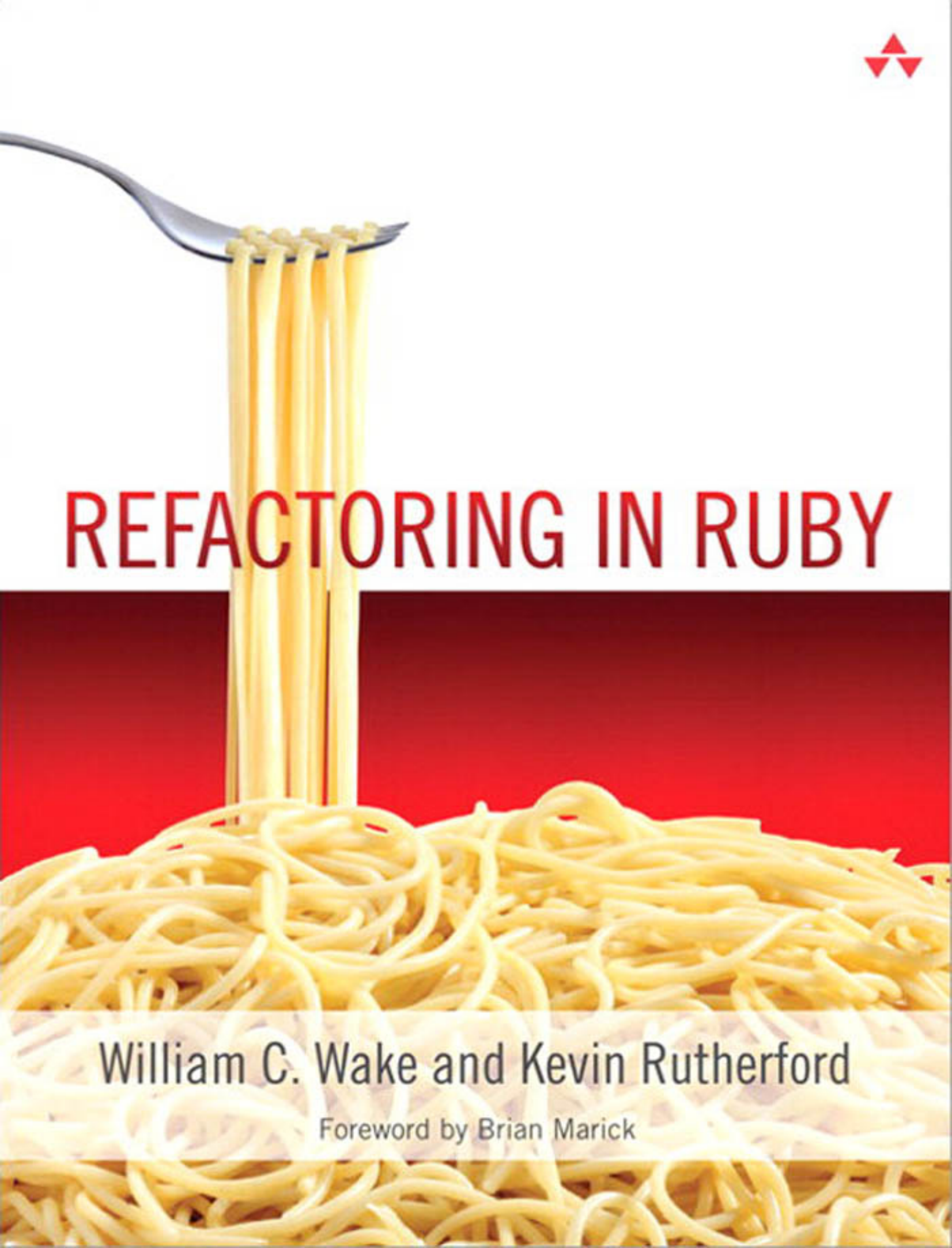
REFACTORING IN RUBY
William C. Wake and Kevin Rutherford
Foreword by Brian Marick

Library of Congress Cataloging-in-Publication Data
Wake, William C., 1960-
Refactoring in Ruby / William C. Wake, Kevin Rutherford.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-54504-6 (pbk. : alk. paper)
1. Software refactoring. 2. Ruby (Computer program language) I. Rutherford, Kevin. II. Title.
QA76.76.R42.W345 2009
005.1'17—dc22

2009032115

ISBN-13: 978-0-321-54504-6
ISBN-10: 0-321-54504-4
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, October 2009

Foreword

I want to give you two reasons to work through this book. The first reason is about *right now*, and the second is about *forevermore*.

The reason you need to work through this book *right now* is, well, us: You and me and all the other Ruby programmers out there. While Ruby's a language that, as the saying goes, makes simple things simple and hard things possible, and while we Ruby programmers are intelligent, virtuous, good-looking, kind to animals, and great fun at parties—we're still human. As such, what we make is often awkward, even if it's Ruby code.

So there's this vast and ever-growing sea of Ruby programmers out there, writing awkward Ruby code. I bet you're working on some of that code now, and I'm sure you'll be working on more of it soon. Do you want to be happy doing that? Or sad?

