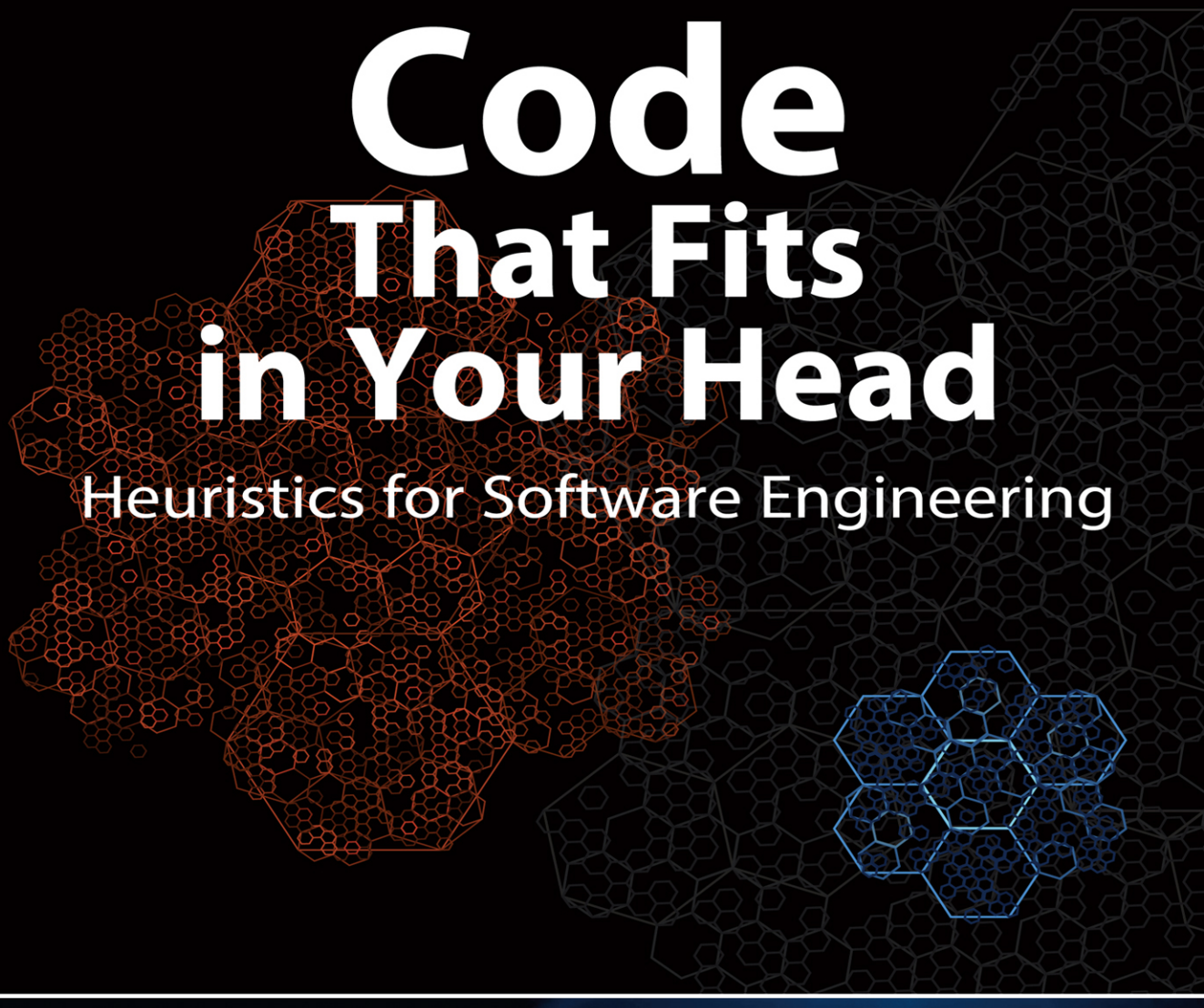
Robert C. Martin Series

Code That Fits in Your Head

Heuristics for Software Engineering



Mark Seemann

Foreword by Robert C. Martin

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Praise for
Code That Fits in Your Head

"We progress in software by standing on the shoulders of those who came before us. Mark's vast experience ranges from philosophical and organisational considerations right down to the precise details of writing code. In this book, you're offered an opportunity to build on that experience. Use it."

—*Adam Ralph, speaker, tutor, and software simplifier, Particular Software*

"I've been reading Mark's blogs for years and he always manages to entertain while at the same time offering deep technical insights. *Code That Fits in Your Head* follows in that vein, offering a wealth of information to any software developer looking to take their skills to the next level."

—*Adam Tornhill, founder of CodeScene, author of* Software Design X-Rays *and* Your Code as a Crime Scene

"My favorite thing about this book is how it uses a single code base as a working example. Rather than having to download separate code samples, you get a single Git repository with the entire application. Its history is hand-crafted to show the evolution of the code alongside the concepts being explained in the book. As you read about a particular principle or technique, you'll find a direct reference to the commit that demonstrates it in practice. Of course, you're also free to navigate the history at your own leisure, stopping at any stage to inspect, debug, or even experiment with the code. I've never seen this level of interactivity in a book before, and it brings me special joy because it takes advantage of Git's unique design in a new constructive way."

—*Enrico Campidoglio, independent consultant, speaker and Pluralsight author*

"Mark Seemann not only has decades of experience architecting and building large software systems, but is also one of the foremost thinkers on how to scale and manage the complex relationship between such systems and the teams that build them."

—*Mike Hadlow, freelance software consultant and blogger*

"Mark Seemann is well known for explaining complex concepts clearly and thoroughly. In this book he condenses his wide-ranging software development experience into a set of practical, pragmatic techniques for writing sustainable and human-friendly code. This book will be a must read for every programmer."

—*Scott Wlaschin, author of* Domain Modeling Made Functional

"Mark writes, 'Successful software endures'—this book will help you to write that kind of software."

