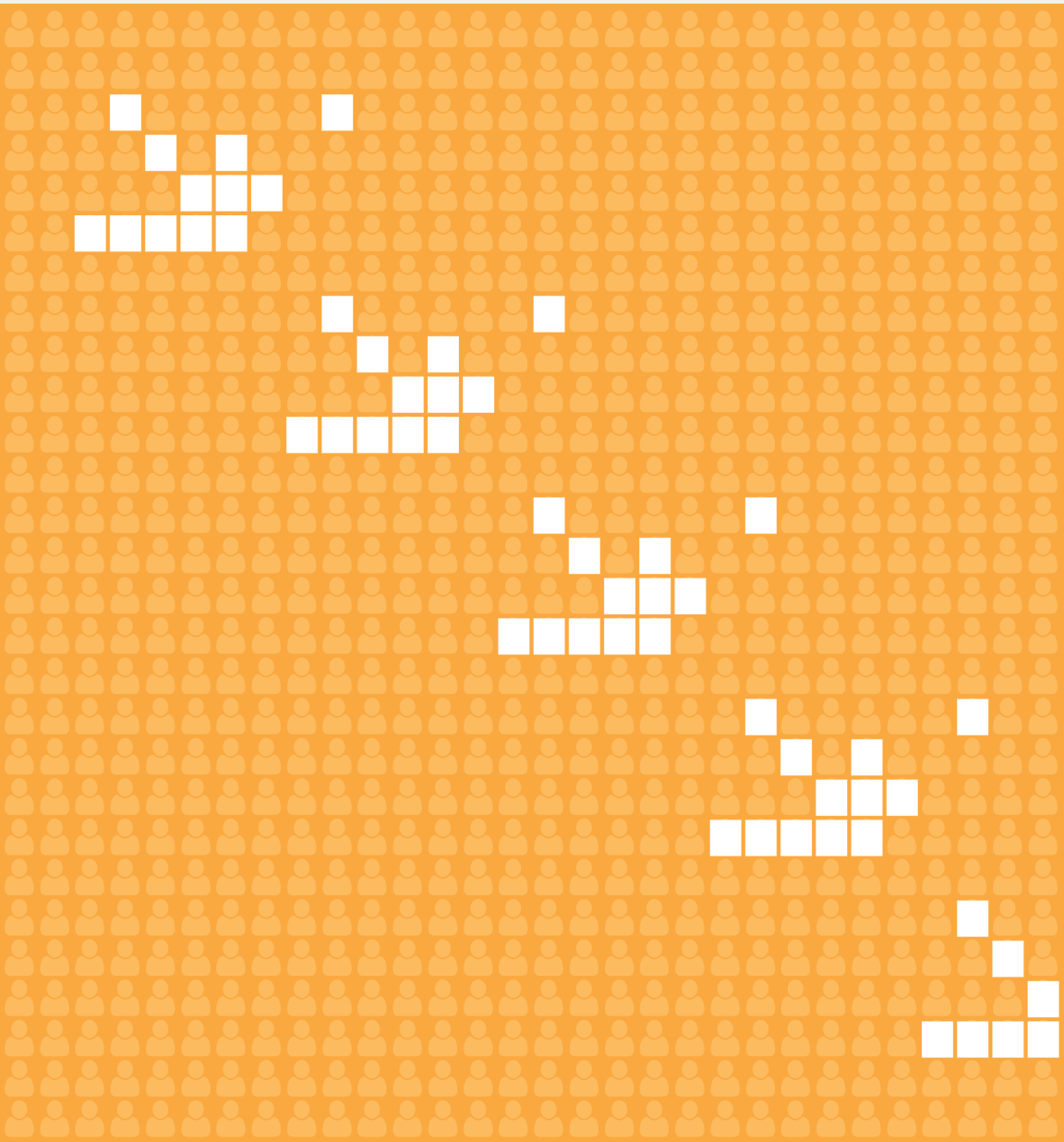
# UNDERSTANDING THE
# 4 RULES OF SIMPLE DESIGN

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


This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

# 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

**Foreword: Kent Beck** . . . . . **i**

**Foreword: Joe Rainsberger** . . . . . **iii**

**Acknowledgements** . . . . . **vi**

**Introduction** . . . . . **viii**

**This Book** . . . . . **x**
- Who It Is For . . . . . x
- What It Is (And Isn't) About . . . . . xi
- Format . . . . . xii
- Why Ruby? . . . . . xii

**Where do these thoughts come from?** . . . . . **1**
- Good Design? . . . . . 1
- Coderetreats . . . . . 4
- Conway's Game of Life . . . . . 7
- 4 Rules of Simple Design . . . . . 10

**Examples** . . . . . **14**
- Test Names Should Influence Object's API . . . . . 15
- Duplication of Knowledge about Topology . . . . . 19
- Behavior Attractors . . . . . 23
- Testing State vs Testing Behavior . . . . . 26

Don't Have Tests Depend on Previous Tests . . . . . 29
Breaking Abstraction Level . . . . . 31
Naive Duplication . . . . . 34
Procedural Polymorphism . . . . . 37
Making Assumptions About Usage . . . . . 42
Unwrapping an Object . . . . . 45
Inverted Composition as a Replacement for Inheritance . 51

**Other Good Stuff** . . . . . **56**

**Other Design Guidelines** . . . . . **57**

**Example constraints** . . . . . **62**

**Some Thoughts On Pair-Programming Styles** . . . . . **64**
Driver-Navigator . . . . . 64
Ping-Pong Pairing . . . . . 65
Which Style Should You Choose? . . . . . 66

**Further Reading** . . . . . **68**
4 Rules of Simple Design . . . . . 68
General Design . . . . . 68
Testing . . . . . 70
Other Things You Probably Should Most Definitely Read 70

# Foreword: Kent Beck

Here's why I wrote the rules, near as I can remember.

As I was coming up as an engineer, the advice I always heard was, "Design for the future. Change is expensive. Make it cheap by anticipating it." What I noticed in practice was that the more change I anticipated, the harder it got to make changes. My incorrect speculations interfered with changes I actually ended up making. Then I would have to choose between working around speculative cruft or ripping it out, both of which delayed progress on what I was trying to accomplish.

I wasn't alone. Lots of folks noticed the cost of speculation. The prevailing response seemed to be that we just weren't good enough at speculation. If we were better speculative designers, we would end up with better designs. This looked like a positive feedback loop to me: more speculation -> worse design -> more speculation.

The good news about disastrous positive feedback loops is that you can generally drive them backwards. I first experimented by ignoring any changes that seemed like they would happen longer than six month in the future. My designs were simpler, I started making progress sooner, and I stressed less about the unknowable future. I shortened the time horizon to three months. More better. One month. More. A week. A day. Oh, hell, what happens if I don't add any design elements not demanded by the current code and tests? Still more better.

Now I had an ethos of software design, but I stupidly labelled it "simple". Talk about a vague, loaded word that everyone will use to justify exactly what they are doing now. I soon tired of debating what "simple" "really" meant. I needed a clear explanation.

My approach to communicating complex ideas at the time was to formulate a simple set of rules, the emergent property of which was the complex outcome I was aiming at (cf patterns). (I have since become disenchanted with this strategy.) I thought about how I recognized simplicity, turned those criteria into actions, sorted them by priority (it's no coincidence that human communication is number two), and posted them on Ward's Wiki[1]. And that's why (and how) I wrote the rules.

– Kent Beck / @kentbeck

April 2014

[1] http://c2.com/cgi/wiki?XpSimplicityRules

# Foreword: Joe Rainsberger

I can trace my interest in Code Retreat back to 2004, although of course, nobody had given it that name yet. Around that time, many of my friends and colleagues had become fascinated with the idea of *intentional practice*. We all looked at each other wondering how we could ever convince a group of already-overworked people to practise. We assumed that most employers would object to practising during work hours. We also assumed that people wouldn't want to fail and learn in front of their peers. At most, we figured that small bands of crazy people might find each other and practise together from time to time, but that it would never really go anywhere. Even as Coding Dojos became more popular, I didn't see intentional practice becoming any more significant than a fun way to pass time at conferences and a curiosity at only the most experimental workplaces. By 2007 I mostly forgot the whole thing.

Imagine my surprise when I received an invitation in January 2009 to come down to Ann Arbor, Michigan to attend the first-ever Code Retreat. I didn't intend to go. I didn't think that I could justify the expense of flying down to participate in a free event. But someone had recently burgled our house, and that left me feeling disconnected from the place. One morning I simply decided that, expensive or not, I wanted to go. I wanted to see my friends, and I figured I'd never get to see so many of them in one place for an event smaller than a conference.

I found myself in a room with about 40 eager programmers, some of which had traveled through snowy conditions from two hours' drive away. I didn't know what to expect. Conway's Game of Life? Over and over again? All day? It sounded strange. It sounded like it wouldn't work. Wow, does it ever work!

And now, five years and hundreds of Code Retreats later, Corey has distilled tens of thousands of hours of collective learning into this powerful little book. I intend to recommend it as a guidebook to anyone who wants to learn the fundamentals of maintainable, cost-effective software design. I've been saying for years that if you simply follow Kent Beck's rules of simple design, then you'll see how every good design principle you've ever heard of reduces to some combination of "remove duplication" and "improve names". Of course, waving my hands like this requires relatively little effort.

