The Addison-Wesley Signature Series

A Mike Cohn Signature Book

Essential Test-Driven Development

Rob Myers



Foreword by Richard Sheridan

FREE SAMPLE CHAPTER |

Praise for *Essential Test-Driven Development*

"Rob Myers's modern, practical take on TDD gets your hands dirty quickly while keeping them safe with clear rules, smart safety nets, and hard-earned wisdom from decades of real experience. Engaging, entertaining, and relevant to every developer."

—Jeffrey J. Langr, veteran software developer and author

"After decades of TDD, Rob delivers what we've needed—a book treating test code as a first-class citizen. His insights on test smells and AI-assisted coding give a glimpse into our future where clean test specifications replace architectural debates. Multi-language examples show TDD's universal principles. A must-have reference."

—Shane Duan, Director of Engineering, Rockfish Data, Inc.

"*Essential Test-Driven Development* is a must-read for anyone who cares about writing quality, confident code. I've seen Rob's approach in action and witnessed firsthand the remarkable impact effective TDD can have. His passion, clarity, and practical guidance make this book an invaluable resource for any serious software professional."

—Jorgen Hesselberg, author of Unlocking Agility*;*
co-founder of Comparative Agility

"Do you want to feel confident while you develop? Do you want to earn the trust of your colleagues and demonstrate trustworthiness to them? Here's how, from someone who has been doing it for a quarter century."

—Kent Beck

"Easily the best book on TDD I've seen in over a decade. Myers avoids the empty sloganeering that treats TDD as an aftermarket bolt-on to old-school approaches, and his design detours take the reader deep into the meat of the subject."

—GeePaw Hill, independent software development technical coach

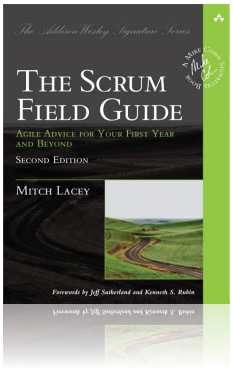
"Rob brings Test-Driven Development back to what it truly is: a craft that connects thinking, quality, and joy. This book inspires developers and teams alike to embrace sustainable technical excellence as the foundation for meaningful, agile co-creation. Highly recommended."

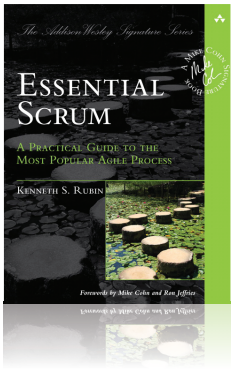
—Björn Jensen, CST, CTC, CEC & AKT

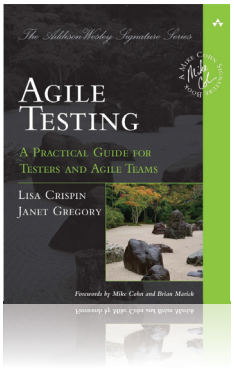
"TDD was a steep learning curve for my teams. And once learned it gave us the unicorn magic. I love how this book lowers that learning curve. You can easily work through it with Rob as your virtual pair."

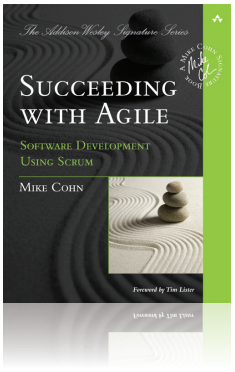
—Lisa Crispin, consultant and author

Pearson Addison-Wesley
Signature Series









The **Pearson Addison-Wesley Signature Series** provides readers with practical and authoritative information on the latest trends in modern technology for computer professionals. The series is based on one simple premise: Great books come from great authors.

Books in the Mike Cohn Signature series are personally chosen by Cohn, a founder of the Agile Alliance and highly regarded Agile expert and author. Mike's signature ensures that he has worked closely with authors to define topic coverage, book scope, critical content, and overall uniqueness. The expert signature also symbolizes a promise to our readers: You are reading a future classic.

Visit **informit.com/awss/vernon** for a complete list of available publications.

Pearson

informIT®

Essential Test-Driven Development

Rob Myers

Addison-Wesley

Hoboken, New Jersey

Cover image: Bernd Schmidt/Shutterstock

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: 2025949034

Copyright © 2026 Robert C. Myers

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: 978-0-13-449415-9
ISBN-10: 0-13-449415-6

$PrintCode

To my late father, Bob Myers, who shared my interests in computers and science fiction. He introduced me to my first dial-up terminal, let me play the turn-based text games "Adventure" and "Super Star Trek" for many hours of many weekends, and showed me how to edit the code and run my own copies of those games.

And to my mother, Alice Myers, who patiently tolerated two nerdy introverts while they spent whole weekends at the kitchen counter, with a dark green telephone cord stretched inconveniently across the entrance to the kitchen.

For their support and encouragement, and for setting me on this fascinating career path, I dedicate this book to them with my love, admiration, and gratitude.

This page intentionally left blank

Contents

Foreword . . . . . **xiii**
Preface . . . . . **xix**
Acknowledgments . . . . . **xxix**
About the Author . . . . . **xxxi**

Part I: Core Techniques **1**

Chapter 1: Thinking Test-Driven **3**
The Safety Net 4
Units of Behavior 4
Behavioral Boundaries 6
A Taste of TDD 9
The Future of Test-Driven Development 14
TDD and Behavior-Driven Development 14
TDD and Functional Programming 16
Daydreams of TDD, Quantum Computers, and No Implementation 17
The Reality of TDD, Artificial Intelligence, and "Vibe Coding" 18
Summary 20

Chapter 2: Basic Moves **21**
The TDD Flowchart 21
Thinking in Tests 23
Three Basic Techniques 26
Requests, Tasks, and a To-Do List 26
The Modest To-Do List 28
Getting Started 28

Exercise Walk-Through 30
Step 1: Write a Test for a New Behavior 32
Step 2: Make It Fail Cleanly 34
Step 3: Make It Pass 35
Step 4: Refactor 36
Repeat the Cycle 37
Next Moves 40
Summary 45

Chapter 3: Build upon Existing Behavior 47

Next Steps 47
Group Tests by Behavior 48
Fake It 49
Triangulate 49
Obvious Implementation 50
Refactoring from Obvious to Better 50
Improving the Tests 53
Invoke the Jetlagged Intern 53
Refactoring the Tests 55
Removing the Symptomless Defect 57
Triangulating Away Side Effects 60
A Picture Is Worth a Thousand Words 62
Combining Behaviors 66
Optimize for the Right Reasons 70
Summary 71
Key Takeaways 71
What to Try Next 72

Chapter 4: Exceptional Behaviors 73

Raising/Throwing 73
A Test That Passes When the Code "Fails" 75
External Exceptions 77
Summary 78

Chapter 5: Sustaining a Test-Driven Practice 79

Attributes of a Good Unit Test 80
Fast 80
Clear 81

Concise/Focused 82
Independent 82
Repeatable 83
Where to Start 83
CRC Cards 84
What to Test 88
Test Behaviors, Not Implementation 88
Test Behaviors, Not Data Structures 89
Test Behavioral Boundaries, Not Data 92
Test Only What Is Known 93
Zombies 94
When Code Is Difficult to Test 96
Test Smells and Refactorings 97
Smell: Unhelpful Test Names 99
Smell: Unhelpful Instance Names 101
Smell: Copy/Paste/Modify 102
Smell: Duplicate Given 102
Smell: Excessive Given 105
Smell: Rejected Given 107
Smell: Forced Return Value 108
Smell: Coupled Assertions 111
Smell: Extended Narrative 115
Smell: Unrelated Assertions 116
Smell: Unfailing Assertion 117
Smell: Redundant Assertion 118
Smell: Repeated Assertions 119
Smell: Branching Code 120
Smell: Calculations 121
Smell: Custom Parent Test Classes 122
Smell: End-to-End Tests 123
Smell: Tear Down 125
Smell: Frequent Changes to *Given* 125
Smell: For Testing Only 128
Smell: Intermittent Failure 129
Smell: Scattered Changes 131
Smell: Multiple Test Doubles 135
Smell: Exposed Implementation 136

Other Common Testing Challenges 136
How to Test a Private Method 136
How to Test an Abstract Class 136
How to Test a Method That Creates Its Own Dependencies 137
How to Test or Refactor Existing Untested Code 138
Summary 138

Part II: Ancillary Practices 139

Chapter 6: Test Doubles 141

The Trouble with Dependencies 142
A Test Double Taxonomy 143
Fakes 143
Stubs 144
Mocks 146
Spies 148
Nullable 152
Additional Recommendations 153
Favor Dependency Injection 153
Wrap External Dependencies 155
Use Simple Objects as Their Own Test Doubles 158
Summary: Use with Care 160
Be Sure to Deploy with Real Dependencies 160
Supplement with Narrow Integration Tests 160
Use Mock Frameworks Sparingly 161

Chapter 7: Testing Legacy Code 163

Characterization Testing 165
The Three Questions 166
Start Writing the First Test 168
Divide and Conquer 169
Build Tests Incrementally 173
Other Considerations 177
Test All Paths and All Results 178
Maximize Coverage While Minimizing Effort 178
Refactor Characterization Tests, Too 178

Recognize That This Is Not Test-Driven Development 179
Wear One Hat at a Time 179
When You Find a Defect 179
Introduce Virtualizing Proxy 180
Step 1: Create a Virtualizing Proxy 182
Step 2: Replace the Original with the Proxy 183
Using a Test Double of the Proxy in a Test 185
Summary: The Complete Toolbox 186

Part III: Return on Investment 189

Chapter 8: The Black Swans 191
The Agilist's Dilemma 192
Working Without a Safety Net 193
The Three Levels of Value 193
Level 1: A Dramatic Reduction in Defects and Rework 194
Level 2: Reduced Feature Time-to-Market 195
Level 3: Implementing the Unexpected 195
The Black Swan User Stories 195
A Major Architectural Change 196
OTIS2 200
The Loan Securitization Pipeline 202
Taming the Black Swans 204
Summary 205

Appendix: Exercises 207

Index 213

This page intentionally left blank

Foreword

Disillusionment

Early on in the pages that follow, author Rob Myers describes his mental state in 1997 when he was considering switching careers. He contemplated leaving software and going for his real estate license. He was falling out of love with a career he thought would carry him for a lifetime.

I know that feeling. I'm sure many of us in the software industry do. It's that moment where you look around and conclude *there has to be a better way*.

Like Rob, I developed a fascination with programming computers when I was just a kid. I chased that feeling through high school and college, and discovered I not only enjoyed programming but was pretty good at it. I later learned I would also be good at leading teams of people to write code together and create more complex systems. That's when the real trouble started.

I noticed some important patterns that would later haunt me and the teams that I led.

The first pattern was that most code could only be maintained by the person who originally wrote it. This silo at once became job security, yet also a prison from which there was no escape.

The second pattern was that programmers, in general, were terrible project managers, and even worse at organizing processes to keep teams of programmers working together cohesively for a long period of time. The biggest project management challenge was the inability to guess how long things were going to take, to the point that the management team paying for the work eventually had to sound the deadline siren, and call for round-the-clock coding to hit the release date. This crunch time inevitably led to the biggest nightmare a software team faces: the QA team finding a critical late-stage bug that had to be fixed before release. Yet, it could not be found, sometimes for months.