In the past ten years or so, we've learned that a wonderful way to be happy working on code is to *refactor* it as you go. Refactoring means that you change the code to be less awkward on the inside without changing what it does. It's something you can do in small, safe steps while adding features or fixing bugs. As you do, the code keeps getting more pleasant, so your life does too.

Before I give you the second reason to work through the book, I want to share my deepest fear: that you'll only read it, not work through it. That would be a horrible mistake. When I think of you doing that, I imagine all the wonderful tricks in the book entering your head through your eyes—and then far, far too many of them sliding right out of your ears, never to be recalled again. What tricks you do remember will be shuffled off to that part of the brain marked "For Rational Use Only," to be taken out rarely, on special occasions. Mere reading will not make you an expert.

You see, expert behavior is often a-rational. Experts typically *act appropriately* without needing to think through a problem. Indeed, experts often have difficulty explaining why a particular action was appropriate. That's because "thinking through a problem" is expensive, so the brain prefers more efficient routes to correct behavior. Those routes are created through repetition—like by doing the exercises in this book. (Gary Klein's *Sources of Power* is a good book about expert behavior, and Read Montague's *Why Choose This Book?* explains why the brain avoids what we think of as problem-solving.)

When it comes to the awkwardness this book teaches you how to correct, efficient thinking and automatic behavior are important. To get good at this stuff, it's not enough to be able to search for awkwardness—it has to leap out at you as you travel the code. Indeed, I'm happy that Kevin and Bill—like most who write about refactoring—describe awkwardness as "code smells." That's because smell is probably the most powerful, primitive, and least controllable of senses. When you open up a container and the smell of rotting meat hits your brain, you *move*. You act. The smell of rotting code should do the same, but it will only do so after practice blazes well-worn trails through your brain.

So: DO THE EXERCISES.

The reason this book will be valuable to you *forevermore* is that computers are strikingly unsuited to most problems that need solving. They pigheadedly insist that we squeeze every last drop of ambiguity out of a world that's flooded with it. That's a ridiculous … impossible … *inhuman* demand that we put up with only because computers are so *fast*. As a result of this fundamental mismatch—this requirement that we *make up* precision—it takes us a long time to craft a program that works well in the world.

The humble and effective way to arrive at such a program is to put a fledgling version out into the world, watch what happens, and then reshape it (the program, not the world—although people try that too) to make the mismatch less awkward. (And then do it again, and again.) That's an intellectual adventure, especially when you spot concepts implicit in the code that no one's ever quite recognized before, concepts that suddenly open up vast new possibilities and require only a few … well, maybe more than a few … minor … well, maybe not so minor … changes.

Without refactoring, and the style it promotes and supports, the changes the program needs will be too daunting too often. With it, you need nevermore look at a program with that familiar sense of hopeless dread.

And won't that be nice?

—Brian Marick
July 4, 2009

Preface

I work mostly as an agile/XP/TDD coach, mostly working with teams developing C++ or C# or Java applications, mostly for Microsoft Windows platforms. Early in any engagement I will inevitably recommend that everyone on the team work through William Wake's *Refactoring Workbook* [26], which I consider to be far and away the best book for any developer who wants to learn to write great code. A short while later in every engagement—and having a UNIX background myself—I urge everyone on the team to improve their project automation skills by adopting a scripting language. I always recommend Ruby because it's easy to learn and object-oriented, and I generally recommend new teams to read Brian Marick's *Everyday Scripting with Ruby* [20] as a starter.

Finally, one day in the summer of 2007, it dawned on me that there was one great book that I couldn't recommend, one that would combine those two facets of all of my projects, but one that hadn't yet been written—a *Refactoring Workbook* for Ruby. So I contacted Bill Wake and suggested we write one, and you're now reading the result.

Compared with Bill's original Java *Refactoring Workbook,* this Ruby edition has a similar overall structure but is otherwise a substantial rewrite. We have retained the core smells, added a few more, and reworked them to apply to Ruby's more dynamic environment. We have replaced all of the code samples, and replaced or revised all of the exercises. We have also rewritten much of the introductory material, principally to reflect the rise in importance of test-driven development during the last five years.

In short, we have tried to create a stand-alone Ruby refactoring workbook for the modern developer, and not a Java book with Ruby code samples. I hope we've come reasonably close to that goal.

—Kevin Rutherford
Summer 2009

What Is This Book About?

Refactoring is the art of *improving the design of existing code* and was introduced to the world by Martin Fowler in *Refactoring* [14]. Fowler's book provides dozens of detailed mechanical recipes, each of which describes the steps needed to change one (usually small) aspect of a program's design without breaking anything or changing any behavior.

But to be skilled in refactoring is to be skilled not only in safely and gradually changing code's design, but also in first recognizing where code needs improvement. The agile community has adopted the term *code smell* to describe the anti-patterns in software design, the places where refactoring is needed.

The aim of this book, then, is to help you practice recognizing the smells in existing Ruby code and apply the most important refactoring techniques to eliminate those smells. It will also help you think about how to design code well and to experience the joy of writing great code.

