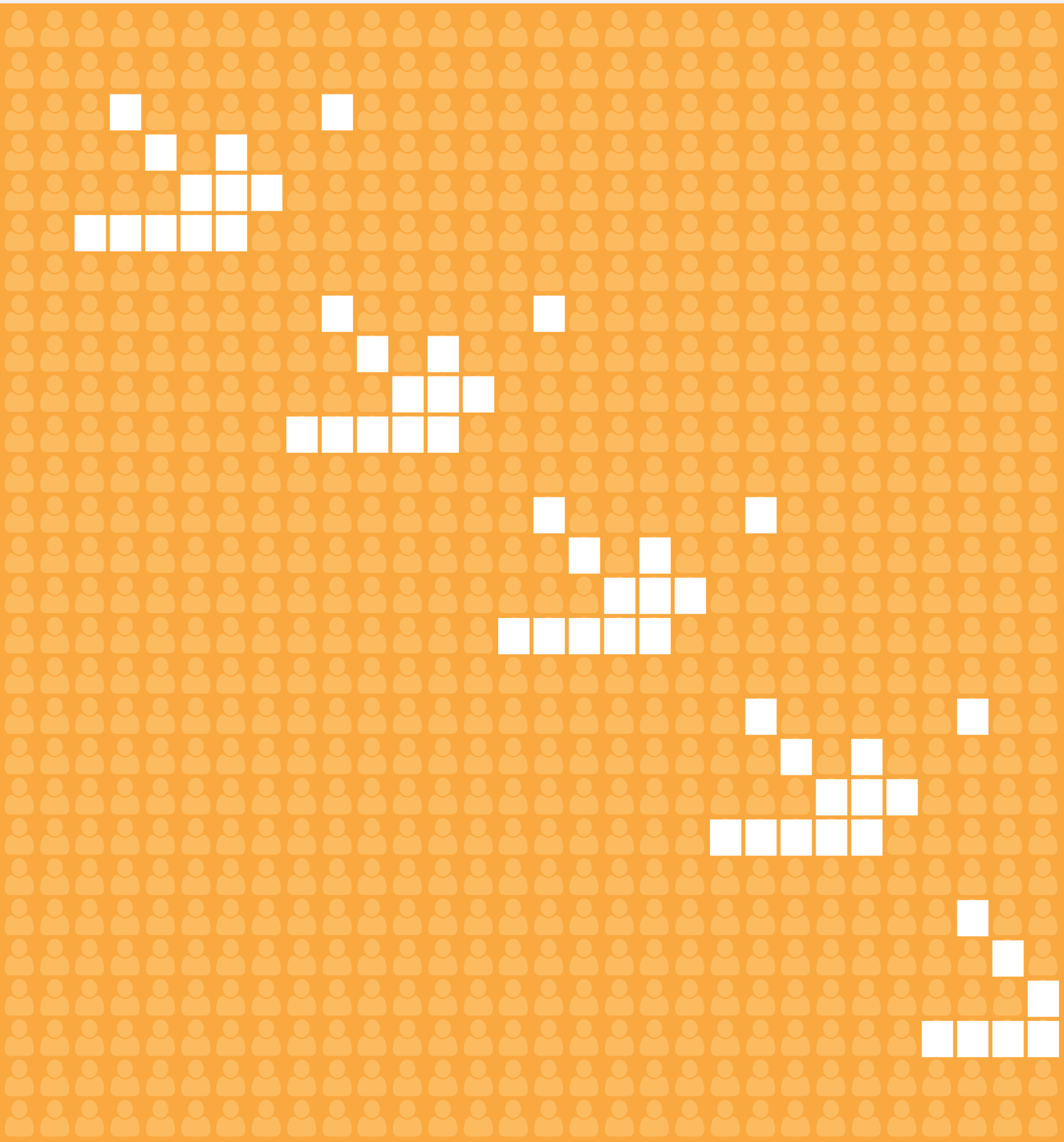
UNDERSTANDING THE

4 RULES OF SIMPLE DESIGN

And other lessons from watching 1000's of pairs work on Conway's Game of Life

by Corey Haines



Understanding the Four Rules of Simple Design

and other lessons from watching thousands of pairs work on Conway's Game of Life

Corey Haines

This book is for sale at http://leanpub.com/4rulesofsimpledesign

This version was published on 2014-06-04

Tweet This Book!