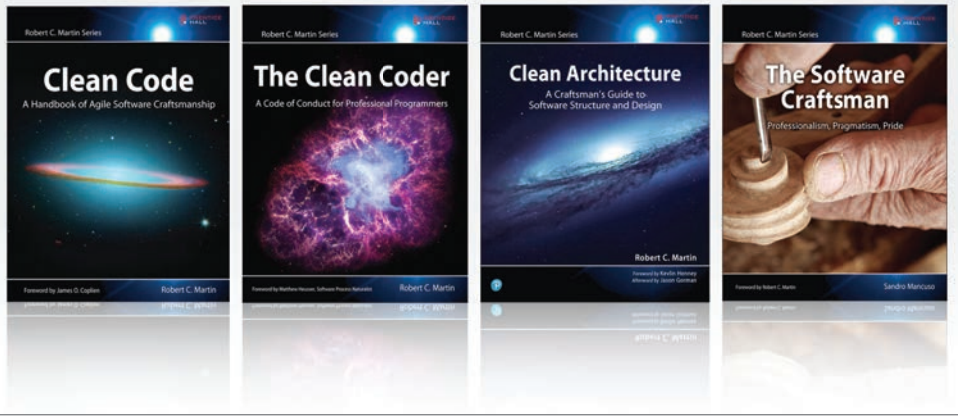
—*Bryan Hogan, software architect, podcaster, blogger*

"Mark has an extraordinary ability to help others think deeply about the industry and profession of software development. With every interview on *.NET Rocks!* I have come away knowing I would have to go back and listen to my own show to really take in everything we discussed."

—*Richard Campbell, co-host*, .NET Rocks!

Code That Fits in Your Head

Robert C. Martin Series



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

The **Robert C. Martin Series** is directed at software developers, team-leaders, business analysts, and managers who want to increase their skills and proficiency to the level of a Master Craftsman. The series contains books that guide software professionals in the principles, patterns, and practices of programming, software project management, requirements gathering, design, analysis, testing, and others.

Code That Fits in Your Head

Heuristics for Software Engineering

Mark Seemann

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Cover: Mark Seeman
Page xxix, author photo: © Linea Vega Seemann Jacobsen
Page 12, Queen Alexandrine's Bridge, Denmark: Ulla Seemann
Page 33, baseball and bat: buriy/123RF
Page 38, illustration of human brain: maglyvi/Shutterstock
Page 38, illustration of laptop computer: grmarc/Shutterstock
Page 157, Figure 8.2: © Microsoft 2021
Page 158, Figure 8.3, scissors: Hurst Photo/Shutterstock
Page 158, Figure 8.3, hand saw: Andrei Kuzmik/Shutterstock
Page 158, Figure 8.3, utility knife: Yogamreet/Shutterstock
Page 158, Figure 8.3, Phillips-head screwdriver: bozmp/Shutterstock
Page 158, Figure 8.3, Swiss military knife: Billion Photos/Shutterstock
Page 159, Figure 8.4: Roman Babakin/Shutterstock
Page 170, Figure 8.5: © Microsoft 2021
Page 239, Figure 12.1: ajt/Shutterstock
Page 259, Figure 13.2, bursting star: Arcady/Shutterstock
Page 269, Figure 13.5: Verdandi/123RF
Page 277, Figure 14.1: Tatyana Pronina/Shutterstock
Page 291, Figure 15.2: kornilov007/Shutterstock
Page 291, hammer: bozmp/Shutterstock
Pages 306, Figure 15.3: Figure based on a screen shot from codescene.io
Pages 307, Figure 15.4: Figure based on a screen shot from codescene.io

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2021944424

ISBN-13: 978-0-13-746440-1
ISBN-10: 0-13-746440-1

ScoutAutomatedPrintCode

To my parents:
My mother, Ulla Seemann, to whom I owe my attention to detail.
My father, Leif Seemann, from whom I inherited my contrarian streak.

This page intentionally left blank

"The future is already here — it's just not very evenly distributed"
—*William Gibson*

This page intentionally left blank

Contents

Series Editor Foreword

My grandson is learning to code.

Yes, you read that right. My 18-year-old grandson is learning to program computers. Who's teaching him? His aunt, my youngest daughter, who was born in 1986, and who 16 months ago decided to change careers from chemical engineering to programming. And who do they both work for? My eldest son, who along with my youngest son, is in the process of starting up his second software consultancy.

Yeah, software runs in the family. And, yeah, I've been programming for a long, long time.

Anyway, my daughter asked me to spend an hour with my grandson teaching him about the basics and the beginnings of computer programming. So we started up a Tuple session and I lectured him on what computers were, and how they got started, and what early computers looked like, and . . . well, you know.

By the end of the lecture I was coding up the algorithm for multiplying two binary integers, in PDP-8 assembly language. For those of you who aren't aware, the PDP-8 had no multiply instruction; you had to write an algorithm

to multiply numbers. Indeed, the PDP-8 didn't even have a subtract instruction; you had to use two's complement and add a pseudo-negative number (let the reader understand).

As I finished up the coding example, it occurred to me that I was scaring my grandson to death. I mean, when I was 18 this kind of geeky detail thrilled me; but maybe it wasn't so attractive to an 18-year-old whose aunt is trying to teach him how to write simple Clojure programs.

Anyway, it made me think of just how hard programming actually is. And it is hard. It's really hard. It may be the hardest thing that humans have ever attempted.

Oh, I don't mean it's hard to write the code to calculate a bunch of prime numbers, or a Fibonacci sequence, or a simple bubble sort. That's not too hard. But an Air Traffic Control system? A luggage management system? A bill of materials system? *Angry Birds*? Now that's hard. That's really, really hard.

I've known Mark Seemann for quite a few years now. I don't remember ever actually meeting him. It may be that we have never actually been together in the same room. But he and I have interacted quite a bit in professional newsgroups and social networks. He's one of my favourite people to disagree with.

He and I disagree on all kinds of things. We disagree on static versus dynamic typing. We disagree on operating systems and languages. We disagree on, well, lots of intellectually challenging things. But disagreeing with Mark is something you have to do very carefully because the logic of his arguments is impeccable.

So when I saw this book, I thought about how much fun it was going to be to read through and disagree with. And that's exactly what happened. I read through it. I disagreed with some things. And I had fun trying to find a way to make my logic supersede his. I think I may have even succeeded in one or two cases—in my head—maybe.

But that's not the point. The point is that software is hard; and much of the last seven decades have been spent trying to find ways to make it a little bit easier. What Mark has done in this book is to gather all the best ideas from those seven decades and compile them in one place.

More than that, he has organized them into a set of heuristics and techniques, and placed them in the order that you would execute them. Those heuristics and techniques build on each other, helping you move from stage to stage while developing a software project.

In fact, Mark develops a software project throughout the pages of this book, while explaining each stage and the heuristics and techniques that benefit that stage.

Mark uses C# (one of the things I disagree with ;-), but that's not relevant. The code is simple, and the heuristics and techniques are applicable to any other language you might be using.

He covers things such as Checklists, TDD, Command Query Separation, Git, Cyclomatic Complexity, Referential Transparency, Vertical Slicing, Legacy Strangulation, and Outside-In Development, just to mention a few.

Moreover, there are gems scattered literally everywhere throughout these pages. I mean, you'll be reading along, and all of a sudden he'll say something like, "Rotate your test function 90 degrees and see if you can balance it on the Act of the Arrange/Act/Assert triplet" or "The goal is not to write code fast. The goal is sustainable software" or "Commit database schema to git".