To a lesser extent this book is also a reference work, providing a checklist to help you review for smells in any Ruby code. We have also described the code smells using a standard format; for each smell we describe

- What to Look For: cues that help you spot it
- Why This Is a Problem: the undesirable consequences of having code with this smell
- When to Leave It: the trade-offs that may reduce the priority of fixing it
- How It Got This Way: notes on how it happened
- What to Do: refactorings to remove the smell
- What to Look for Next: what you may see when the smell has been removed

This should help keep the smell pages useful for reference even when you've finished the challenges.

This book does not attempt to catalog or describe the mechanics of refactorings in Ruby. For a comprehensive step-by-step guide to Ruby refactoring recipes, we recommend *Refactoring, Ruby Edition,* by Jay Fields, Shane Harvie, and Martin Fowler [11], which is a Ruby reworking of Fowler's *Refactoring*. It is also not our intention to describe smells in tests; these are already covered well by Gerard Meszaros in *XUnit Test Patterns* [22].

Who Is This Book For?

This book is intended for practicing programmers who write and maintain Ruby code and who want to improve their code's "habitability." We have tried to focus primarily on the universal principles of good design, rather than the details of advanced Ruby-*fu*. Nevertheless, we do expect you to be familiar with most aspects of the Ruby language, the core classes, and the standard libraries. For some exercises you will also need an existing body of Ruby code on hand; usually this will be from your own projects, but you could also use open source code in gems or downloaded applications. Familiarity with

refactoring tools or specific IDEs is not assumed (but the examples in this book will provide great help if you wish to practice using such tools).

As mentioned above, it will be helpful to have Fields et al., *Refactoring, Ruby Edition* [11], handy as you work through the exercises. In addition to the mechanics of refactorings, we frequently refer to design patterns, particularly those cataloged by Gamma et al. [16]; you may also find it useful to have available a copy of Russ Olsen's *Design Patterns in Ruby* [24].

What's in This Book?

This book is organized into three sections.

Part I, "The Art of Refactoring," provides an overview of the art of refactoring. We begin with an example; Chapter 1, "A Refactoring Example," takes a small Ruby script containing some common smells and refactors it toward a better design. Chapter 2, "The Refactoring Cycle," takes a brief look at the process of refactoring—when and how to refactor with both legacy code and during test-driven development—while Chapter 3, "Refactoring Step by Step," looks in detail at the tools used and steps taken in a single refactoring. Finally, Chapter 4, "Refactoring Practice," suggests some exercises that you can apply in your own work and provides suggestions for further reading.

Part II, "Code Smells," is the heart of the book, focusing on Ruby code smells. Each chapter here consists of descriptions of a few major code smells, followed by a number of exercises for you to work through. The challenges vary; some ask you to analyze code, others to assess a situation, others to revise code. Not all challenges are equally easy. The harder ones are marked "Challenging"; you'll see that these often have room for variation in their answers. Some exercises have solutions (or ideas to help you find solutions) in Appendix A, "Answers to Selected Questions." Where an exercise relies on Ruby source code you can download it from `www.refactoringinruby.info`.

Part III, "Programs to Refactor," provides a few "large" programs to help you practice refactoring in a variety of domains.

Part IV, "Appendices," provides selected answers to exercises and brief descriptions of currently available Ruby refactoring tools.

How to Use This Book

This is a *workbook:* Its main purpose is to help you understand the art of refactoring by practicing, with our guidance. There's an easy way to do the exercises: Read the exercise, look up our solution, and nod because it sounds plausible. This may lead you to many insights. Then there's a harder but far better way to do the exercises: Read the exercise,

solve the problem, and only then look up our solution. This has a much better chance of leading you to your own insights. Solving a problem is more challenging than merely recognizing a solution and is ultimately much more rewarding.

As you work through the problems, you'll probably find that you disagree with us on some answers. If so, please participate in the community and discuss your opinions with others. That will be more fun for all of us than if you just look at our answers and nod. See Chapter 4, "Refactoring Practice," to learn how to join the discussion.

We think it's more fun to work with others (either with a pair-partner or in a small group), but we recognize this isn't always possible.

Almost all of the code examples need to be done at a computer. Looking for problems, and figuring out how to solve them, is different when you're looking at a program in your environment. Hands-on practice will help you learn more, particularly where you're asked to modify code. Refactoring is a skill that requires practice.

Good luck!

Acknowledgments

Brian Marick has been a huge supporter of the original *Refactoring Workbook* project, and an inspiration with his writing and teaching.

We'd like to thank our core reviewers: Pat Eyler, Micah Martin, Russ Olsen, and Dean Wampler. Their encouragement and suggestions really helped us along the way.

Our involvement in this writing project has placed demands and strains on our families, and we both thank them deeply for their endless patience and support.

Kevin thanks the many people who read drafts of various chapters and provided reactions and feedback, notably Lindsay McEwan; and many thanks to Ashley Moran for pushing the development of Reek, and for introducing lambdas into the Robot tests.

Bill thanks his friends Tom Kubit and Kevin Bradtke for being sounding boards on agile software and other ideas. (Tom gets a double nod for his reviews and discussion of the earlier book.)

Finally, thanks to Chris Guzikowski, Chris Zahn, Raina Chrobak, Kelli Brooks, Julie Nahil, and the others at Pearson who have helped us pull this together.

