, verified changes in, 463

three laws of, 603

teams

cleanliness standards for, 28

consistent formatting by, 118, 132–133

maintenance vs. improvement, 12

teamwork oath, 539–541

TemplateMethod pattern, 174, 176, 338

temporal coupling, 177

Teradyne, 545

testing

100% coverage by tests, 333–334, 415, 529–530

acceptance testing, 313–316, 457

AI prompts, 327

cleaning and, 187, 188–204, 210, 222, 297–298

clean tests, 299–300

code expressivity and, 337

confirmability via, 406

decoupled from production code, 68

design flaws in, 191

documentation via, 295, 395, 605, 606

domain-specific language for, 307–310

F.I.R.S.T best practices tool, 310–311

Grok3 test improvements, 39

hand coding in, 436

impracticality of, 298–299

instantiating derivatives and, 64–68

invalid condition tests, 30, 40

learning tests, 470–473

merging functions in, 34

mutation testing, 530

need for further, 28

pure functions, 153

quick, sure, and repeatable, 518

readable tests, 303, 305–306, 308

reducing test noise, 201

refactoring and, 68, 187–188

role of design in, 422

Roman numeral conversion test, 22–23

semantic stability via, 530–531

Single Act Rule for, 309–310

small unit tests, 285–286, 300, 406–415, 603, 605, 606

specificity of tests, 197, 210

speed of, 534–535

SRP classes, 272

TDD approach to, 189

test data patterns, 37

test design, 311

test names, 190, 223

test resilience, 210, 229

test suite, 297–298, 300

third-party code, 470–473

threaded code, 432–433

turning off test cases, 96

well-specified requirements for, 2

See also refactoring

- testing disciplines
 - benefits of disciplines, 294–299, 501
 - design and, 293–294, 296
 - requirement to follow, 501–502
 - small bundles, 293, 604, 605, 607, 610, 611–612
 - TCR, 289, 292–293, 501
 - TDD, 37, 138, 189, 197, 289–292
 - See also* TDD
 - TestNG, 119, 123
 - Therac-25 radiation therapy machine, 491
 - third-party frameworks
 - boundaries in, 462–474
 - impracticality of testing, 298
 - IoT (Internet of Things) system, 462–466
 - testing, 470–473
 - UI/application boundary in, 466–474
 - this argument, 160
 - threads
 - decoupling via concurrency, 424
 - independence of, 428
 - locking, 565
 - nonthreaded code and, 433
 - OS and threading policy, 434
 - parallel rendering of, 440
 - pluggable threaded code, 434
 - race conditions and, 95, 165
 - separating safe/unsafe multithread
 - actions, 438
 - spurious failures as issue with, 433
 - testing, 432–433
 - tunable threaded code, 434
 - thread-safe collections, 429
 - Tichy, Walter, 522
 - Time and Money, 119
 - time constraints, 4–5, 11, 503, 549–551
 - time management, 538
 - timestamps, 93–94
 - TMI (too much information) comments, 109–110
 - TODO comments, 106
 - toggles vs. branches, 525–526
 - Tomcat, 99, 119
 - Toyota software catastrophes, 492, 498, 499
 - trainwrecks, 256, 257
 - transaction isolation, 438, 439
 - transitive dependencies, 351
 - traversal code, 175, 338
 - trim function, 96–97
 - try/catch blocks
 - error name in, 77
 - extracted into functions, 105, 167
 - noise comments in, 104
 - Turing, Alan, 483–484
 - type code, 230–231
 - type encoding, 80, 81
 - types
 - design changes and new, 264–265
 - hierarchy of, 231
 - multiple types in single file, 268
 - subtypes and, 351
- ## U
- UI/application boundary, 466–474
 - unclean code
 - business effects of, 3, 10–13
 - challenges of, 1, 392
 - clean vs., 269
 - harm of, 495–502
 - production of, 4
 - project pressures and, 4–5
 - reading/understanding, 26–27
 - resisting slide toward, 14–15
 - slogging/wading in, 4
 - slowed production via, 5
 - TDD and unclean code, 607–608
 - transformation of, 6
 - understanding code
 - editing and, 26–27
 - small/well-named/organized elements, 42
 - Unified Modeling Language (UML), 272, 273
 - unit testing, 285–286, 300, 406–415, 524, 603, 605, 607, 612
 - Unix, 522
 - urgency vs. importance, 506–507
 - use cases, 449, 450, 451, 477–478
 - user interface
 - hiding switch statements in, 144
 - name prefixes for, 81
 - tests in, 222
 - UI/application boundary, 466–474
 - UI isolation, 468
- ## V
- valid tests, 30
 - variables
 - declaring, 122–124
 - denoting close association for, 121–122

- extraction and, 238–239
- in-function, 219
- functional languages vs., 179
- modification of, 154
- naming, 76, 78–79, 85
- prefixing member, 81
- replacing comments with, 106–107
 - short names in local, 78
- variadic arguments, 161
- VAXes, 522
- vertical formatting, 118–127
- virtual offices, 540–541
- viscosity, 533–535
- visibility, comment, 90–91, 93, 582, 587
- volatility, isolating, 358–359, 378, 386, 504
- Volkswagen EPA testing scandal, 490
- von Neumann, John, 269, 485

W

- warnings, comments as, 96
- We, Programmers*, 249, 317, 617
- Wheeler, David, 269
- while loops, 132, 213
- while statements, 139, 564
- Wiki, invention of, 8, 558, 559
- Wilkes, Maurice, 269

- Windows C API, 80
- Wirth, Niklaus, 249, 515
- workable code, 6, 37, 142, 184, 186
- writing code
 - AI support in, 40
 - comments/names, 27
 - editing and, 13–14, 20, 24, 87, 183
 - historic process of, 185–186
 - listening to music while, 536
 - reading/writing differences, 246–247
 - role of functions in, 184
 - workable code writing, 186
 - writing tests before, 291, 415

X

- Xeno's paradox, 12
- XP Immersion, 111

Y

- YAGNI (You Aren't Going to Need It), 56, 57, 332, 333

Z

- zeroth subscript, 73
- Zone of Pain, 386
- Zone of Uselessness, 386