My team at Interface Systems in Ann Arbor, Michigan, faced such a nightmare scenario in the mid-1990s. It went on for several weeks as the programmers scrambled to find that elusive bug that absolutely could not be worked around. As the director of that team, I was called into regular update meetings with my bosses, who were frustrated and demanded a solution.

Finally, one morning, one of my long-time lead developers told me, as I walked by his office, that he had found and solved the problem. Eureka! I asked him how he did it. He mentioned that he was perusing some directories of files containing the code and stumbled upon a test bed he had written months before; he dusted it off and ran it. He said it exposed the problem immediately and took him right to the problem.

While I was thrilled to have this nightmare behind me, I knew it was only a temporary state of bliss, as there would inevitably be another monster bug hiding in the dark recesses of the code waiting to pounce at the most inconvenient time.

I, too, began plotting my escape. My version was starting a canoe camp in the boundary waters of Minnesota.

There has to be a better way, I thought.

This Isn't New

Rob writes that even 25-plus years after Kent Beck and others started espousing Test-Driven Development, unit testing, and automated unit test frameworks in the late 1990s, most still don't truly practice it. Even those who do still are in need of good coaching and discipline to bring it into the flow of standard coding activity.

My experience tells me it's been far longer since we actually knew what to do. I was at the University of Michigan pursuing degrees in computer science and engineering from 1978 to 1982. In the summer of 1980, I got a phone call from Professor John Saylor, who asked if I'd be interested in joining a cohort of Michigan computer science majors for a summer internship at a high-flying entrepreneurial firm in town called Manufacturing Data Systems, Inc. (MDSI). I jumped at the chance.

For the first four weeks of the internship, the entire group of eager students was put through an extensive and exciting boot camp to learn MDSI's approach to writing Pascal code. One of the key constructs Dr. Saylor taught us was that we would write tests *before* we wrote the code; we would catalog the tests, then write the code, and run the tests to determine how well we did with our coding. We would then pay as much attention to our test code as we would our production code. I honestly thought I had been given a key to a portal for learning truly modern ways of coding. I couldn't believe how lucky I was that I would be able to spend the summer refining this skill.

After this four-week session was completed, we interns were assigned to our various departments and bosses for the summer. I was assigned to Vince B. to work on the CODE and CAPP projects. As soon as I settled in to my new cube, Vince called me into his office, closed the door, and told me that I wasn't to use any of the techniques we had spent the last four weeks learning. He had a delivery style that made it

quite clear I wasn't to argue a case to try and change his mind. His basic message was "We don't have time for this bullsh*t; we've got deadlines."

The experience of those four weeks never left me—nor did the frustration of being denied the chance to use that experience. Later on in my tenure at MDSI, I had one chance to use these modern techniques with one other programmer. It resulted in the best code I had ever written professionally to that point. It was the only chance I'd had to apply Dr. Saylor's approach.

The Java Factory at Interface Systems

In 1997, I was promoted to VP of R&D at Interface Systems. Now, I believed, I had the perch from which I could start pursuing that *better way* I had always believed was possible. For the first two years, though, things improved only here and there by applying the standard approach: working harder.

In 1999, some important coincidences began to occur. I started working with a couple of consultants named James Goebel and Tom Meloche, who worked for a company that had consultants placed at Chrysler. Those consultants had the good fortune to brush up against another consultant who was bringing radical new ideas to the Chrysler Comprehensive Compensation system, after that system went through its own disastrous history of implementation failure. They needed a fresh, new approach. Kent Beck was apparently quite persuasive in convincing the management team there to try something new. The result was that Tom and James began teaching me about Extreme Programming, which included the concepts of writing tests before you write the code and then automating those tests so that they could be run over and over again as the coding progressed. I was reminded of my time with Dr. Saylor and my own experience.

I never looked back. The rest is history for me, and that history is well captured in my own book, *Joy, Inc.: How We Built a Workplace People Love*.

I eventually warmed to all the other ideas of Extreme Programming, especially pair programming. But it was TDD and automated unit testing that truly drew the honeybee to the flower. It eventually became the basis for launching my own company, with Tom and James, called Menlo Innovations.

What's in a Name?

The dot-com bubble burst in early 2001, and the dream at Interface Systems ended with a pop when the California company that had purchased it in September 2000

had to shutter every remote office it had, including mine in Ann Arbor. James, Tom, and I knew we had built a great engine room. Sadly, the first time we did it, we built it inside the *Titanic*. The fate of the *Titanic* was not caused by the engine room. We knew how to build a great engine room and we could do it again. Looking back on the experience now, we had the great fortune of having been able to build the prototype of Menlo Innovations inside of Interface Systems.

We started the company as most begin—in the basement of my home. The company's name was derived from my inspiration of seeing a replica of the Menlo Park, New Jersey, lab of Thomas Edison in Greenfield Village in Dearborn, where Henry Ford honored his friend Edison by reconstructing the lab.

James built the original website but did not include the address of the company in the website (my wife didn't really think it was appropriate to invite physical traffic to our home in a cozy Ann Arbor neighborhood). We did, however, include our cell phone numbers so that people could contact us.

Just a couple of weeks after launching the website, I received a call from someone who was passionately interested in what we were doing, especially since we were espousing ties to Kent Beck's thinking and Extreme Programming. He wanted to come work for us. He was in northern California. I explained that we were not in hiring mode quite yet, but promised him I'd be in touch. I couldn't imagine how someone in northern California would come work for us. (Remember, this was 2001.)

About a month later, he called again and was looking for an update. I explained that things were starting slowly, but I promised I would be in touch. He pressed harder and said: "Look, I'll just come down for lunch. I want to meet you guys."

Perplexed, I said, "Come down for lunch? What do you mean?"

"Menlo Park isn't that far for me."

I couldn't help but laugh out loud. I said, "Rob, we are in Ann Arbor, Michigan!"

He didn't hesitate. He said he would get on a plane. And he did.

Rob Myers commuted from California every two weeks for two years as he helped us rebuild the Organ Transplant Information System (OTIS) at the University of Michigan Health System. He joined our efforts to rebuild a critical piece of technology infrastructure at one of the leading organ transplant centers in the United States. When Menlo completed its portion of the work, there were almost 18,000 unit tests after three years of development. We had trained the Michigan team how to write tests and maintain existing ones. At the height of its operation, there were almost 30,000 unit tests faithfully being run each time the code was changed. That system stayed in operation for more than 15 years.

If we hadn't named our company Menlo Innovations, I might never have met Rob. We cemented our relationship in the years between 2001 and 2003, and we've stayed in touch ever since.

The Effect

Menlo started with all of these practices in place. We've never faltered. We worked on several systems (like OTIS) that quite literally hold the lives of people in the data and the algorithms—for example, a flow cytometer that is FDA-validated for monitoring the treatment of people with blood-borne cancers.

In all of our nearly 25 years to date, we have experienced exactly two software emergencies. Both were resolved within 24 hours. My life used to be firefighting every day.

I still don't see these techniques being systematically taught in university curricula. They're touched on, but not taught, as John Saylor taught me in 1980.

I first touched a computer to program it in 1971. I can say without a doubt that what you are about to learn—from one of the best teachers in this discipline—is the most revolutionary concept in computer programming ever discovered. Hands down.

In my early days, I believed there was a *better way*. There is, and you are about to learn all about it.

It is safe to say that it saved my career and brought me back to the joy of the profession that I had yearned for from my earliest days.

Thank you, Rob Myers, for taking a chance on a company that you thought was just down the road from you but turned out to be across the country. My life was enriched for having had you be such an important influence in our earliest days. I am forever grateful.

—Richard Sheridan
CEO, Co-founder, and Chief Storyteller, Menlo Innovations;
author, *Joy, Inc.: How We Built a Workplace People Love* (2013),
and *Chief Joy Officer: How Great Leaders Elevate Human Energy and Eliminate Fear* (2018)

This page intentionally left blank

Preface

Test-Driven Development (TDD) is a straightforward practice that provides software developers, teams, and organizations with numerous short-term and long-term benefits. It is the development practice of writing an automated test or "specification" for a software behavior just before writing the code that implements that behavior and makes that test pass.

TDD is often described as merely a way to produce software with fewer defects. What people often miss is that each existing test continues to play an essential role throughout the life cycle of the software. TDD not only catches most implementation mistakes instantly, but also leaves behind a representative guardian for each behavior. This safety net of existing tests allows developers to reshape the internal structure of the code without worrying about breaking another part of the system written months earlier. It gives them the confidence to make room for unexpected and innovative features. This ability to modify existing code without breaking anything has always been key to maintaining and enhancing software.

TDD is the combination of two earlier Extreme Programming (XP) practices. In 1996, Kent Beck[1] formulated XP, which provided a more tenable and sane approach to software development than many of the leading methods of the time. Two of the earliest practices in XP were (1) "test-first"—writing a unit test before writing the implementation that would make it pass—and (2) "refactoring"—reshaping the internal design of the resulting implementation. Beck saw value in combining and describing those two practices independent of XP, and in 2002 published *Test-Driven Development by Example*.

It's now more than 20 years later. Sadly, most software development teams still struggle with wholeheartedly embracing TDD—and still suffer the consequences of avoiding a test-driven approach.

Why This Book?

In 1997, I had been writing software professionally for about a dozen years, and I was sick of it. No matter how much upfront design or testing we did, the customer would eventually create or uncover a scenario that no one had anticipated. The code would

1. https://en.wikipedia.org/wiki/Kent_Beck

have to change, often dramatically, often wrecking our careful design considerations or creating a new and hidden defect.

So, in 1997, I was seriously pondering a career change. *"Perhaps real estate?"*

But in 1998, I had my first encounter with most of the practices described in this book. Suddenly programming made sense to me again! For the next six years, I had great fun writing the most well-designed and *most defect-free software of my career*, even when—*especially* when—the customers asked for something unexpected.

My personal mission in writing this book is a simple one: to help people craft quality software products and enjoy their careers. I wrote this book so the modern software developer can experience the *ease and joy* I've found using TDD.

The purpose of this book is to give software developers a set of simple—and *essential!*—practices they can use to build and maintain their code. At its core, TDD is truly simple. It gives you a clear set of steps through which you can channel all your creativity, talent, experience, and enjoyment of this craft. With these simple techniques, you can create, enhance, *and maintain* the most complex software imaginable.

Test-Driven Development in Brief

Test-Driven Development (TDD) is the practice of first considering what it is we want the software to do before deciding how we want the software to do it. We record each unique description of inputs and expected outcomes in an executable and repeatable format (a test) before writing the code that we presume will make the test pass.

TDD requires us to notice the thinking that must already occur before we can write any code and to preserve those thoughts in a readable and repeatable test suite. Its process is summarized in the following steps:

Step 1. Write a failing test. Write one test for a desired nonexistent behavior, run it, and ensure it fails in a predictable manner.

Step 2. Write just enough code to make that test pass. Write a minimal implementation that will pass the test without writing any more than is needed for this one test, and without breaking any previous tests.

Step 3. Refactor. Look for opportunities to clarify and simplify the structure of the code and the tests, and make those changes in such a way that all tests continue to pass. This step is ***not optional***. Often, refactoring helps make the next test easier to write.