Contact Us

Feel free to contact us:

Kevin: `kevin@rutherford-software.com`
`http://www.kevinrutherford.co.uk`
Bill: `william.wake@acm.org`
`http://xp123.com`

CHAPTER 1

A Refactoring Example

Rather than start with a lot of explanation, we'll begin with a quick example of refactoring to show how you can identify problems in code and systematically clean them up. We'll work "at speed" so you can get the feel of a real session. In later chapters, we'll touch on theory, provide deeper dives into problems and how you fix them, and explore moderately large examples that you can practice on.

Sparkline Script

Let's take a look at a little Ruby script Kevin wrote a while back. The script generates a *sparkline* (a small graph used to display trends, without detail) and does it by generating an SVG document to describe the graphic. (See Figure 1.1.)

The original script was written quickly to display a single sparkline to demonstrate the trends that occur when tossing a coin. It was never intended to live beyond that single use, but then someone asked Kevin to generalize it so that the code could be used to create other sparklines and other SVG documents. The code needs to become more reusable and maintainable, which means we'd better get it into shape.

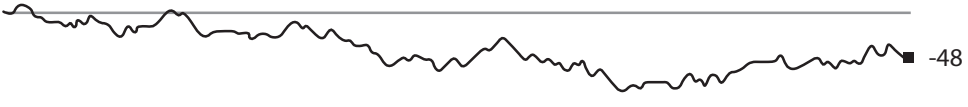

Figure 1.1 A sparkline

Here's the original code:

```
NUMBER_OF_TOSSES = 1000
BORDER_WIDTH = 50

def toss
  2 * (rand(2)*2 - 1)
end

def values(n)
  a = [0]
  n.times { a << (toss + a[-1]) }
  a
end

def spark(centre_x, centre_y, value)
  "<rect x=\"#{centre_x-2}\" y=\"#{centre_y-2}\"
    width=\"4\" height=\"4\"
    fill=\"red\" stroke=\"none\" stroke-width=\"0\" />
  <text x=\"#{centre_x+6}\" y=\"#{centre_y+4}\"
    font-family=\"Verdana\" font-size=\"9\"
    fill=\"red\" >#{value}</text>"
end

$tosses = values(NUMBER_OF_TOSSES)
points = []
$tosses.each_index { |i| points << "#{i},#{200-$tosses[i]}" }

data = "<svg xmlns=\"http://www.w3.org/2000/svg\"
     xmlns:xlink=\"http://www.w3.org/1999/xlink\" >
  <!-- x-axis -->
  <line x1=\"0\" y1=\"200\" x2=\"#{NUMBER_OF_TOSSES}\" y2=\"200\"
            stroke=\"#999\" stroke-width=\"1\" />
  <polyline fill=\"none\" stroke=\"#333\" stroke-width=\"1\"
    points = \"#{points.join(' ')}\" />
  #{spark(NUMBER_OF_TOSSES-1, 200-$tosses[-1], $tosses[-1])}
</svg>"

puts "Content-Type: image/svg+xml
Content-Length: #{data.length}

#{data}"
```

Forty lines of code, and what a mess! Before we dive in and change things, take a moment to review the script. Which aspects of it strike you as convoluted, or unreadable, or even unmaintainable? Part II, "Code Smells," of this book lists over forty common code problems: Each kind of problem is known as a *code smell*, and each has very specific

characteristics, consequences, and remedies. For the purposes of this quick refactoring demonstration, we'll use the names of these smells (so that you can cross-reference with Part II, "Code Smells," if you wish), but otherwise we just want to get on with fixing the code. Here are the more obvious problems we noticed in the code:

- **Comments:** There's a comment in the SVG document (line 29). As a comment in the SVG output that's not a bad thing, because the SVG is quite opaque. But it also serves to comment the Ruby script, which suggests that the string is too complex.
- **Inconsistent Style:** Part of the SVG document is broken out into a separate method (line 34), whereas most is built inline in the `data` string.
- **Long Parameter List:** Strictly speaking, the list of properties of the XML elements aren't Ruby parameters. But they are long lists, and we feel sure they will cause problems later.
- **Uncommunicative Name:** The code uses `data` as the name of the SVG document, `i` as an iterator index (line 25), `a` as the name of an array (line 9), and `n` as the number of array elements (line 8).
- **Dead Code:** The constant `BORDER_WIDTH` (line 2) is unused.
- **Greedy Method:** `toss` tosses a coin and also scales it to be –2 or +2.
- **Derived Value:** Most of the numbers representing SVG coordinates and shape sizes could probably be derived from the number of tosses and the sparkline's max and min values.
- **Duplicated Code:** The text markers for the start and end tags of XML elements are repeated throughout the code; the calculation `200-tosses[x]` is repeated (lines 25, 34).
- **Data Clump:** The SVG components' parameters include several x-y pairs that represent points on the display canvas (lines 15, 18, 30). Some have further parameters that go to make up a rectangle (lines 16, 30). Strictly, these are parameters to SVG elements, and this is therefore a problem in the definition of SVG.
- **Global Variable:** Why is `tosses` a global variable at all?
- **Utility Function:** One might argue that all of the methods here (lines 4, 8, 14) are Utility Functions.
- **Greedy Module:** The script isn't a class, as such, but it does have multiple responsibilities: Some of the script deals with tossing coins, some deals with drawing pictures, and some wraps the SVG document in an HTTP message.
- **Divergent Change:** The `data` string (lines 27–35) is probably going to need to be different for almost every imaginable variation on this script.

