Chapter 2

Refactoring

In this chapter I offer a few thoughts on what refactoring is and what you need to do to be good at it. This chapter is best read in accompaniment with the chapter "Principles in Refactoring" [F].

What Is Refactoring?

A **refactoring** is a "behavior-preserving transformation" or, as Martin Fowler defines 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" [F, 53].

The **process of refactoring** involves the removal of duplication, the simplification of complex logic, and the clarification of unclear code. When you refactor, you relentlessly poke and prod your code to improve its design. Such improvements may involve something as small as changing a variable name or as large as unifying two hierarchies.

To refactor safely, you must either manually test that your changes didn't break anything or run automated tests. You will have more courage to refactor and be more willing to try experimental designs if you can quickly run automated tests to confirm that your code still works.

Refactoring in small steps helps prevent the introduction of defects. Most refactorings take seconds or minutes to perform. Some large refactorings can require a sustained effort for days, weeks, or months until a transformation has been completed. Even such large refactorings are implemented in small steps.

It's best to refactor continuously, rather than in phases. When you see code that needs improvement, improve it. On the other hand, if your manager needs you to finish a feature before a demo that just got scheduled for tomorrow, finish the feature and refactor later. Business is well served by continuous refactoring, yet the practice of refactoring must coexist harmoniously with business priorities.

What Motivates Us to Refactor?

While we refactor code for many reasons, the following motivations are among the most common.

- *Make it easier to add new code.*
  When we add a new feature to a system, we have a choice: we can quickly program the feature without regard to how well it fits with an existing design, or we can modify the existing design so it can easily and gracefully accommodate the new feature. If we go with the former approach, we incur design debt (see *Design Debt*, *15*), which can be paid down later by refactoring. If we go with the latter approach, we analyze what will need to change to best accommodate the new feature and then make whatever changes are necessary. Neither approach is better than the other. If you have little time, it may make more sense to quickly add the feature and refactor later. If you have more time or you perceive that you'll go faster by paving the way for the feature prior to programming it, by all means refactor before adding the feature.

- *Improve the design of existing code.*
  By continuously improving the design of code, we make it easier and easier to work with. This is in sharp contrast to what typically happens: little refactoring and a great deal of attention paid to expediently adding new features. Continuous refactoring involves constantly sniffing for coding smells (see Chapter 4, 37) and removing smells immediately after (or soon after) finding them. If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code. You may even enjoy your job more.

- *Gain a better understanding of code.*
  Sometimes we look at code and have no idea what it does or how it works. Even if someone could stand next to us and explain the code, the next person to look at it could also be totally confused. Is it best to write a comment for such code? No. If the code isn't clear, it's an odor that needs to be removed by refactoring, not by deodorizing the code with a comment.

  When we refactor such code, it is usually best to do so in the presence of someone who fully understands the code. If that person isn't available, see if he or she can help explain the code by e-mail, chat, or phone. Failing that, refactor only what you truly understand. In the end, your efforts will make it easier for everyone to understand the code.

- *Make coding less annoying.*
  I've often wondered what propels me to refactor code. Sure, I can say that I refactor to remove duplication, to simplify or clarify the code. But what actually propels me to refactor? Emotions. I often refactor simply to make code less annoying to work with.

  For example, I once joined a project that had some significant design debt. In particular, there was one enormous class with way too many responsibilities. Because much of what we did involved changing this enormous class, every time we checked in code (which was often, since we practiced continuous integration), we would have to deal with a complex merge involving the enormous class. As a result, everyone took longer than necessary to integrate code. This was very annoying. So another programmer and I set off on a three-week odyssey to break apart the enormous class into smaller classes. It was hard work that just had to be done. When we finished this work, integrating code took far less time and the overall programming experience was much more pleasant.

Many Eyes

When the Declaration of Independence was still a draft, Benjamin Franklin, sitting beside Thomas Jefferson, revised Jefferson's wording of "We hold these truths to be sacred and undeniable" to the now-famous phrase, "We hold these truths to be self-evident." According to biographer Walter Isaacson, Jefferson was distraught by Franklin's edits. So Franklin, aware of his friend's state, sought to console Jefferson by telling him the tale of his friend John Thompson.

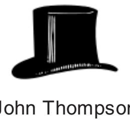
John had just started out in the hat-making business and wanted a sign for his shop. He composed his sign like so:

John Thompson, hatter, makes
and sells hats for ready money.