Some of these gems are profound, some are just idle mentions, others are speculations, but all of them are examples of the deep insight that Mark has acquired over the years.

So read this book. Read it carefully. Think through Mark's impeccable logic. Internalise these heuristics and techniques. Stop and consider the insightful gems as they pop out at you. And just maybe, when it comes time for you to lecture your grandchildren, you won't scare the devil out of them.

—Robert C. Martin

This page intentionally left blank

Preface

In the second half of the 2000s, I began doing technical reviews for a publisher. After reviewing a handful of books, the editor contacted me about a book on Dependency Injection.

The overture was a little odd. Usually, when they contacted me about a book, it would already have an author and a table of contents. This time, however, there was none of that. The editor just requested a phone call to discuss whether the book's subject matter was viable.

I thought about it for a few days and found the topic inspiring. At the same time, I couldn't see the need for an entire book. After all, the knowledge was out there: blog posts, library documentation, magazine articles, even a few books all touched on related topics.

On reflection, I realised that, while the information was all out there, it was scattered, and used inconsistent and sometimes conflicting terminology. There'd be value in collecting that knowledge and presenting it in a consistent pattern language.

Two years later, I was the proud author of a published book.

After some years had gone by, I began to think about writing another book. Not this one, but a book about some other topic. Then I had a third idea, and a fourth, but not this one.

A decade went by, and I began to realise that when I consulted teams on writing better code, I'd suggest practices that I'd learned from better minds than mine. And again, I realised that most of that knowledge is already available, but it's scattered, and few people have explicitly connected the dots into a coherent description of how to develop software.

Based on my experience with the first book, I know that there's value in collecting disparate information and presenting it in a consistent way. This book is my attempt at creating such a package.

Who Should Read This Book

This book is aimed at programmers with at least a few years of professional experience. I expect readers to have suffered through a few bad software development projects; to have experience with unmaintainable code. I also expect readers seeking to improve.

The core audience is 'enterprise developers'—particularly back-end developers. I've spent most of my career in that realm, so this simply reflects my own expertise. But if you're a front-end developer, a games programmer, a development tools engineer, or something else entirely, I expect you will still gain a lot from reading this book.

You should be comfortable reading code in a compiled, object-oriented language in the C family. While I've been a C# programmer for most of my career, I've learned a lot from books with example code in C++ or Java[1]. This book turns the tables: Its example code is in C#, but I hope that Java, TypeScript, or C++ developers find it useful, too.

1. If you're curious about which books I mean, take a look at the bibliography.

Prerequisites

This isn't a beginner's book. While it deals with how to organise and structure source code, it doesn't cover the most basic details. I expect that you already understand why indentation is helpful, why long methods are problematic, that global variables are bad, and so on. I don't expect you to have read *Code Complete* [65], but I assume that you know of some of the basics covered there.

A Note for Software Architects

The term 'architect' means different things to different people, even when the context is constrained to software development. Some architects focus on the big picture; they help an entire organisation succeed with its endeavours. Other architects are deep in the code and mainly concerned with the sustainability of a particular code base.

To the degree that I'm a software architect, I'm the latter kind. My expertise is in how to organise source code so that it addresses long-term business goals. I write about what I know, so to the degree this book is useful to architects, it will be that type of architect.

You'll find no content about Architecture Tradeoff Analysis Method (ATAM), Failure Mode and Effects Analysis (FMEA), service discovery, and so on. That kind of architecture is outside the scope of this book.

Organisation

While this is a book about methodologies, I've structured it around a code example that runs throughout the book. I decided to do it that way in order to make the reading experience more compelling than a typical 'pattern catalogue'. One consequence of this decision is that I introduce practices and heuristics when they fit the 'narrative'. This is also the order in which I typically introduce the techniques when I coach teams.

The narrative is structured around a sample code base that implements a restaurant reservation system. The source code for that sample code base is available at informit.com/title/9780137464401.

If you want to use the book as a handbook, I've included an appendix with a list of all the practices and information about where in the book you can read more.

About the Code Style

The example code is written in C#, which is a language that has rapidly evolved in recent years. It's picking up more and more syntax ideas from functional programming; as an example, *immutable record types* were released while I was writing the book. I've decided to ignore some of these new language features.

Once upon a time, Java code looked a lot like C# code. Modern C# code, on the other hand, doesn't look much like Java.

I want the code to be comprehensible to as many readers as possible. Just as I've learned much from books with Java examples, I want readers to be able to use this book without knowing the latest C# syntax. Thus, I'm trying to stick to a conservative subset of C# that ought to be legible to other programmers.

This doesn't change the concepts presented in the book. Yes, in some instances, a more succinct C#-specific alternative is possible, but that would just imply that extra improvements are available.

To Var or Not to Var

The `var` keyword was introduced to C# in 2007. It enables you to declare a variable without explicitly stating its type. Instead, the compiler infers the type from the context. To be clear, variables declared with `var` are exactly as statically typed as variables declared with explicit types.

For a long time the use of this keyword was controversial, but most people now use it; I do, too, but I occasionally encounter pockets of resistance.

While I use `var` professionally, writing code for a book is a slightly different context. Under normal circumstances, an IDE isn't far away. A modern development environment can quickly tell you the type of an implicitly typed variable, but a book can't.

I have, for that reason, occasionally chosen to explicitly type variables. Most of the example code still uses the `var` keyword because it makes the code shorter, and line width is limited in a printed book. In a few cases, though, I've deliberately chosen to explicitly declare a variable's type, in the hope that it makes the code easier to understand when read in a book.

Code Listings

The majority of the code listings are taken from the same sample code base. It's a Git repository, and the code examples are taken from various stages of development. Each such code listing includes a relative path to the file in question. Part of that file path is a Git commit ID.

For example, listing 2.1 includes this relative path: *Restaurant/f729ed9/Restaurant.RestApi/Program.cs*. This means that the example is taken from commit ID `f729ed9`, and the file is `Restaurant.RestApi/Program.cs`. In other words, to view this particular version of the file, you check out that commit:

```
$ git checkout f729ed9
```

When you've done that, you can now explore the `Restaurant.RestApi/Program.cs` file in its full, executable context.

A Note on the Bibliography

The bibliography contains a mix of resources, including books, blog posts, and video recordings. Many of my sources are online, so I have of course supplied URLs. I've made an effort to mostly include resources that I have reason to believe have a stable presence on the Internet.

Still, things change. If you're reading this book in the future, and a URL has become invalid, try an internet archive service. As I'm writing this, https://archive.org is the best candidate, but that site could also be gone in the future.