- **Reinvented Wheel:** There are already Ruby libraries for manipulating XML elements, and even for creating SVG documents.

Which should we address first? When faced with a long to-do list of code smells it's easy to feel a little intimidated. It's important to remember at this stage that we can't fix everything in one sitting; we'll have to proceed in small, safe steps. We also want to avoid planning too far ahead—the code will change with every step, and right now it would be a futile waste of energy to attempt to visualize what the code might be like even a few minutes from now.

So in the next few sections we're simply going to address the smells that strike us as "next" on the to-do list, without regard to what "next" might mean, or to what will happen after that. It is entirely likely that you would address the smells in a different order, and that's just fine; experience suggests that we're likely to finish up at approximately the same place later.

First, let's tidy up a little.

Consistency

We can easily remove the **Dead Code** and change the **Global Variable**; at the same time we'll create a simple method for each SVG element type we use, and convert those quoted strings too:

```
NUMBER_OF_TOSSES = 1000

def toss
  2 * (rand(2)*2 - 1)
end

def values(n)
  a = [0]
  n.times { a << (toss + a[-1]) }
  a
end

def rect(centre_x, centre_y)
  %Q{<rect x="#{centre_x-2}" y="#{centre_y-2}"
    width="4" height="4"
    fill="red" stroke="none"  stroke-width="0" />"}
end
```

```
def text(x, y, msg)
  %Q{<text x="#{x}" y="#{y}"
    font-family="Verdana" font-size="9"
       fill="red" >#{msg}</text>"}
end

def line(x1, y1, x2, y2)
  %Q{<line x1="#{x1}" y1="#{y1}" x2="#{x2}" y2="#{y2}"
    stroke="#999" stroke-width="1" />}
end

def polyline(points)
  %Q{<polyline fill="none" stroke="#333" stroke-width="1"
    points = "#{points.join(' ')}" />"}
end

def spark(centre_x, centre_y, value)
  "#{rect(centre_x, centre_y)}
   #{text(centre_x+6, centre_y+4, value)}"
end

tosses = values(NUMBER_OF_TOSSES)
points = []
tosses.each_index { |i| points << "#{i},#{200-tosses[i]}" }

data = %Q{<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink" >
  <!-- x-axis -->
  #{line(0, 200, NUMBER_OF_TOSSES, 200)}
  #{polyline(points)}
  #{spark(NUMBER_OF_TOSSES-1, 200-tosses[-1], tosses[-1])}
</svg>}

puts "Content-Type: image/svg+xml
Content-Length: #{data.length}

#{data}"
```

The overall **Greedy Module** is now somewhat more apparent, as we have more methods dealing with SVG elements now. However, note that each of the methods we just added is also a **Greedy Method**, because each knows something about an SVG element and something about how we want the sparkline to look. So we've traded some problems for others, and that's a very subjective process.

Testability

We changed quite a lot of code there, and each time we extracted a method we re-ran the script to make sure we hadn't broken the sparkline. But the HTTP wrapper (lines 52–54) forces us into a particularly unfriendly test environment. So to improve testability, we'll delete that HTTP wrapper and simply replace it with:

```
puts data
```

More on testing as we proceed, but for now that little change makes it easier to run `sparky.rb`.

Greedy Methods

Each of the SVG drawing methods we extracted is greedy, because they know about SVG *and* sparkline formatting. We want to address that next, because those two kinds of knowledge are likely to cause change at different rates in the future.

We'll begin with `rect`: we passed in two parameters from the caller, but to make this method fully independent of the sparklines application we need to pass in 5 more:

```
def rect(centre_x, centre_y, width, height,
         fill, stroke, stroke_width)
  %Q{<rect x="#{centre_x}" y="#{centre_y}"
    width="#{width}" height="#{height}"
    fill="#{fill}" stroke="#{stroke}"
    stroke-width="#{stroke_width}" />}
end
```

This is ugly, but right now it's what the code seems to want. We're trading one smell for another again here, but little bits of flexibility and maintainability are created as by-products.

The caller changes to match:

```
SQUARE_SIDE = 4

def spark(centre_x, centre_y, value)
  "#{rect(centre_x-(SQUARE_SIDE/2), centre_y-(SQUARE_SIDE/2),
          SQUARE_SIDE, SQUARE_SIDE, 'red', 'none', 0)}
   #{text(centre_x+6, centre_y+4, value)}"
end
```

The changes to `spark` made some **Derived Values** apparent, so we also took the opportunity to fix that by introducing a constant for the size of the little red square.