You then return to step 1, gradually growing the behaviors, the structural design, and the suite of tests that protect those behaviors during all future changes.

This cycle is often abbreviated as "Red–Green–Refactor" or "Red–Green–Clean," as illustrated in Figure P.1.

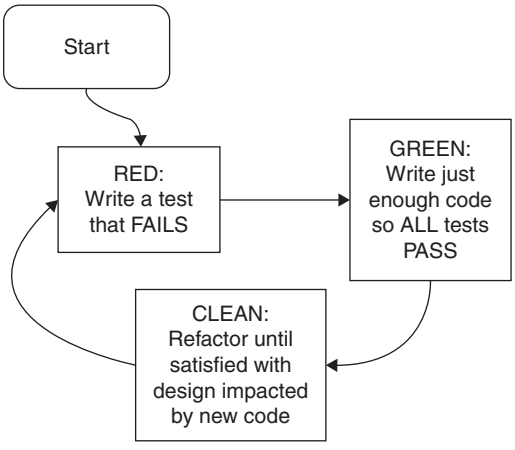


Figure P.1 *The simplest TDD flowchart, Red–Green–Clean.*

TDD can be summarized as this simple set of mechanical steps. However, when developers apply their problem-solving skills, creativity, and experience to TDD practices, they can build high-quality software with speed borne of confidence. With each iteration through the TDD cycle, they produce high-quality and well-designed software, and their growing suite of fast and comprehensive tests protects all prior investments in the solution.

For the team using TDD, the tests in a team's test suite are the result of all previously completed requests. The test suite serves as a detailed, up-to-date, and runnable engineering specification, describing precisely what the software currently does. A good engineering specification is clear, readable, and easily understood by both current and future developers.

Rather than having later requests become increasingly challenging, consume ever more time, and lead to building irritation among team members, with TDD each request can be met with the same confident enthusiasm the team would have given it had it been their first request on the very first day of a project. Moreover, because each new request can be completed without having to tiptoe over existing functionality, time-to-market is significantly reduced.

Terms, Themes, and Conventions

Here are a few clarifying definitions for terms that are used throughout the book.

Definitions

Implementation: The code that delivers valuable software behaviors. In contrast to test code, which tests those behaviors.

Refactoring: Any change to the software system that preserves existing overt behaviors and improves maintainability.

- *Overt behavior:* Behavior that is expressly described in the code, through loops, branches, method calls, and so on. A counterexample is speed of execution: Speed is an implicit behavior based on many complicated factors; no programming language has a `faster` keyword.
- *Maintainability:* A measure of how easily the software can be enhanced or repaired. It is a factor of both the code's design and the thoroughness of the team's test suite.

Design: The internal structure of the software's code; that is, how developers structure the implementation and how they choose to name the parts of that structure. Good software design clarifies and communicates how behaviors are implemented. A good design allows new functionality to be added without excessive rework.

I avoid using the term "design" when defining "refactoring" because both are often misunderstood to be topics that impact only developers, such that only developers would care about them. Maintainability, in contrast, is understood as how software grows in quality and value.

Request: Something the software development team is being asked to build that will incrementally increase the value of the software delivered. Agile methods have popularized the term "user story," meaning a very brief narrative describing an example of a user's interaction with the software to accomplish that user's immediate goal. A user story is typically much shorter than a "feature request," but I use the shortened "request" herein as an umbrella term for either.

Test: An automated script that defines and executes a behavior and checks that the results of that behavior are correct.

In this book, I will occasionally use "specification" or "scenario" instead of "test" whenever there is need to emphasize the test's role as *specification* (i.e., documentation) or as a *scenario* (i.e., an example).

Scenario: A single example of a behavior that includes what needs to be true before the behavior occurs and what the results should be once it is complete. Many scenarios make up a *specification*.

Specification: Documentation describing all detailed *requests*. For a complex product, there are often two specifications: the product specification and the engineering specification. There can be overlap between the two, but the engineering specification often covers technical details that do not need to be visible to a user, business analyst, or product manager.

Unit test: The term "unit test" has been around for decades, and people often assume that everyone knows what it means. Many modern developers dislike the term. Developers are often told "you must *unit-test* your code" as a requirement, and view it as a chore. Many earnest attempts to comply by testing individual methods or lines of code are likely to result in frustration and discouragement.

The industry has tried to solve the problem by changing the name. "Microtest," "spec," and "developer test" are sometimes used for this purpose, and they all express something true about these tests. For example, "spec" reminds us that we are writing a bit of engineering specification, and suggests that we may want to write it out before writing the implementation. "Developer test" conveys that these tests are written by developers, are read by developers, and exist primarily to support an ongoing development process.

The terminology isn't the problem; instead, the definition of "unit test" is the problem. It lacks clarity: What exactly is a *unit* of software?[2] A class, a function or method, a branch or loop, a line of code? All of those are the content and structure of the code, not what the code *does*.

I use "unit" throughout this book[3] to mean this: A *unit* of software is the smallest bit of *runtime behavior* that can be described independently. The implementation of a unit of *behavior,* as you'll see later, can span methods or can hide within part of a Boolean conditional clause.

Two important themes are baked into this book: *essentialism* and *emergence*. Rarely are they mentioned by name, but they serve as guiding principles.

2. "Microtest" suffers the same vagueness as "unit": How small is "micro," and what are we comparing it against?

3. I did run an informal poll on LinkedIn, and "unit test" won by a large margin, but the comments explaining my colleagues' various opinions were even more helpful.

Recommended Reading

McKeown, Greg. *Essentialism: The Disciplined Pursuit of Less*. New York: Crown Business, 2014.

Essentialism

The choice of the word "essential" in the title of this book is intentional. The practices herein are *essential*, meaning they are necessary. *Essential* also implies that there is nothing extra. In his book *Essentialism*, McKeown quotes Dieter Rams, the original industrial designer at Braun: "Less, but better."

How can you build valuable, quality software without causing your team to burn out? By using a practice that is, at its core, so simple that it stays out of the way, yet so powerful that it cuts away confusion, mistakes, and rework, replacing those with flexibility and confidence.

Emergence

> *"… a process whereby larger entities, patterns, and regularities arise through interactions among smaller or simpler entities that themselves do not exhibit such properties."*[4]

Most of the exercises and examples in this book come from the technical courses I teach, and every exercise is done using a test-driven approach and working in pairs. There could be 20 people in a class (that is, 10 pairs working separately on the same exercise), and I would see 10 perfectly acceptable but unique designs.

There is never one "right" design. Software design is highly contextual: Who is on the team, the experiences and education they have had, the technologies that are being used, and many other factors—all intertwine to make every software design unique.

A metaphor I use to describe what is called "emergent design" is the pruning and shaping of a Japanese maple tree. One cannot entirely decide up front what the tree will look like. You start with a little sapling, and the sapling, the soil, and the weather all have a say in how the little tree will grow up. You get feedback from the tree and work gradually with the tree, making small changes each season to reach a mutually

4. http://en.wikipedia.org/wiki/Emergence

acceptable aesthetic result. The discipline is a creative one that requires knowledge, and gets easier with experience.

Test-driven software development is like that: The tests and the implementation offer continuous feedback to each other. The developers who listen for that feedback fare better than those who try to force their preconceptions upon the software.

Who Is This Book For?

This book is designed to equip developers with the *essential* skills and techniques that can be applied to diverse business domains, programming languages, frameworks, and tools that they will encounter throughout their software development careers, including—and perhaps most urgently—modern AI agentic tools.

Practices and techniques tend to greatly outlive specific technologies. TDD—particularly the test-driven mindset and practices for maintaining a team's suite of clear, fast tests—is applicable whether you will be writing unit tests, Gherkin scenarios, or prompts for an AI agent.

I've worked with teams who have used the practices described in this book to build and maintain AWS Lambda functions; JavaScript using Promises, Angular, or React; node.js applications; Ruby on Rails apps; ASP.Net/ADO.Net applications; PL-SQL; and even assembler code.

I have endeavored to write examples that are understandable even when written in a programming language unfamiliar to the reader. Examples are in a variety of programming languages (mostly Java, C#, and Ruby) to convey (1) that TDD is an effective practice regardless of language, and (2) developers can comprehend code in unfamiliar languages if the tests and implementation are written clearly. I often chose the language that makes the example a bit clearer (e.g., C#'s override keyword makes an override obvious).

Book Organization

This book is arranged into three parts: Core Techniques, Ancillary Practices, and Return on Investment.

Part I: Core Techniques

- Chapter 1, "Thinking Test-Driven," provides an overview of the TDD discipline and an investigation into TDD's utility in a rapidly changing industry.
- Chapter 2, "Basic Moves," walks you through an example, demonstrating both the mechanics and the mindset that make up an effective TDD discipline. Part of that mindset is treating TDD as a cooperative win-win game that includes the development computer as one of the players.
- Chapter 3, "Build upon Existing Behavior," continues the example from Chapter 2 and goes deeper into the test-driven mindset and how to approach your implementation from a test-driven perspective.
- Chapter 4, "Exceptional Behaviors," briefly covers how to use TDD to design behaviors that throw exceptions or raise errors.
- Chapter 5, "Sustaining a Test-Driven Practice," describes common pitfalls and provides additional techniques for maintaining your test suite and implementation in the long term. A discipline that loses its effectiveness over time isn't worth starting, so I draw from the experiences of TDD teams that worked successfully together for six months or more. A catalog of common "test smells" is provided to assist you in maintaining your own test suites.

Part II: Ancillary Practices

- Chapter 6, "Test Doubles," covers a variety of techniques to substitute for troublesome external dependencies and greatly reduce the amount of time the test suite runs. Without some way to swap out external dependencies within your tests, your test suite will slow to a crawl, become an impractical burden, and eventually be abandoned.
- Chapter 7, "Testing Legacy Code," covers techniques to test existing untested code whenever it requires maintenance or enhancement. Testing legacy code does not have to be a horrible or futile effort. Practicing these techniques will also encourage and strengthen your TDD skills and will provide clarity on why TDD is the smarter "path of least resistance."

Part III: Return on Investment

- Chapter 8, "The Black Swans," provides the business case for using TDD. Three tiers of value are described: (1) fewer defects and much less time spent on debugging, (2) shortened "cycle time" (that is, time spent building a feature), and (3) greater adaptability to valuable, innovative, and unforeseen feature requests. This is a good chapter to share with a team's skeptics, including product managers, functional managers, architects, testers, and developers, regardless of their years of experience.

Register your copy of *Essential Test-Driven Development* 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/register and log in or create an account. Enter the product ISBN (9780134494159) and click Submit. 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

Acknowledgments

This book is far better thanks to the many people who reviewed, critiqued, and edited it.

To my early reviewers:

- Lisa Crispin, whose deep agile testing experience helped me clarify vague descriptions and examples.
- James Grenning, for offering very agreeable feedback and allowing me to share his ZOMBIESSS acronym.
- Jeff Langr, for going through all 75,000-plus words of the original draft. His thorough, detailed feedback encouraged me to streamline entire chapters and remove many "side-quests," reducing the size of the book by roughly a third. (Worry not! I didn't cut anything *essential!*)
- The small group of anonymous reviewers. Thank you, whoever you are!

