From

The Essence of
Objecte Oriented
Programming with Java
and UML

Bruce E. Wampler

Addison-Wesley
(Boston, 2002)

CHAPTER 8
Refactoring

Up to this point, we've focused on using object orientation to design and develop new programs. Understanding objects and using object-oriented techniques is one of the best ways to develop programs that are easy to understand, easy to write, and perhaps most important, easy to modify and maintain.

The fact is, most programming involves maintaining and modifying existing code. If the existing code is not written in an OO language, then using object orientation probably won't help much with changing it. If the existing code is written in an OO language, there is hope. Unfortunately, as we've noted before, just because a program uses Java or C++ doesn't mean it uses OO techniques. And even if the program started out with a decent OO design, if it has been maintained and modified over time, it is likely to have lost some of its initial elegance.

So what are you going to do? One of the most recent object-oriented techniques to be formalized and developed into an essential programming tool is called refactoring. While programmers have always spent some time cleaning up their code, refactoring takes this a bit further. Refactoring is a disciplined approach to improving the design of existing code. With it, the overall design and structure of an existing program is improved while its observable functionality remains unchanged. Once its design has been improved, it is easier to maintain.

Software maintenance usually has one of two goals. The first is to fix bugs; the second is to add features. While the goal of refactoring is neither of these, it can greatly improve how easy it is to do either. At first, it may seem a waste to refactor code without adding new functionality. However, trying to modify a poorly constructed program can take far more time and effort than first refactoring and then modifying. Refactoring can reveal flaws in the structure of the existing code that are the underlying causes of bugs or incorrect behavior.

Ward Cunningham and Kent Beck were two of the first software experts to recognize the importance of refactoring and to help develop it into a formal technique. The principal refactoring resource is *Refactoring: Improving the Design of Existing Code*, by Martin Fowler. Although refactoring can be useful for almost any kind of object-oriented programming, it is an essential part of the Extreme Programming (XP) methodology (see Chapter 9).

What Is Refactoring?

Refactoring should be considered a basic principle of programming and does not require any special methodology. Over time, as code is changed, it tends to deteriorate. Changes are often made on the fly, under time pressure, without regard for the overall structure of the code. This can lead to code entropy. Refactoring helps to undo code entropy.

The Basic Refactoring Process

While the basic refactoring process is not that complicated, experience with programming helps (especially object-oriented programming). The main goal of refactoring is to improve the overall design and structure of an existing program, without changing its observable behavior. This means that you don't refactor and add functionality at the same time. Once the refactoring has been done, it is then easier to add functionality.

The first step in refactoring is to understand the existing code. As part of the process of reading the code, you will almost certainly find problems with it. Refactoring isn't just about finding problems and making small improvements. It is a well-defined and structured technique for improving the code. Each individual refactoring may make only a small difference, but the cumulative effect of applying many refactorings can result in greatly improved overall quality, readability, and design of the code.

Refactorings

The developers of refactoring have identified a list and descriptions of known refactorings that can help improve code. Many of these refactorings are listed and described in Fowler's book, and more can be found on the refactoring Web site, `www.refactoring.com`. As you become more experienced, many specific refactorings will become familiar, and you will start to recognize cases in code that can benefit from the technique.

Reduce Risk of Change

Any time you change code, it is risky. You can introduce new bugs. You can change the behavior of the program. You can break things. By using disciplined refactoring, you can reduce the risks involved in making changes. This is a major difference between a simple code cleanup and the formal refactoring process. By carefully following the refactoring process, you reduce the risk of making changes and, at the same time, improve its design and make it easier to change in the future.

Don't Change Functionality

One of the first rules of refactoring is: Do not change the functionality of the existing code. By functionality, in this context, we mean the outside observed behavior of the code. The program should behave exactly the same before and after the refactoring. If the behavior changes, it will be impossible to know for sure that the refactoring hasn't broken other things as well. If you need to change the behavior, do that as a separate step. Improve the code with refactoring, then make the change.

One Thing at a Time

To be sure you don't change behavior, it is important to apply only one refactoring at a time. While going over the code, you may often find several things that can benefit from refactoring. But to reduce the risk of making changes, refactoring requires that you make only one change at a time.

