Chapter 2

What Is Refactoring?

"Refactoring: Improving the design of existing code."
—Martin Fowler

Refactoring is the process of improving the design of code without affecting its external behavior. We refactor so that our code is kept as simple as possible, ready for any change that comes along.

See Martin Fowler's book *Refactoring* (1999) for a full discussion of the subject.

In this chapter, we'll start with some realistic code and work our way through several refactorings. Our code will become more clear, better designed, and of higher quality.

What do we need for refactoring?

- Our original code
- Unit tests (to ensure we haven't unwittingly changed the code's external behavior)
- A way to identify things to improve

- ✧ A set of refactorings we know how to apply
- ✧ A process to guide us

Original Code

The following code was designed to generate a Web page, by substituting strings for %CODE% and %ALTCODE% in a template read from a file. The code works, but a performance test showed that it is too slow; the prime reason is that it creates too many temporary strings. Once our attention is called to it, we see that this code needs a good cleanup as well.

```
import java.io.*;
import java.util.*;

/** Replace %CODE% with requested id, and
**  replace %ALTCODE% w/"dashed" version of id.
*/
public class CodeReplacer {
    public final String TEMPLATE_DIR = "templatedir";
    String sourceTemplate;
    String code;
    String altcode;

  /**
  * @param reqId java.lang.String
  * @param oStream java.io.OutputStream
  * @exception java.io.IOException The exception description.
  */
  public void substitute(String reqId, PrintWriter out)
                                  throws IOException
  {
           // Read in the template file
    String templateDir = System.getProperty(TEMPLATE_DIR, "");
    StringBuffer sb = new StringBuffer("");
    try {
      FileReader fr = new FileReader(templateDir +
                                     "template.html");
      BufferedReader br = new BufferedReader(fr);
```

```
      String line;
      while(((line=br.readLine())!="") && line!=null)
         sb = new StringBuffer(sb + line + "\n");
      br.close();
      fr.close();
    } catch (Exception e) {
    }
    sourceTemplate = new String(sb);

    try {
      String template = new String(sourceTemplate);
          // Substitute for %CODE%
      int templateSplitBegin = template.indexOf("%CODE%");
      int templateSplitEnd = templateSplitBegin + 6;
      String templatePartOne = new String(
               template.substring(0, templateSplitBegin));

      String templatePartTwo = new String(
              template.substring(templateSplitEnd,
                                 template.length()));
      code = new String(reqId);
      template = new String(
            templatePartOne+code+templatePartTwo);

          // Substitute for %ALTCODE%
      templateSplitBegin = template.indexOf("%ALTCODE%");
      templateSplitEnd = templateSplitBegin + 9;
      templatePartOne = new String(
            template.substring(0, templateSplitBegin));
      templatePartTwo = new String(
            template.substring(templateSplitEnd,
                               template.length()));
      altcode = code.substring(0,5) + "-" +
                    code.substring(5,8);
      out.print(templatePartOne+altcode+templatePartTwo);
    } catch (Exception e) {
      System.out.println("Error in substitute()");
    }
    out.flush();
    out.close();
  }
}
```

Unit Tests

The first step in refactoring is to create unit tests that verify the basic functionality. If you're doing Extreme Programming (XP) with incremental, test-first programming, those tests exist already as a by-product of that process.

The following test requires a file named `template.html` that contains the text `"xxx%CODE%yyy%ALTCODE%zzz"`.

```
import java.io.*;
import junit.framework.*;

public class CodeReplacerTest extends TestCase {
  CodeReplacer replacer;

  public CodeReplacerTest(String testName){super(testName);}

  protected void setUp() {replacer = new CodeReplacer();}

  public void testTemplateLoadedProperly() {
    try {
      replacer.substitute("ignored",
                    new PrintWriter(new StringWriter()));
    } catch (Exception ex) {
      fail("No exception expected, but saw:" + ex);
    }

    assertEquals("xxx%CODE%yyy%ALTCODE%zzz\n",
                 replacer.sourceTemplate);
  }

  public void testSubstitution() {
    StringWriter stringOut = new StringWriter();
    PrintWriter testOut = new PrintWriter (stringOut);
    String trackingId = "01234567";

    try {
      replacer.substitute(trackingId, testOut);
    } catch (IOException ex) {
      fail ("testSubstitution exception - " + ex);
    }
```

```
    assertEquals("xxx01234567yyy01234-567zzz\n",
                 stringOut.toString());
  }
}
```

This code uses the JUnit unit-testing framework introduced in the previous chapter.