To my infinitely patient executive editor, Haze Humbert, who set a daily cadence and weekly goals (that approach seems so familiar somehow). To my earlier editor, Chris Guzikowski, who greenlit this book ten years ago. To the expert editing team at Pearson, including Chris Cleveland, Julie Nahil, Jill Hobbs, Vani Rana, and Jayaprakash P (JP). If there are any errors remaining, they're all on me.

To those who worked with me while porting my training materials to another programming language or while updating examples in the book. Each encounter confirmed that the most fun way to learn a new programming language is test-driven pair-programming:

- Matt Hargett, who first converted my Java exercises to C++
- R. Jason Kerney, who paired with me using F# and proper functional programming
- Pat Maddox, an excellent Ruby developer and incredibly patient mentor
- William Pietri, who kindly helped me port everything to Python within mere days

- Lars Thorup, who pair-ported and co-taught several C++ and JavaScript TDD courses
- Ted M. Young, whose deep knowledge and experience saved my archaic Java examples from embarrassing obsolescence

To my programming mentors across almost 50 years (chronologically): dear old Dad, the late Larry Allen, Dr. Steve Wampler, the late *great* Dr. Melvin K. Neville, Fred George, and Dr. Bob Kass.

To the many friends and colleagues who helped me grow as a coach, instructor, and author: Scott Bain, Dale Emery, James Goebel, Peter Green, Bob Hartman, Elisabeth Hendrickson, Jorgen Hesselberg, Joshua Kerievsky, Richard Lawrence, Jeff McKenna, Alan Shalloway, Richard Sheridan, James Shore, Chris Sims, and Roshi Jisho Warner.

Finally, to the wonderful Operatives of Serendipity who made me the person I am today:

- Uncle Pete, who gave me his old science-fiction paperback collection when I was ten, rescuing me from despair and igniting a lifelong passion for all things nerdy.
- Dear friend Ted Levie, my first "pair-programming" partner circa 1985. We'd sit in the third floor NAU graphics lab until 3 A.M. drinking cheap, flavored instant coffee and attempting to solve Dr. Neville's absurdly challenging projects. Forty years later, we still have our hikes, mountain bike outings, and road trips filled with deep conversation (and better coffee).
- My husband Dave, without whom this book would not exist. No, really! Had I left Minneapolis as planned before we met, I would never have received Fred George's excellent XP training in 1998. Dave still patiently tolerates my enthusiastic banter—to anyone who will listen—about TDD, XP, and the pros and cons of AI-augmented coding.

About the Author

Rob Myers discovered his fondness for programming as a teenager, while using his father's portable terminal to study and modify mainframe computer games. His software development career began in 1986 as an intern in Phoenix before graduating from Northern Arizona University with a degree in computer science and engineering. He encountered Extreme Programming (XP) in 1998 and credits that event with rescuing his career from the typical product-development challenges of the day. In 2002, Rob began training and mentoring organizations in XP, TDD, and other modern development practices. He started his own training company, Agile Institute, in 2008, and has enjoyed teaching his fun, immersive, and extra-nerdy courses to hundreds of developers ever since, including his flagship course, Essential Test-Driven Development, which formed the basis for this book.

This page intentionally left blank

It is, as far as he knows, the only way of coming downstairs, but sometimes he feels that there really is another way, if only he could stop bumping for a moment and think of it.

—A. A. Milne, Winnie-the-Pooh

This page intentionally left blank

Part I

Core Techniques

This page intentionally left blank

Chapter 1

Thinking Test-Driven

> *Those who want really reliable software will discover that they must find means of avoiding the majority of bugs to start with, and as a result the programming process will become cheaper. If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.*
>
> —Edsger W. Dijkstra, *The Humble Programmer* (1972)[1]

The mechanics of Test-Driven Development (TDD) are simple, but TDD is not an invitation to disengage your intellect, nor does it require you to forget your own previous experiences and education.

People naturally talk about software behaviors using examples, though they aren't always aware that they're doing so. While working with a team, I will often hear someone offer a simple example in a verbal sentence or two. By capturing that example in a small bit of automated, executable specification (a test), the team can be sure of the following:

- The example doesn't get lost in translation later.
- Once complete, the implementation works correctly.
- The behavior captured by that example is never mistakenly broken during the entire life of the product.

For developers, many examples will occur to them while they are writing code, but that dialog is often internal: "What do I want my next few lines of code to do?" Before a developer writes even the tiniest bit of logic, they already know what they

1. *Communications of the ACM*, vol. 15, no. 10, Oct. 1972, pp. 859–866, https://doi.org/10.1145/355604.361591.

expect that code to do. TDD asks the developer to record that thought by turning it into a concrete description of the example and its expected results.

This book encourages a deeper dialog: "What if a caller does *this*? What do I want the software to do? How can I be sure I got it right, and that *I don't break something else*?"

Writing that thought in a familiar language and having it instantly added to an automated test suite costs the developer very little extra time and could easily save them hours upon hours of debugging later.

The Safety Net

The team's collection of fast, automated, and comprehensive tests that grow from a diligent TDD practice is much more than a typical suite of tests. For the test-driven team, the test suite is a powerful *safety net*.

Although each tiny individual unit test might seem quite simple by itself, in the aggregate that safety net protects the team's entire previous efforts and the business's investment in the software produced. Whereas individual tests instantly catch common mistakes in implementation, the comprehensive safety net also protects against mistakes during refactoring.

Units of Behavior

The following specification (in Ruby's rSpec) describes software that emulates a simple box. We can add items to the box, remove them, and verify whether the box contains a specific item. In each scenario, the tested behavior is emphasized.

```
it 'knows what has been added' do
  @box = Box.new
  @box.add("red pen")
  expect(@box.has("red pen")).to be true
end

it 'knows what it does not have' do
  @box = Box.new
  @box.add("red pen")
  expect(@box.has("purple smartphone")).to be false
end
```

```
it 'can remove something it has' do
    @box = Box.new
    @box.add("red pen")
    @box.remove("red pen")
    expect(@box.has("red pen")).to be false
end
```

Notice that `add()` cannot be tested without `has()`, and vice versa. That is, the *behavior* is spread out over two or three method calls on the same object. The *implementations* of these behaviors are encapsulated by the `Box` object, and the *specification* of each unit of behavior is clearly described by a single *scenario*. In other words, behavior is most clearly described in the tests.

The next example demonstrates how a single line of code can be part of multiple distinct behaviors. Imagine we are designing a retro-style volume control dial for a smartphone (Figure 1.1). The control can be set to only integer values, and our user interface (UI) developer tells us the dial will graphically "snap" to the next value as the user rotates the dial, perhaps with haptic feedback to emulate that memorable "click."



Figure 1.1 *A stereo system's volume dial circa 1980. (Photo: atm2003/123rf).*

In a brief whiteboard discussion, we decide we need a volume-control application programming interface (API) that accepts an integer from 0 to 10. Zero is effectively "off" and 10 is the maximum volume (not 11![2]).

The UI developer assures us that we will not receive an out-of-range integer. However, we agree that if this somehow happens, our code will throw an `IllegalArgumentException`. In Java, this could look like the following:

```
if (volumeZeroToTen > 10 || volumeZeroToTen < 0)
    throw new IllegalArgumentException("Volume cannot be " +
                                       valueZeroToTen);

audioManager.setStreamVolume(volumeZeroToTen);
```

There are three scenarios:

1. When `volumeZeroToTen` is greater than 10
2. When `volumeZeroToTen` is less than 0
3. When `volumeZeroToTen` is within the acceptable range

Each of those three scenarios is different because of the first `if` conditional, and each deserves its own distinct unit test. The behavior is more clearly described by expressly writing tests for those three examples.

Behavioral Boundaries

A *behavioral boundary* is the conceptual boundary separating two (or more) related scenarios. In practice, it is often the boundary between two similar input values that give different results.

Let's return to the volume dial example. Suppose that rather than having the dial's API set the actual phone volume (as previously shown), we decide to take an event-driven approach instead. An observer of a `Dial` object will need to query the setting whenever the graphical dial is turned. What the observer wants to know is not which value was set on the dial, but what percent of the maximum volume the setting represents.

2. https://en.wikipedia.org/wiki/This_Is_Spinal_Tap

A developer might be tempted to aim for the "happy path" (the scenario that typically avoids all edge cases and error conditions), write a simple implementation (for example, `return currentSettingFromZeroToTen * 10;`), and be done with it:

```
@Test
public void levelAsPercent_HappyPath_RightDownTheMiddle() {
    Dial dial = new Dial("Volume");
    dial.setTo(5);
    assertThat(dial.levelAsPercent()).isEqualTo(50);
}
```

This is a good start, but there is another way to approach this kind of problem, particularly when you are working with something more sophisticated than a volume dial.

TDD is first and foremost a practice that facilitates and supports a developer's thinking and coding. Writing one test scenario often leads to an awareness of other related scenarios. Writing tests for those related scenarios is known as the "triangulation" technique; it is covered in detail in later chapters. In its most refined form, triangulation helps the developer identify behavioral boundaries by asking, *"What value(s) will cause us to write smarter code?"*

Let's review what we know about `Dial`:

- Negative numbers should throw an exception.
- Numbers greater than 10 should throw the same exception with a different message.
- Zero is effectively "off."
- All other values represent a percentage of the maximum volume.

There is a behavioral boundary between –1 (an error) and 0 (off), and another behavioral boundary between 10 (maximum) and 11 (another error). However, there are no behavioral boundaries between any two of the integers between 1 and 10. The behavior—the complete code path—is the same for all: 1, 2, 5, 6, 9, and 10 must all exhibit the same behavior (in this case, a simple calculation) resulting in different answers.

Because 0 (off) could also be represented by 0 percent, is there a behavioral boundary between a dial setting of 0 and 1? It's impossible to say without learning more about the product specification. If we are asked to have the software turn on a little green light whenever the sound system is turned on, that would call for a behavioral

boundary between 0 and 1 (though not necessarily implemented by `Dial`). Until that behavior is requested, however, the test-driven developer would not build it.

When you spot a behavioral boundary, you can write your tests for the values that represent the "edges" of that boundary:

```
@BeforeEach
public void initializeVolumeDial() {
    dial = new Dial("Volume");
}

@Test
public void dialThrowsExceptionForValueUnderMin() {
    assertThatIllegalArgumentException().isThrownBy(() -> {
        dial.setTo(-1);
    });
}

@Test
public void levelAsPercent_WhenOff() {
    dial.setTo(0);
    assertThat(dial.levelAsPercent()).isEqualTo(0);
}

@Test
public void levelAsPercent_AtMaximum() {
    dial.setTo(10);
    assertThat(dial.levelAsPercent()).isEqualTo(100);
}

@Test
public void dialThrowsExceptionForValueOverMax() {
    assertThatIllegalArgumentException().isThrownBy(() -> {
        dial.setTo(11);
    });
}
```

What if you hadn't noticed the boundaries before writing your tests, and you had written that earlier happy path test for the dial setting of 5? You could either leave the original test within the suite or delete it. If the happy path test provides even the tiniest additional clarity in the specification, leave it. It would take perhaps one

additional microsecond for each run. Alternatively, you could adjust the happy path test to be right at the edge of the boundary by changing the 5 to a 0 or a 10 and by changing the expected outcome to 0 or 100, respectively.