Quoting Myself

Apart from other resources, the bibliography also includes a list of my own work. I'm aware that, as far as making a case, quoting myself doesn't constitute a valid argument in itself.

I'm not including my own work as a sleight of hand. Rather, I'm including these resources for the reader who might be interested in more details. When I cite myself, I do it because you may find an expanded argument, or a more detailed code example, in the resource I point to.

Acknowledgements

I'd like to thank my wife Cecilie for love and support during all the years we've been together, and my children Linea and Jarl for staying out of trouble.

Apart from family, my first thanks go to my invaluable long-time friend Karsten Strøbæk, who not only has tolerated my existence for 25 years, but who was also the first reviewer on this book. He also helped me with various LaTeX tips and tricks, and added more entries to the index than I did.

I'd also like to thank Adam Tornhill for his feedback on the section about his work.

I'm indebted to Dan North for planting the phrase *Code That Fits in Your Head* in my subconscious, which might have happened as early as 2011 [72].

About the Author

Mark Seemann is a bad economist who's found a second career as a programmer, and he has worked as a web and enterprise developer since the late 1990s. As a young man Mark wanted to become a rock star, but unfortunately had neither the talent nor the looks – later, however, he became a Certified Rockstar Developer. He has also written a Jolt Award-winning book about Dependency Injection, given more than a 100 international conference talks, and authored video courses for both Pluralsight and Clean Coders. He has regularly published blog posts since 2006. He lives in Copenhagen with his wife and two children.

This page intentionally left blank

12 TROUBLESHOOTING

Professional software development consists of more than feature development. There are also meetings, time reports, compliance activities, and ... defects.

You run into errors and problems all the time. Your code doesn't compile, the software doesn't do what it's supposed to, it runs too slowly, et cetera.

The better you get at solving problems, the more productive you are. Most of your troubleshooting skills may be based on "the shifting sands of individual experience" [4], but there *are* techniques that you can apply.

This chapter presents some of them.

12.1 UNDERSTANDING

The best advice I can think of is this:

> Try to understand what's going on.

If you don't understand why something doesn't work[1], then make understanding it a priority. I've witnessed a fair amount of 'programming by coincidence' [50]: throw enough code at the wall to see what sticks. When it looks as though the code works, developers move on to the next task. Either they don't understand why the code works, or they may fail to understand that it doesn't, really.

If you understand the code from the beginning, chances are that it'll be easier to troubleshoot.

12.1.1 Scientific Method

When a problem manifests, most people jump straight into troubleshooting mode. They want to *address* the problem. For people who program by coincidence [50], addressing a problem typically involves trying various incantations that may have worked before on a similar problem. If the first magic spell doesn't work, they move on to the next. This can involve restarting a service, rebooting a computer, running a tool with elevated privileges, changing small pieces of code, calling poorly-understood routines, etc. When it looks like the problem has disappeared, they call it a day without trying to understand why [50].

Needless to say, this isn't an effective way to deal with problems.

Your first reaction to a problem should be to understand why it's happening. If you have absolutely no idea, ask for help. Usually, though, you already have some inclination of what the problem may be. In that case, adopt a variation of the scientific method [82]:

- Make a prediction. This is called a *hypothesis*.
- Perform the experiment.
- Compare outcome to prediction. Repeat until you understand what's going on.

1. Or, if you don't understand why something *does* work.

Don't be intimidated by the term 'scientific method'. You don't have to don a lab coat or design a randomised controlled double-blind trial. But *do* try to come up with a falsifiable hypothesis. This might simply be a prediction, such as *"if I reboot the machine, the problem goes away,"* or *"if I call this function, the return value will be 42."*

The difference between this technique and 'programming by coincidence' is that the goal of going through these motions isn't to address the problem. The goal is to understand it.

A typical experiment could be a unit test, with a hypothesis that if you run it, it'll fail. See subsection 12.2.1 for more details.

12.1.2 Simplify

Consider if *removing* some code can make a problem go away.

The most common reaction to a problem is to add more code to address it. The unspoken line of reasoning seems to be that the system 'works', and the problem is just an aberration. Thus, the reasoning goes, if the problem is a special case, it should be solved with more code to handle that special case.

This may occasionally be the case, but it's more likely that the problem is a manifestation of an underlying implementation error. You'd be surprised how often you can solve problems by *simplifying* the code.

I've seen plenty of examples of such an 'action bias' in our industry. People who solve problems I never have because I work hard to keep my code simple:

- People develop complex Dependency Injection Containers [25] instead of just composing object graphs in code.
- People develop complicated 'mock object libraries' instead of writing mostly pure functions.
- People create elaborate package restore schemes instead of just checking dependencies into source control.

- People use advanced diff tools instead of merging more frequently.
- People use convoluted object-relational mappers (ORMs) instead of learning (and maintaining) a bit of SQL.

I could go on.

To be fair, coming up with a simpler solution is *hard*. For example, it took me a decade of erecting increasingly more elaborate contraptions in object-oriented code before I found simpler solutions. It turns out that many things that are difficult in traditional object-oriented programming are simple in functional programming. Once I learned about some of these concepts, I found ways to use them in object-oriented contexts, too.

The point is that a catchphrase like KISS[2] is useless in itself, because *how* does one keep things simple?

You often have to be *smart* to keep it simple[3], but look for simplicity anyway. Consider if there's a way you can solve the problem by *deleting* code.

12.1.3 RUBBER DUCKING

Before we discuss some specific problem-solving practices, I want to share some general techniques. It's not unusual to be stuck on a problem. How do you get unstuck?

You may be staring at a problem with no clue as to how to proceed. As the above advice goes, your first priority should be to understand the problem. What do you do if you're drawing a blank?

If you don't manage your time, you can be stuck with a problem for a long time, so *do* manage your time. Time-box the process. For example, set aside 25 minutes to look at the problem. If, after the time is up, you've made no progress, take a break.

2. Keep It Simple, Stupid.
3. Rich Hickey discusses simplicity in *Simple Made Easy* [45]. I owe much of my perspective on simplicity to that talk.

When you take a break, physically remove yourself from the computer. Go get a cup of coffee. Something happens in your brain when you get out of your chair and away from the screen. After a couple of minutes away from the problem, you'll likely begin to think about something else. Perhaps you meet a colleague as you're moving about. Perhaps you discover that the coffee machine needs a refill. Whatever it is, it temporarily takes your mind off the problem. That's often enough to give you a fresh perspective.

I've lost count of the number of times I return to a problem after a stroll, only to realise that I've been thinking about it the wrong way.

If walking about for a few minutes isn't enough, try asking for help. If you have a colleague to bother, do that.