Test Each Step

Perhaps the most important principle of refactoring is to thoroughly test the program after each refactoring. This is how you reduce the risk of change—by ensuring that you haven't changed functionality or broken other parts of the program. Besides the identification of a large set of known refactorings, the combination of preserving functionality, of making only one change at a time, and of testing after each step is one of the most important contributions of refactoring.

Summary

The following list summarizes the refactoring process.

1. Review code to identify refactorings.
2. Apply only one refactoring at a time, without changing functionality.
3. Test the refactoring.
4. Repeat to find more refactorings.

When Do You Refactor?

To use refactoring effectively, it is important to know just when to refactor. You don't always need to refactor working code. There are some guidelines for deciding when to refactor.

First, when you plan to add some functionality to a program, be prepared to refactor. As we've noted before, one important benefit of object-oriented program design is that the code is easier to maintain and modify. So when it comes time to add new functionality to a program, it is important that code be as well-designed as possible. This is precisely the goal of refactoring. Refactoring should be applied to the code until its design has improved enough to make it easy to modify. Then, after the refactoring, new functionality should be much easier to add.

Refactoring is also useful when you need to find bugs. Part of finding a bug is understanding a program. The fact that a program has bugs often means that the code isn't clear enough to spot them in the first place. Refactoring while you are going over code to hunt for bugs improves the quality of the code and even reduces the number of bugs.

One important part of almost every project is the code review. For code reviews with just two or three programmers (probably the most productive kind), refactorings can be suggested and applied as the group goes over the code and gets better at understanding its design. In fact, the pair programming of Extreme Programming can be considered pair code reviews, and refactoring is an important part of XP.

Code Smells

Kent Beck and Martin Fowler have also developed a list of what they call "code smells" to help determine when to refactor. If you sniff these out in existing code, refactoring is in order. Here's a brief list of some of the smells they've identified.

- **Duplicate code** Duplicate code means you need to extract some methods.
- **Long method** Too long is hard to understand. Extract methods.
- **Large class** A class that does too much needs to be split.
- **Long parameter list** A long list makes it hard to read. Consider passing objects.
- **Divergent change** This is code degradation as a result of too many chaotic changes to a class.

- **Shotgun surgery** Shotgun surgery means too many undisciplined changes to classes and attributes.
- **Feature envy** One class is interested in too many details of another class.
- **Data clumps** Data that is used together everywhere should be in a class of its own.
- **Primitive obsession** A program can use too many primitive data types that should really be part of a class.
- **Switch statements** Switch statements can mean you are not using polymorphism effectively.
- **Parallel inheritance hierarchies** Repeating class definitions in parallel classes is more duplication to eliminate.
- **Lazy class** A class should do enough to pay its own way or be eliminated.
- **Speculative generality** Designing for future flexibility before it is needed can increase complexity unnecessarily.
- **Message chain** Too many messages in a chain are hard to follow.
- **Middleman** Sometimes, it is better to work with an object directly.
- **Inappropriate intimacy** Classes shouldn't need to know too much about each other.
- **Incomplete library class** Sometimes, you can't get it all and need to do some yourself.
- **Data class** Classes need something to do.
- **Refused bequest** Subclasses should use most of what their parents give them.
- **Comments** Could a comment be eliminated by providing a better name for a method or variable?

When Not to Refactor

Knowing when not to refactor is also important. One of the main reasons not to refactor is when the code is so bad that it needs to be rewritten from scratch. Eventually, code can become so outdated, so difficult to understand, or so buggy that it would be more cost-effective to start over than to try to fix it or add new features.

This decision can also apply to code that is written in a non-object-oriented language. Most refactorings apply to object-oriented languages. Obviously, these refactorings have limited use for non-OO languages. It may be time to rewrite the program using an OO language.

Some Refactorings

The identification of refactorings gives you a catalog of things to look for in existing code. Each refactoring has been given a name, much like the design patterns we discussed in Chapter 7. There are many more refactorings than there are design patterns, and most refactorings are much simpler and easier to understand.

In this section, we will go over a few refactorings. It is not the goal here to make you into a master of refactoring, but to give you an idea of some of the specifics. Just as design patterns belong in every good programmer's toolbox, refactoring has its place there as well.

Refactoring Categories

More than 70 specific refactorings have been identified in Fowler's book, and many more are identified on the refactoring Web site, with more added all the time. The refactorings have been organized into the following categories. Specific individual refactorings are shown in *italics*. This summary mentions only a fraction of the total number.