We can now introduce extra parameters to `text`, `line`, and `polyline` in the same way:

```
def text(x, y, msg, font_family, font_size, fill)
  %Q{<text x="#{x}" y="#{y}"
   font-family="#{font_family}" font-size="#{font_size}"
   fill="#{fill}" >#{msg}</text>}
end

def line(x1, y1, x2, y2, stroke, stroke_width)
  %Q{<line x1="#{x1}" y1="#{y1}" x2="#{x2}" y2="#{y2}"
   stroke="#{stroke}" stroke-width="#{stroke_width}" />}
end

def polyline(points, fill, stroke, stroke_width)
  %Q{<polyline fill="#{fill}" stroke="#{stroke}"
   stroke-width="#{stroke_width}"
   points = "#{points.join(' ')}" />}
end
```

The calling code changes to match, for example:

```
SQUARE_SIDE = 4
SPARK_COLOR = 'red'

def spark(centre_x, centre_y, value)
  "#{rect(centre_x-(SQUARE_SIDE/2), centre_y-(SQUARE_SIDE/2),
          SQUARE_SIDE, SQUARE_SIDE, SPARK_COLOR, 'none', 0)}
   #{text(centre_x+6, centre_y+4, value,
          'Verdana', 9, SPARK_COLOR)}"
end
```

Note that we have again traded problems. The four drawing methods are no longer greedy, but now their callers know some SVG magic (color names, font names, and drawing element dimensions). This kind of trading is a completely natural part of refactoring, as we create areas of stability within the code. We'll return to address this **Inappropriate Intimacy** (**General Form**) later.

Greedy Module

That may not be the last we see of **Greedy Methods**, but code changes in the previous section have highlighted another of the problems in the original code: There's now an even clearer distinction between code that knows how to write an SVG document and code that knows what a sparkline should look like.

To fix that, we're going to extract a module for the SVG methods. We'll put it in a new source file called `svg.rb`:

```
module SVG
  def self.rect(centre_x, centre_y, width, height, fill,
      stroke, stroke_width)
    %Q{<rect x="#{centre_x}" y="#{centre_y}"
     width="#{width}" height="#{height}"
     fill="#{fill}" stroke="#{stroke}"
     stroke-width="#{stroke_width}" />}
  end

  # etc...
end
```

A quick glance at this module shows that the **Data Clumps** and **Long Parameter Lists** we predicted are now a reality. (And in fact, each of these SVG elements can take more parameters than we have provided here, so the problem is much worse than it seems.) Note also that we haven't yet moved all of the XML into the SVG module, but to do that we'll have to decide how to deal with nested XML elements. We want to make the calling script a little clearer before diving into the design of the SVG interface.

Comments

There's a comment in the SVG document generated by the script:

```
  <!-- x-axis -->
```

The comment is there because it's difficult to match the magic SVG words and symbols to the format and structure of a sparkline. We don't like commenting source code, but we have no problem creating a self-documenting SVG document, so we're happy to keep the comment. The problem is that one comment isn't enough; the output SVG needs to have a few more! Worse, the script doesn't communicate the sparkline's structure to us, its readers, and so we could easily break it accidentally in the future. We'll fix both of these issues by extracting a method for each component of the sparkline's structure:

```
def sparkline(points)
  "<!-- sparkline -->
  #{SVG.polyline(points, 'none', '#333', 1)}"
end
```

```
def spark(centre_x, centre_y, value)
  "<!-- spark -->
  #{SVG.rect(centre_x-(SQUARE_SIDE/2), centre_y-(SQUARE_SIDE/2),
             SQUARE_SIDE, SQUARE_SIDE, SPARK_COLOR, 'none', 0)}
  <!-- final value -->
  #{SVG.text(centre_x+6, centre_y+4, value,
             'Verdana', 9, SPARK_COLOR)}"
end

def x_axis(points)
  "<!-- x-axis -->
  #{SVG.line(0, 200, points.length, 200, '#999', 1)}"
end
```

While extracting `x_axis` we also removed its dependency on the constant `NUMBER_OF_TOSSES`. In fact, we now see no reason for the constant to exist; we'll inline it in the call to `values`, and recalculate its value in the call to `spark`:

```
tosses = values(1000)

#...

data = %Q{<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink" >
  #{x_axis(points)}
  #{sparkline(points)}
  #{spark(tosses.length-1, 200-tosses[-1], tosses[-1])}
</svg>}
```

Whole Objects

Leaving aside the horrors of that last string for a moment, look inside it at the call to `spark`: We have a **Long Parameter List** in which every parameter is calculated from `tosses`. Let's use Preserve Whole Object by pushing those calculations into the `spark` method:

```
def spark(y_values)
  final_value = y_values[-1]
  centre_x = y_values.length-1
  centre_y = 200 - final_value
  "<!-- spark -->
  #{SVG.rect(centre_x-(SQUARE_SIDE/2), centre_y-(SQUARE_SIDE/2),
             SQUARE_SIDE, SQUARE_SIDE, SPARK_COLOR, 'none', 0)}
  <!-- final value -->
  #{SVG.text(centre_x+6, centre_y+4, final_value,
             'Verdana', 9, SPARK_COLOR)}"
end
```

`spark`'s parameter could represent coin tosses, stock prices, or temperatures, so we renamed it while we remembered.