I've experienced this often enough: I start explaining the problem, but halfway in, I break off in mid-sentence: *"Never mind, I've just gotten an idea!"*

The mere act of explaining a problem tends to produce new insight.

If you don't have a colleague, you may try explaining the problem to a rubber duck, such as the one shown in figure 12.1.

Figure 12.1 A rubber duck. Talk to it. It'll solve your problems.

It doesn't really have to be a rubber duck, but the technique is known as *rubber ducking* because one programmer actually did use one [50].

Instead of using a rubber duck, I typically begin writing a question on the Stack Overflow Q&A site. More often than not, I realise what the problem is before I'm done formulating the question[4].

And if realisation *doesn't* come, I have a written question that I can publish.

12.2 Defects

I once started in a new job in a small software startup. I soon asked my co-workers if they'd like to use test-driven development. They hadn't used it before, but they were keen on learning new things. After I'd shown them the ropes, they decided that they liked it.

A few months after we'd adopted test-driven development, the CEO came by to talk to me. He mentioned in passing that he'd noticed that since we'd started using tests, defects in the wild had significantly dropped.

That still makes me proud to this day. The shift in quality was so dramatic that the CEO had noticed. Not by running numbers or doing a complex analysis, but simply because it was so significant that it called attention to itself.

You can reduce the number of defects, but you can't eliminate them. But do yourself a favour: don't let them accumulate.

> The ideal number of defects is zero.

Zero bugs isn't as unrealistic as it sounds. In lean software development, this is known as *building quality in* [82]. Don't push defects in front of you to 'deal with them later'. In software development, *later* is *never*.

4. When that happens, I *don't* succumb to the sunk cost fallacy. Even if I've spent time writing the question, I usually delete it because I deem that it's not, after all, of general interest.

When a bug appears, make it a priority to address it. Stop what you're doing[5] and fix the defect instead.

12.2.1 Reproduce Defects as Tests

Initially, you may not even understand what the problem is, but when you think that you do, perform an experiment: The understanding should enable you to formulate a hypothesis, which again enables you to design an experiment.

Such an experiment may be an automated test. The hypothesis is that when you run the test, it'll *fail*. When you actually do run the test, if it *does* fail, you've validated the hypothesis. As a bonus, you also have a failing test that reproduces the defect, and that will later serve as a regression test.

If, on the other hand, the test succeeds, the experiment failed. This means that your hypothesis was wrong. You'll need to revise it so that you can design a new experiment. You may need to repeat this process more than once.

When you finally have a failing test, 'all' you have to do is to make it pass. This can occasionally be difficult, but in my experience, it usually isn't. The hard part of addressing a defect is understanding and reproducing it.

I'll show you an example from the online restaurant reservation system. While I was doing some exploratory testing I noticed something odd when I updated a reservation. Listing 12.1 shows an example of the issue. Can you spot the problem?

The problem is that the `email` property holds the `name`, and vice versa. It seems that I accidentally switched them around somewhere. That's the initial hypothesis, but it may take a little investigation to figure out *where*.

Have I not been following test-driven development? Then how could this happen?

5. Isn't it wonderful that with Git you can simply stash your current work?

Listing 12.1 Updating a reservation with a PUT request. A defect is manifest in this interaction. Can you spot it?

```
PUT /reservations/21b4fa1975064414bee402bbe09090ec HTTP/1.1
Content-Type: application/json
{
  "at": "2022-03-02 19:45",
  "email": "pan@example.com",
  "name": "Phil Anders",
  "quantity": 2
}

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
{
  "id": "21b4fa1975064414bee402bbe09090ec",
  "at": "2022-03-02T19:45:00.0000000",
  "email": "Phil Anders",
  "name": "pan@example.com",
  "quantity": 2
}
```

This could happen because I'd implemented `SqlReservationsRepository`[6] as a Humble Object [66]. This is an object so simple that you may decide not to test it. I often use the rule of thumb that if the cyclomatic complexity is *1*, a test (also with a cyclomatic complexity of *1*) may not be warranted.

Even so, you can still make mistakes even when the cyclomatic complexity is *1*. Listing 12.2 shows the offending code. Can you spot the problem?

Given that you already know what the problem is, you can probably guess that the `Reservation` constructor expects the `email` argument before the `name`. Since both parameters are declared as `string`, though, the compiler doesn't complain if you accidentally swap them. This is another example of stringly typed code [3], which we should avoid[7].

6. See for example listing 4.19.

7. One way to avoid stringly typed code is to introduce `Email` and `Name` classes that wrap their respective `string` values. This prevents some cases of accidentally swapping these two arguments, but as it turned out when I did it, it wasn't entirely foolproof. You can consult the example code's Git repository if you're interested in the details. The bottom line was that I felt that an integration test was warranted.

Listing 12.2 The offending code fragment that causes the defect shown in listing 12.1. Can you spot the programmer error?
(Restaurant/d7b74f1/Restaurant.RestApi/SqlReservationsRepository.cs)

```
using var rdr =
    await cmd.ExecuteReaderAsync().ConfigureAwait(false);
if (!rdr.Read())
    return null;

return new Reservation(
    id,
    (DateTime)rdr["At"],
    (string)rdr["Name"],
    (string)rdr["Email"],
    (int)rdr["Quantity"]);
```

It's easy enough to address the defect, but if I can make the mistake once, I can make it again. Thus, I want to prevent a regression. Before fixing the code, write a failing test that reproduces the bug. Listing 12.3 shows the test I wrote. It's an integration test that verifies that if you update a reservation in the database and subsequently read it, you should receive a reservation equal to the one you saved. That's a reasonable expectation, and it reproduces the error because the `ReadReservation` method swaps `name` and `email`, as shown in listing 12.2.

That `PutAndReadRoundTrip` test is an integration test that involves the database. This is new. So far in this book, all tests have been running without external dependencies. Involving the database is worth a detour.

12.2.2 Slow Tests

Bridging the gap between a programming language's perspective on data and a relational database is error-prone[8], so why not test such code?

In this subsection, you'll see an outline of how to do that, but there's a problem: such tests tend to be slow. They tend to be orders of magnitudes slower than in-process tests.

8. Proponents of object-relational mappers (ORMs) might argue that this makes the case for such a tool. As I've stated elsewhere in this book, I consider ORMs a waste of time: they create more problems than they solve. If you disagree, then feel free to skip this subsection.

