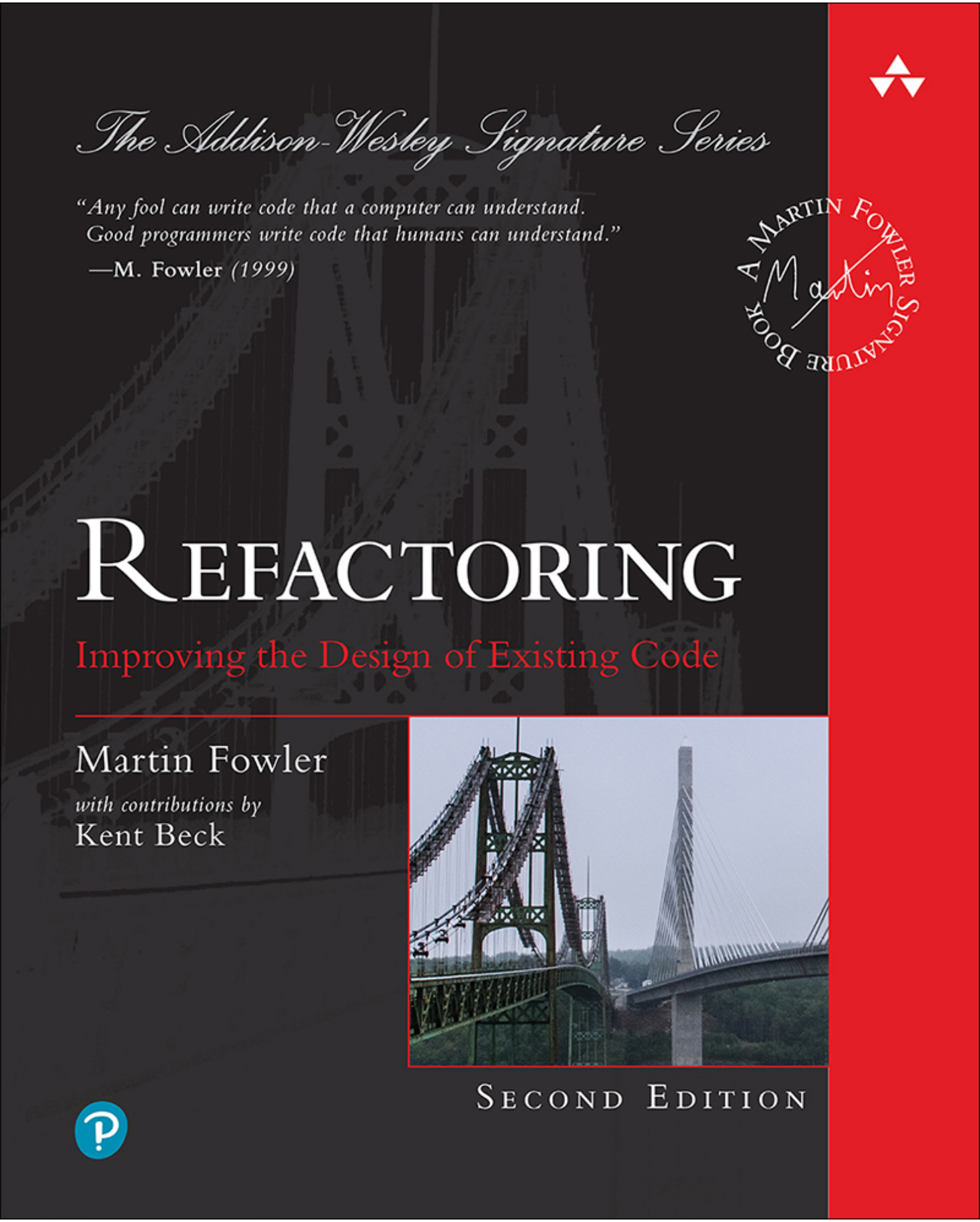

The Addison-Wesley Signature Series

"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand."
—M. Fowler (1999)

A Martin Fowler Signature Book

Refactoring

Improving the Design of Existing Code

Martin Fowler
with contributions by
Kent Beck

Second Edition

Refactoring

Improving the Design of Existing Code

Second Edition

Martin Fowler

with contributions by
Kent Beck

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

Library of Congress Control Number: 2018950015

Cover photo: Waldo-Hancock Bridge & Penobscot Narrows Bridge by Martin Fowler
Lightbulb graphic: Irina Adamovich/Shutterstock

ISBN-13: 978-0-13-475759-9
ISBN-10: 0-13-475759-9

1 18

Chapter 1
Refactoring: A First Example

How do I begin to talk about refactoring? The traditional way is by introducing the history of the subject, broad principles, and the like. When somebody does that at a conference, I get slightly sleepy. My mind starts wandering, with a low-priority background process polling the speaker until they give an example.