Now take another look at `x_axis`—it only cares how many y-values there are, but it isn't interested in the points. We can pass in the y-values instead:

```
 def x_axis(y_values)
  "<!-- x-axis -->
  #{SVG.line(0, 200, y_values.length, 200, '#999', 1)}"
end
```

This means that the only code that cares about `points` is the `sparkline` method. We can move the calculation of `points` into that method:

```
def sparkline(y_values)
  points = []
  y_values.each_index { |i| points << "#{i},#{200-y_values[i]}" }
  "<!-- sparkline -->
   #{SVG.polyline(points, 'none', '#333', 1)}"
end
```

And so finally (and after a little tidying up), the creation of the SVG document looks like this:

```
puts %Q{<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink" >
       #{x_axis(tosses)}
       #{sparkline(tosses)}
       #{spark(tosses)}
      </svg>}
```

Feature Envy

Look again at that sequence of method calls taking `tosses` as the single parameter. That chunk of code has more affinity with the `tosses` array than it does with the rest of the script. Same goes for the three methods `spark`, `sparkline`, and `x_axis`—they all do more with the array of `y_values` than they do with anything else. There's a missing class here, one whose state is the array, and which has methods that know how to draw the pieces of a sparkline. Instances of this missing class represent sparklines, so finding a name for it is easy. First, we'll create a simple stub to hold the array:

```
class Sparkline
  attr_reader :y_values
```

```
  def initialize(y_values)
    @y_values = y_values
  end

end
```

Then we'll update the final `puts` call to use it:

```
sp = Sparkline.new(values(1000))
puts %Q{<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink" >
       #{x_axis(sp.y_values)}
       #{sparkline(sp.y_values)}
       #{spark(sp.y_values)}
      </svg>}
```

Now we're going to move the three methods (and that huge string) onto the new class. In real life we would do them one by one, testing as we go; but for the sake of brevity here let's cut to the final state of the new class:

```
class Sparkline

  def initialize(y_values)
    @y_values = y_values
  end

  def to_svg
    %Q{<svg xmlns="http://www.w3.org/2000/svg"
          xmlns:xlink="http://www.w3.org/1999/xlink" >
       #{x_axis}
       #{sparkline}
       #{spark}
      </svg>}
  end

private

  def x_axis
    "<!-- x-axis -->
    #{SVG.line(0, 200, y_values.length, 200, '#999', 1)}"
  end

  def sparkline
    points = []
    y_values.each_index { |i| points << "#{i},#{200-y_values[i]}" }
    "<!-- sparkline -->
     #{SVG.polyline(points, 'none', '#333', 1)}"
  end

  SQUARE_SIDE = 4
  SPARK_COLOR = 'red'
```

```
  def spark
    final_value = y_values[-1]
    centre_x = y_values.length-1
    centre_y = 200 - final_value
    "<!-- spark -->
     #{SVG.rect(centre_x-(SQUARE_SIDE/2), centre_y-(SQUARE_SIDE/2),
                SQUARE_SIDE, SQUARE_SIDE, SPARK_COLOR, 'none', 0)}
     <!-- final value -->
     #{SVG.text(centre_x+6, centre_y+4, final_value,
                'Verdana', 9, SPARK_COLOR)}"
  end

end
```

Notice that the `attr_reader` for `y_values` is no longer necessary, so we deleted it. The public accessor was needed in the early phases of that refactoring step so that we could introduce the new class without breaking any other code. But after the methods had all migrated into the new class, the array is used only internally, and thus can be hidden.

For completeness, here's what remains of the original script:

```
require 'sparkline'

def toss
  2 * (rand(2)*2 - 1)
end

def values(n)
  a = [0]
  n.times { a << (toss + a[-1]) }
  a
end

puts Sparkline.new(values(1000)).to_svg
```

Uncommunicative Names

Now the script is so short, the **Uncommunicative Names** really stand out. Here's an alternative version with better names for anything we thought wasn't communicating clearly:

```
 require 'sparkline'

def zero_or_one() rand(2) end
```

```
def one_or_minus_one
  (zero_or_one * 2) - 1
end

def next_value(y_values)
  y_values[-1] + one_or_minus_one
end

def y_values
  result = [0]
  1000.times { result << next_value(result) }
  result
end

puts Sparkline.new(y_values).to_svg
```

While fixing the names we discovered a 2 being used to scale the sparkline vertically; we removed it in the interest of honest statistics. We find defects often during the course of refactoring. Usually this is because the process of refactoring has revealed something that previously wasn't obvious. It's okay to fix these defects, provided you consciously switch hats for a few moments while doing so.