Before using the new sign, John decided to show it to some friends to seek their feedback. The first friend thought that the word "hatter" was repetitive and unnecessary because it was followed by the words "makes . . . hats," which showed that John was a hatter. The word "hatter" was struck out. The next friend observed that the word "makes" could be removed because his customers would not care who made the hats. So "makes" was struck out. A third friend said he thought the words "for ready money" were useless, as it was not the custom to sell hats on credit. People were expected to purchase hats with money. So those words were omitted.

The sign now read, "John Thompson sells hats."

"Sells hats!" said his next friend. "Why, nobody will expect you to give them away. What then is the use of that word?" "Sells" was stricken. At this point there was no use for the word "hats" since a picture of one was painted on the sign. So the sign was ultimately reduced to:



In his book *Simple and Direct*, Jacques Barzun explains that all good writing is based upon revision [Barzun, 227]. Revision, he points out, means to re-see. John Thompson's sign was gradually revised by his friends, who helped him remove duplicate words, simplify his language, and clarify his intent. Jefferson's words were revised by Franklin, who saw a simpler, better way to express Jefferson's intent. In both cases, having many eyes revise one individual's work helped produce dramatic improvements.

The same is true of code. To get the best refactoring results, you'll want the help of many eyes. This is one reason why extreme programming suggests the practices of pair-programming and collective code ownership [Beck, XP].

Human-Readable Code

Every now and then I run across a piece of code that so impresses me, I spend the next several months and years telling people about it. Such was the case when I studied a piece of code written by Ward Cunningham. If you don't know Ward, you may know one of his many excellent innovations. Ward pioneered Class-Responsibility-Collaboration (CRC) cards, the Wiki Web (a simple, fast, read/write Web site), extreme programming (XP), and the FIT testing framework (*http://fit.c2.com*).

The code I was studying came from a fictional payroll system that was meant for use in a refactoring workshop. As one of the workshop instructors, I needed to study this code prior to teaching. I began by looking over the test code. The first test method I studied checked a payroll amount based on a date. What immediately struck my eye was the date. The code said:

```
november(20, 2005)
```

This code called the following method:

```
public void Date november(int day, int year)
```

I was surprised and delighted. Even in a piece of test code, Ward had taken the trouble to produce a thoroughly human-readable method. If he had not cared to produce code that was simple and easy to understand, he could have written this:

```
java.util.Calendar c = java.util.Calendar.getInstance();
c.set(2005, java.util.Calendar.NOVEMBER, 20);
c.getTime();
```

While the above code produces the same date, it doesn't do what Ward's `november()` method does, which:

- Reads like spoken language
- Separates important code from distracting code

Now here's a very different story. It involves a method called `w44()`. I discovered the `w44()` function in a heap of Turbo Pascal spaghetti code that passed for a loan risk calculator for a large Wall Street bank. I spent the first three weeks of my professional programming career exploring this morass of code. I eventually figured out that 44 is the ASCII symbol for a comma, and "w" stands for "with." So `w44()` was the programmer's way of communicating that his routine returned a number, formatted as a string with commas. How intuitive! Either the programmer was shooting for big-time job security or he just didn't have a way with names.

Martin Fowler said it best:

> Any fool can write code that a computer can understand. Good programmers write code that humans can understand. [F, 15]

Keeping It Clean

Keeping code clean is a lot like keeping a room clean. Once your room becomes a mess, it becomes harder to clean. The worse the mess becomes, the less you want to clean it.

Suppose you do one giant cleanup of your room. Now what? If you want your room to remain clean, you can't leave things on the floor (like those socks) or allow books, magazines, glasses, or toys to pile up on tables. You must practice continuous hygiene.

Have you ever been in this situation? I have. If I can keep my room clean for several weeks, continuous hygiene starts to become a habit. Then I don't have to think so hard about whether I should throw my socks on the floor or deposit them in the laundry hamper. My habit propels me to put the socks in the hamper.

Unfortunately, new habits often run the risk of being compromised by old habits. One day you're too tired to pick your clothes up off the floor. Then several books get knocked off a shelf by a certain toddler. Before you know it, your room is a mess again.

To keep code clean, we must continuously remove duplication and simplify and clarify code. We must not tolerate messes in code, and we must not backslide into bad habits. Clean code leads to better design, which leads to faster development, which leads to happy customers and programmers. Keep your code clean.

Small Steps

Once upon a time a young, bright programmer was attending an intensive testing and refactoring workshop I was teaching. Everyone in this class was participating in a coding challenge that involved refactoring code with nearly all of the coding smells (see Chapter 4, *37*) described in this book and in *Refactoring* [F]. During this challenge, pairs of programmers must discover a smell, find a refactoring for the smell, and demonstrate the refactoring by programming it on a projector while the rest of the classes watches.

