Refactoring

Change and decay in all around I see . . .

> H. F. Lyte, “Abide With Me”

As a program evolves, it will become necessary to rethink earlier decisions and rework portions of the code. This process is perfectly natural. Code needs to evolve; it’s not a static thing.

Unfortunately, the most common metaphor for software development is building construction (Bertrand Meyer [Mey97b] uses the term “Software Construction”). But using construction as the guiding metaphor implies these steps:

1. An architect draws up blueprints.

2. Contractors dig the foundation, build the superstructure, wire and plumb, and apply finishing touches.

3. The tenants move in and live happily ever after, calling building maintenance to fix any problems.

Well, software doesn’t quite work that way. Rather than construction, software is more like gardening—it is more organic than concrete. You plant many things in a garden according to an initial plan and conditions. Some thrive, others are destined to end up as compost. You may move plantings relative to each other to take advantage of the interplay of light and shadow, wind and rain. Overgrown plants get split or pruned, and colors that clash may get moved to more aesthetically pleasing locations. You pull weeds, and you fertilize plantings that are in need of some extra help. You constantly monitor the health of the garden, and make adjustments (to the soil, the plants, the layout) as needed.

Business people are comfortable with the metaphor of building construction: it is more scientific than gardening, it’s repeatable, there’s a rigid reporting hierarchy for management, and so on. But we’re not building skyscrapers—we aren’t as constrained by the boundaries of physics and the real world.

The gardening metaphor is much closer to the realities of software development. Perhaps a certain routine has grown too large, or is trying
to accomplish too much—it needs to be split into two. Things that don’t work out as planned need to be weeded or pruned.

Rewriting, reworking, and re-architecting code is collectively known as refactoring.

When Should You Refactor?

When you come across a stumbling block because the code doesn’t quite fit anymore, or you notice two things that should really be merged, or anything else at all strikes you as being “wrong,” don’t hesitate to change it. There’s no time like the present. Any number of things may cause code to qualify for refactoring:

- **Duplication.** You’ve discovered a violation of the DRY principle (*The Evils of Duplication*, page 26).

- **Nonorthogonal design.** You’ve discovered some code or design that could be made more orthogonal (*Orthogonality*, page 34).

- **Outdated knowledge.** Things change, requirements drift, and your knowledge of the problem increases. Code needs to keep up.

- **Performance.** You need to move functionality from one area of the system to another to improve performance.

Refactoring your code—moving functionality around and updating earlier decisions—is really an exercise in *pain management*. Let’s face it, changing source code around can be pretty painful: it was almost working, and now it’s really torn up. Many developers are reluctant to start ripping up code just because it isn’t quite right.

Real-World Complications

So you go to your boss or client and say, “This code works, but I need another week to refactor it.”

We can’t print their reply.

Time pressure is often used as an excuse for not refactoring. But this excuse just doesn’t hold up: fail to refactor now, and there’ll be a far greater time investment to fix the problem down the road—when there are more dependencies to reckon with. Will there be more time available then? Not in our experience.
You might want to explain this principle to the boss by using a medical analogy: think of the code that needs refactoring as a "growth." Removing it requires invasive surgery. You can go in now, and take it out while it is still small. Or, you could wait while it grows and spreads—but removing it then will be both more expensive and more dangerous. Wait even longer, and you may lose the patient entirely.

Tip 47

Refactor Early, Refactor Often

Keep track of the things that need to be refactored. If you can't refactor something immediately, make sure that it gets placed on the schedule. Make sure that users of the affected code know that it is scheduled to be refactored and how this might affect them.

How Do You Refactor?

Refactoring started out in the Smalltalk community, and, along with other trends (such as design patterns), has started to gain a wider audience. But as a topic it is still fairly new; there isn't much published on it. The first major book on refactoring ([FBB+99], and also [URL 47]) is being published around the same time as this book.

At its heart, refactoring is redesign. Anything that you or others on your team designed can be redesigned in light of new facts, deeper understandings, changing requirements, and so on. But if you proceed to rip up vast quantities of code with wild abandon, you may find yourself in a worse position than when you started.

Clearly, refactoring is an activity that needs to be undertaken slowly, deliberately, and carefully. Martin Fowler offers the following simple tips on how to refactor without doing more harm than good (see the box on page 30 in [FS97]):

1. Don't try to refactor and add functionality at the same time.

2. Make sure you have good tests before you begin refactoring. Run the tests as often as possible. That way you will know quickly if your changes have broken anything.
Automatic Refactoring

Historically, Smalltalk users have always enjoyed a class browser as part of the IDE. Not to be confused with Web browsers, class browsers let users navigate through and examine class hierarchies and methods.

Typically, class browsers allow you to edit code, create new methods and classes, and so on. The next variation on this idea is the refactoring browser.

A refactoring browser can semiautomatically perform common refactoring operations for you: splitting up a long routine into smaller ones, automatically propagating changes to method and variable names, drag and drop to assist you in moving code, and so on.

As we write this book, this technology has yet to appear outside of the Smalltalk world, but this is likely to change at the same speed that Java changes—rapidly. In the meantime, the pioneering Smalltalk refactoring browser can be found online at [URL 20].

3. Take short, deliberate steps: move a field from one class to another, fuse two similar methods into a superclass. Refactoring often involves making many localized changes that result in a larger-scale change. If you keep your steps small, and test after each step, you will avoid prolonged debugging.

We’ll talk more about testing at this level in Code That’s Easy to Test, page 189, and larger-scale testing in Ruthless Testing, page 237, but Mr. Fowler’s point of maintaining good regression tests is the key to refactoring with confidence.

It can also be helpful to make sure that drastic changes to a module—such as altering its interface or its functionality in an incompatible manner—break the build. That is, old clients of this code should fail to compile. You can then quickly find the old clients and make the necessary changes to bring them up to date.

So next time you see a piece of code that isn’t quite as it should be, fix both it and everything that depends on it. Manage the pain: if it hurts now, but is going to hurt even more later, you might as well get it over
with. Remember the lessons of *Software Entropy*, page 4: don’t live with broken windows.

**Related sections include:**
- *The Cat Ate My Source Code*, page 2
- *Software Entropy*, page 4
- *Stone Soup and Boiled Frogs*, page 7
- *The Evils of Duplication*, page 26
- *Orthogonality*, page 34
- *Programming by Coincidence*, page 172
- *Code That’s Easy to Test*, page 189
- *Ruthless Testing*, page 237

### Exercises

**38.** The following code has obviously been updated several times over the years, but the changes haven’t improved its structure. Refactor it.

```java
if (state == TEXAS) {
    rate = TX_RATE;
    amt = base * TX_RATE;
    calc = 2*basis(amt) + extra(amt)*1.05;
} 
else if ((state == OHIO) || (state == MAINE)) {
    rate = (state == OHIO) ? OH_RATE : ME_RATE;
    amt = base * rate;
    calc = 2*basis(amt) + extra(amt)*1.05;
    if (state == OHIO)
        points = 2;
} 
else {
    rate = 1;
    amt = base;
    calc = 2*basis(amt) + extra(amt)*1.05;
}
```

**39.** The following Java class needs to support a few more shapes. Refactor the class to prepare it for the additions.

```java
public class Shape {
    public static final int SQUARE = 1;
    public static final int CIRCLE = 2;
    public static final int RIGHT_TRIANGLE = 3;
    private int shapeType;
    private double size;
    public Shape(int shapeType, double size) {
        this.shapeType = shapeType;
        this.size = size;
    }
    // ... other methods ...
```