If you don't test the edges of the behavioral boundaries, your team's safety net will not be as clear and thorough as possible. For example, if we use 42 instead of 11 as the erroneous input greater than 10, and later another developer mistakenly increases the maximum value to 30, our test suite wouldn't catch this problem. Our team might not learn about the defect until some unlucky consumer receives a very unpleasant (and very loud) surprise.

Because thinking test-driven requires human thought, mistakes will still happen, but far less frequently. When TDD is used diligently, every software defect is either a missing test or a vaguely specified test (that is, a misunderstanding). In either case, the mistake is a gap in the team's safety net. It often takes just one more test to close the gap.

A Taste of TDD

Let's return to the Box software and give it the ability to indicate whether it is empty or not.

This example is written in JavaScript using Jasmine. In case you want to follow along, you won't need anything except a text editor, a browser, and a download of the Jasmine library.

1. Write a single test for a tiny bit of behavior:

```
describe("box", function() {
    it ("starts out empty", function() {
        var box = new Box();
        expect(box.isEmpty()).toBeTruthy();
    });
});
```

2. Write just enough code so your test fails due to an assertion (also known as an expectation):

```
Box = function() {
};

Box.prototype = {
```

```
    isEmpty: function() {
        return false;
    },

};
```

3. Run ***all*** of the tests—that is, develop the habit of running the whole suite each time. Make sure the new test fails with an informative message, as illustrated in Figure 1.2.

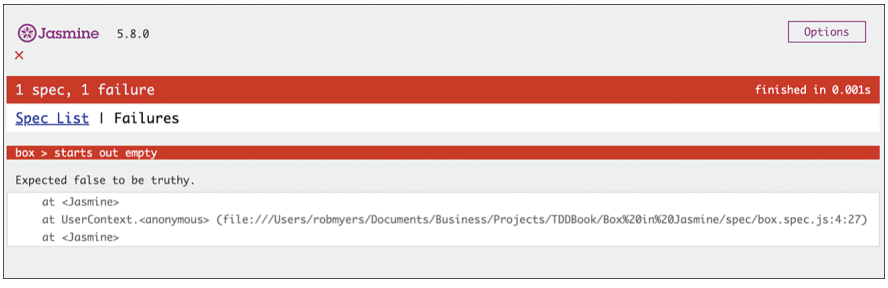


Figure 1.2 *Jasmine's SpecRunner.html page showing the clean test failure.*

(*Source:* Jasmine Software. Screenshot of testing, 2025.)

Note that Jasmine highlights the name of the test that has failed. The test ("spec") name and failure message together describe the expected behavior. Seeing the test fail and reading the failure message is how you "test the test": You know you've asked for behavior that has not yet been implemented.

4. Write just enough code to get the test to pass as quickly as possible:

```
isEmpty: function() {
    return true;
},
```

All we did was change false to true. This technique, called *Fake It*, will be covered in detail in Chapter 2, "Basic Moves."

5. Run ***all*** of the tests again. They should all pass now, as shown in Figure 1.3.



Figure 1.3 *Jasmine's test-results browser page showing the tests passing.*

(*Source:* Jasmine Software. Screenshot of testing, 2025.)

We're not done with this behavior, but this seemingly trivial code is *sufficient to pass all existing tests*.

6. Refactor diligently!
   Because this is our very first test, there's nothing to refactor here. Nevertheless, you should always take a moment to review both the tests and the implementation to see if you can reduce any duplication or improve the clarity.

Back to step 1!

1. Write a single test …
   Where there's a *true*, there's usually a *false*—for example, when a `Box` isn't empty. We'll give `Box` an `add()` method:

```
describe("box", function() {
    it ("isn't empty after adding", function() {
        var box = new Box();
        box.add("red pen");
        expect(box.isEmpty()).toBeFalsy();
    });

    it ("starts out empty", function() {
        var box = new Box();
        expect(box.isEmpty()).toBeTruthy();
    });
});
```

2. Write just enough code to make your test fail cleanly. It won't fail cleanly until it has a stub for `add()`. Resist the temptation to write an implementation for `add()` before you see the test fail:

```
Box = function() {
};

Box.prototype = {
    add: function(item) {
    },
    isEmpty: function() {
        return true;
    },
};
```

3. Run ***all*** of the tests.
   Of course, the new test fails, as illustrated in Figure 1.4.

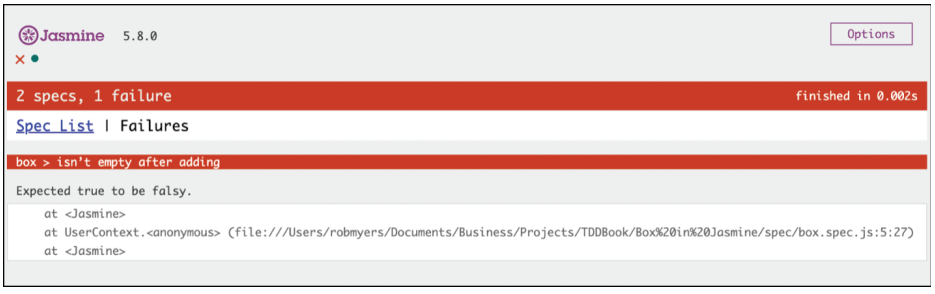


Figure 1.4 *A new failing test.*

(*Source:* Jasmine Software. Screenshot of testing, 2025.)

4. Write just enough code to get both tests to pass:

```
Box = function() {
    this.items = [];
};

Box.prototype = {
    add: function(item) {
        this.items.push(item);
    },
    isEmpty: function() {
        return this.items.length === 0;
    },
};
```

5. Run ***all*** of the tests to see them pass, as shown in Figure 1.5.

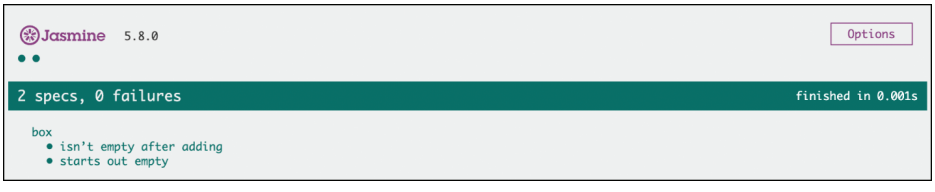


Figure 1.5 *Both tests passing.*

(*Source:* Jasmine Software. Screenshot of testing, 2025.)

6. Refactor diligently!
   Refactoring the tests is as important as refactoring the implementation. Doing so often helps you write later tests.

There is a tiny bit of duplication between the two tests, and that's enough. The duplicated code is highlighted in the following code:

```
describe("box", function() {
    it ("isn't empty after adding", function() {
        var box = new Box();
        box.add("red pen");
        expect(box.isEmpty()).toBeFalsy();
    });

    it ("starts out empty", function() {
        var box = new Box();
        expect(box.isEmpty()).toBeTruthy();
    });
});
```

All unit-testing frameworks have a way to perform common setup for each test in the suite. In Jasmine, it's called `beforeEach()`, and it runs once before each test within the containing `describe` block:

```
describe("box", function() {
    var box;
    beforeEach(function() {
        box = new Box();
    });

    it ("isn't empty after adding", function() {
        box.add("red pen");
        expect(box.isEmpty()).toBeFalsy();
    });

    it ("starts out empty", function() {
        expect(box.isEmpty()).toBeTruthy();
    });
});
```

Another run of the tests confirms that the changes made did not break any tests (Figure 1.6).



Figure 1.6 *Checking the refactoring.*

(*Source:* Jasmine Software. Screenshot of testing, 2025.)

Here's what we've accomplished with this tiny example:

- We have some tested, production-quality code.
- We have written part of an engineering specification.
- We have written two distinct *behaviors* that this object provides.
- We've reduced duplication in the tests.

In Chapter 2, we'll walk through a richer example and explore more test-driven thinking and design choices.

The Future of Test-Driven Development

One of my favorite personal mottoes is "*Never* an absolutist."[3] Do I believe TDD is a practice that will never change and will never disappear?

Of course not: It's already changing. What follows are some of those changes, along with their implications for the future of TDD. In each case, the TDD game is slightly modified, but the thought processes, techniques, and human activities remain essentially the same.

As far as TDD disappearing, well, that won't happen as long as humans are needed to explain to a computer what it is that *we* want it to do for *us*.

TDD and Behavior-Driven Development

Behavior-Driven Development (BDD) is like TDD in many ways. It is the whole-team practice of building software by writing brief, human-readable scenarios, and

3. For the record, I was saying this long before we heard Obiwan Kenobi's equally silly paradoxical utterance, "Only a Sith deals in absolutes."

getting them to work one at a time. BDD is defined as the practice of "exploring desired system behaviors with examples in conversations and formalizing those examples into automated tests to guide development."[4]

BDD isn't merely a renaming of TDD. As Matt Wynne wrote in *The Cucumber Book* (Pragmatic Bookshelf, 2017), "BDD builds upon TDD by formalizing the good habits of the best TDD practitioners."

In practice, BDD differs from TDD in the following ways:

1. The tests are "business-facing." That is, they express business rules and requests. The suite of tests ("scenarios" or "examples") represents a runnable product specification, whereas the emphasis of TDD is often on a runnable engineering specification. To keep the scenarios readable to the entire team—whether technically inclined or otherwise—BDD examples are written in a ubiquitous domain language and use a handful of keywords.
2. BDD is a whole-team activity and is best implemented with nearly continuous collaboration between the product and development specialists. Although this approach might seem burdensome on the surface, BDD can reduce the need for many traditional meetings and hand-offs. The team works closely together to plan, specify, build, test, and demonstrate working software within hours or even minutes.
3. TDD is included as a practice within the BDD cycle. There are often situations where finer-grained, "infrastructural" testing is required to complete a BDD scenario without cluttering the product specification with engineering details. Ergo, there's no need to choose between BDD or TDD. They go together nicely, like chocolate and peanut butter.

Many of the test-driven techniques described in this book apply equally to a team's BDD scenarios: They favor developing more, smaller tests with fewer assertions; setting up just enough to test the business rule; reducing duplication (for example, using Cucumber's Gherkin *Background* keyword); making the scenarios readable; strategically replacing challenging dependencies with test doubles; and taking small, quick, and safe steps toward the team's goal.

4. As defined in *Behavior-Driven Development with Cucumber* by Richard Lawrence and Paul Rayner (Addison-Wesley, 2019). distilled from the ideas of Liz Keogh and Daniel Terhorst-North.

Recommended Reading

To learn more about BDD, check out any one of the following highly rated books:

- *The Cucumber Book* by Matt Wynne, Aslak Hellesøy, and Steve Tooke (Pragmatic Bookshelf, 2017).
- *Behavior-Driven Development with Cucumber* by Richard Lawrence and Paul Rayner (Addison-Wesley, 2019).
- *BDD in Action* by John Ferguson Smart (Manning, 2023).

TDD and Functional Programming

When using Functional Programming (FP), whenever you create a type with a rigorous contract, you are effectively testing a lot of assumptions at compile-time. But there is also behavioral (runtime) code, and where there's behavior, there's the opportunity to get it wrong. Ergo, unit-testing FP behaviors is still beneficial, and building those behaviors by using a test-driven approach is as critical to quality FP as it is to object-oriented development.