At about five minutes before noon, the class had been refactoring for nearly an hour. Since lunch had already been brought into the room, I asked if anyone had a small refactoring they'd like to complete before we broke for lunch. The young programmer raised his hand and said he had a small refactoring in mind. Without mentioning a specific smell or associated refactoring, this fellow described a rather large problem in the code and explained what he intended to do to fix it. A few students expressed their doubt that such a problem could be fixed in only five minutes, but the programmer insisted that he could complete the work, so we all agreed to let him and his pair partner try it.

Five minutes pass very quickly when you're refactoring something that is complicated.

The programmer and his partner found that after moving and altering some code, many of the unit tests were now failing. Failing unit tests show up as a big red bar in the unit-testing tool, which looks awfully big and red when it is being projected onto a large screen. As the programmers attempted to fix the broken unit tests, one by one people began to leave so they could eat lunch at a nearby table. Fifteen minutes later I took a break too. As I stood in the lunch line, I watched the programming action on the projector.

Twenty minutes into their work, the big red bar still hadn't turned green (which signals that all tests are passing). At that point the young programmer and his partner got up to get some food. Then they quickly returned to the computer. Many of us watched as one programmer attempted to eat his lunch with one hand while continuing to refactor with the other. Meanwhile, the minutes were ticking by.

At ten minutes to one (nearly fifty-five minutes after beginning their refactoring), the big red bar turned green. The refactoring was complete. As the class reassembled, I asked everyone what had gone wrong. The young programmer provided the answer: he had not taken small steps. By combining several refactorings into a single, large step, he thought he would go faster; but just the opposite was true. Each big step generated failures in the unit tests, which took a good deal of time to fix, not to mention that some of the fixes needed to be undone during later steps.

Many of us have had similar experiences—we take steps that are too large and then struggle for minutes, hours, or even days to get back to a green bar. The better I get at refactoring, the more I proceed by taking very small, safe steps. In fact, the green bar has become a gyroscope, a device that helps me stay on course. If the bar stays red for too long—more than a few minutes—I know that I'm not taking small enough steps. Then I backtrack and begin again. I nearly always find smaller, simpler steps I can take that will get me to my goal faster than if I'd taken bigger steps.

Design Debt

If you ask your manager to let you spend time continuously refactoring your code to "improve its design," what do you think the response will be? Probably "No" or an extended outburst of laughter or a harsh look. Keeping up with an endless stream of feature requests and defect reports is hard enough! Who has time for design improvements? What planet are you living on?

The technical language of refactoring doesn't communicate effectively with the vast majority of management. Instead, Ward Cunningham's financial metaphor of design debt [F, 66] works infinitely better. Design debt occurs when you don't consistently do three things.

1. Remove duplication.
2. Simplify your code.
3. Clarify you code's intent.

Few systems remain completely free of design debt. Wired as we are, humans just don't write perfect code the first time around. We naturally accumulate design debt. So the question becomes, "When do you pay it down?"

Due to ignorance or a commitment to "not fix what ain't broken," many programmers and teams spend little time paying down design debt. As a result, they create a Big Ball of Mud [Foote and Yoder]. In financial terms, if you don't pay off a debt, you incur late fees. If you don't pay your late fees, you incur higher late fees. The more you don't pay, the worse your fees and payments become. Compound interest kicks in, and as time goes on, getting out of debt becomes an impossible dream. So it is with design debt.

Discussing technical problems using the financial metaphor of design debt is a proven way to get through to management. I routinely take out a credit card and show it to managers when I'm speaking about design debt. I ask them, "How many months in a row do you not pay down your debt?" While some don't always pay off their debt in full each month, nearly all don't let debt accumulate for long. Discussions like this help managers acknowledge the wisdom of continuously paying down design debt.

Once management accepts the importance of continuous refactoring, the organization's entire way of building software can change. Suddenly, everyone from executives to managers to programmers agrees that going too fast hurts everyone. Programmers now have management's blessing to refactor. Over time, the small, hygienic acts of refactoring accumulate to make systems easier and easier to extend and maintain. When that happens, everyone benefits, including the makers, managers, and users of the software.

Evolving a New Architecture

There was once a company that had an older system, with the all-too-common problems of poor design, instability, and difficult maintenance. The company decided to refactor the system's architecture rather than rewrite everything from scratch.