With this book, Corey demonstrates the Implementers Rule: implementers rule. I've wanted to see a book like this for years, but never had the energy to write it. Corey has literally spent years and traveled thousands of kilometers making it happen. That makes me happy.

In five short years, I have attended and helped facilitate perhaps a dozen Code Retreats in almost as many countries. I've participated in the Global Day of Code Retreat in Gent, Stockholm, and a five-minute walk from my house in Atlantic Canada. I've helped launch Legacy Code Retreat[2], a member of the Code Retreat family, which has taken root in Europe and already helped hundreds of programmers practise rescuing legacy code in a safe environment. Code Retreat seems intent on staying with us for years to come. Those facts and this book form a significant part of Corey's professional legacy to us. I get to say, "I knew him when…" That makes me happy, too.

As you practise test-driven development and use the four elements of simple design to guide your decisions, you're going to notice patterns. You'll recognise that when you see duplication over here, that means that some code over here wants to become a brand new module. You'll notice a recurring pattern in names that nudges you gradually-but-firmly towards moving this code over there. You'll find some of those patterns in this book, but more importantly,

[2] http://www.legacycoderetreat.org

you'll notice patterns that you won't find in this book. New patterns. Patterns that we know, but that nobody has taken the time to write down before. Perhaps even patterns that nobody has ever noticed before. When that happens, I hope that you'll think back to this book and recognise how it and Code Retreat have helped you reach that wonderful learning moment.

That would make me *really* happy.

– J. B. Rainsberger / @jbrains

Summerside, PEI, Canada

March 2014

# Acknowledgements

First and foremost, this book benefits from the thousands of developers who gave up a day of their lives to spend time with me throwing away their code. They came, wrote code and shared their learnings with each other. It has been a huge honor to see the coderetreat family grow to a level beyond my wildest dreams. There are a lot of people who had a significant influence, but I want to highlight a few in particular.

Coderetreat would have not found such an effective format without the early contributions of two people who believed it could work. Alex Bolboaca and Maria Diaconu, from Mozaic Works in Romania, heard the idea when I was there for the Open Agile conference in 2009. Through that year, while I was running coderetreats in the United States, Maria and Alex ran and actively experimented with coderetreat format in Romania. When we regrouped in 2010, the stable format that we know today came out of our conversations and shared experiences. As one of many examples, I think one of the most powerful aspects of the format is the closing circle and the 3 questions: Alex and Maria are to thank for these.

The past couple years have seen an explosive growth in the number of coderetreats and the influence it has had on our industry. While my travels and talks have contributed, Erik Talboom and Adrian (Adi) Bolboaca's influence in Europe has played a significant role. Adi and Erik have not only been spreading the classic form of coderetreat, but also helped spread another important style, legacy coderetreat, based on the ideas of Joe Rainsberger and others.

In 2013, the third annual Global Day of Coderetreat (GDCR) spanned a record number of timezones, countries, cities and developers participating. This is very much due to the organizing work of Adi

Bolboaca, Jim Hurne, Martin Klose and Alissa Conaty. With their guidance and hard work, the coderetreat and the GDCR continues to grow, and I look forward to seeing what they do in 2014.

And, of course, the original idea of coderetreat came from conversations with Gary Bernhardt, Nayan Hajratwala and Patrick Welsh at the 2009 Codemash conference.

The 4 Rules of Simple Design, of course, aren't my original thoughts; they were coined by Kent Beck in the late 90s and continue to show us how following simple rules, we can make our designs flexible and clear. My understanding and application of the 4 rules come from conversations with and learnings from a lot of people over the years. Joe Rainsberger, Bob Martin, Michael Feathers, David Chelimsky and Cory Foy are just a few of the people that have been influential over the years.

I'd also like to thank James Rosen for his detailed editing of this book. He found a lot of typos, grammatical issues and just things that just didn't make sense. Mike Gehard and his gSchool cohort provided feedback on whether the book was at all understandable to beginners. Jacky Sum swooped in and gave me some great feedback from a beginner's perspective, as well.

And, of course, if you love the cover as much as I do, Zach Walsh is to thank for that. I came to him with an idea of combining people and Conway's Game of Life, and he exceeded all my expectations. Thanks, Zach!

When I started this journey in 2009, I never imagined the growth that could come from sharing and learning from so many developers around the world. Without the support and interest of a worldwide community of developers, the ideas and focus of this book couldn't have happened. Thank you all.

– Corey Haines

# 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*[3]. During these day-long events, we worked on improving our ability to make good choices around the minute-by-minute decisions we make while writing.

Over those years, I watched thousands of pairs of programmers work on exactly the same system, Conway's Game of Life[4]. 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* —.

[3] http://coderetreat.org/

[4] 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.

# Where do these thoughts come from?

## Good Design?

As developers, we often enjoy discussing what makes a good design. These conversations are useful and important, but I think they are best done after-hours, in a relaxed atmosphere. The idea of "Good Design" can often lead to a feeling that there is a pinnacle, that there is an Aristotelian ideal for the design of a system. Unfortunately, this just isn't true. That is why discussions about this ideal are best done away from the stresses of day-to-day work, the pressures that cause us to want to "just get it done."

If you ask a room of developers what makes a "Good Design," you'll most likely get as many answers as there are respondents. I think this variety makes these conversations valuable. Comparing thoughts and ideas on this topic can sometimes yield insight into techniques for improving a codebase, especially if the discussion centers around a concrete piece of code. However, they are ultimately fruitless when trying to reach some ideal of "Good Design."

Instead, I prefer to talk about "Better Design." This takes us to a more concrete footing, allowing us to entertain the idea that perhaps there are more than one design that works, depending on the situation. It also removes the conflict inherent in "your design is bad, because it isn't 'good'" when talking. If we can look at

things from a comparison point of view, perhaps we can find some fundamental ideas about "better." When talking about fundamental ideas, it can be important to talk about what, if anything, we truly know about software development.

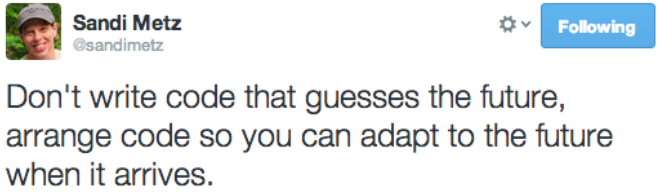
The one constant that we know for sure in software development is that things are going to change. Whether it is our personal projects or a system for a multi-national corporation, the desired functionality will change over time. And these changes to the system require changes to the underlying codebase. If we don't pay attention, our code rots, calcifying into a hardened mass that resists change, pushing back on us whenever we try to add something new.

Simple design, though, is one that is easy to change. Striving for a simple design — one that is adaptable to changing needs — is the key to a "better design." Whenever we have a choice to make, look for the choice which would be easier to change.

It is important to keep in mind that this does not mean you should strive for huge, xml-configuration-based systems, making everything configurable. Quite the opposite. When we plan and build explicit extension- and configurability points, we are going against the idea of simple design. There is a second constant that we know to be true in software development: we don't know exactly what is going to need to change. Every configuration and extensibility point you explicitly plan and build is a belief about the evolution of the system. A concrete statement "this is going to change in the future, so it is worth my investment right now." But, as the old saying goes, "we'll never be more ignorant than we are at this moment." Rather than planning for change points, we build systems, by applying simple design principles, that can change easily at ANY point.

Sandi Metz[5] had a tweet that captures this well.

[5] https://twitter.com/sandimetz




As time goes on, and we learn more about what places are candidates for frequent change, we then move, based on that knowledge, to make those parts of the system even easier to change. As more changes are applied to a simple codebase, it often naturally exudes the right extension mechanism.

So, what are these "simple design principles?"

As a developer learns more about design, they start to find out about different design principles and guidelines. Things like the SOLID principles, Law of Demeter, Design Patterns, and so forth. All of these are important at different levels of the development lifecycle, but they can often be a bit too abstract. They also seem to fall out naturally when applying some basic principles. These principles are called the "4 Rules of Simple Design." And this is what we'll focus on primarily in this book: applying these principles on small sample bits of code, but large enough to really see the thought process during refactoring.

## Coderetreats

Generally, when we are developing, we have a goal of putting code into production. No matter if we are getting paid, or if we are working on some side project, we generally have a feeling that we want to get it done. Of course, this adds pressure to do things the way in which we are most familiar and comfortable. In general, we don't get paid for trying new things, we get paid for building things for production.

We might occasionally pick up a side project to learn something new. However, even these projects tend to have a goal of "getting it done." Also, when learning, we have to live with the mistakes we make at the beginning, during our initial learning phase. If you ask someone to delete the past week's worth of learning, you'll receive a pretty big resistance; even though most people would agree that the second (or third) time you write something, it is done faster and in a better way.

What if there were a day where you were encouraged to try new things? Not just new things, things that you've never even thought of? And, you didn't have to live with the mistakes you've made during learning, you can just throw it away. That's coderetreat.