FP unit tests contain all the attributes of good tests, as described in Chapter 5, "Sustaining a Test-Driven Practice." The following example gives a taste of unit testing in F# (a .Net FP language):

```
[<Test>]
let ``intersection of two sets should only contain common elements``
() =
    let left =
        initialSet
        |> add "Rob"
        |> add "Awesome"

    let right =
        initialSet
        |> add "Jason"
        |> add "Awesome"

    let result =
        left
        |> intersection right
```

```
    result
    |> contains "Awesome"
    |> expectsToBeTrue

    result
    |> contains "Rob"
    |> expectsToBeFalse

    result
    |> contains "Jason"
    |> expectsToBeFalse
```

Here is the passing implementation of `intersection` as another taste of FP syntax:

```
let intersection right left =
    left
    |> List.filter(fun element ->
        right
        |> List.contains element
    )
```

Daydreams of TDD, Quantum Computers, and No Implementation

We developers used to fear the day when computers could write their own code. We were worried not only because such computers might build Terminator robots (or worse, mountains of paperclips[5]) and take over the world, but also because we'd be out of a job.

For many years, I've speculated that we would eventually see a computing breakthrough that would allow the computer to write the entire implementation. At the time, I envisioned a quantum computer (QC) that could take a team's specifications and search a "solution space" for a set of machine instructions that would pass all the tests.

If the computer were fast enough, it could perhaps rewrite the entire implementation each time the team added a new test scenario. In other words, neither

5. Read about the "Paperclip Problem" in https://en.wikipedia.org/wiki/Instrumental_convergence.

the team nor the development computer would ever need to read or refactor the implementation.

Teams could then focus entirely on writing, refactoring, and maintaining the test suite. Future high-level programming languages could be limited to the syntax and structure needed for good test-writing (for example, no more loops or branching statements). Developers would write engineering specifications as unit tests, or co-author product scenarios while working side-by-side with product designers. Or perhaps the team roles of product designer, developer, and tester would blend and merge into something new. A test-driven approach would be the de facto standard for building and maintaining software.

While I was daydreaming about the impacts of this hypothetical QC, a different computing breakthrough was happening. Noting the surging power and popularity of artificial intelligence[6] (AI) and large language models (LLMs), I began to wonder whether these new tools would someday make my predictions a reality.

The Reality of TDD, Artificial Intelligence, and "Vibe Coding"

Using a rudimentary OpenAI script that contained a simple prompt including all my tests (I could easily add tests with each run of the script), I explored whether OpenAI could generate code to pass all my tests. And it did so, repeatedly and successfully.

With each new run of the script, my AI agent did not have access to any previous prompt, dialog, or existing implementation. There was nothing for me or the AI to refactor, because it always started from scratch and overwrote all the code with each run.

My experiment was only that: a simple proof-of-concept. I had the script build simple bits and pieces of various classroom exercises (for example, the *Salvo* game described in the Exercises Appendix at the end of this book).

There are several ways that AI tools are currently assisting real developers. Some integrated development environments (IDEs) offer AI "autocomplete" options that are quite good at predicting what the developer intended to write next, and the complete code appears much faster than the developer could have typed it. Others read all the existing code and will suggest refactorings. The developer merely needs to glance over the proffered code and press a single key to accept the changes.

AI agents can even write tests for you. However, the generated tests that I saw merely confirmed that the code did what it did, not necessarily that it did what was

6. Not to be confused with the "artificial intelligence" from the pre-LLM science fiction tales, which has been relabeled artificial *general* intelligence (AGI), and most likely does not exist at the time of this writing.

wanted. This is the gap in the workflow where humans are still needed: Someone needs to come up with descriptive and detailed examples of what they want the software to do.

Right now, various methods for incorporating AI into the full development workflow are vying for our attention. One of those approaches is called "vibe coding," which was described to me as allowing the AI agent to write the implementation without any review. The person receiving the code then tests it and gives the AI feedback. This is happening in a few different ways:

- The human manually tests the application and then explains to the AI where it failed, often using an example. They do this repeatedly until the human is satisfied with the results. At least one person who used this method expressed occasional frustration with the AI agent's tendency to seemingly reinterpret older requirements or to incorrectly anticipate unspoken needs.
- The human lets the AI write its own tests, and the human reviews and runs those tests.
- The human gives the AI examples, either one at a time or in a batch, and adds to those examples if either the human realizes there is a gap in the examples (a common and natural occurrence with a test-driven approach) or the AI delivers a solution that is in some way too generalized (also common when pairs or ensembles develop code using a test-driven approach). This style most closely resembles my early experiment with OpenAI.

Some things I noted about these reported experiences:

- In every case, clear, specific examples—either written by or approved by the person making the requests—were necessary at some point in the workflow.
- Except for the time saved by having the AI write code (which was certainly a significant savings), little time was saved during interactions with the AI. Put another way, the person interacting with the AI still spent about the same amount of overall time explaining what they wanted.
- None of the applications described to me could directly impact a user's health, finances, or safety. I asked AI guru Scott Werner if he would use vibe coding for financial, medical, or safety applications, and he responded, "No, probably not anything life critical."

Computers can't read our minds, and they don't do well with ambiguous instructions. When it comes to our safety, health, finances, and other critical domains, we will need to describe—to the computer and to each other—all desired outcomes using complete, descriptive, and detailed examples. That is exactly the practice of TDD.

Summary

The real power of TDD comes from thinking about what the software needs to do and coming up with examples to represent that behavior. The red–green–clean steps are not meant to be mere mechanics, but rather a scaffolding to allow developers to explore, guide, and preserve those thoughts.

The next two chapters will use an example application to explore the thinking process in detail. Approach this material as you would approach the rules of a new game. These chapters provide the rules and basic strategies by walking you through a sample "game." The example app is a simple one, and the resulting implementation will be trivial, but you will come away with an understanding of the power of test-driven thinking.

Index

A

abstract class testing, 136–137
abstractions, 87, 95, 136
acceptance tests, 123, 202
accessors, data, 73, 90
AfterEach, 125, 175
Agilist's Dilemma, 191, 192–193, 195
AI, coding assistance from, 18–19
A Light Introduction to Nullables (Shore), 152
alternative return paths, 73
anti-fragile systems, 196
APIs
  building, 156
  calling external, 157
  testing Filesystem API, 159
applications
  Black Swan stories for, 196–205
  correct deployment of, 131
  exceptions thrown in, 74
  programmatic errors in, 111
arrays, merging, 50
artificial general intelligence (AGI), 18
Art of Agile Development (Shore), 163
ASP.Net, 164
assertions (expectations)
  Coupled Assertions smell, 111–113
  custom, 112–113, 122
  defined, 22
  exception-thrown, 120
  multiple, 49
  one per test, 115
  passing/failing, 75
  as prerequisite for tests, 25
  Redundant Assertion smell, 118–119
  Repeated Assertion smell, 119–120
  syntax, 120
  in TDD flowchart, 21, 22
  test failures due to, 9
  in test scenarios, 108
  test writing and, 33
  Unfailing Assertion smell, 117–118
  Unrelated Assertions smell, 116–117
assertThrows(), 75, 76
autocomplete code, 18
automated testing, 81

B

baby steps
  vs. large steps, 96
  Obvious Implementation as, 39
  in refactoring, 52
  vs. Scattered Changes smell, 131
  Sprinting Baby, 32
  vs. uncertainty, 35
Beck, Kent, 26, 61, 84
BeforeEach, 13, 82, 103, 104, 107, 114, 133, 175
behavior-driven development, 14–16
behavior(s), 32–33
  added to legacy systems, 163, 164, 170
  added to simplify testing, 57
  assertions unrelated to, 117
  behavioral boundaries, 6–9, 38
  both sides of boundaries in, 95
  changeable, 128
  characterization tests and new, 175
  characterization tests for multiple, 178
  characterization tests to preserve, 164, 165, 179
  clean test failures and new, 10
  clear tests for, 81
  exceptional, 73–78
  functions of isolated, 97, 106, 108
  grouping tests by, 48–49
  human logic for, 18–19
  knowing assertions for, 25
  in legacy code, 166–167
  multi-behavior methods, 108–109
  in TDD flowchart, 22
  testing, 73, 88–93, 94
  testing small units of, 3–6, 9, 23, 124
  test-resistant, 79
  test suite as protection of, 37
  thinking at unit level of, 83
  triangulating related, 102
  writing tests for new, 32–33
Black Swans, 195–205
The Black Swan (Taleb), 195
blind spots, noticing, 53–55
blocks, 120
Boolean values, 40
bottlenecks, performance, 70
boundaries, behavioral
  both sides of, 95
  choosing which side to implement, 88, 89
  on CRC cards, 87

defined, 38
errors when reaching, 91
and external dependencies, 146
Fake It as revealing, 49
loops, 120
overlooking of, 39
test-data boundaries vs., 53
testing at edges of, 122
testing boundaries, not data, 92–93
and test names, 100
tests written for, 6–9
thinking in tests and, 83
brainstorming techniques, 64
Branching Code smell, 119, 120, 121–122
bugs
avoided at the outset, 3
characterization tests and discovering, 179
fear of introducing, 192
as missing tests, 9
TDD discipline and reduced, 194
thrown exceptions as, 75
well-crafted safety net vs., 98
Builder, 137

C

C#
Extract Method in, 134
interface keyword, 25
internal constructors, 155
internal for methods in, 172
optional parameters in, 133
redirecting output in, 175
stubs in, 144
virtualizing proxy, 183
C++ classes, 180
Calculations smell, 121–122
call stack, tracing exceptions in, 74
camel case, 100
case sensitivity, 100
catch clause, 74
catch statements, 120
challenges
best practices for, 63–64
Black Swan events, 195–205
good tests vs., 80–83
learning from, 97
in legacy code, 167, 169
See also code smells
changes
characterization tests for, 166, 179
fear of frequent, 192
saving, 45
Scattered Changes smell, 131–135
characterization tests, 164–180, 185, 186
clarity
as attribute of good test, 81–82
checking for excruciatingly obvious, 82
improvements, 11, 43
via test names, 101
classes
abstract, 87
analyzing legacy, 166
class names, 100
in CRC cards, 84
objects and, 84
parent classes, 122
testing abstract, 136–137
clean code, 37
clean test failures
examples of, 41
as second TDD step, 33–34
via swapping objects, 49
in TDD flowchart, 22
testing tests and, 10, 11
code
autocomplete, 18
behavior described by, 3–4, 5
characterization tests and errors in, 179
clear/readable test code, 25
computer-written, 17–18
creating "seams" in, 171
difficult-to-test code, 96–97
efficiency/optimization of, 70–71
fake code, 35–36
in IDE, 34
Nullables and production, 152
procedural, 106
separating known/unknown in, 93
stubs as test code, 145
teams and idioms in, 61
testing untested, 138, 149
test vs. implementation, 80, 98
writing tests before, 136
code smells
defined, 37
and difficult-to-test code, 96–97
good tests vs., 80–83
intentional enhancement of, 129
Magic Number, 81
multiple assertions as, 49
multiple wrapper layers, 158
null keyword as, 160
refactoring to prevent, 36, 55
tipping point with, 127
collaborators, 84
combinedWith(), 53
comments, to-do, 30
compiling, in TDD flowchart, 21, 22
complexity
dependencies and increased, 143
incremental addition of, 94
of legacy code, 178
mocks and increased, 151
comprehensive tests, 43
concise tests, 82
constants, 56
constructors, 155, 174
contains(), 42, 43, 48

containsAll(), 56, 62, 63, 65, 69, 70–71
context-switching, 80, 86
Copy/Paste/Modify test smell, 102, 164
cost
  of Black Swan events, 198, 204
  of defects, 194
  of dependencies, 143
  performance optimization and, 71
coupling
  Coupled Assertions smell, 111–113, 116
  dependency injection and, 154
  unnecessary, 76, 98, 106
COVID-19 pandemic, 196
CPU-intensive calls, 70
CRC (class–responsibilities–collaborators) cards, 24, 64, 84–88, 106, 129
creating mock objects, 147
Cucumber, 15, 16, 24, 131, 169
Cunningham, Ward, 84, 93
CUPID, 61
Custom Parent Test Classes smell, 122