Listing 12.3 Integration test of `SqlReservationsRepository`.
(Restaurant/645186b/Restaurant.RestApi.SqlIntegrationTests/SqlReservationsRepositoryTests.cs)

```
[Theory]
[InlineData("2032-01-01 01:12", "z@example.net", "z", "Zet", 4)]
[InlineData("2084-04-21 23:21", "q@example.gov", "q", "Quu", 9)]
public async Task PutAndReadRoundTrip(
    string date,
    string email,
    string name,
    string newName,
    int quantity)
{
    var r = new Reservation(
        Guid.NewGuid(),
        DateTime.Parse(date, CultureInfo.InvariantCulture),
        new Email(email),
        new Name(name),
        quantity);
    var connectionString = ConnectionStrings.Reservations;
    var sut = new SqlReservationsRepository(connectionString);
    await sut.Create(r);

    var expected = r.WithName(new Name(newName));
    await sut.Update(expected);
    var actual = await sut.ReadReservation(expected.Id);

    Assert.Equal(expected, actual);
}
```

The time it takes to execute a test suite matters, particularly for developer tests that you continually run. When you refactor with the test suite as a safety net, it doesn't work if it takes half an hour to run all tests. When you follow the Red Green Refactor process for test-driven development, it doesn't work if running the tests takes five minutes.

The maximum time for such a test suite should be ten seconds. If it's much longer than that, you'll lose focus. You'll be tempted to look at your email, Twitter, or Facebook while the tests run.

You can easily eat into such a ten-second budget if you involve a database. Therefore, move such tests to a second stage of tests. There are many ways you can do this, but a pragmatic way is to simply create a *second* Visual Studio

solution to exist side-by-side with the day-to-day solution. When you do that, remember to also update the build script to run this new solution instead, as shown in listing 12.4.

Listing 12.4 Build script running all tests. The `Build.sln` file contains both unit and integration tests that use the database. Compare with listing 4.2. *(Restaurant/645186b/build.sh)*

```
#!/usr/bin/env bash
dotnet test Build.sln --configuration Release
```

The `Build.sln` file contains the production code, the unit test code, as well as integration tests that use the database. I do day-to-day work that doesn't involve the database in another Visual Studio solution called `Restaurant.sln`. That solution only contains the production code and the unit tests, so running all tests in that context is much faster.

The test in listing 12.3 is part of the integration test code, so only runs when I run the build script, or if I explicitly choose to work in the `Build.sln` solution instead of in `Restaurant.sln`. It's sometimes practical to do that, if I need to perform a refactoring that involves the database code.

I don't want to go into too much detail about how the test in listing 12.3 works, because it's specific to how .NET interacts with SQL Server. If you're interested in the details, they're all available in the accompanying example code base, but briefly, all the integration tests are adorned with a `[UseDatabase]` attribute. This is a custom attribute that hooks into the xUnit.net unit testing framework to run some code before and after each test case. Thus, each test case is surrounded with behaviour like this:

1. Create a new database and run all DDL[9] scripts against it.
2. Run the test.
3. Tear down the database.

9. Data Definition Language, typically a subset of SQL. See listing 4.18 for an example.

Yes, each test *creates a new database* only to delete it again some milliseconds later[10]. That *is* slow, which is why you don't want such tests to run all the time.

Defer slow tests to a second stage of your build pipeline. You can do it as outlined above, or by defining new steps that only run on your Continuous Integration server.

12.2.3 Non-deterministic Defects

After running the restaurant reservation system for some time, the restaurant's maître d' files a bug: once in a while, the system seems to allow overbooking. She can't deliberately reproduce the problem, but the state of the reservations database can't be denied. Some days contain more reservations than the business logic shown in listing 12.5 allows. What's going on?

You peruse the application logs[11] and finally figure it out. Overbooking is a possible race condition. If a day is approaching capacity and two reservations arrive simultaneously, the `ReadReservations` method might return the same set of rows to both threads, indicating that a reservation is possible. As figure 12.2 shows, each thread determines that it can accept the reservation, so it adds a new row to the table of reservations.

This is clearly a defect, so you should reproduce it with a test. The problem is, however, that this behaviour isn't deterministic. Automated tests are supposed to be deterministic, aren't they?

It is, indeed, best if tests are deterministic, but do entertain, for a moment, the notion that nondeterminism may be acceptable. In which way could this be?

10. Whenever I explain this approach to integration testing with a database, I'm invariably met with the reaction that one can, instead, test by rolling back transactions. Yes, except that this means that you can't test database transaction behaviour. Also, using transaction rollback *may* be faster, but have you measured? I have, once, and found no significant difference. See also section 15.1 for my general position on performance optimisation.
11. See subsection 13.2.1.

Tests can fail in two ways: A test may indicate a failure where none is; this is called a false positive. A test may also fail to indicate an actual error; this is called a false negative.

Listing 12.5 Apparently, there's a bug in this code that allows overbooking. What could be the problem? *(Restaurant/dd05589/Restaurant.RestApi/ReservationsController.cs)*

```
[HttpPost]
public async Task<ActionResult> Post(ReservationDto dto)
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));

    var id = dto.ParseId() ?? Guid.NewGuid();
    Reservation? r = dto.Validate(id);
    if (r is null)
        return new BadRequestResult();

    var reservations = await Repository
        .ReadReservations(r.At)
        .ConfigureAwait(false);
    if (!MaitreD.WillAccept(DateTime.Now, reservations, r))
        return NoTables500InternalServerError();

    await Repository.Create(r).ConfigureAwait(false);
    await PostOffice.EmailReservationCreated(r).ConfigureAwait(false);

    return Reservation201Created(r);
}
```

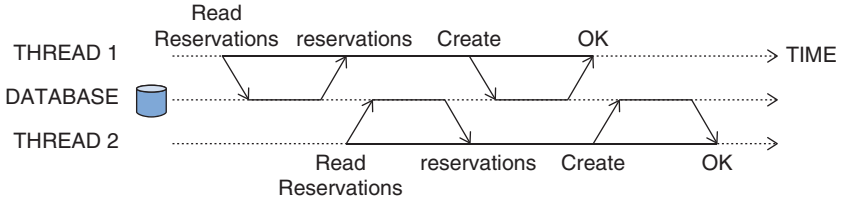


Figure 12.2 A race condition between two threads (e.g. two HTTP clients) concurrently trying to make a reservation.

False positives are problematic because they introduce noise, and thereby decrease the signal-to-noise ratio of the test suite. If you have a test suite that often fails for no apparent reason, you stop paying attention to it [31].

False negatives aren't quite as bad. Too many false negatives could decrease your trust in a test suite, but they introduce no noise. Thus, at least, you know that if a test suite is failing, there *is* a problem.

One way to deal with the race condition in the reservation system, then, is to reproduce it as the non-deterministic test in listing 12.6.