Composing Methods

One common problem comes from code that has methods that are too long. The Composing Methods group of refactorings is intended to help reduce the size of methods and to help improve the readability of the code by replacing sequences of code with calls to methods that are built from the original code. Refactorings in this category include *Extract Method*, *Inline Method*, and *Replace Temp with Query*.

Moving Features Between Objects

During object design, it is important to decide where to place various responsibilities. Sometimes, responsibility can be placed in the wrong class. Some classes end up with too many responsibilities. Such refactorings as *Move Method*, *Move Field*, and *Extract Class* can be used to help put responsibilities where they belong.

Organizing Data

Sometimes objects can be used instead of simple data items. Refactorings such as *Replace Data Value with Object* or *Replace Array with Object* can make working with a class easier. They can also clarify what the data item is being used for and make it easier to work with.

Simplifying Conditional Expressions

Conditional expressions can be some of the most complicated and confusing parts of any program to understand. Such refactorings as *Decompose Conditional* or *Consolidate Duplicate Conditional Fragments* can be used to simplify code.

Making Method Calls Simpler

Defining the interface to a class can be difficult. Just what the methods are named and how they are called can lead to confusion or simplicity. Refactorings such as *Rename Method*, *Add Parameter*, and others from this category can help improve the interface to a class.

Dealing with Generalization

One guideline we discussed for good object design was moving methods as high up the inheritance hierarchy as possible. Getting methods and subclasses in just the right place is the goal of this category of refactorings. Some of them include *Pull Up Method*, *Push Down Method*, *Extract Subclass*, and *Extract Superclass*. All are meant to help refine the inheritance hierarchy.

Some Specific Refactorings

In Fowler's book, each individual refactoring description includes the name of the refactoring, a short description of the problem, a short description of the solution, a more detailed discussion of the motivation for using the refactoring, and a discussion of the mechanics for carrying out the refactoring.

Currently, refactoring is mostly a manual operation. It is up to the programmer to identify specific refactorings and then actually rewrite the code. As this book is written, a few refactoring software tools that can help with some of the mechanical aspects of the different refactorings are emerging. The refactoring Web site, `www.refactoring.com`, keeps information about the latest refactoring tools.

The following descriptions of some refactorings are neither complete nor intended to imply that they are the most important refactorings. They were chosen simply to illustrate some typical refactorings from each category.

Extract Method

Extract Method is used when you have a code fragment that has meaning when taken by itself. That code is extracted and turned into a method whose name clearly explains the purpose of the method. Short, well-named methods can make code clearer. A well-named method can eliminate the need for a comment. Sometimes, you can even find duplicated code that belongs in a method.

Replace Temp with Query

Replace Temp with Query is used when you find a temporary variable used to hold the results of an expression. By extracting the expression into a method and then replacing all references to the temp with the method call, the meaning can be clearer, and you can reuse the method in other places.

Move Method

If you find a method is being used more often by another class than the one where it is defined, you can use Move Method to move the method to the other class. You remove the original definition, and invoke the new method from the original class.

Extract Class

If you find you have one class doing work that should really be done by two, use Extract Class. You can create a new class and move the relevant methods and attributes from the old class into the new one.

Decompose Conditional

One way to improve complicated conditional statements is to extract the code that makes up the *then* and *else* parts into methods with meaningful names. This reduces the complexity, makes the statements more meaningful to read, and often results in methods that can be reused.

Rename Method

Rename Method is one of the simplest refactorings, yet it can lead to code that is much easier to understand. If the name of a method (or even a variable) does not indicate its purpose, then it should be renamed so that it is meaningful.

Pull Up Method

If you find methods in different subclasses that have identical results, you can use Pull Up Method to move them to the superclass. Eliminating this duplicate behavior makes the code easier to maintain and understand.

Extract Subclass

If a class has methods that are used by only some of the instances of that class, those instances should have their own subclass. Extract Subclass is used to extract those features into a new subclass.

Chapter Summary

- Refactoring is a programming tool that can improve the design of existing code.
- A major goal of refactoring is to reduce the risk of change by providing a well-defined approach to improving code.
- Several things indicate you need to refactor, including "code smells."
- There are many refactorings in several categories.

Resources

Refactoring: Improving the Design of Existing Code, Martin Fowler, Addison-Wesley, 1999, ISBN 0-201-48567-2.

Refactoring Web site: `www.refactoring.com`.