D

data structure
  testing behavior to drive, 89–92
  testing boundaries instead of, 92–93
Date, 81, 84, 130, 131, 158, 159
dates, 32
DateTime, 130, 131, 158
debugging
  avoiding, 3, 36
  TDD discipline and reduced, 194
  as time drag, 4, 36
decomposing requests to unit level, 83
defects. *See* bugs
dependencies
  calling external, 148
  cascading, 143
  in characterization tests, 177
  coupled, 124
  decoupling, 98
  dependency injection, 153–155
  deploying real vs. fake, 160
  external, 73, 74, 77, 130, 141–142
  injected into proxies, 185
  in legacy code, 167
  legacy code and external, 180
  test doubles to replace external, 125, 141
  testing methods that create, 137–138
  as test obstacles, 142–143
  and virtualizing proxy, 180–186
  wrapping external, 155–158
  See also test doubles
describe block, 13, 48
describe clause, 81, 100, 103, 107, 114, 115
describe clause, 113
design
  cautious programmers and, 193
  constructors vs. production code in, 155
  copy/paste/modify in, 164
  CRC cards for, 84, 87, 88
  data structure and, 89
  and difficult-to-test code, 96, 149
  emergent, 198
  grouping domain exceptions, 76
  implementation, 106
  iterative test design, 92, 191
  refactoring to clarify, 43
  state- and interaction-based tests, 149
  surprise changes to, 204
  team knowledge and good, 61
  test-resistant, 79
  test-suite, 79
  test vs. object-oriented, 98
  wrapper objects in, 155
  "zero" case in, 94
  ZOMBIES model for, 94–96
Dijkstra, Edsger W., 3
discipline, TDD, 191–195, 205
DLL, 25
domain exceptions, 74, 76
domain values, 55
doubles, test. *See* test doubles, 87
drawings of sticking points, 63–64
driver role, 29
Duplicate Given smell, 102–105, 133
duplication, reducing
  custom assertions for, 56
  Duplicate Given smell, 102–105
  and eliminating, 57–59
  importance of, 164
  refactoring for, 50, 98
  reviewing tests/implementation to, 11
  Rule of Three/Two for, 137
dynamic link library (DLL), 76

E

edge cases
  avoiding, 7
  both sides of, 95
  loops and, 120
  in overriding built-in equals(), 67
  testing, 67
emergent design, 198
empty groups, 56
end-to-end tests, 123–125, 135
equality comparisons, 66
equals(), 66–67, 67
errors
  avoiding error conditions, 7
  characterization tests and discovering, 179
  data in error messages, 74, 76–77
  excruciatingly obvious messages for, 92
  fear of introducing, 192
  for invalid user input, 74, 92
  one-off-errors, 179

out-of-range, 6
programmatic errors, 111
symptomless, 57–59
TDD discipline and reduced, 194
throwing exceptions/raising errors, 73–75
tool for spotting undetectable, 53–54
writing tests to anticipate/correct, 61
See also exceptions
event-driven coding, 6
exceptions
avoiding pointer/reference, 95
as coding/refactoring mistakes, 110
external exceptions, 77
passing tests when code fails, 75–77
raising/throwing exceptions, 73–75
tracing programmatic errors, 111
unexpected, 120
in ZOMBIES model, 96
See also bugs; errors
Excessive Given smell, 105–106, 135
excruciatingly obvious tests, 81, 82, 92, 121
expectations (assertions). *See* assertions (expectations)
experiments with the unfamiliar, 94
Exploratory Testing, 81
Exposed Implementation smell, 136, 149
Extended Narrative test smell, 115–116, 118, 136, 151
external dependencies. *See* dependencies
Extract Class refactoring, 136
Extract Method refactoring, 56, 76, 130, 133, 134, 136, 137, 169, 171, 172, 174
Extreme Programming, 61, 200

F

Factory Method, 137
failures
avoiding multiple, 88
characterization test, 176
context via failure reports, 99
and halting of tests, 115
Intermittent Failure smell, 129–130
passing tests when code fails, 75–77
red state and failing tests, 29
from typos, 56
See also clean test failures
fake code, 35–36
Fake It ('Til You Make It) technique
boundaries revealed by, 49
defined, 26
examples of, 10, 41
to make tests pass, 35–36, 49
"zero" case and, 94
fakes, 143–144, 160
fast tests, 80–81
Feathers, Michael, 80, 163, 167, 168
final classes, 180
flowchart, TDD, 21–22
focused tests, 82
Forced Return Value smell, 108–111
formatting changes, Black Swan story of, 202–204
For Testing Only smell, 128–129, 130
Fortran, 179
Four Rules of Simple Design, 61
Fowler, Martin, 37
FoxPro, 200
FP (Functional Programming), 16–17
Frequent Changes to *Given* smell, 125–128
Fukushima, 196
functionality, performance optimization for, 71
functions, known/unknown, 93

G

generalization, over-, 19
getters/setters, 73, 90
Gherkin scenarios, 169
given objects, 103, 105, 107, 115–116, 118, 124, 125, 144
given/when/then structure, 23, 115
greenfield projects, 163
green state
considerations in, 29, 30
defined, 29
via passing tests, 35
refactoring following, 106
Grenning, James, 94
groups, comparisons for equal, 66
guard clauses, 110

H

hardware drivers, 141
has(), 52
The Humble Programmer (Dijkstra), 3

I

idioms, code, 61
if statements, 6, 120
IllegalArgumentException, 6
immutable objects, 32, 159
implementation
of abstractions, 95
clean test failures and, 10–11
computer-written, 17–18
design improvements, 106
encapsulated implementation details, 172
Exposed Implementation smell, 136
fake it code before, 36
implementation/test code ratio, 98
minimum passing, 71
Obvious Implementation, 26
performance/speed of, 70
refactoring the, 12
rules for adding to, 21
services/relationships as preceding, 84

single-behavior, 5
test code in, 121
testing behaviors instead of, 88–89
tests and, 23
tests written before, 32–33
triangulation and improving, 37
undetectable breaks in, 53
of the unexpected, 195
virtualizing proxy and legacy, 183
well-written tests and clean, 79, 96
working vs. finalized, 88
incremental building of tests, 173–177
independent tests, 82
index card use, 27, 30
injecting mocks, 147
injection of dependencies, 153–155
inputs. *See* given objects
integrated development environments (IDEs)
AI assistance in, 18
built-in refactoring, 52
introduce parameter in, 133
rote work done by, 34
test method creation by, 68
tracking to-do list in, 30
integration, reducing merge conflicts in, 88
integration tests, 123, 131, 160
IntelliJ, 34
interaction-based tests, 148, 149
interactions, Nullables and turning off, 152
interface
behavioral, 90
designing for user, 96
mocking the, 148
object, 25
stubbing methods on, 145–146
virtualizing proxy, 184
wrapping ResultSet in, 156
"zero" case in designing, 94
interface keyword, 25
Intermittent Failure smell, 129–130
internal keyword, 25
Introduce Virtualizing Proxy refactoring, 180, 181
isEqualTo(), 66, 68–70
iterative test design, 92

J

Jasmine, 9, 10, 48, 100, 114
Java
converting UTF8 to Unicode in, 197–199
equals() in, 66
final classes, 180
interface keyword, 25
naming in, 40
null exceptions, 75
package-private methods in, 172
protected constructors in, 155
refactoring in, 37
stubs in, 144, 146
test frameworks in, 24
virtualizing proxy in, 182
writing tests in, 34
JavaScript, 9, 24, 37
JDBC, 157, 158, 160, 180
JDBCException, 74
Jetlagged Intern Mind practice, 53–54, 61, 71, 93, 149
JUnit, 22, 24, 100, 103, 122, 144, 174
JUnit5, 48

K

knowable factors, coding, 93

L

lambdas, 61, 120
languages
case sensitivity in, 100
class in OOP, 84
experiments with unfamiliar, 67
merging arrays in, 50
mock as noun/verb, 146
testing frameworks in, 23–24, 48
for test writing, 18
See also specific language by name
large datasets, 70
leadership for TDD teams, 191
legacy code testing
and adding new behaviors, 163–164
via characterization tests, 164–180
disconnecting from network for, 168
and external dependencies, 180
need for, 163
testing all paths/results, 178
testing long blocks of, 184
and virtualizing proxy, 180–186
wrappers in, 158
LLMs (large language models), 18
logic
mistakes and human, 9
TDD as record of developer, 3–4, 20
of TDD as unchanging, 14
test-driven, 60
loops
Branching Code smell, 120
in custom assertions, 120
in legacy code, 167

M

Magic Number code smell, 81
maintenance, 79–80, 97, 136
manual testing, 81
memory, test run in, 80
Menlo Innovations, 200
merge conflicts, reducing, 88
Meszaros, Gerard, 143

methods
  analyzing legacy, 166
  to call a virtualizing proxy, 180–186
  calls in Ruby, 90
  classes with static, 180
  dependencies created by, 137–138
  extract methods, 136, 137, 183
  guard clauses in, 110
  method signatures, 131
  multiple behaviors in, 108–109
  naming, 40, 129
  overload methods, 131
  parameter names, 41
  Rename Method refactorings, 52, 101, 108, 171
  stubbing, 145–146
  testing private, 136, 172, 173
  trying to do too much in one, 108
Miro, 85
missed specifications, 42
Mocha, 24, 48, 100
mock frameworks, 144, 146, 150, 151, 160, 161, 180, 185
mocks, 146–148, 149, 153
modifying tests, 43–44
Move Method refactoring, 136
MSTest, 24, 100, 103, 175
Multiple Test Doubles smell, 135
Mural, 85
mutators, data, 73

N

naming
  methods, 40, 129
  refactoring for clear, 55
  term re-use, 100
  Unhelpful Test Names smell, 99–101
narrow integration tests, 160
navigation tests, 123
navigator role, 29
.Net, 66, 75, 180, 183
North, Daniel Terhorst, 61
Novell's Developer Portal, 196
Nullables, 152–153
null exceptions, 117
null keyword, 160, 174, 185
Null Object returns, 95
null pointer/reference exceptions, 75, 95
NUnit, 24

