Refactoring To Patterns

version 0.17

Joshua Kerievsky
joshua@industriallogic.com
Industrial Logic, Inc.
http://industriallogic.com

Introduction

> Patterns are a cornerstone of object-oriented design, while test-first programming and merciless refactoring are cornerstones of evolutionary design. To stop over- or under-engineering, it's necessary to learn how patterns fit into the new, evolutionary rhythm of software development.

The great thing about software patterns is that they convey many useful design ideas. It follows, therefore, that if you learn a bunch of these patterns, you'll be a pretty good software designer, right? I considered myself just that once I'd learned and used dozens of patterns. They helped me develop flexible frameworks and build robust and extensible software systems. After a couple of years, however, I discovered that my knowledge of patterns and the way I used them frequently led me to over-engineer my work.

Once my design skills had improved, I found myself using patterns in a different way: I began refactoring to patterns, instead of using them for up-front design or introducing them too early into my code. My new way of working with patterns emerged from my adoption of Extreme Programming design practices, which helped me avoid both over- and under-engineering.

Zapping Productivity

When you make your code more flexible or sophisticated than it needs to be, you over-engineer it. Some do this because they believe they know their system's future requirements. They reason that it's best to make a design more flexible or sophisticated today, so it can accommodate the needs of tomorrow. That sounds reasonable, if you happen to be a psychic.

But if your predictions are wrong, you waste precious time and money. It's not uncommon to spend days or weeks fine-tuning an overly flexible or unnecessarily sophisticated software design—--leaving you with less time to add new behavior or remove defects from a system.

What typically happens with code you produce in anticipation of needs that never materialize? It doesn't get removed, because it's inconvenient to do so, or because you expect that one day the code will be needed. Regardless of the reason, as overly flexible or unnecessarily sophisticated code accumulates, you and the rest of the programmers on your team, especially new members, must operate within a code base that's bigger and more complicated than it needs to be.

To compensate for this, folks decide to work in discrete areas of the system. This seems to make their jobs easier, but it has the unpleasant side effect of generating copious amounts of duplicate code, since everyone works in his or her own comfortable area of the system, rarely seeking elsewhere for code that already does what he or she needs.

Over-engineered code affects productivity because when someone inherits an over-engineered design, they must spend time learning the nuances of that design before they can comfortably extend or maintain it.

Over-engineering tends to happen quietly: Many architects and programmers aren't even aware they do it. And while their organizations may discern a decline in team productivity, few know that over-engineering is playing a role in the problem.

Perhaps the main reason programmers over-engineer is that they don't want to get stuck with a bad design. A bad design has a way of weaving its way so deeply into code that improving it becomes an enormous challenge. I've been there, and that's why up-front design with patterns appealed to me so much.

The Patterns Panacea

When I first began learning patterns, they represented a flexible, sophisticated and even elegant way of doing object-oriented design that I very much wanted to master. After thoroughly studying the patterns, I used them to improve systems I'd already built and to formulate designs for systems I was about to build. Since the results of these efforts were promising, I was sure I was on the right path.

But over time, the power of patterns led me to lose sight of simpler ways of writing code. After learning that there were two or three different ways to do a calculation, I'd immediately race toward implementing the Strategy pattern, when, in fact, a simple conditional expression would have been simpler and faster to program—a perfectly sufficient solution.

On one occasion, my preoccupation with patterns became quite apparent. I was pair programming, and my pair and I had written a class that implemented Java's TreeModel interface in order to display a graph of Spec objects in a tree widget. Our code worked, but the tree widget was displaying each Spec by calling its toString() method, which didn't return the Spec information we wanted. We couldn't change Spec's toString() method since other parts of the system relied on its contents. So we reflected on how to proceed. As was my habit, I considered which patterns could help. The Decorator pattern came to mind, and I suggested that we use it to wrap Spec with an object that could override the toString() method. My partner's response to this suggestion surprised me. "Using a Decorator here would be like applying a sledgehammer to the problem when a few light taps with a small hammer would do." His solution was to create a small class called NodeDisplay, whose constructor took a Spec instance, and whose one public method, toString(), obtained the correct display information from the Spec instance. NodeDisplay took no time to program, since it was less than 10 simple lines of code. My Decorator solution would have involved creating over 50 lines of code, with many repetitive delegation calls to the Spec instance.

Experiences like this made me aware that I needed to stop thinking so much about patterns and refocus on writing small, simple, straightforward code. I was at a crossroads: I'd worked hard to learn patterns to become a better software designer, but now I needed to relax my reliance on them in order to become truly better.

Going Too Fast

Improving also meant learning to not under-engineer. Under-engineering is far more common than over-engineering. We under-engineer when we become exclusively focused on quickly adding more and more behavior to a system without regard for improving its design along the way. Many programmers work this way—I know I sure have. You get code working, move on to other tasks and never make time to improve the code you wrote. Of course, you'd love to have time to improve your code, but you either don't get around to it, or you listen to managers or customers who say we'll all be more competitive and successful if we simply don't fix what ain't broke.

That advice, unfortunately, doesn't work so well with respect to software. It leads to the "fast, slow, slower" rhythm of software development, which goes something like this:

1. You quickly deliver release 1.0 of a system, but with junky code.
2. You attempt to deliver release 2.0 of the system, but the junky code slows you down.
3. As you attempt to deliver future releases, you go slower and slower as the junky code multiplies, until people lose faith in the system, the programmers and even the process that got everyone into this position.

That kind of experience is far too common in our industry. It makes organizations less competitive than they could be. Fortunately, there is a better way.