Listing 12.6 Non-deterministic test that reproduces a race condition. *(Restaurant/98ab6b5/Restaurant.RestApi.SqlIntegrationTests/ConcurrencyTests.cs)*

```
[Fact]
public async Task NoOverbookingRace()
{
    var start = DateTimeOffset.UtcNow;
    var timeOut = TimeSpan.FromSeconds(30);
    var i = 0;
    while (DateTimeOffset.UtcNow - start < timeOut)
        await PostTwoConcurrentLiminalReservations(
            start.DateTime.AddDays(++i));
}
```

This test method is only an orchestrator of the actual unit test. It keeps running the `PostTwoConcurrentLiminalReservations` method in listing 12.7 for 30 seconds, over and over again, to see if it fails. The assumption, or hope, is that if it can run for 30 seconds without failing, the system may actually have the correct behaviour.

There's no guarantee that this is the case. If the race condition is as scarce as hen's teeth, this test could produce a false negative. That's not my experience, though.

When I wrote this test, it only ran for a few seconds before failing. That gives me some confidence that the 30-second timeout is a sufficiently safe margin, but I admit that I'm guessing; it's another example of the art of software engineering.

It turned out that the system had the same bug when updating existing reservations (as opposed to creating new ones), so I also wrote a similar test for that case.

Listing 12.7 The actual test method orchestrated by the code in listing 12.6. It attempts to post two concurrent reservations. The state of the system is that it's almost sold out (the capacity of the restaurant is ten, but nine seats are already reserved), so only one of those reservations should be accepted.
(Restaurant/98ab6b5/Restaurant.RestApi.SqlIntegrationTests/ConcurrencyTests.cs)

```
private static async Task PostTwoConcurrentLiminalReservations(
    DateTime date)
{
    date = date.Date.AddHours(18.5);
    using var service = new RestaurantService();
    var initialResp =
        await service.PostReservation(new ReservationDtoBuilder()
            .WithDate(date)
            .WithQuantity(9)
            .Build());
    initialResp.EnsureSuccessStatusCode();

    var task1 = service.PostReservation(new ReservationDtoBuilder()
        .WithDate(date)
        .WithQuantity(1)
        .Build());
    var task2 = service.PostReservation(new ReservationDtoBuilder()
        .WithDate(date)
        .WithQuantity(1)
        .Build());
    var actual = await Task.WhenAll(task1, task2);

    Assert.Single(actual, msg => msg.IsSuccessStatusCode);
    Assert.Single(
        actual,
        msg => msg.StatusCode == HttpStatusCode.InternalServerError);
}
```

These tests are examples of slow tests that ought to be included only as second-stage tests as discussed in subsection 12.2.2.

There are various ways you can address a defect like the one discussed here. You can reach for the Unit of Work [33] design pattern. You can also deal with the issue at the architectural level, by introducing a durable queue with a single-threaded writer that consumes the messages from it. In any case, you need to serialise the reads and the writes involved in the operation.

I chose to go for a pragmatic solution: use .NET's lightweight transactions, as shown in listing 12.8. Surrounding the critical part of the `Post` method with a

`TransactionScope` effectively serialises[12] the reads and writes. That solves the problem.

Listing 12.8 The critical part of the `Post` method is now surrounded with a `TransactionScope`, which serialises the read and write methods. The highlighted code is new compared to listing 12.5. *(Restaurant/98ab6b5/Restaurant.RestApi/ReservationsController.cs)*

```
using var scope = new TransactionScope(
    TransactionScopeAsyncFlowOption.Enabled);
var reservations = await Repository
    .ReadReservations(r.At)
    .ConfigureAwait(false);
if (!MaitreD.WillAccept(DateTime.Now, reservations, r))
    return NoTables500InternalServerError();

await Repository.Create(r).ConfigureAwait(false);
await PostOffice.EmailReservationCreated(r).ConfigureAwait(false);
scope.Complete();
```

In my experience, most defects can be reproduced as deterministic tests, but there's a residual that eludes this ideal. Multithreaded code infamously falls into that category. Of two evils, I prefer nondeterministic tests over no test coverage at all. Such tests will often have to run until they time out in order to give you confidence that they've sufficiently exercised the test case in question. You should, therefore, put them in a second stage of tests that only runs on demand and as part of your deployment pipeline.

12.3 BISECTION

Some defects can be elusive. When I developed the restaurant system I ran into one that took me most of a day to understand. After wasting hours following several false leads, I finally realised that I couldn't crack the nut only by staring long enough at the code. I had to use a *method*.

12. *Serialisability*, here, refers to making sure that database transactions behave as though they were serialised one after another [55]. It has nothing to do with converting objects to and from JSON or XML.

Fortunately, such a method exists. We can call it *bisection* for lack of a better word. In all its simplicity, it works like this:

1. Find a way to detect or reproduce the problem.
2. Remove half of the code.
3. If the problem is still present, repeat from step 2. If the problem goes away, restore the code you removed, and remove the other half. Again, repeat from step 2.
4. Keep going until you've whittled down the code that reproduces the problem to a size so small that you understand what's going on.

You can use an automated test to detect the problem, or use some ad hoc way to detect whether the problem is present or absent. The exact way you do this doesn't matter for the technique, but I find that an automated test is often the easiest way to go about it, because of the repetition involved.

I often use this technique when I *rubber duck* by writing a question on Stack Overflow. Good questions on Stack Overflow should come with a *minimal working example*. In most cases I find that the process of producing the minimal working example is so illuminating that I get unstuck before I have a chance to post the question.

12.3.1 Bisection with Git

You can also use the bisection technique with Git to identify the commit that introduced the defect. I ultimately used that with the problem I ran into.

I'd added a secure resource to the REST API to list the schedule for a particular day. A restaurant's maître d' can make a `GET` request against that resource to see the schedule for the day, including all reservations and who arrives when. The schedule includes names and emails of guests, so it shouldn't be available without authentication and authorisation[13].

This particular resource demands that a client presents a valid JSON Web Token (JWT). I'd developed this security feature with test-driven development and I had enough tests to feel safe.

13. For an example of what this looks like, see subsection 15.2.5.

Then one day, as I was interacting with the deployed REST API, I could no longer access this resource! I first thought that I'd supplied an invalid JWT, so I wasted hours troubleshooting that. Dead end.

It finally dawned on me that this security feature *had* worked. I'd interacted with the deployed REST API earlier and seen it work. At one time it worked, and now it didn't. In between these two known states a commit must have introduced the defect. If I could identify that particular code change, I might have a better chance of understanding the problem.

Unfortunately, there was some 130 commits between those two extremes.

Fortunately, I'd found an easy way to detect the problem, if given a commit.

This meant that I could use Git's `bisect` feature to identify the exact commit that caused the problem.