Derived Values

Now it's time to tackle all those **Derived Values** we noticed right at the outset. They have all migrated into `Sparkline`, which is nicely convenient. I'll begin with the 200s: The x-axis is drawn halfway down the canvas, at y-coordinate 200, and so every `y_value` is scaled vertically by 200. (Y-coordinates increase down the page; so point `(0, 0)` is at the top-left corner and point `(0, 200)` is 200 drawing units below that.) In fact, `200-y` does two things: It translates the line vertically downward by 200 units *and* it flips the line over so that positive y-values appear above negative y-values. These are *transforms* of the image: Reflection followed by translation. SVG (currently) has no reflection transform, but it does offer translation, and we feel we'll get simpler Ruby code if we use it. First, then, we'll invert the sparkline's y-values in the constructor:

```
 def initialize(y_values)
  @height_above_x_axis = y_values.max
  @height_below_x_axis = y_values.min
  @final_value = y_values[-1]
  @y_values = reflect_top_and_bottom(y_values)
end

def reflect_top_and_bottom(y_values)
  y_values.map { |y| -y }
end
```

and change `sparkline` and `spark` correspondingly:

```
def sparkline
  points = []
  y_values.each_index { |i| points << "#{i},#{y_values[i] + 200}" }
  "<!-- sparkline -->
    #{SVG.polyline(points, 'none', '#333', 1)}"
end

def spark
  centre_x = y_values.length-1
  centre_y = y_values[-1] + 200
  "<!-- spark -->
    #{SVG.rect(centre_x-(SQUARE_SIDE/2), centre_y-(SQUARE_SIDE/2),
               SQUARE_SIDE, SQUARE_SIDE, SPARK_COLOR, 'none', 0)}
   <!-- final value -->
    #{SVG.text(centre_x+6, centre_y+4, @final_value,
               'Verdana', 9, SPARK_COLOR)}"
end
```

Next, we use an SVG transform to move the whole graphic down the screen by 200 units:

```
def to_svg
  %Q{<svg xmlns="http://www.w3.org/2000/svg"
        xmlns:xlink="http://www.w3.org/1999/xlink" >
     <g transform="translate(0,200)">
       #{x_axis}
       #{sparkline}
       #{spark}
     </g>
   </svg>}
end
```

And now we can remove those magic `200`s from the drawing methods. For example, `x_axis` now becomes

```
def x_axis
  "<!-- x-axis -->
   #{SVG.line(0, 0, y_values.length, 0, '#999', 1)}"
end
```

We now have more SVG magic—the `<g>` element—in the code, but also there is less duplication, and we consider that much more important.

We have now removed all but one of the magic `200`s; before going any further, we want to document its meaning:

```
def to_svg
  height_above_x_axis = 200
  %Q{<svg xmlns="http://www.w3.org/2000/svg"
        xmlns:xlink="http://www.w3.org/1999/xlink" >
     <g transform="translate(0,#{height_above_x_axis})">
       #{x_axis}
       #{sparkline}
       #{spark}
     </g>
   </svg>}
end
```

It is now clear that the `200` is simply a guess as to what a reasonable value might be. If the sparkline's y-values stray outside of the range –200..200 we'll find the line disappears off the edge of the graphic. We spoke to our customer just now, and he agrees that we should replace the `200` with the maximum y-value:

```
def to_svg
  %Q{<svg xmlns="http://www.w3.org/2000/svg"
        xmlns:xlink="http://www.w3.org/1999/xlink" >
     <g transform="translate(0,#{height_above_x_axis})">
       #{x_axis}
       #{sparkline}
       #{spark}
     </g>
   </svg>}
end

def initialize(y_values)
  @height_above_x_axis = y_values.max
  @final_value = y_values[-1]
  @y_values = reflect_top_and_bottom(y_values)
end
```

Wabi-Sabi

We've made a number of refactoring changes to the code, and in the process its structure has altered a great deal. Have we finished? No, and in a sense we never will. Software can never be perfect, and there's usually little point in chasing down that last scintilla of design perfection. Any code will always be a "work in progress"—the important thing is to have removed the major problems, and to know what slight odors remain.

The title of this section is also the name of the Japanese artistic style that celebrates the incomplete, the unfinished, and the transitory. Try to become used to thinking of your code as a *process* and not simply an *artifact;* aim for *better*, not *best*. Read more in Leonard Koren's *Wabi-Sabi: For Artists, Designers, Poets and Philosophers* [19], for example.

Summing Up

Here's the current state of the main script after the refactorings:

```
require 'sparkline'

def zero_or_one() rand(2) end

def one_or_minus_one
  (zero_or_one * 2) - 1
end

def next_value(y_values)
  y_values[-1] + one_or_minus_one
end

def y_values
  result = [0]
  1000.times { result << next_value(result) }
  result
end

puts Sparkline.new(y_values).to_svg
```

(You can get complete copies of the "before" and "after" states of the code from our download, which you can find online at `http://github.com/kevinrutherford/rrwb-code`.)

The code still has some smells: `sparkline.rb` still knows too much about SVG; `svg.rb` still has long parameter lists; and the functionality of the SVG module duplicates that of a standard Ruby library. Notice also that the code has expanded from 40 lines to 100, and from one source file to three—all without increasing the script's functionality.

Overall, though, the code is much more readable and maintainable than it was before. We have traded size for flexibility, and in the future it will be much easier to reuse any of the various parts of this code. This is a reasonable place to stop for now.

What's Next

Now that we've seen a quick example of how refactoring can improve code, we'll look at how refactoring fits into the development process, and then consider different problems in code and examples of how to address them.

Index

Footnote references are indicated with "n," followed by the footnote number.

J

K

L

M

N

O

P

R

S

T