Coderetreat is a day-long workshop focused on analyzing and practicing the decisions we make when writing code. With a set format, evolved over a couple years of learning, the exercises are focused on practicing and studying the small minute-by-minute decisions we make when writing code. While there is value in larger design activities, the small steps of refactoring are equally as important.

The format of a standard coderetreat is simple:

- Full day (5 to 6 sessions)
- Participants write code in pairs (pair programming)

- 45-minute sessions
- Conway's Game of Life is the problem
- Code is deleted after each session
- New pairs each session
- At the end of the day, we do a short retrospective where everyone answers the following questions
  - What, if anything, did you learn today?
  - What, if anything, surprised you today?
  - What, if anything, will you do differently moving forward?

For each session, a set of constraints is given. These constraints are generally a bit extreme. They have the goal of breaking the participants out of their usual way of thinking. Most people will begin working on a problem in the way they are comfortable. The constraints are there to remove the ability to code in a familiar, comfortable way. You can find some examples of session constraints in the appendix.

The urge to rush to finishing is strong. One of the goals of the morning sessions is to break this feeling, allowing people to relax into the idea of not finishing. It emphasises enjoying the feeling of explicitly not thinking about the end, but paying attention to the minute-by-minute coding, living for the moment-to-moment decisions when writing and refactoring. Often the resistance comes from a feeling of ownership: "this is my code, its existence represents me." Or, sometimes it is a sense of value: "this code is valuable, due to the time I've spent on it."

All these attitudes are learned habits and can be transcended by practice. In the context of coderetreat, the practice is repeatedly deleting the code you've written. Being interrupted and asked to delete it, starting over.

Pretty rapidly, most people gain a previously unrealized perspective about code ownership. In fact, after deleting and starting again

enough times, I've often heard people say they have a feeling of freedom they've not experienced before. The separation of identity from code frees them to experiment with new ideas. When value isn't tied to amount (or quality) of code, they can more readily accept that an attempt isn't working and discard it.

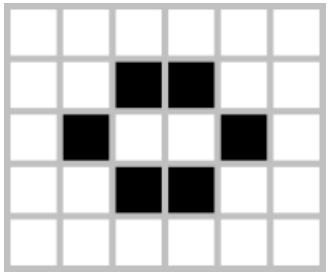
## Conway's Game of Life

At coderetreats we traditionally work on Conway's Game of Life. This application is very simple and easily understandable, yet the underlying domain and structure can hold a lot of subtle lessons in low-level design. When we couple this problem with a time limit and constraints to pull ourselves out of our comfort zone, it becomes even more rich.

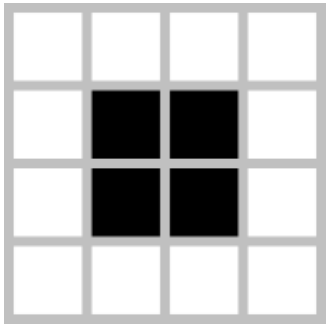
So, what is the game?

Conway's Game of Life (GoL) is what is known as a zero-player game. Sounds fun, right? It actually is a fascinating system called a cellular automaton. We set up an initial pattern on a board, start the program running, and the system evolves the board through a series of generations.

The game is played on an infinite two-dimensional grid. Each cell in this grid is considered either alive or dead.



**A Beehive of Living Cells**

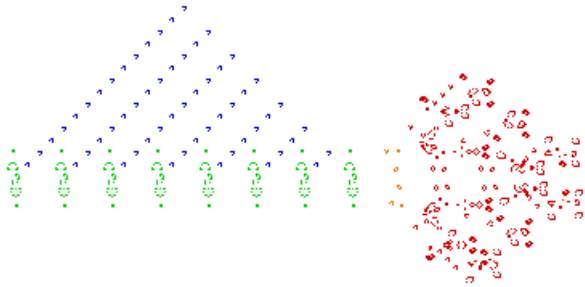


**A Block of Living Cells**

When we run the game, the program goes over each cell and calculates whether it will be alive or dead in the next generation. The determination is based on four simple rules that take into account the number of living neighbors. It is worth noting that there are eight neighbors to a cell, diagonals count.

1. If a living cell has less than two living neighbors, it is dead in the next generation, as if by underpopulation.
2. If a living cell has two or three living neighbors, it stays alive in the next generation.
3. If a living cell has more than three living neighbors, it is dead in the next generation, as if by overcrowding.
4. If a dead cell has exactly three living neighbors, it comes to life in the next generation.

Each tick of the game calculates the next generation based on these four simple rules. The beauty comes out when you see some of the fantastically complex structures that arise from such simplicity[6].



**An Active Breeder Pattern**

This system is the backdrop for the examples in this book. I would challenge you to spend a little bit of time thinking about how you would build this system, how you would design it. Assume that changes are coming, but you don't know what they are. But, don't just settle with one idea, see if you can come up with several different approaches. Take some time and think about it now. The rest of the book can wait. Check out the wikipedia entry[7] for more information. Once you are done, feel free to check out the sample list of coderetreat session constraints and ask yourself how they impact your proposed solution(s).

[6]Breeder Pattern picture by wikipedia user HyperDeath. For more information on licensing, see the picture's wikipedia page.

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

# 4 Rules of Simple Design

So, what are these 4 Rules of Simple Design?

Originally codified by Kent Beck in the late 90's, these rules outline some fundamental concepts around software design. The two core rules can guide us as we make our small, code-level refactorings.

Here they are in a simplified form.

1. Tests Pass
2. Expresses Intent
3. No Duplication (DRY)
4. Small

Let's look at these in order and see what they mean.

1. Tests Pass

   It makes sense that this would be the first one. After all, if you can't verify that your system works, then it doesn't really matter how great your design is, does it? With the modern tools that exist, we generally mean that these tests are automated. But, notice that the rule doesn't say "Automated Tests Pass," just "Tests Pass." It is about correctness and verification. Looking at this from the point of view of "easier to change," though, you can see that the length of time it takes to make sure your "Tests Pass" can be a significant factor in making changes. If you can type a command and have your system verified in a matter of seconds, or less, then you can change your system more readily than if you have to wait hours, or even days. So, when looking at your testing strategy, tend towards automated, and tend towards making them fast(er). I have a saying that I like to use:

   "If you have to ask how fast your test suite should be, it should be faster."

2. Expresses Intent

   How often have you went looking at a piece of code and found a method with name like `process_transaction`, but after looking more closely, you realize it neither processes nor has anything to do with transactions? This is an extreme case, but highlights an important problem when we are writing, and especially when you are updating, code: it is easy for the names we give things to stray from what they represent.

   One of the most important qualities of a codebase, when it comes time to change, is how quickly you can find the part that should be changed. The first step is identifying the code related to the functionality we are addressing. Paying attention to the names and how our code expresses itself is the key to making our lives easy when we come back to it.

   Also, over time, as we change the functionality of our system, classes and methods can become filled with unrelated behaviors. This makes it difficult to have the name effectively express their intent. As we start to see structures getting large, the difficulty in finding an expressive name is a red flag that it is doing too much and should be refactored.

3. No Duplication (DRY)

   This is the most subtle of the rules. We tend to think of duplication at a code level — a mechanical "this looks like that, so duplication!" level. However, this rule isn't about code duplication; it is about *knowledge* duplication.

   A lot of people are introduced to this idea through the DRY principle, or Don't Repeat Yourself. This was established in the book, The Pragmatic Programmer[8], by Dave Thomas and Andy Hunt.

   The DRY principle states "Every piece of knowledge should have one and only one representation." This rule also has been expressed as "Once and Only Once."

[8] http://pragprog.com/book/tpp/the-pragmatic-programmer

Instead of looking for code duplication, always ask yourself whether or not the duplication you see is an example of core knowledge in the system.

4. Small

   Once we've applied the above rules, it is important to look back and make sure that you don't have any extraneous pieces. Some questions I like to ask myself when I take a step back after writing some code.

- Do I have any vestigial code that is no longer used?

  This is an easy one. Sometimes, as we are working through our system, we build things that aren't used in the final product. Maybe they seemed like a good idea at the time, but the capability never came to fruition. If so, no questions asked, just delete that.

- Do I have any duplicate abstractions?

  In the course of refactoring, we often end up extracting abstractions, whether they be methods or new types. While we strive to keep duplication down, per the DRY principle, sometimes we find that we missed something. Perhaps the duplication is far apart in the codebase. Perhaps it is was hard to see the similarity when focused on the small. Take a moment to see if you notice anything now. If so, combine them.

  Sometimes, though, it isn't that the full abstractions are duplicate, but just that they have some similar characteristics, perhaps a behavior, or two. If so, then we might be missing another common abstraction that they can rely on. Don't wait, extract it.

- Have I extracted too far?

  In the course of writing, we can sometimes over-extract. A common case of this is when we extract a method for readability, to better express our intent. However, once we

> are done with the rest of our cleanup, we can inline the extracted method. This is a great example of the fluidity of a codebase's expressiveness over time.