The examples wake me up because I can see what is going on. With principles, it is too easy to make broad generalizations—and too hard to figure out how to apply things. An example helps make things clear.

So I'm going to start this book with an example of refactoring. I'll talk about how refactoring works and will give you a sense of the refactoring process. I can then do the usual principles-style introduction in the next chapter.

With any introductory example, however, I run into a problem. If I pick a large program, describing it and how it is refactored is too complicated for a mortal reader to work through. (I tried this with the original book—and ended up throwing away two examples, which were still pretty small but took over a hundred pages each to describe.) However, if I pick a program that is small enough to be comprehensible, refactoring does not look like it is worthwhile.

I'm thus in the classic bind of anyone who wants to describe techniques that are useful for real-world programs. Frankly, it is not worth the effort to do all the refactoring that I'm going to show you on the small program I will be using. But if the code I'm showing you is part of a larger system, then the refactoring becomes important. Just look at my example and imagine it in the context of a much larger system.

THE STARTING POINT

In the first edition of this book, my starting program printed a bill from a video rental store, which may now lead many of you to ask: "What's a video rental store?" Rather than answer that question, I've reskinned the example to something that is both older

and still current.

Image a company of theatrical players who go out to various events performing plays. Typically, a customer will request a few plays and the company charges them based on the size of the audience and the kind of play they perform. There are currently two kinds of plays that the company performs: tragedies and comedies. As well as providing a bill for the performance, the company gives its customers "volume credits" which they can use for discounts on future performances—think of it as a customer loyalty mechanism.

The performers store data about their plays in a simple JSON file that looks something like this:

plays.json...

Click here to view code image

```
{
  "hamlet": {"name": "Hamlet", "type": "tragedy"},
  "as-like": {"name": "As You Like It", "type": "comedy"},
  "othello": {"name": "Othello", "type": "tragedy"}
}
```

The data for their bills also comes in a JSON file:

invoices.json...

Click here to view code image

```
[
  {
    "customer": "BigCo",
    "performances": [
      {
        "playID": "hamlet",
        "audience": 55
      },
      {
        "playID": "as-like",
        "audience": 35
      },
      {
        "playID": "othello",
        "audience": 40
      }
```

```
    ]
  }
]
```

The code that prints the bill is this simple function:

Click here to view code image

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency: "USD",
                         minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
    case "tragedy":
      thisAmount = 40000;
      if (perf.audience > 30) {
        thisAmount += 1000 * (perf.audience - 30);
      }
      break;
    case "comedy":
      thisAmount = 30000;
      if (perf.audience > 20) {
        thisAmount += 10000 + 500 * (perf.audience - 20);
      }
      thisAmount += 300 * perf.audience;
      break;
    default:
        throw new Error(`unknown type: ${play.type}`);
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience

    // print line for this order
    result += `  ${play.name}: ${format(thisAmount/100)} (${perf.audience}
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

Running that code on the test data files above results in the following output:

Click here to view code image

```
Statement for BigCo
  Hamlet: $650.00 (55 seats)
  As You Like It: $580.00 (35 seats)
  Othello: $500.00 (40 seats)
Amount owed is $1,730.00
You earned 47 credits
```

COMMENTS ON THE STARTING PROGRAM

What are your thoughts on the design of this program? The first thing I'd say is that it's tolerable as it is—a program so short doesn't require any deep structure to be comprehensible. But remember my earlier point that I have to keep examples small. Imagine this program on a larger scale—perhaps hundreds of lines long. At that size, a single inline function is hard to understand.