Common code would be accessible from a new framework layer. Applications would use the framework layer for common services. This separation would allow framework programmers to gradually improve underlying framework code, with minimal impact on applications.

The company decided to form a framework team. Application teams would rely on the framework team for common services.

While this plan sounds reasonable, it's actually quite risky. If the framework team members lose touch with application needs, they can easily build the wrong code. If the application team members don't get what they need, they may bypass the framework to meet deadlines or slow down just to wait for what they need. Bypassing the framework is a return to the legacy architecture, while waiting for code is also a poor option.

Evolutionary design provides a better way. It suggests that you:

- Form one team
- Drive the framework from application needs
- Continuously improve applications and the framework by refactoring

With one team, the framework and the applications can't fall out of alignment. With the framework driven by real application needs, only valuable framework code gets produced. Continuous refactoring is essential to this process, for it's what keeps framework and application parts separate.

The company in this story decided to follow this evolutionary path, hiring coaches to train and guide them. Despite initial concerns about not having one team focus exclusively on framework development, the process resulted in continuous improvement in architecture, continuous delivery of high-quality applications, and evolution of a lean, general-purpose framework.

Refactoring is the essential ingredient here. It's what allows the team to effectively and efficiently evolve a new architecture.

Composite and Test-Driven Refactorings

Composite refactorings are high-level refactorings composed of low-level refactorings. Much of the work performed by low-level refactorings involves moving code around. For example, *Extract Method* [F] moves code to a new method, *Pull Up Method* [F] moves a method from a subclass to a superclass, *Extract*

Class [F] moves code to a new class, and *Move Method* [F] moves a method from one class to another.

Nearly all of the refactorings in this book are composite refactorings. You begin with a piece of code you want to change and then incrementally apply various low-level refactorings until a desired change has occurred. Between applying low-level refactorings, you run tests to confirm that modified code continues to behave as expected. Testing is thus an integral part of composite refactoring; if you don't run tests, you'll have a hard time applying low-level refactorings with confidence.

Testing also plays an altogether different role in refactoring; it can be used to rewrite and replace old code. A **test-driven refactoring** involves applying test-driven development to produce replacement code and then swap out old code for new code (while retaining and rerunning the old code's tests).

Composite refactorings are used far more frequently than test-driven refactorings because a good deal of refactoring work simply involves relocating existing code. When it isn't possible to improve a design this way, test-driven refactorings can help you produce a better design safely and effectively.

Substitute Algorithm [F] is a good example of a refactoring that is best implemented using test-driven refactorings. It essentially involves completely changing an existing algorithm for one that is simpler and clearer. How do you produce the new algorithm? You can't produce it by transforming the old algorithm into the new one because your logic for the new algorithm is different. You can program the new algorithm, substitute it for the old algorithm, and then see if the tests pass. But if the tests don't pass, you're likely to find yourself on a long date with a debugger. A better way to program the algorithm is to use test-driven development. This tends to produce simple code, and it also produces tests that later allow you or others to confidently apply low-level or composite refactorings.

Encapsulate Composite with Builder (96) is another example of a test-driven refactoring. In this case, you want to make it easier for clients to build a Composite by simplifying the build process. A Builder, which provides a simpler way of building a Composite, is where you'd like to take the design. Yet if that design is far different from the existing design, you will likely be unable to use low-level or composite refactorings to produce the new design. Once again, test-driven development provides an effective way to reimplement and replace old code.

The refactoring *Replace Implicit Tree with Composite (178)* is both a composite refactoring and a test-driven refactoring. Choosing how to implement this refactoring depends on the nature of the code you encounter. In general, if it's difficult to implement the *Extract Class* [F] refactoring on the code, the test-driven approach may be easier. *Replace Implicit Tree with Composite (178)* includes an example that uses test-driven refactoring.

Move Embellishment to Decorator (144) is not a test-driven refactoring; however, the example for this refactoring shows how test-driven refactoring is used to move behavior from outside a framework to inside the framework. This example involves moving code around, so you might think it would be more convenient to use composite refactorings to implement it. In fact, because the changes involve updating numerous classes, it turns out to be easier to use test-driven development to make the design transformation.

In your practice of refactoring, you're likely to use low-level and composite refactorings most of the time. Just remember that the "reimplement and replace" technique, as performed by using test-driven refactoring, is another useful way to refactor. While it tends to be most helpful when you're designing a new algorithm or mechanism, it may also provide an easier path than applying low-level or composite refactorings.