An important thing to realize about these rules is that they iterate over each other. Frequently, fixing a naming issue will uncover some duplication. Eliminating that duplication will then reveal some expressiveness that can be improved. Joe Rainsberger wrote a great blog post about this iterative nature of the 4 rules[9].

There are many very interesting articles on the internet about the 4 rules of simple design. You can find them in the further reading section.

[9] http://blog.thecodewhisperer.com/2013/12/07/putting-an-age-old-battle-to-rest/

# Examples

While running coderetreats, I have the opportunity to see a lot of people working on Conway's Game of Life. As we go through the day, I make comments about design, both in the large and in the small. Over the years, I've seen similar patterns pop up across many different developers.

This section contains some of these concrete examples for the 4 rules of simple design and other lessons from coding Conway's 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, 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.

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.

## Duplication of Knowledge about Topology

Given that we are building with a cell abstraction, we can start to think about their locations. A common next step is to set certain cells to be alive at a given location, check for living cells at a location, etc.

A common, and pretty reasonable, implementation is to have something like a `World` class that contains these behaviors. A naive implementation might look at our 2-d grid and build the methods directly.

```
class World
  def set_living_at(x, y)
    #...
  end
  def alive_at?(x, y)
    #...
  end
end
```

And, of course, we might decide to add the coordinates to our `Cell` classes. After all, the cells are placed at a certain location.

```
class LivingCell
  attr_reader :x, :y
end
class DeadCell
  attr_reader :x, :y
end
```

On the surface, this seems okay. But, there is a subtle, not always obvious duplication of knowledge here: knowledge of our topology.

A good way to detect knowledge duplication is to ask what happens if we want to change something. What effort is required? How many places will we need to look at and change? For example, what if we want to change our topology to 3 dimensions? In our design, we would have quite a few places to change. This is duplication of knowledge; we have spread the knowledge of our topology — the fact that we are working on a 2-dimensional grid — all over the codebase. Eliminating this duplication relies on a strategy of *reification*. This is the act of taking a concept and making it real by extraction. So, let's extract the `x,y` to create a `Location` abstraction.

```
class Location
  attr_reader :x, :y
end
```

Now, doing this gives us a way to eliminate our duplication.

```
class World
  def set_living_at(location)
    #...
  end
  def alive_at?(location)
    #...
  end
end
class LivingCell
  attr_reader :location
end
class DeadCell
  attr_reader :location
end
```

By isolating this knowledge, we have made it easier to handle any change in our topology. Our code becomes more adaptable, along with making it much more clear.

While we looked at this refactoring from the perspective of duplication, we can also approach this as a naming problem: a lack of effectively expressing our intent.

To start with, the parameters x and y have horrible names. The fact that we "know" what they mean is a convention, rather than a result of being explicit. When encountering poor names, we often can find a missing abstraction by thinking about what the poorly-named variables represent.

```
class World
  def set_living_at(x, y)
    #...
  end
end
```

In our case, as a pair, the x and y represent a location in our system. A better way to represent it, to be explicit, is to name the pair.

```
class World
  def set_living_at(location)
    #...
  end
end
```

This then tells us that this parameter represents a single object, an instance of something.

Of course, we could take a small, interim step by making this a tuple on the caller side.

```
world.set_living_at([x, y])
```

A good step, but this does not solve the naming issue, it really just pushes it elsewhere in the code. That can be good as a small

step solution, but we'll want to clean it up there, too. Applying some judicious name fixing at that level could push us closer to a `Location` object.

## Behavior Attractors

A common and reasonable starting point for building Game of Life is to set living cells at locations. As per the rules, cells can be either living or dead. Here's an example of how this could be implemented.

```
class World
  def set_living_at(x, y)
    #...
  end
  def alive_at?(x, y)
    #...
  end
end
class Cell
  attr_reader :x, :y
  def alive_in_next_generation?
  # run rules
  end
end
```

Imagine that we are happily moving along with this design when we find ourselves in need of asking for the neighbor locations for a given x, y. Perhaps we want something like the following method.

```
def neighbors_of(x, y)
  # calculate the coordinates of neighbors
end
```

Whenever we have a new method — a new behavior — an important question is "where do we put it?" What type does this belong to? Unfortunately it isn't always immediately clear where to put it. Determining the right place for a new behavior can often be very challenging.

Should this go on the `Cell` class? It could make sense to ask a cell for its neighbors' coordinates. After all, the cell does have knowledge about where it is positioned on the grid. Of course, it is also focused on implementing the rules for evolving to the next generation. That feels like we are starting to put unrelated responsibilities onto the `Cell` class.

Perhaps a better place would be the `World` class? After all, this represents the larger world, where we can set living cells and query whether certain locations are alive. It clearly has knowledge of the grid. But, just thinking about the name "World," we can kind of get a sense that it has the makings of a God Class[10]. Perhaps we should stay away from putting more things there.

How many times have you run into this problem? You know you need a behavior, but there is a bit of confusion around its proper place. Too often, our solution is to punt on really analyzing the problem. Instead, we just put it in whatever file is open at the time. After all, we can always justify it later. Or, we tell ourselves we'll move it later, once we have more information. Once it starts getting used, though, moving it becomes less and less likely.

However, there could be a much more natural place. In an earlier example, we eliminated the knowledge duplication around the location, reifying a `Location` concept. Had we done this here, we might find that we already have a place that is just right.

```
class Location
  attr_reader :x, :y
end
```

Our other classes reference this, the `Location`, and rely on it to be entirely focused on the topology. What better place to put a behavior than the type that is concerned about the topology? Our behavior is really about asking for what locations constitute

[10] http://c2.com/cgi/wiki?GodClass

the neighborhood around a given location. Sounds like a natural behavior for the Location class.

```
class Location
  attr_reader :x, :y
  def neighbors
    # calculate a list of locations
    # that are considered neighbors
  end
end
```

This is an example of what I call a **Behavior Attractor**.

By aggressively eliminating knowledge duplication through reification, we often find that we have built classes that naturally accept new behaviors that arise. They not only accept, but *attract* them; by the time we are looking to implement a new behavior, there is already a type that is an obvious place to put it.

As a corollary to this, we can use this idea to notice potentially missing abstractions. If we are working on a new behavior, but are not sure where to place it — what object it belongs to — this might be an indication that we have a concept that isn't expressed well in our system.

## Testing State vs Testing Behavior

When starting a problem, especially when we are taking an outside-in approach, it is common to begin with the outermost object. We might start with some form of coordinator object for the use case we are writing. In Conway's Game of Life, we might think of a `World` class; an object that might coordinate the grid, the changing state of the cells, etc.

In our case, a great first test might be to see whether a `World` starts empty.

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

This is reasonable. It is simple. It establishes that we have a world. And, it establishes that there are no living cells in a new one.

The next test, of course, continues down this path. The previous test was looking at the idea of emptiness, so it seems natural to continue in that vein. How about placing a living cell? After all, that is how we start the game: placing cells. Since we've established an `empty?` method, we might make a simple test that the world isn't empty after placing a living cell.

```
def test_world_is_not_empty_after_setting_a_living_cell
  world = World.new
  location = Location.random
  world.set_living_at(location)
  assert_false world.empty?
end
```

Nice and simple tests, for sure. They seem to follow naturally. You probably can see the path forward now. "Nice and simple" is true.

It is worth noting, though, that they are leading to a very state-focused test suite. We are doing something, then checking what, if any, state change occurred.

An alternate way to develop a system is to focus on the behavior rather than the state of the objects. Think about what behaviors you expect and have our tests center around those. The idea of "focusing on behavior" is a common topic in software development conversations, but it isn't always that clear how to do it.

Building our system in a behavior-focused way is about only building the things that are absolutely needed and only at the time they are needed. This way, we end up with a system that has just enough code to support our use cases. When doing this, there is a handy tool I use to keep myself building only what is needed.

When I think there is something I want to build, I ask myself a simple question: "What behaviour of my system requires this?" Once I answer that question, I move to building that behavior.

In our case above, this formula generates two questions:

- How do we know that we want to set an individual cell?
- How do we know that we want to check that the world is empty?

Once we answer these questions — usually with a statement that "this behaviour will need it" — we can take a step back and build our tests around that behaviour.

Why do we need to set an individual cell? Above, we said that this might be how we set up the initial pattern. This leads to another question.

Why do we need the initial pattern? The point of the game is to calculate the next generation.

And there is where we have identified a fundamental behavior: calculating the next generation.

In our system, this fundamental behaviour happens with the tick, moving to the next generation. This is what triggers everything. So, let's start testing that. Then, as it needs behaviors, we can build those.

So, what is a very simple thing we say about a tick? The empty world should tick into another empty world.

```
def test_an_empty_world_stays_empty_after_a_tick
end
```

Now a question has come up. We just had a question about having our first test be about checking that a new world was empty. And, it seems like we've moved ourselves into a position where we need to do this. Since the test dictates that we start with an empty world, we probably should postpone this test and make sure that a new world is empty, so we can write the original test.

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

After this, we can move to our original test. We know that a new world is empty. So, we can fill our behavior-focused test with that knowledge.

```
def test_an_empty_world_stays_empty_after_a_tick
  world = World.new
  next_world = world.tick
  assert_true next_world.empty?
end
```

Huzzah! We're now a little more behavior-focused!