Socratic Development

Test-first programming and merciless refactoring, two of the many excellent Extreme Programming practices, dramatically improved the way I build software. I found that these two practices have helped me and the organizations I've worked for spend less time over-engineering and under-engineering, and more time designing just what we need: well-built systems, produced on time.

Test-first programming enables the efficient evolution of working code by turning programming into what Kent Beck once likened to a Socratic dialogue: Write test code to ask your system a question, write system code to respond to the question and keep the dialogue going until you've programmed what you need. This rhythm of programming put my head in a different place. Instead of thinking about a design that would work for every nuance of a system, test-first programming enabled me to make a primitive piece of behavior work correctly before evolving it to the next necessary level of sophistication.

Merciless refactoring is an integral part of this evolutionary design process. A refactoring is a "behavior-preserving transformation," or, as Martin Fowler defined it, "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior." [Fowler, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999)].

Merciless refactoring resembles the way Socrates continually helped dialogue participants improve their answers to his questions by weeding out inessentials, clarifying ambiguities and consolidating ideas. When you mercilessly refactor, you relentlessly poke and prod your code to remove duplication, clarify and simplify.

The trick to merciless refactoring is to not schedule time to make small design improvements, but to make them *whenever* your code needs them. The resulting quality of your code will enable you to sustain a healthy pace of development. Martin Fowler et al.'s book, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999), documents a rich catalog of refactorings, each of which identifies a common need for an improvement and the steps for making that improvement.

Why Refactor To Patterns?

On various projects, I've observed what and how my colleagues and I refactor. While we use many of the refactorings described in Fowler's book, we also find places where patterns can help us improve our designs. At such times, we refactor to patterns, being careful not to produce overly flexible or unnecessarily sophisticated solutions.

When I explored the motivation for refactoring to patterns, I found that it was identical to the motivation for implementing non-patterns-based refactorings: to reduce or remove duplication, simplify the unsimple and make our code better at communicating its intention.

However, the motivation for refactoring to patterns is not the primary motivation for using patterns that is documented in the patterns literature. For example, let's look at the documented Intent and Applicability of the Decorator pattern and then examine Erich Gamma and Kent Beck's motivation for refactoring to Decorator in their excellent, patterns-dense testing framework, JUnit.

Decorator's Intent [*Design Patterns*, page 175]:
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator's Applicability (GoF, page 177):

- To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- For responsibilities that can be withdrawn.
- When extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and could produce an explosion of subclasses to support every combination, or a class definition may be hidden or otherwise unavailable for subclassing.

Motivation for Refactoring to Decorator in JUnit
Erich remembered the following reason for refactoring to Decorator:

> "Someone added TestSetup support as a subclass of TestSuite, and once we added RepeatedTestCase and ActiveTestCase, we saw that we could reduce code duplication by introducing the TestSetup , Decorator." [private email]

Can you see how the motivation for refactoring to Decorator (reducing code duplication) had very little connection with Decorator's Intent or Applicability (a dynamic alternative to subclassing)? I noticed similar disconnects when I looked at motivations for refactorings to other patterns. Consider these examples:

Pattern	Intent (GoF)	Refactoring Motivations
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations.	Simplify code Remove duplication Reduce creation errors
Factory Method	Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses.	Remove duplication Communicate intent
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.	Remove duplication

Based on these observations, I began to document a catalog of refactorings to patterns to illustrate when it makes sense to make design improvements with patterns. For this work, it's essential to show refactorings from real-world projects in order to accurately describe the kinds of forces that lead to justifiable transformations to a pattern.

My work on refactoring to patterns is a direct continuation of work that Martin Fowler began in his excellent catalog of refactorings, in which he included the following refactorings to patterns:

- Form Template Method (345)
- Introduce Null Object (260)
- Replace Constructor with Factory Method (304)
- Replace Type Code with State/Strategy (227)
- Duplicate Observed Data (189)

Fowler also noted the following:

> There is a natural relation between patterns and refactorings. Patterns are where you want to be; refactorings are ways to get there from somewhere else. Fowler, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999)

This idea agrees with the observation made by the four authors of the classic book, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994):

> Our design patterns capture many of the structures that result from refactoring. … Design patterns thus provide targets for your refactorings.

Evolutionary Design

Today, after having become quite familiar with patterns, the "structures that result from refactoring," I know that understanding good reasons to refactor to a pattern are more valuable than understanding the end result of a pattern or the nuances of implementing that end result.

If you'd like to become a better software designer, studying the evolution of great software designs will be more valuable than studying the great designs themselves. For it is in the evolution that the real wisdom lies. The structures that result from the evolution can help you, but without knowing why they were evolved into a design, you're more likely to misapply them or over-engineer with them on your next project.

To date, our software design literature has focused more on teaching great solutions than teaching evolutions to great solutions. We need to change that. As the great poet Goethe said, "That which thy fathers have bequeathed to thee, earn it anew if thou wouldst possess it." The refactoring literature is helping us reacquire a better understanding of good design solutions by revealing sensible evolutions to those solutions.

If we want to get the most out of patterns, we must do the same thing: See patterns in the context of refactorings, not just as reusable elements existing apart from the refactoring literature. This is perhaps my primary motivation for producing a catalog of refactorings to patterns.

By learning to evolve your designs, you can become a better software designer and reduce the amount of work you over- or under-engineer. Test-first programming and merciless refactoring

are the key practices of evolutionary design. Instill refactoring to patterns in your knowledge of refactorings and you'll find yourself even better equipped to evolve great designs.