Please help Corey Haines by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Just bought ”Understanding 4 rules of simple design”. Check it out! #4rulesbook https://leanpub.com/4rulesofsimpledesign

The suggested hashtag for this book is #4rulesbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#4rulesbook

This book is dedicated to the thousands of people who have both attended and facilitated coderetreats around the world. Without you, the lessons in this book would have been much more difficult to convey.

I'd also like to dedicate this to Sarah Gray, who laughed when I came back from India and said "Huh, I think I'm writing a book." She's an amazing companion.

And, of course, to Zak the Cat for being awesome! MEW!!!!

Contents

Introduction

From 2009 to 2014, I traveled the world working with software developers, both individually and in teams, to improve their craft. I did this primarily through a training workshop format called *coderetreat*[1]. During these day-long events, we worked on improving our ability to make good choices around the minute-by-minute decisions we make will writing.

Over those years, I watched thousands of pairs of programmers work on exactly the same system, Conway's Game of Life[2]. As a facilitator of these coderetreat workshops, I had the unique opportunity to provide feedback, both direct and through questions, on improving the act of writing adaptable, simple code. As time progressed, I began to see patterns arise, common techniques and designs that spanned languages, stayed the same between companies, and crossed national borders. My job as a facilitator was to ask the questions that would push people past these common ideas, gently (and sometimes not so gently) prodding the participants into opening their minds to alternate ways to approach their design.

This book contains some of those patterns, designs and my responses to them. Grouped into a series of examples against the backdrop of Conway's Game of Life, I've done my best not just to write the mechanics — the *What* — of the refactorings, but to focus on the ideas behind them — the *Why* —.

[1]http://coderetreat.org/

[2]http://en.wikipedia.org/wiki/Conway%27s_game_of_life

Zak!

So, enjoy this picture of Zak, and let’s get started.

Corey Haines

March 2014

This Book

Who It Is For

This book is for developers. I know that sounds a bit vague, but I'm not sure how better to state it. Okay, let me try.

It is for beginners.
Are you just learning to program? WELCOME! Perhaps you are in the throes of discovering what it's like to make, what it's like to wake up in the morning and create something that wasn't there yesterday. Exciting, right? And it's a wonderful career/hobby. This book is for you. You haven't had to maintain a larger application, yet. You haven't had the pleasure of wading through reams and reams of spaghetti code, trying to track down that one elusive place to make your change safely. And then you find that codebase was written by you. The ideas contained in this book are about some of the fundamentals of software development, principles you need to think about when writing an application to do your best to ensure this doesn't happen.

It is for intermediate developers.
Nice! You've been programming a while, written a few systems, and you're feeling pretty solid. Maybe you've been coding for a handful of years, and have established "your way" to be effective. Unfortunately, at this stage, it is easy to hit a plateau with your skills. Once effective, it can happen that you stop learning, either consciously or subconsciously. After all, you know what you're doing, right? Now is the time to go back to the fundamentals, really analyze the "Why"

behind the decisions you make. By stripping your thoughts down to the core, you can build them back up with even more insight and understanding.

It is for advanced practitioners.

You've been doing this for a very long time. Over the years, you've figured out how to build systems that can stand the test of time, easily accepting any changes that come. Awesome! This book is for you, too. It is easy to lose sight of the fact that others have to maintain your code, often without the context you have. And sometimes we forget the fundamentals. After all, we tend not to think about them anymore. Going back and thinking about the basics, though, can often shed light on some of the decisions we make, helping us continue to fine-tune our practice.

What It Is (And Isn't) About

Throughout the building of a system, there are many levels of design decisions, ranging from the large up-front thinking (à la hammock-driven development) to the almost continuous decisions made around things such as naming variables and extracting methods.

This book is focused on the latter. While there are important considerations and thoughts to be had at all stages of the software development lifecycle, I'm choosing, for the purposes of this book, to assume they have happened. Instead, the examples here are low-level, focused on decisions that are made in the minute-by-minute rush of writing code.

This book is not about any particular technique. It's not about any particular language. While the examples use an object-based/object-oriented language, most of the ideas transcend that and focus instead on the fundamentals of writing adaptable code — code that can accept change as it is needed.

And lastly, this book is not a step-by-step guide to building Conway's Game of Life. In fact, we spend very little time on the actual system itself. As with coderetreat, GoL is just a backdrop that we use to investigate how best to apply the 4 rules of simple design, and other design guidelines, at the micro-level when writing code.

Format

This book is built as a series of essays, a series of examples, highlighting different ways to think about your code in the context of the 4 rules of simple design. Rather than grouping them by the individual rule, however, I'm celebrating the fact that the rules feed into each other iteratively. Often, completing a refactoring based on "Expresses Intent," for example, will highlight a further refactoring based on "Eliminate Duplication." Because of this, while the examples can stand on their own, they are best read through in the order presented.