## Don't Have Tests Depend on Previous Tests

Let's look at our previous example's "behavior-based" test.

```
def test_an_empty_world_stays_empty_after_a_tick
  world = World.new
  next_world = world.tick
  assert_true next_world.empty?
end
```

Unfortunately, there is a subtle problem here.

How do we know that a newly-initialized `World` is empty? The test name indicates we are starting with an empty world, but the test code does not specify this explicitly. We talked about having our test names correspond to the test code in a previous example. Is this a problem here, though? We do have another test verifying this.

And there is our problem.

This test implicitly depends on the validity of a different, previous test: there is an assumption here that new worlds are empty. This causes a subtle, but important, problem; that lack of explicitness, combined with the coupling to the previous test, makes this test contribute to a fragile test suite. What happens if we change the parameters around a new world? What if we decide to make it not empty, but rather start with a stable structure, such as the block? In that case, our original "new world is empty" test fails, as it should. However, we'll get another failure "an empty world stays empty after a tick". We'll look at that test and wonder why it is failing. That's not good. We want test failures to be explicit, quickly and effectively pointing us to the problem. How should we resolve this?

Let's look back at the idea of letting the test name influence the test code and use that to make the test code a bit more explicit. Rather

than riding with the assumption that a new world is empty, let's explicitly ask for an empty world.

```
def test_an_empty_world_stays_empty_after_a_tick
  world = World.empty
  next_world = world.tick
  assert_true next_world.empty?
end
```

Now, if we change the default constructor to return something other than an empty world, this test will continue to pass. Only if we change what we mean by an empty world, created by `World.empty`, will this test fail. And, if we do that in such a way that the next world isn't empty, then this test will fail. And it should, because the statement we are verifying will no longer be true.

In fact, over time I've developed a guideline for myself that external callers can't actually use the base constructor for an object. Put another way: the outside world can't use `new` to instantiate an object with an expectation of a specific state. Instead, there must be an explicitly named builder method[11] on the class to create an object in a specific, valid state.

[11] I had originally used the term *factory method* here. Thanks to Ian Whitney for pointing out that it could be confusing, as this is really closer to the builder pattern.

## Breaking Abstraction Level

Automated unit test suites can have a tendency towards fragility, breaking for reasons not related to what the test is testing. This can be a source of pain when maintaining or making changes to a system. Some people have even gone to the extreme of moving away from unit- or micro-tests and only writing full-stack integration tests. Of course, this is the wrong reaction. Instead, we should investigate the source of the fragility and react with changes to our design.

It isn't always a problem with our system design, though. Sometimes fragility can come about because of problems in our tests. Let's take a look at a test we had in an earlier example. It is fairly small, and the assertion matches the test description.

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

But, there is a problem. Can you see it?

Our test talks about the world being empty and adding cells. However, looking at the test code, we can see details about the topology: the `(1,1)` tuple. We want to strive to have our tests be concise and clear about the behavior we are describing. However, in this case, our test code is implying that the `empty?` method is somehow dependent on the coordinates, themselves.

This is an example of breaking the level of abstraction. We are testing the behavior of the world, but we are including details that it isn't concerned with. If the actual topology knowledge is encapsulated in the location object, then the world should be relying on that object to manage those particulars. By tying this test to

concrete implementation of `2` dimensions, via the `(1,1)` tuple, rather than the `Location` abstraction, we are laying the groundwork for fragile tests: change the topology and tons of tests fail that are not related to the coordinate system. This coupling can be seen as another example of duplication: spreading the knowledge of the topology not just throughout the code, but also throughout the test suite.

To improve this, we work to hide the details of the topology from the world object. One way to do this is to use a stand-in, a test double for the location object. This can be as simple as creating a new, plain object.

```
def test_world_is_not_empty_after_adding_a_cell
  world = World.empty
  world.set_living_at(Object.new)
  assert_false world.empty?
end
```

Or, if you don't like the use of test doubles, you can use a builder method that provides a location without exposing implementation details.

```
def test_world_is_not_empty_after_adding_a_cell
  world = World.empty
  world.set_living_at(Location.random)
  assert_false world.empty?
end
```

Note: We could have used a more concrete location, like `Location.center`, but we aren't guaranteed that our grid has a center, especially if it is infinite.

By isolating ourselves from changes to the topology, the internals of the `Location`, we help ensure that this test won't break if we

change something about the underlying coordinate system. We also emphasize that the actual coordinates of the location are irrelevant in this test.

Personally, I like to use a test double in this case, as it highlights that we aren't using any specific attributes of the location object. And, if we find that we need some interaction with the location, we can specify it as constraints on the double. The result is that our test clearly expresses what behaviors of the location object we depend on. If we want to be even more explicit, we can give the test double a name. This can increase the readability of the test.

```
def test_world_is_not_empty_after_adding_a_cell
  world = World.empty
  world.set_living_at(double(:location_of_cell))
  assert_false world.empty?
end
```

By using a test double, we gain feedback that can help minimize the coupling of the behavior under test: we must be explicit about every interation. Because we have to specify the coupling points, we can be clear and confident about how many touch points our objects have with each other. This helps identify any abstraction problems; for example, if this test needs 3 methods stubbed on the location double, then that is a potential indication that we are missing an abstraction, or perhaps `set_living_at` is doing too much.

## Naive Duplication

Let's look at encoding the actual rules of a cell's evolution, how it transforms from generation to generation. A common design is to have a cell class with some sort of state to specify being alive or dead. We give it some sort of method to calculate its next state according to the evolution rules.

```
class Cell
  attr_reader :alive # true | false

  def alive_in_next_generation?
    if alive
      number_of_neighbors == 2 ||
            number_of_neighbors == 3
    else
      number_of_neighbors == 3
    end
  end
end
```

Let's start refactoring this. Any noticeable duplication?

Aha! That check around whether number of neighbors is 3 looks suspicious. Let's get rid of the duplication.

```
class Cell
  # ...
  def alive_in_next_generation?
    (alive && number_of_neighbors == 2) ||
          number_of_neighbors == 3
  end
end
```

We definitely got rid of the two instances of the number 3, but we have introduced new issues. This is due to what I consider naive, mechanical elimination of duplication: a refactoring that stems from a fundamental misunderstanding of the idea of DRY.

Let's review the idea of duplication. The Don't Repeat Yourself, or DRY, principle states:

> Every piece of knowledge has one and only one representation

Notice that it doesn't say anything about code. In fact, it has very little to do with code. Just looking at code that appears similar and combining them misses the point of the DRY principle.

With this clarity in hand, let's analyze our example a bit more closely.

These 3s are not the same. Thinking they are is a result of seeing a magic number without some sense of what it represents in terms of our domain. When thinking about duplication, it can help to expand the scope of our view, in this case to include the equality check, and to think about what it represents. In our alive case, the 3 is more closely linked to the 2 in the concept of a "stable neighborhood," while in the dead case, it is linked to something like a "genetically fertile neighborhood."

One good technique to keep from mistaking similar-looking code as actual knowledge duplication is to explicitly name the concepts before you try to eliminate the duplication. In our case, we would end up with something like this.

```
class Cell
  # ...
  def alive_in_next_generation?
    if alive
      stable_neighborhood?
    else
      genetically_fertile_neighborhood?
    end
  end
end
```

After this small refactoring, we can see clearly that the 3s represent different things. This is the power of paying close attention to the expressiveness of our code before blindly trying to eliminate duplication.

## Procedural Polymorphism

Take a look at the code that we have around the cell's evolution.

```
class Cell
  # ...
  def alive_in_next_generation?
    if alive
      stable_neighborhood?
    else
      genetically_fertile_neighborhood?
    end
  end
end
```

Notice that it contains a bit too much implementation detail. The method name `alive_in_next_generation?` is more about implementation, the move from generation to generation, rather than a description of the behavior we want. It is more of a state-oriented statement "alive in next generation?" rather than a question about behavior.

When we find these very generic names, we are looking at an expressiveness problem. Why is "alive" the state we are interested in? What if we add another state?

However, if we think about a better name, we have a hard time. In the case of a living cell, this is really whether it stays alive. In the case of a dead cell, though, it is about the cell coming to life. How can we reconcile this inconsistency?

Before diving straight into tackling the reconciliation, let's start at a lower level, inside the method, and see if we can gain any insight.

Starting at the top, let's look at the branching variable, `alive`; there are a few different questions we could ask ourselves about it.

The name of this variable captures a default, or preferred, state: alive. Why is this the thing we highlight? Each cell is really in one of two states; why not highlight dead? What if we change the concept of living? What if it isn't binary? Changing this means we have to change code also related to the other two states. We also are spreading the concept in several places: alive has to do with both the variable and the method that uses it.

A seemingly quick solution would be to make it something like `state`, but that masks our intention a bit. What are the possible states?

```
class Cell
  # ...
  def alive_in_next_generation?
    if state == ALIVE
      stable_neighborhood?
    elsif state == DEAD
      genetically_fertile_neighborhood?
    end
  end
end
```

This isn't much better; We now have even more of an expressiveness problem with this branching: do we really know these are the only ones? I also feel a bit uncomfortable when I see an `if-elsif` sequence without a raw else.