The Benefits of Composite Refactorings

The composite refactorings in this book, each of which targets a particular pattern, have some of the following benefits.

- *They describe an overall plan for a refactoring sequence.*
  The mechanics of a composite refactoring describe the sequence of low-level refactorings you can apply to improve a design in a particular way. Do you need such a sequence? If you already know the low-level refactorings, you can certainly apply them in whatever order you see fit.

  However, the refactoring sequences in this catalog may prove to be more safe, effective, or efficient in improving your design than your own refactoring sequences are. I once followed certain low-level refactorings to refactor to the State [DP] pattern. Then I learned a better, safer sequence. Then someone suggested improvements to that sequence. By the time I got to the fifth version of the sequence, I knew I had a much better way of refactoring to the State pattern, and it was far different from my initial approach.

- *They suggest nonobvious design directions.*
  Composite refactorings begin at a source and take you to a destination. The destination may or may not be obvious, given your source. Much depends on your familiarity with patterns, each of which defines a destination as well as the forces that suggest the need to go towards or to that

destination. The composite refactorings in this book make these nonobvious design directions clearer by describing real-world cases in which it made sense to move in the direction of a pattern.

- *They provide insights into implementing patterns.*
  Because there is no right way to implement a pattern (see *There Are Many Ways to Implement a Pattern*, 26), it's useful to consider alternative pattern implementations. This is particularly true of patterns that solve different kinds of design problems. For example, this book contains three different refactorings to Composite [DP] and three different ways to refactor to Visitor [DP]. How you refactor to these patterns and others will vary depending on the initial problem you face. In recognition of that, the refactoring sequences in this book vary in how they ultimately implement a pattern.

Refactoring Tools

The early pioneers of refactoring tools—people like William Opdyke, Ralph Johnson, John Brant, and Don Roberts—envisioned a world in which we could look at code that needed a refactoring and simply ask a tool to perform the refactoring for us. In the mid-1990s, John and Don built such a tool for Smalltalk. Software development hasn't been the same since.

After the 1999 publication of *Refactoring* [F], Martin Fowler challenged tool vendors to produce automated refactoring tools for mainstream languages such as Java. These tool vendors responded, and before long, many programmers throughout the world could execute automated refactorings from their integrated development environments (IDEs). Over time, even die-hard users of programming editors began transitioning to IDEs, largely due to automated refactoring support.

As refactoring tools continue to implement low-level refactorings, like *Extract Method* [F], *Extract Class* [F], and *Pull Up Method* [F], it becomes easier to transform designs by executing sequences of automated refactorings. This has important implications for pattern-directed refactorings because the mechanics for these refactorings are composed of low-level refactorings. When tool vendors automate the majority of low-level refactorings, they will automatically create support for the refactorings in this book.

Using automated refactorings to move towards, to, or away from a pattern is completely different from using a tool to generate pattern code. In general, I've found that pattern code generators provide an excellent way to over-engineer your code. In addition, they generate code that doesn't contain tests, which further limits your ability to refactor as and when needed. By contrast, refactoring lets you discover small design improvements you can safely make to go towards, to, or away from a pattern implementation.

Because refactoring is the act of performing behavior-preserving transformations, you might think that you would not need to run test code after you perform an automated refactoring. Well, you do, much of the time. You may have complete confidence in your automated refactoring tool for some refactorings, while you may not completely trust it for other refactorings. Many automated refactorings prompt you to make choices; if you make the wrong choices, you can easily cause your test code to stop running correctly (which is another way to say that the automated refactoring you performed did add or remove some behavior). In general, it's useful to run all of your tests after refactoring to confirm that the code is behaving as you expect.

If you lack tests, can you trust automated refactoring tools to preserve behavior in your code and not introduce unwanted behavior? You may be able to trust many of the refactorings, while others, which may be just out of production, are less stable or trustworthy. In general, if you lack test coverage for your code, you really won't have much success with refactoring, unless the tools become substantially more intelligent.

Advances in automated refactorings can impact what steps you follow in the mechanics of a refactoring. For example, a recent automation of *Extract Method* [F] is so smart that if you extract a chunk of code from one method and that same chunk of code exists in another method, both chunks of code will be replaced by a call to the newly extracted method. That capability may change how you approach the mechanics from a refactoring, given that some of the work from multiple steps may be automated for you.

What is the future of refactoring tools? I hope that we see more automated support for low-level refactorings, tools that suggest which refactorings could help improve certain pieces of code, and tools that allow you to explore how your design would look if several refactorings were applied together.