Why Ruby?

Most of the examples in this book are written in Ruby.

I chose Ruby because it has a readable syntax with a minimum of ceremony. The code snippets are small, and I try to use little-to-no ruby-specific functionality. If you have a familiarity with any type of language, you should be able to understand the examples with little effort.

Some Samples

Thanks for being interested in my book, *Understanding the 4 Rules of Simple Design*[3]. I've enjoyed writing it. It comes from the past 5 years of working with software developers to improve their understanding of fundamental techniques through the coderetreat workshop format. Don't worry if you've never been to a coderetreat, the book doesn't assume any knowledge of it.

Here are two sample sections from the book. The sample design example is small and focused, based on the domain present in Conway's Game of Life (GoL). You don't have to be an expert in GoL to understand it, but I'd recommend at least skimming the introductory information at the excellent Wikipedia page on Conway's Game of Life[4]. The book, itself, goes over these rules, especially the relevant concepts for all the examples.

The first sample is a concrete example of thinking about the 4 rules of simple design with regard to your test names and test code.

The second sample is from the "Other Good Stuff" section and contains some thoughts on a pair-programming style called "ping-pong pairing."

The book contains more concrete examples, plus some discussion of other design principles, like SOLID and Law of Demeter.

I hope you like it. Thanks!!

[3]https://leanpub.com/4rulesofsimpledesign
[4]http://en.wikipedia.org/wiki/Conway%27s_game_of_life

Test Names Should Influence Object's API

The idea of naming, and how it relates to the intent of your code, can be seen when looking at the symmetry between test names and the test code. When talking about test descriptions, we often say that they can stand in for documentation. Unfortunately, it is easy to lose sight of this when writing the code inside the test.

In Conway's Game of Life[5], a common approach is to start with a `World` class. Since one of the techniques we practice at coderetreat is test-driven development, we start with a test. A common starting point is that a living cell can be added. I see the following two tests quite often.

```
def test_a_new_world_is_empty
  world = World.new
  assert_equal 0, world.living_cells.count
end

def test_a_cell_can_be_added_to_the_world
  world = World.new
  world.set_living_at(1, 1)
  assert_equal 1, world.living_cells.count
end
```

On the surface, these seem like reasonably well-written tests. However, if we look at it from the idea that the tests should express intent, then there is an obvious mismatch between the test names and the code in the test.

[5] http://en.wikipedia.org/wiki/Conway%27s_game_of_life

Let's look at the first one, since this is the simple one that we might write first.

```
def test_a_new_world_is_empty
  world = World.new
  assert_equal 0, world.living_cells.count
end
```

The test name talks about an empty world. The test code, though, has no concept of an empty world, no mention of an empty world. Instead, it is brutally reaching into the object, yanking out some sort of collection (only a lack of living cells represents that the world is empty?) and counting it.

When we write our tests, we should be spending time on our test names. We want them to describe both the behavior of the system and the way we expect to use the component under test. When starting a new component, we can use our test names to influence and mold our API. Think of the test as the first consumer of the component, interacting with the object the same way as the rest of the system. Do we want the rest of the system to be reaching in and grabbing the internal collection? No, of course we don't. Instead, think about letting the code in the test be a mirror of the test description. How about something like this.

```
def test_a_new_world_is_empty
  world = World.new
  assert_true world.empty?
end
```

This hides the internals of the object, while building up a usable API for the rest of the system to consume.

Now, let's look at the second test.

```
def test_a_cell_can_be_added_to_the_world
  world = World.new
  world.set_living_at(1, 1)
  assert_equal 1, world.living_cells.count
end
```

After the discussion around the first test, we can see the lack of symmetry here. The test name talks about adding to the world, but the verification step isn't looking for the cell that was added. It is simply looking to see if a counter was incremented on some internal collection. Let's apply the symmetry again and have the test code actually reflect what we say is being tested.

```
def test_a_cell_can_be_added_to_the_world
  world = World.new
  world.set_living_at(1, 1)
  assert_true world.alive_at?(1, 1)
end
```

This now adds to our API. Additional tests, of course, will flesh out the behavior of these methods, but we now have begun to build up the usage pattern for this object.

We also could add a test around the `empty?` method using `set_-living_at`.

```
def test_after_adding_a_cell_the_world_is_not_empty
  world = World.new
  world.set_living_at(1, 1)
  assert_false world.empty?
end
```

This is another way of slowly building up the API, especially the beginnings of the `set_living_at` behavior.