Variables named `state` are also a huge red flag for expressiveness. Does a cell really change state? Do dead cells change state into living cells? Or are living cells created? To be honest, too often *state* variables are usually just an indiciation that we've given up on really understanding and encoding our intention.

Resolving this requires us to talk a bit about polymorphism in general. Polymorphism is about being able to call a method/send

a message to an object and have more than one possible behavior. This can be one of the most powerful techniques in programming.

In our case, we are providing a form of polymorphism with this method. When this method is called, the caller can expect one of two different behaviors: either the ruleset for living cells or the ruleset for dead cells. Which ruleset gets run is based on an internal state, hidden from the outside world. In a way, this is good; the caller shouldn't have to care. But it is worth looking at the method we use to achieve the goal.

When we use a branching construct inside a method like this, we run into several problems. We've talked about the expressiveness problem, but we also have issues with changing this code. If we are going to add a state, or change rules around the states, we will find ourselves modifying existing code. Not just existing code, but code that is unrelated to the change we are making. If we add a state, why would we force ourselves to modify the code related to the other states? When we begin to overload concepts in our system, especially method names, we run into this "everything goes here" situation.

In general, `if` statements (or other branching constructs) are imperative, procedural mechanisms. While they do provide a form of polymorphism, they provide a form that I call **Procedural Polymorphism**. It satisfies our needs for selecting a behavior, but their procedural background leads to tightly-coupled code, joining these often unrelated behaviors together.

Luckily, object-based and object-oriented languages provide a preferred method for polymorphism, what I call **Type-Based Polymorphism**. The idea is one central to object-oriented design: use different types for the different branches. The general approach is to analyze what the branching condition is, identify the concepts, and reify them into first-class concepts in our system.

In our example, we can take our *state* and raise it to types: `LivingCell` and `DeadCell`.

```
class LivingCell
  def alive_in_next_generation?
    # neighbor_count == 2 || neighbor_count == 3
    stable_neighborhood?
  end
end
class DeadCell
  def alive_in_next_generation?
    # neighbor_count == 3
    genetically_fertile_neighborhood?
  end
end
```

At this point we have separated out the concepts. And, if we choose to, we can also inline the business rule methods without sacrificing too much.

```
class LivingCell
  def alive_in_next_generation?
    neighbor_count == 2 || neighbor_count == 3
  end
end
class DeadCell
  def alive_in_next_generation?
    neighbor_count == 3
  end
end
```

We also have higher-level names for our concepts, which makes it easier to find where changes need to occur.

A huge benefit of this is that we also have provided ourself a safer method for adjusting the different states a cell can be in. If we need

to add a new one, we add a new class. We *extend* our system, rather than modify it. This is an example of the open-closed principle[12].

```
class ZombieCell
  def alive_in_next_generation?
    # new, possibly more complex rules
  end
end
```

It also provides a clear method for fixing the names of our methods to match the actual concepts in our system, focusing on specific behaviors, rather than a generic idea of `alive_in_next_generation`.

```
class LivingCell
  def stays_alive?
    neighbor_count == 2 || neighbor_count == 3
  end
end
class DeadCell
  def comes_to_life?
    neighbor_count == 3
  end
end
```

At this point, we now have very explicit statements of the intent of the types and their behaviors. But, changing these names takes away the polymorphism! We no longer can call a single method and have the appropriate rules applied. This is true. This could be an indication that the idea of having the initially-desired polymorphism isn't a good design. Naturally it depends on how we end up using the cells, but focusing heavily on explicitness in this fashion can raise flags about desired or "planned" designs.

[12]Components should be open for extension, but closed for modification. See the Other Design Guidelines section in the back of this book.

## Making Assumptions About Usage

Let's look at the `Cell` classes.

```
class LivingCell
  def stays_alive?(number_of_neighbors)
    number_of_neighbors == 2 ||
          number_of_neighbors == 3
  end
end
class DeadCell
  def comes_to_life?(number_of_neighbors)
    number_of_neighbors == 3
  end
end
```

It seems reasonable that those methods would be there. After all, the following reads okay. Or, at least, it feels familiar.

```
cell.stays_alive?(number_of_neighbors)
```

But, there are a couple possible flags here.

First, notice that we are talking about **entity classes** here. That is, we have objects representing concrete abstractions: Cells. Classes of this nature tend to encapsulate and provide behavior around state. Methods on them are generally involved in working with that state. For example, query methods provide a way to access the state. In this case, though, the methods are not accessing internal state, at all. In fact, they are primarily using the passed-in value, `number_of_neighbors`.

It is true that we could say that the rules, themselves, the comparisons are related to the cell and constitute cell-focused knowledge. While cell-focused, they really represent the rules. But why is `Cell`

our abstraction around executing the rules? Why don't we reify the idea of a rule? One of the key parts of being easier to change (i.e. a better design) is being able to more easily find where the changes need to occur; this is what good naming contributes to. So, if we were to come to a large system, and we wanted to change the rules for evolution, you might look at a `Cell` class. But imagine if there was a `Rule` class. That could probably be an even larger signpost. Let's play with this a bit by just adding Rules to the class names.

```
class LivingCellRules
  def stays_alive?(number_of_neighbors)
    number_of_neighbors == 2 ||
          number_of_neighbors == 3
  end
end
class DeadCellRules
  def comes_to_life?(number_of_neighbors)
    number_of_neighbors == 3
  end
end
```

This looks interesting. Of course, we've now lost an abstraction, the Cell. This will influence our location objects. Are the locations linked to the current rules depending on the state, or is there still some placeholder idea of a cell? Do we even need a reified cell abstraction? What is causing us to have it? In fact, if we think about it, the concept of a `DeadCell` has a potential trap in it. We are working with an infinite grid. So, which dead cells are we keeping track of? Which locations are we tracking? How do we know that we should instantiate a location object for a given `(x,y)` pair? We can't keep track of all of them. Perhaps it does make sense to question the concept of a concrete cell class.

A lot of questions that arise have a "*do we need this abstraction*" flavor. This happens quite frequently when following an inside-out development style. We start somewhere in our domain, making

a very large assumption that the abstractions we are building will be needed sometime. As we've seen, new abstractions can be developed and investigated through refactorings, but it can be easy to work yourself into a corner. The fundamental thought that is hidden in "*do we need this abstraction*" is "use influences structure." So, should we have `LivingCellRules` and get rid of `LivingCell`? Should location objects keep a link to the rule, rather than the cell? Perhaps the location object doesn't actually contain this link at all. Perhaps the existence of an instantiated location object implies `LivingCellRules`. So many answers not just disappear but never come up when building abstractions and behaviors through actual usage. This is often what happens when using an outside-in development method.

## Unwrapping an Object

A common (and perhaps one of my favorite) constraints in coderetreat is writing your code with "no return values." Most of the time, in our daily lives, we build in a very imperative style, asking several objects for their data, perhaps some calculations, then we enact an algorithm and stuff the results back into other objects. By eliminating the ability to return values from our functions, we force ourselves to rely instead on telling objects to enact behaviors.

Another side effect of this constraint is that you no longer can have properties on your objects — no methods for querying the internal state. By eliminating the ability to query for data, we begin to build objects that are very tightly encapsulated. We can rely on the objects alone to manage their internal state.

Whenever this constraint comes up, there is an inevitable question: "How do you test for equality?" As people work on the problem, they notice that they need a way to compare whether two location objects represent the same place on the grid.

As an example, imagine we have the following two location objects.

```
class Location
  attr_reader :x, :y
end

location1 = Location.new(1, 1)
location2 = Location.new(1, 2)

if location1.equals?(location2)
  # Do something interesting
end
```

Ordinarily, you'd write something like this.

```
class Location
  attr_reader :x, :y
  def equals?(other_location)
    self.x == other_location.x &&
          self.y == other_location.y
  end
end

location1.equals?(location2)
```

Of course, `equals?` as a method name here is a bit poor. This isn't really looking to see if the locations are equal, as much as representing the same place in space. But, for our purposes here, this is good enough.

But, this `equals?` method doesn't conform to our constraint: it is asking other_location to *return* its x and y. This isn't allowed.

This can be a very common sticking point. Most of us have been trained to use properties to access internal state of an object. We pretend, of course, that we are using properties to encapsulate state, but really it is just a way to allow the outside world to reach inside us and do what they want. In a world where you can't return anything, though, how do you get around this?

The key idea is in a technique that I call **unwrapping**. Take a look at the following alternate form of equals.

```
class Location
  attr_reader :x, :y
  def equals?(other_location)
    other_location.equals_coordinate?(self.x,
                                      self.y)
  end
  def equals_coordinate?(other_x, other_y)
    self.x == other_x && self.y == other_y
  end
end
```

Look what this is doing. Inside the first object (*location1*), we have access to our own internals. Rather than taking the approach of asking the other object (*location2*) for its internals, let's just pass our own to it. So, we are comparing internals without having to reach inside the other object.