Git can run an automated bisection for you if you have an automated way to detect the problem. Usually, you don't. When you bisect, you're looking for a commit that introduced a defect that *went unnoticed at the time*. This means that even if you have an automated test suite, the tests didn't catch that bug.

For that reason, Git can also bisect your commits in an interactive session. You start such a session with `git bisect start`, as shown in listing 12.9.

Listing 12.9 The start of a Git bisect session. I ran it from Bash, but you can run it in any shell where you use Git. I've edited the terminal output by removing irrelevant data that Bash tends to show, so that it fits on the page.

```
~/Restaurant ((56a7092...))
$ git bisect start

~/Restaurant ((56a7092...)|BISECTING)
```

This starts an interactive session, which you can tell from the Git integration in Bash (it says `BISECTING`). If the current commit exhibits the defect you're investigating, you mark it as shown in listing 12.10

Listing 12.10 Marking a commit as bad in a bisect session.

```
$ git bisect bad

~/Restaurant ((56a7092...)|BISECTING)
```

If you don't provide a commit ID, Git is going to assume that you meant the current commit (in this case `56a7092`).

You now tell it about a commit ID that you know is good. This is the other extreme of the range of commits you're investigating. Listing 12.11 shows how that's done.

Listing 12.11 Marking a commit as good in a bisect session. I've trimmed the output a little to make it fit on the page.

```
$ git bisect good 58fc950
Bisecting: 75 revisions left to test after this (roughly 6 steps)
[3035c14...] Use InMemoryRestaurantDatabase in a test

~/Restaurant ((3035c14...)|BISECTING)
```

Notice that Git is already telling you how many iterations to expect. You can also see that it checked out a new commit (`3035c14`) for you. That's the half-way commit.

You now have to check whether or not the defect is present in this commit. You can run an automated test, start the system, or any other way you've identified to answer that question.

In my particular case, the half-way commit didn't have the defect, so I told Git, as shown in listing 12.12.

Listing 12.12 Marking the half-way commit as good in a bisect session. I've trimmed the output a little to make it fit on the page.

```
$ git bisect good
Bisecting: 37 revisions left to test after this (roughly 5 steps)
[aa69259...] Delete Either API

~/Restaurant ((aa69259...)|BISECTING)
```

Again, Git estimates how many more steps are left and checks out a new commit (`aa69259`).

Listing 12.13 Finding the commit responsible for the defect, using a Git bisect session.

```
$ git bisect bad
Bisecting: 18 revisions left to test after this (roughly 4 steps)
[75f3c56...] Delete redundant Test Data Builders

~/Restaurant ((75f3c56...)|BISECTING)
$ git bisect good
Bisecting: 9 revisions left to test after this (roughly 3 steps)
[8f93562...] Extract WillAcceptUpdate helper method

~/Restaurant ((8f93562...)|BISECTING)
$ git bisect good
Bisecting: 4 revisions left to test after this (roughly 2 steps)
[1c6fae1...] Extract ConfigureClock helper method

~/Restaurant ((1c6fae1...)|BISECTING)
$ git bisect good
Bisecting: 2 revisions left to test after this (roughly 1 step)
[8e1f1ce] Compact code

~/Restaurant ((8e1f1ce...)|BISECTING)
$ git bisect good
Bisecting: 0 revisions left to test after this (roughly 1 step)
[2563131] Extract CreateTokenValidationParameters method

~/Restaurant ((2563131...)|BISECTING)
$ git bisect bad
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[fa0caeb...] Move Configure method up

~/Restaurant ((fa0caeb...)|BISECTING)
$ git bisect good
2563131c2d06af8e48f1df2dccbf85e9fc8ddafc is the first bad commit
commit 2563131c2d06af8e48f1df2dccbf85e9fc8ddafc
Author: Mark Seemann <mark@example.com>
Date:   Wed Sep 16 07:15:12 2020 +0200

Extract CreateTokenValidationParameters method

Restaurant.RestApi/Startup.cs | 32 ++++++++++++++++++++-------------
1 file changed, 19 insertions(+), 13 deletions(-)

~/Restaurant ((fa0caeb...)|BISECTING)
```

I repeated the process for each step, marking the commit as either good or bad, depending on whether or not my verification step passed. This is shown in listing 12.13.

After just eight iterations, Git found the commit responsible for the defect. Notice that the last step tells you which commit is the 'first bad commit'.

Once I saw the contents of the commit, I immediately knew what the problem was and could easily fix it. I'm not going to tire you with a detailed description of the error, or how I fixed it. If you're interested, I wrote a blog post [101] with all the details, and you can also peruse the Git repository that accompanies the book.

The bottom line is that bisection is a potent technique for finding and isolating the source of an error. You can use it with or without Git.

12.4 Conclusion

There's a significant degree of personal experience involved in troubleshooting. I once worked in a team where a unit test failed on one developer's machine, while it passed on another programmer's laptop. The exact same test, the same code, the same Git commit.

We could have just shrugged and found a workaround, but we all knew that making the symptom go away without understanding the root cause tends to be a myopic strategy. The two developers worked together for maybe half an hour to reduce the problem to a minimal working example. Essentially, it boiled down to string comparison.

On the machine where the test failed, a comparison of strings would consider `"aa"` less than `"bb"`, and `"bb"` less than `"cc"`. That seems fine, doesn't it?

On the machine where the test *succeeded*, however, `"bb"` was still less than `"cc"`, but `"aa"` was *greater than* `"bb"`. What's going on?

At this point, I got involved, took one look at the repro and asked both developers what their 'default culture' was. In .NET, the 'default culture' is an

Ambient Context [25] that knows about culture-specific formatting rules, sort order, and so on.

As I expected, the machine that considered `"aa"` greater than `"bb"` was running with the Danish default culture, whereas the other machine used US English. The Danish alphabet has three extra letters (Æ, Ø, and Å) after Z, but the Å used to be spelled *Aa* in the old days, and since that spelling still exists in proper nouns, the *aa* combination is considered to be equivalent to *å*. Å being the last letter in the alphabet is considered greater than B.

It took me less than a minute to figure out what the problem was, because I'd run into enough problems with Danish sort orders earlier in my career. That's still the *shifting sands of individual experience*—the art of software engineering.

I'd never been able to identify the problem if my colleagues hadn't first used a methodology like bisection to reduce the problem to a simple symptom. Being able to produce a minimal working example is a superpower in software troubleshooting.

Notice what I *haven't* discussed in this chapter: debugging.

Too many people rely exclusively on debugging for troubleshooting. While I do occasionally use the debugger, I find the combination of the scientific method, automated testing, and bisection more efficient. Learn and use these more universal practices, because you can't use debugging tools in your production environment.

INDEX

B

C

D

G

H

P

Q

S

T

U