Focusing on the symmetry between a good test name and the code under tests is a subtle design technique. It is definitely not the only design influence that our tests can have on our code, but it can be an important one. So, next time you are flying through your tdd cycle, take a moment to make sure that you are actually testing what you say you are testing.

Some Thoughts On Pair-Programming Styles

Coderetreat workshops encourage pair-programming as a form of sharing and learning together. While the majority of this book consists of concrete examples of code-level design decisions, this section addresses some patterns I've seen around pair-programming.

Driver-Navigator

Traditionally, pair-programming has been introduced via the Driver-Navigator form. In this form, one member has the keyboard and control of the input. Their job is to type and focus on the minute-to-minute coding. The other member is the navigator. Their job is to pay attention to the code being written, but keep the larger picture in mind, guiding the driver in the right direction.The pair should swap roles frequently.

Unfortunately, too often this form of pair-programming leads to what I call the "Driver-Twitterer" style of collaboration. In this mode, the person with the keyboard is writing code while the other person watches intently for a short time. Then, after a bit, the navigator starts to lose interest. Perhaps the driver isn't talking, perhaps the navigator doesn't want to disturb them. Sometimes I've seen where the driver says "just a second, I've got an idea," and then proceeds to code in silence for minutes on end. This can have the effect of boring the navigator. So, what do they do? Naturally, they check twitter. Or email. Or some other non-code-focused task.

As with every aspect of development, communication is key here. But, without practice, driver-navigator level of communication is lacking. In order for this style to work, the pair needs to have good communication habits, constantly keeping the other abreast of what thoughts are going through their head. Unfortunately, this level of communication isn't necessarily built-in to a new pair. Because of this intense communication requirement, I generally consider the driver-navigator style of pair-programming to be a more intermediate level style.

It is quite common for a coderetreat workshop to be a person's first time pair-programming, their introduction to the practice of writing code as a team. Because of this, I like to introduce a style that has the necessary level of communication built-in to the practice. The style I introduce is called "Ping-Pong Pairing."

Ping-Pong Pairing

There are two basic forms of ping-pong, but they both share on very important aspect: both members are writing code frequently. Because of this, I stress the importance of having two sets of live input devices, one for each participant. So, there would be two keyboards and two mouses, all live. I find that having this setup minimizes the context shift when switching who is typing. Having two live input devices isn't a requirement, of course, but it definitely smooths over some inherent friction in having to pass the keyboard back and forth.

The first style of ping-pong is where one member takes on the role of test writer, and the other takes on the role of getting the tests to pass. I like to call the test writer the "test redder" and the one getting them to pass the "test greener." The table below illustrates the flow of writing.

Ping-Pong Form 1

member 1	member 2
write test	
	make test green
write test	
	make test green

The second style of ping-pong is where the role of "test redder" passes between participants. This is done by having the first member write a test, then control is passed to the other member. That person gets the test to pass, to turn green, then they are responsible for writing the next test. The table below illustrates the flow of writing.

Ping-Pong Form 2

member 1	member 2
write test	
	make test green write next test
make test green write next test	
	make test green write next test

The primary difference between these two is that in the first form, the role is stable, but control is passed. In the second, the role is passed along with control. Both are effective and great ways to introduce people to pair-programming.

Which Style Should You Choose?

If you are an experienced pair, or at least both members are experienced at this style of collaborative code writing, then it doesn't matter which style you use. In fact, it is common to see all styles used through a pairing session. With experience, participants generally have developed the level of communication necessary for working in whatever form is useful at the moment.

As I mentioned above, though, I consider driver-navigator a more intermediate style. So, if one, or both, of the participants are new to pair-programming, then ping-pong can be a fantastic way to introduce the concepts. I generally recommend a specific ping-pong style based on the level of testing experience of the members.

Only one member has experience writing tests: Form 1
: Having the experienced person writing most of the tests is the most effective. Over time, the less-experienced person can start picking up test writing. By watching the tests being written, though, they can learn the thought process behind test-driven development.

Both members have experience writing tests: Form 2
: Since the tests guide the design, it can be useful to have both members influencing that aspect. Passing the test-writing role back and forth can help keep both members interested.

Pairing is a fantastic way to develop software. I've written some of my best code, my best systems, when two people's hands were on the keyboard. At the end of a coderetreat, it is very common to get the feedback from first-timers that they didn't expect working in a pair to be so productive. Pair-programming is listed frequently in the closing questions as both surprising and what they will take with them going forward.