Of course, in a language with signature-based overloading, you wouldn't have to have two methods.

```
public class Location
{
  private int x;
  private int y;
  public boolean Equals(Location otherLocation) {
    return otherLocation.Equals(this.x, this.y);
  }
  public boolean Equals(int otherX, int otherY) {
    return this.x == otherX && this.y == otherY
  }
}
```

But, wait, you say. Doesn't `equals?` return a boolean? The constraint is that we can't return anything. So, we are violating that.

This is true. Now that we have a way to do the comparison without querying for an object's state, we can tackle this aspect.

Let's take a step back and look at this from a behavioral point of view, returning to the fundamental question "why do we need this behavior?" Or, "why do we care if they are equal?" In general, we look for equality in order to react in a certain way. So, if they are equal, we'll do something. As a simple example, let's increment a counter.

Since we can't return the boolean, let's rewrite our code to remove that. In Ruby, every method returns something, so we have to be explicit to get rid of the boolean return.

```
class Location
  attr_reader :x, :y
  def equals?(other_location)
    other_location.equals_coordinate?(self.x, self.y)
    nil
  end
  def equals_coordinate?(other_x, other_y)
    self.x == other_x && self.y == other_y
    nil
  end
end
```

So, now we can't get access to it. That satisfies the constraint, but it doesn't do us much good. We want to do something if they are equal. Since we are can't react to the comparison outside the objects, we need to move the behavior inward closer to where the action is happening. Notice that `equals_coordinate?` does the comparison. So, this is where we need to do the behavior.

Ordinarily, we would write something with a simple `if` statement.

```
count_of_locations = 0
if location1.equals?(location2)
  count_of_locations++
end
```

Instead, let's take the behavior, wrap it in a lambda, and move it to where the comparison is happening.

```
count_of_locations = 0
location1.equals?(location2, -> { count_of_locations++ \
})
```

In this code, we expect the lambda to be called if the locations turn out to be equal. Let's fix our code to support this.

```
class Location
  attr_reader :x, :y
  def equals?(other_location, if_equal)
    other_location.equals_coordinate?(self.x, self.y, i\
f_equal)
    nil
  end
  def equals_coordinate?(other_x, other_y, if_equal)
    if self.x == other_x && self.y == other_y
      if_equal.()
    end
    nil
  end
end
```

Now, we have a situation where we are telling a location object (*location1*) "Here is another location object (*location2*). If you are equal to it, do this (*if_equal*)."

Note: In most languages, there is some form of first-class function which makes this technique fairly straight-forward. Sadly, Java only recently got these. So, you have to solve this using some form of a command object. Is this bad? Not necessarily, although it can be a bit cumbersome.

## Inverted Composition as a Replacement for Inheritance

Take a look at these cell classes.

```
class LivingCell
  attr_reader :location
end
class DeadCell
  attr_reader :location
end
```

We've extracted the location object. One benefit of this, of course, is that gives us a centralized place for our topology knowledge. This is very nice, but we can see another duplication here, as well. Both the living cell and the dead cell have a location attribute.

Is this knowledge duplication? What is the knowledge we are duplicating? Since these are two different objects, and this is "just" an attribute, we can be tempted to say it isn't. As we look at this code in light of the 4 rules, we want to make sure that what we have is actual knowledge duplication, rather than just incidental, implementation similarity. After all, extracting the location object was about taking the "actual" knowledge and representing it in one place.

We can look at this as knowledge duplication, since this location attribute represents the fact that our cells are linked to a specific position on the grid. It is an interesting case here, where eliminating a specific duplication didn't eliminate **all** the duplication, just part of it.

So, let's look at ways to eliminate this duplication.

A common attempt at a solution to this is to jump to inheritance. We could do something like the following.

```
class Cell
  attr_reader :location
end
class LivingCell < Cell
end
class DeadCell < Cell
end
```

Wait, though, let's look at this code a minute.

Now, it does seem to simplify our code a bit if we think in terms of lines of code. But, is it really simpler? It does add another type after all. I often say having more classes isn't bad, as long as they are the correct abstractions. But, unlike the extraction of the `Location` class, this extraction doesn't introduce a new domain concept; this abstraction increases the complexity without adding additional information about our domain. This feels like a violation of the fourth rule, "small."

Inheritance is often used as way of creating "reuse" rather than eliminating duplication. We are assuming that both the `LivingCell` and `DeadCell` need to have access to their location (do they?), so we provide access through the base class. Even if we support our assumption, however, the objects don't need access to their location necessarily, they really would need access to the behaviors that the location object exposes. And, of course, at this point, we haven't even talked about whether they truly do.

So, let's ask again: is it really eliminating the duplication? The location attribute is still there on the objects. Our two different types still contain the same knowledge.

Base classes of this nature, extracted entirely to eliminate apparent duplication can have a tendency to hide actual duplication. Also, it is very common for these base classes to become buckets of unrelated behavior.

So, if inheritance isn't really eliminating the knowledge, what other options do we have?

In Ruby, we do have modules. This might be a good use for them.

```
class LivingCell
  include HasLocation
end
class DeadCell
  include HasLocation
end
```

And the `HasLocation` module adds `attr_reader :location` to the including class. Modules, when used this way, though, are just a way to implement multiple inheritance. The same arguments arise as in the above discussion of straight subclassing.

I do believe this is slightly better than using a base class, `Cell`. Modules are often used as a way of grouping aspects of different classes, and this can be useful for code organization. But this technique should be used very judiciously. Primarily, I use modules in this way as a step in the path towards a better design. Separating out aspects of a class into modules can help find hidden dependencies, as well as highlight all the different responsibilities a class has. But they are rarely the place to stop.

So, with that option off the table, how do we eliminate the duplication? Let's look at what we are trying to accomplish. Our goal is to have a link between the `Cell` and the `Location` it is at. Or, rather, our system needs to know this link. We haven't actually seen anything to indicate the `Cell` classes, themselves, need the link. Our assumption here is that something needs to see the link.

When having two types containing a link to the same type `(Living|Dead)Cell` and `Location`, a useful technique is to reverse the dependency.

```
class Location
  attr_reader :x, :y
  attr_reader :cell
end
class LivingCell
  def stays_alive?(number_of_neighbors)
    number_of_neighbors == 2 ||
          number_of_neighbors == 3
  end
end
class DeadCell
  def comes_to_life?(number_of_neighbors)
    number_of_neighbors == 3
  end
end
```

At this point, our cell classes are indeed just focused on information related to the cell (for example, rules). The topology is also further abstracted from the rules of the game. We can start to see that the `Location` class is taking on a structural role, providing the link between the topology and the cell that exists there. The cell classes are now focused on rules around evolution.

While the refactoring is good, it highlights a potential naming issue. Is `Location` the correct name for this class? From a reading point of view, it seems like a `Cell` should have a `Location`, not the other way around. This is arguable, of course, but it seems like potentially we chose the wrong name for the `Location` class. Perhaps it is better as a `Coordinate`.

```
class Coordinate
  attr_reader :x, :y
  attr_reader :cell
end
```

I'm not saying it is, or not, at this point. I only wanted to mention it is interesting how eliminating the duplication highlighted a possible naming issue. This is a good example of how applying these rules can often lead to other refactoring opportunities and insight into our design.

# Other Good Stuff

This section contains material either not directly related to the focus of the book, or is supplementary to the content outlined in the book.


- Other Design Guidelines
- Examples of Session Constraints
- Some Thoughts on Pair-Programming Styles
- Further Reading

# Other Design Guidelines

While the 4 rules of simple design are fundamental concepts in building an adaptable codebase, there are a lot of other design guidelines out there. It is worth studying these, as well.

Just like Design Patterns are best used as descriptions of designs, rather than prescriptions of how to build systems, design guidelines, especially higher-level ideas like SOLID, can be best used when describing where you are, and why it is appropriate. Along with that, looking back at a design, it can be useful to explain why a piece of code does NOT abide by this or that design guideline. It can be difficult, though, at the time of writing, to really apply most of these principles. In the end, most design guidelines are best internalized and applied subconsciously. Once this happens, these guidelines can move into the realm of descriptive usage.

The 4 rules, on the other hand, are simple enough to apply consciously. Thinking about good naming to express intent and looking for duplication of knowledge are two techniques that can have real effects on the code you write at the time of writing.

Over the years, I've come to see the 4 rules of simple design as the most useful concrete, coding-time principles to keep in mind. While the examples in this book primarily use the 4 rules to guide the code, let's look at how they lead us naturally to code that satisfies these other principles.

## The SOLID Principles

The SOLID principles[13] were originally codified by Robert "Uncle Bob" Martin, bringing a set of existing design principles together in

[13] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

a more easily-understandable format. Like the 4 rules, they focus on making systems flexible and adaptable when changes are required. As we'll see, focusing on the 4 rules of simple design can lead us to satisfying these principles. Let's look at these principles, and how they can relate to flexible designs.