Given that the program works, isn't any statement about its structure merely an aesthetic judgment, a dislike of "ugly" code? After all, the compiler doesn't care whether the code is ugly or clean. But when I change the system, there is a human involved, and humans do care. A poorly designed system is hard to change—because it is difficult to figure out what to change and how these changes will interact with the existing code to get the behavior I want. And if it is hard to figure out what to change, there is a good chance that I will make mistakes and introduce bugs.

Thus, if I'm faced with modifying a program with hundreds of lines of code, I'd rather it be structured into a set of functions and other program elements that allow me to understand more easily what the program is doing. If the program lacks structure, it's usually easier for me to add structure to the program first, and then make the change I need.

When you have to add a feature to a program but the code is not structured in a convenient way, first refactor the program to make it easy to add the feature, then add the feature.

In this case, I have a couple of changes that the users would like to make. First, they want a statement printed in HTML. Consider what impact this change would have. I'm faced with adding conditional statements around every statement that adds a string to

the result. That will add a host of complexity to the function. Faced with that, most people prefer to copy the method and change it to emit HTML. Making a copy may not seem too onerous a task, but it sets up all sorts of problems for the future. Any changes to the charging logic would force me to update both methods—and to ensure they are updated consistently. If I'm writing a program that will never change again, this kind of copy-and-paste is fine. But if it's a long-lived program, then duplication is a menace.

This brings me to a second change. The players are looking to perform more kinds of plays: they hope to add history, pastoral, pastoral-comical, historical-pastoral, tragical-historical, tragical-comical-historical-pastoral, scene individable, and poem unlimited to their repertoire. They haven't exactly decided yet what they want to do and when. This change will affect both the way their plays are charged for and the way volume credits are calculated. As an experienced developer I can be sure that whatever scheme they come up with, they will change it again within six months. After all, when feature requests come, they come not as single spies but in battalions.

Again, that `statement` method is where the changes need to be made to deal with changes in classification and charging rules. But if I copy `statement` to `htmlStatement`, I'd need to ensure that any changes are consistent. Furthermore, as the rules grow in complexity, it's going to be harder to figure out where to make the changes and harder to do them without making a mistake.

Let me stress that it's these changes that drive the need to perform refactoring. If the code works and doesn't ever need to change, it's perfectly fine to leave it alone. It would be nice to improve it, but unless someone needs to understand it, it isn't causing any real harm. Yet as soon as someone does need to understand how that code works, and struggles to follow it, then you have to do something about it.

THE FIRST STEP IN REFACTORING

Whenever I do refactoring, the first step is always the same. I need to ensure I have a solid set of tests for that section of code. The tests are essential because even though I will follow refactorings structured to avoid most of the opportunities for introducing bugs, I'm still human and still make mistakes. The larger a program, the more likely it is that my changes will cause something to break inadvertently—in the digital age, frailty's name is software.

Since the `statement` returns a string, what I do is create a few invoices, give each invoice a few performances of various kinds of plays, and generate the statement strings. I then do a string comparison between the new string and some reference

strings that I have hand-checked. I set up all of these tests using a testing framework so I can run them with just a simple keystroke in my development environment. The tests take only a few seconds to run, and as you will see, I run them often.

An important part of the tests is the way they report their results. They either go green, meaning that all the strings are identical to the reference strings, or red, showing a list of failures—the lines that turned out differently. The tests are thus self-checking. It is vital to make tests self-checking. If I don't, I'd end up spending time hand-checking values from the test against values on a desk pad, and that would slow me down. Modern testing frameworks provide all the features needed to write and run self-checking tests.

Before you start refactoring, make sure you have a solid suite of tests. These tests must be self-checking.

As I do the refactoring, I'll lean on the tests. I think of them as a bug detector to protect me against my own mistakes. By writing what I want twice, in the code and in the test, I have to make the mistake consistently in both places to fool the detector. By double-checking my work, I reduce the chance of doing something wrong. Although it takes time to build the tests, I end up saving that time, with considerable interest, by spending less time debugging. This is such an important part of refactoring that I devote a full chapter to it (*Building Tests (85)*).

DECOMPOSING THE `STATEMENT` FUNCTION

When refactoring a long function like this, I mentally try to identify points that separate different parts of the overall behavior. The first chunk that leaps to my eye is the switch statement in the middle.

Click here to view code image

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;
```