O

object-oriented programming languages, 84, 153
objects
  BeforeEach mechanism and, 103
  changing states of, 97, 108
  classes and, 84
  and Coupled Assertions smell, 112
  determining equality of, 66
  immutable, 32, 159
  interface, 25
  known/unknown, 93
  mock, 146–147
  multi-behavior, 128
  names and distinct, 101
  object-to-object delegation, 97, 108
  simple, as test doubles, 158–159
  swapping call/parameter, 49
  testing dependencies in, 137
  test vs. object-oriented design, 98
  and Unrelated Assertions smell, 116
Obvious Implementation technique, 26, 43, 50, 61
one-off-errors, 179
online collaboration, 85
onsite customer, 22
OpenAI, 18, 19
optimizing, reasons for, 70–71
optional parameters, 133
Organ Transplant Information System (OTIS), 200
OTIS 2, 200–202
output string formatting, 73
overlapping groups, 60–61
overload methods, 131, 133
override methods, 132
overriding built-in equals(), 66–67

P

pair programming, 27, 29, 61, 72
parent classes, 122
Pascal, 179
passing tests
  making tests pass, 35–36
  via Obvious Implementation, 39–40, 71
  refactoring as following, 50, 58
  simplest code for, 42, 49
Password Exercise, 125–128, 207–209
PDF formatting, 200–202
performance. *See* speed
permutation tests, 128
Playwright library, 131
primitive objects, 66
private methods, 136, 172, 173
product advocate/owner, 22, 83, 86
production class, 48
products, thinking at unit level to develop, 83
programmatic errors, 111
programmers
  AI assistance for, 18
  computers as replacing, 17–18
  discipline for TDD, 191, 192
  overly cautious, 193
  research done by, 93
  TDD as support for, 7
  TDD pairs/teams of, 26–28, 29
  tests as recorded logic of, 3–4, 83
programming
  by computers, 17–18
  effective, 3

excess of caution in, 192–193
protected constructors, 155
public interface, refactoring only the, 52
public keyword, 25
Python, 24, 50, 72

Q

quantum computer (QC), 17

R

raising errors, 73–75
readable test code, 25, 43, 81
red–green–clean steps
considerations in red vs. green, 29, 30
and programmer logic, 20
red and green defined, 29
in TDD flowchart, 21–22
red state
avoiding lengthy considerations in, 29, 30
clean test failures as getting to, 34
Redundant Assertion smell, 118–119
refactoring
after passing tests, 50, 58
AI-assisted, 18
in Black Swan stories, 198, 201, 205
of characterization tests, 178
to clarify nested describe clauses, 114
cleaning as you go, 36–37
for dependency-creating methods, 137
diligent, 11, 12
for distinct tests, 55
to-do list of upcoming, 28
vs. duplication, 98
for existing untested code, 138
Extract Method, 56, 76, 130, 133, 134, 136, 137, 169, 171, 172, 174
for good design, 61
of known/unknown factors, 93
of legacy code, 164, 171, 179
from minimal implementation, 50
as a must, 44–45
removing excess behaviors via, 128
Rename refactorings, 52, 101, 108, 171
running tests after, 45
small changes after, 97
speed of, 79–80
surgical, 171, 174, 180, 184
and test resistance, 79
tests, 55, 72
via virtualizing proxy, 180, 184
writing tests following, 51
Refactoring: Changing the Design of Existing Code (Fowler), 37
Rejected Given test smell, 104, 107
relationships
coupled assertions, 112
and services, 84
Rename Method refactorings, 52, 101, 108, 171
repeatable tests, 83, 142
Repeated Assertion smell, 119–120
requests
decomposed to unit level, 83
tasks and, 26–28
research of the unknown, 93–94
resilient vs. anti-fragile systems, 196
resistance to testing, 79
responsibilities, 84, 98, 136
results. *See* then, in given/when/then structure
ResultSet, 156, 157, 158
retry loops, 74
return paths, 73
return values, forced, 108
rSpec, 4, 24, 48, 100, 114
Ruby, 4, 24, 48, 132, 133
Rule of Three, 137
Rule of Two, 137

S

safety net, test suite as
and behavioral edges, 9
defined, 4
end-to-end tests, 123
gaps in, 130
legacy code and absence of, 164
legacy code testing to patch, 163–187
maintenance of, 79
narrow integration tests in, 161
payback from strong, 98
redundant assertions and lack of trust for, 118
steps for closing, 53–54
working without, 193
Salvo game, 89–92, 132, 144, 147, 153, 210–212
Scattered Changes smell, 131–135
scenarios
CRC cards for, 85, 86
Cucumber, 131, 169
to-do list of upcoming, 28
pair brainstorming of, 29
test doubles in test, 141
testing multiple, 115–116
tests as single-behavior, 5, 23
writing tests for related, 7
Scrum sprints, 192
Scrum teams, 22
sealed classes, 180
services and relationships, 84
setters/getters, 73
Shore, James, 143, 152, 160, 163, 196, 197
side effects, 60–61
skillful procrastination, 94
slow tests, 80
sociable tests, 158
software
antifragile, 196
behavioral examples, 3–4
design and test design, 98
three functions of, 97, 108

specification, missed, 42
specificity, 8, 9, 20
speed
  dependencies and decreased test, 142
  fast tests, 80–81
  optimizing for, 70
spies, 136, 148–152, 153
Sprinting Baby, 32, 71
SQLException, 156, 157, 158
stack trace, 74, 111
state-based tests, 148, 152, 153
static methods, classes with, 180
stubs, 144–146, 146–147, 152, 180, 185
surgical refactoring, 171, 174
symptomless defects, 57–59
syntax
  assertion, 120
  experiments with unfamiliar, 67

T

Taleb, Nassim Nicholas, 195
targeted tests, 82
tasks
  CRC cards as, 88
  order of completing, 41
  requests and, 26–28
TDD (Test-Driven Development)
  basic techniques for, 26, 37–40, 42, 45–46
  best practices for challenges in, 63–64
  characterization testing and, 164–165, 177, 179
  code defects in, 42
  defined, 45
  discipline needed for, 191, 205
  exception guidelines, 74
  flowchart for, 21–22
  future of, 14–20
  isolating code from external dependencies in, 142
  jetlagged intern practice, 53–55
  leadership for, 191
  as record of developer logic, 3–4, 23, 26
  spies in, 148
  testing legacy code with, 163
  thought/coding as supported by, 7
  unit-level thinking for, 83–84
teams
  on Black Swan projects, 196–205
  business language of, 81
  CRC card creation by, 85
  delivering exceptions to support, 74
  design preferences of, 79
  good design and, 61
  leadership for, 191
  online collaboration for, 85
  pair programming, 29, 72
  Product Advocate on, 22
  team tasks, 26–28
  time spent on maintenance by, 79–80
Tear Down smell, 125, 175
temporary (fake) code, 36
test-data boundaries, 53
test doubles
  best practices for, 153–159, 160–161
  decoupling dependencies via, 98
  for external exceptions, 77
  fakes, 143–144
  for incomplete responsibilities, 87
  for independent tests, 82
  mocks, 146–148
  Multiple Test Doubles smell, 135
  of proxy in characterization test, 185–186
  to replace external dependencies, 125
  resolving smells with, 135
  spies, 136
  stubs, 144–146, 147
  taxonomy for, 143–153
  test speed and, 80
  and virtualizing proxy, 180, 183
  when to use, 141–142
tests
  of abstract classes, 136–137
  adding to/modifying, 43–44
  AI assistance for, 19
  attributes of good, 80–83
  automated feedback on, 34
  automated testing, 193
  of behavior, 88–93
  as behavioral units, 3–6
  bugs as missing tests, 9
  characterization tests, 164–180
  clean implementation and good, 79, 96
  clean test failures, 10
  clear/readable, 43, 46
  common setup for, 13
  copy/paste vs. from scratch, 102, 164
  dependencies as obstacles to, 142–143
  design, and software design, 98
  difficult-to-test code, 96–97
  excruciatingly obvious, 81, 82, 92, 121
  for existing untested code, 138, 149
  given/when/then structure of, 23–25
  grouped by behavior, 48–49
  and implementation code, 23, 121
  implementation/test code ratio, 98
  improving test data, 53–55
  incremental building of, 173–177, 193
  integration tests, 160
  of knowable factors, 93–96
  manual vs. automated, 81
  mathematical operators, 121
  maximizing coverage/minimizing effort in, 178
  multiple tests, 88
  passing tests when code fails, 75–77
  performance, 70
  of private methods, 136, 172, 173
  readability of, 81, 114
  as record of developer logic, 3–4, 83
  refactoring, 55, 72
  single-behavior, 115, 120, 124, 135
  small vs. large, 43
  sociable tests, 158

state- and interaction-based, 148
with test doubles, 141
test smells, 98, 99–136
triangulation of, 7
written before implementation, 32–33
written for behavioral "edges", 8–9
test suite
behavioral edges in, 8–9
maintenance of, 79
readability of, 81
running entire, 10, 22, 34
as a safety net. *see* safety net, test suite as
safety of comprehensive, 45
tiny gaps in, 36
then, in given/when/then structure, 23, 24–25, 118, 124
thinking in tests, 83
throwing exceptions/raising errors, 73–75
time
dependencies and increased test, 142
integration test time, 123
reducing time-to-market, 195
saved via AI coding assistance, 19
spent on test-suite maintenance, 79–80, 97
to-do comments, 30
to-do lists
best practices for, 29–30
kept short, 88
modifications to, 47
order of tasks on, 41
of upcoming scenarios, 28
training mock objects, 147
Triangulation technique
as basic TDD technique, 26
and behavioral edges, 7, 37, 49
to break fake code, 92
missing opportunities for, 53
vs. Obvious Implementation, 42
for side effects, 60–61
and single-behavior tests, 115–116
try/catch/wrap blocks, 158
typos, failures via, 56

U

unexpected, implementing the, 195
Unfailing Assertion smell, 117–118, 118
Unhelpful Instance Names smell, 101–102
Unhelpful Test Names smell, 99–101
unit tests
for adding characterization tests, 169
attributes of good, 80–83
BeforeEach mechanism, 103
defined, 80, 83
to describe behavior, 3–6
end-to-end tests, 123
framework for, 24
implementation code and, 32–33
loops and, 120
one behavior per test, 115
single-behavior, 124
sociable tests as, 159
to streamline test suite, 98
trying to do too much in, 105
unit-test framework, 67
University of Michigan, 200
unknowns, mitigating risks of, 93–96
unpredictable test outcomes, 142
Unrelated Assertions smell, 116–117, 136
untested code, 138, 149
users
designing for, 96
exception alerts for, 74
unexpected implementations for, 195–205
user controls, 164

V

values
domain, 55
stubs for predetermined, 144
Venn diagrams, 64
verifying object/mock interactions, 147, 149
vibe coding, 19
virtualizing proxy, 180–186

W

Werner, Scott, 19
when step, 23, 24, 49, 114
wins, quick, 28, 32
Working Effectively with Legacy Code (Feathers), 80, 163, 167, 168
wrapping external dependencies, 155–158
writing tests
best practices for, 57
before code, 136
as first TDD step, 32–33
following refactoring, 51
for legacy code, 167

X

XML format, 202
XP teams, 22, 196, 200
XSLT, 203

Z

"zero" case, 94
ZOMBIES, 94–96