**Single Responsibility (SRP)**

> "A class (component) should have one, and only one, reason to change"
>
> The Single Responsibility Principle is by far one of the most popular, while simultaneously one of the least understood. Because of the name, discussions around this consist of defining what is meant by "responsibility" of a component, while the definition talks only about change. "Change" makes it a bit more concrete, but still leaves a lot open for discussion: what level of change do we look at?
>
> Systems that satisfy the SRP are flexible with isolated behaviors contained in small, cohesive packages. This allows us to safely make changes to functionality. At its core, SRP is another way of maximizing cohesion. After vigorously eliminating duplication and making sure that our pieces are named appropriately and expressively, we generally find that our code satisfies the SRP.

**Open-Closed (OCP)**

> "A system should be open for extension, but closed for modification"
>
> Changing code is dangerous; once we have it written and tested, we want to minimize the chance for bugs to be introduced. By introducing or altering behavior only through extension, we benefit from the stability of small, stable core pieces that won't change out from under us.
>
> There is a danger when focusing too much on the OCP. If we plan for extensibility, our systems become riddled with

unnecessary and unwieldy extension points and extensibility mechanisms. To counter this, we should focus on isolating knowledge, naturally building only the extension points that truly represent the pieces of our system that will change.

**Liskov Substitution (LSP)**

"Derived types should be substitutable for their base types"

Polymorphism is a key part of a flexible design. Being able to substitute a more specific type when a general type is expected allows us to provide different behaviors without having complex branching. There is a danger, though, if the specialized type significantly changes fundamental expectations of the more general type's behavior. Derived types should enhance any base behaviors, rather than change it.

A healthy focus on the names we give our types can help in abiding by LSP. A specialized type's name should reflect that it is an enhancement of the base, not a change.

**Interface Segregation**

"Interfaces should be small, focused on a specific use case"

The surface area of a class has a direct influence on how easy it is to use. Although a class might have several different ways to use it, any specific client should see only those behaviors specific to its needs.

When we focus on effectively grouping *and naming* the behaviors of our class, we naturally build small interfaces that provide a clear, cohesive view of what our class does. If it is difficult to name, that is feedback that our class is getting too large.

**Dependency Inversion**

"Depend on abstractions, rather than concrete implementations"

One of the most dangerous parts when changing a system is having your changes unexpectedly influence other, unrelated

parts of your system. We want to guard against the situation where a change ripples through the whole system, causing waves and possible bugs throughout. By depending on abstractions, decoupling ourselves from concrete implementations, we can set up walls between behaviors. Abstractions better move us into standardized communication methods between components, making it easier to independently replace or change things.

## Law of Demeter

Contrary to popular belief, the Law of Demeter (Lod) was not originally described by a person named Demeter. Nor was it a reference to the ancient Greek god, Demeter (patron god of measurement, of course). Instead, it was developed during the Demeter project as a simple guideline for the code there.

The original statement can sound a bit cryptic to ears raised on current languages, but in a simple form, you can think of it as:

A method can access either locally-instantiated variables, parameters passed in, or instance variables.

A much simpler way to think about it is:

Only one dot per statement.

At its heart, the LoD is about encapsulation. We don't want to reach inside an object and manipulate its insides; that's just mean. Instead, we want to ask objects to perform some action for us. Let the object deal with its collaborators.

The LoD can also be thought of in terms of knowledge duplication. By exposing the internals of an object, we are spreading structural knowledge through our code. Both the object *and* the outside collaborator know about its internals.

Personally, I find the LoD to be an extremely simple, incredibly powerful mechanism for helping ensure proper encapsulation and

decoupling of behaviors across an object graph. The best way, of course, to get a sense of its power is to read the Original Paper[14].

[14]http://www.ccs.neu.edu/research/demeter/papers/law-of-demeter/oopsla88-law-of-demeter.pdf

# Example constraints

Some constraints that I've used:

**Lines of code per method <= 3**
: We all have seen methods we consider too large. But what size is reasonable for a method? Let's go to the extreme and say three lines is the maximum.

**No in-method branching statements**
: Branching statements, such as the notorious `if`, are a form of procedural polymorphism. Let's work on improving our object skills by finding other mechanisms to get the same effect. Prefer type-based polymorphism or lookup tables.

**No primitives across method boundaries (input or output)**
: The only types that can be passed across method boundaries (inputs and outputs) are ones that we have defined. No data primitives, such as booleans, integers or strings. You also are not allowed to use primitive data structures such as Lists, Arrays or Enumerables. Focus instead on understanding what the types represent and building types for those concepts.

**Mute ping-pong pairing**
: One member of the pair writes the unit tests, the other member writes the code to turn those tests green. You can think of the roles as "test redder" and "test greener." This is standard. However, the only communication allowed between partners is through the tests and the code. And no cheating by putting a bunch of comments!

**Find the loophole**
: Generally coupled with ping-pong pairing, one pair writes

the tests, the other pair tries to get those tests passing. The catch is that the pair working to get the tests passing writes the wrong code. How long can you go before the tests force you into a "correct" algorithm. Here's the catch, though: you must write production-level code, think of it as code you would show a prospective employer.

**No return values**

Similar to the principle of "tell, don't ask" this constraint takes away your ability to return values from a function. Focus entirely on sending messages to objects.

**Program like it's 1969**

Compilation is expensive and not real-time. You can only run your code twice during the session: first at the 30-minute mark, the second at the 44-minute mark. Better pay attention to syntax!

**Object Calisthenics**

Practicing different aspects of object-oriented design is great, but what happens when you put everything together in an extreme situation? Object Calisthenics[15] provides a set of rules that really work out your OO understanding.

[15] http://www.cs.helsinki.fi/u/luontola/tdd-2009/ext/ObjectCalisthenics.pdf

# 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.

# Further Reading

## 4 Rules of Simple Design

Here are some significant places to read more about the 4 rules of simple design.

There is some interesting discussion on the c2 wiki page for XP Simplicity Rules[16]. And, of course the c2 wiki will have links to a plethora of discussions around related principles.

For example, there is also information and discussion on the c2 wiki about the DRY Principle[17].

Conversations with Joe Rainsberger have had a significant impact on my thoughts around applying the 4 rules. He has a blog post titled "The Four Elements of Simple Design"[18]. He also has a wonderful post that explains the apparent discrepancy in how different people order rules 2 and 3, titled "Putting an Age-Old Battle to Rest"[19].

And, of course, these are just a start. I highly recommend that you do a search for more[20].

## General Design

Here are some suggestions for books and papers to read on the general topic of design.

[16] http://c2.com/cgi/wiki?XpSimplicityRules

[17] http://c2.com/cgi/wiki?DontRepeatYourself

[18] http://www.jbrains.ca/permalink/the-four-elements-of-simple-design

[19] http://blog.thecodewhisperer.com/2013/12/07/putting-an-age-old-battle-to-rest/

[20] https://www.google.com/#q=4+rules+of+simple+design

**Practical Object-Oriented Design in Ruby**[21] **by Sandi Metz**

This is a fantastic book that covers ideas and guidelines for building maintainable systems. Sandi is an experienced developer with tons of object-oriented design experience. She definitely knows her stuff and how to explain it in an understandable fashion. Although this book has Ruby in the title, most of the lessons in it are applicable across all object languages.

**On the criteria to be used in decomposing systems into modules**[22] **by David Parnas**

Wow, what can I say about this paper? It is from 1971. It covers an investigation into two types of modularization and the effect they have on maintainability. You know it is going to be good by the first line of the abstract, "This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time." Read it!

**Law of Demeter**[23]

There is a whole section on this in the other design guidelines section, so I won't say too much. Go read this paper, though. You won't be sorry.

**Clean Code**[24] **by Robert Martin**

This book lays out in fine detail some general rules around how to write robust, maintainable code. Through concrete examples and guidelines, the book is a wonderful tour through techniques that can make a codebase stand the test of time.

---

[21] http://www.poodr.com/

[22] http://repository.cmu.edu/cgi/viewcontent.cgi?article=2979&context=compsci

[23] http://www.ccs.neu.edu/research/demeter/papers/law-of-demeter/oopsla88-law-of-demeter.pdf

[24] http://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882/

## Testing

**Growing Object-Oriented Software Guided By Tests**[25] **by Nat Pryce and Steve Freeman**
Walk through building a system using "London-style" test-driven development by the guys who literally invented the style of heavy isolation through mocks. The examples are in Java, but the techniques can easily be translated over to other languages.

**Mock Roles, not Objects**[26] **by Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes**
A classic paper about effective use of test doubles when doing unit testing.

**Test-Driven Development Screencast**[27] **by Kent Beck**
This is a fabulous short screencast series where Kent Beck goes through building a component using TDD. All the while, he outlines his thought process. Rare chance to watch and listen to the father of test-driven development use the technique.

## Other Things You Probably Should Most Definitely Read

**The Pragmatic Programmer**[28] **by Dave Thomas and Andy Hunt**
This book literally changed my life. Filled with tips and ideas about how to be effective as a software developer, including a section on the DRY principle.

[25] http://www.growing-object-oriented-software.com/
[26] http://jmock.org/oopsla2004.pdf
[27] http://pragprog.com/screencasts/v-kbtdd/test-driven-development
[28] http://pragprog.com/book/tpp/the-pragmatic-programmer