Code Smells

Martin Fowler and Kent Beck use the metaphor of "code smells" to describe what you sense when you look at code. Code smells tend to be a "bad sign," rather than an indication that something is necessarily wrong. You may have heard a similar idea described as "anti-patterns" (after Brown et al. 1988) or "Spidey-sense" (after Stan Lee's Spider-man, 1996).

What potential danger signs might you see in code?

- Classes that are too long
- Methods that are too long
- Switch statements (instead of polymorphism)
- "Struct" classes (getters and setters but not much functionality)
- Duplicate code
- Almost (but not *quite*) duplicate code
- Overdependence on primitive types (instead of introducing a more domain-specific type)
- Useless (or wrong!) comments
- Many more...

Some smells are obvious right away; you may not detect others until you're in the middle of refactoring.

Look at the original code above, and see what problems you can identify. (Don't restrict yourself to this list!)

A Catalog of Refactorings

About half of Martin Fowler's *Refactoring* book is devoted to a catalog of refactorings. Each of these is a relatively simple transformation; Fowler explains the mechanics of the change and provides examples (Figure 2.1). For another example, not in Fowler's book, see Figure 2.2.

Extract Method

Before

```
// Assume all are instance
// variables

void f() {
    ...
    // Compute score
    score = a * b + c;
    score -= discount;
}
```

After

```
void f() {
    ...
    computeScore();
}

void computeScore() {
    score = a * b + c;
    score -= discount;
}
```

FIGURE 2.1 Sample Refactoring
Source: Fowler, Martin. 1999. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley.

Replace String with StringBuffer

Before

```
String a, b, c;
  :
return a + b + c;
```

After

```
String a, b, c;
  :
StringBuffer sb = new
  StringBuffer(a);
sb.append(b);
sb.append(c);
return sb.toString();
```

FIGURE 2.2 Sample Refactoring

This refactoring lets us replace the easy-to-read String version with a potentially more efficient StringBuffer version (or vice versa: many refactorings are appropriate to use "backward"). Some compilers can apply this particular rule for us automatically; then we can keep the readability and let the compiler do the work. For the example below, we'll apply this refactoring ourselves.

If all compilers did this reorganization automatically, would we still want this refactoring? I believe we would. The first form is shorter, but relies on the overloading of "+" in Java. The second version uses the (otherwise) uniform operator "." to handle method calls. If we ever wanted to replace the data type (using something other than String/StringBuffer), we'd be better off starting from the StringBuffer version.

Process

Work in an environment that lets you alternate testing and changing your code.

Apply one refactoring, then run the unit tests. Repeat this process until your code expresses its intent clearly, simply, and without duplication. At first, it may feel awkward to run the tests so often, but it will speed you up to do so. (It takes a few seconds to run the tests, but it reassures you that several minutes worth of changes are OK.)

When are we done? When the code

1. Passes its tests (works)
2. Communicates everything it needs to communicate
3. Has no duplication
4. Has as few classes and methods as possible

These goals are in priority order: if duplication is required to communicate, the code is duplicated. The goals are often compressed to the phrase "once and only once" ("once" to work; "only once" to avoid duplication).

In *The Pragmatic Programmer*, Dave Thomas and Andy Hunt (2000) discuss a similar but more general rule known as the DRY principle ("Don't Repeat Yourself"): "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

Going to Work

When you considered the sample code, what smells did you find? Here is what I saw:

- Long class
- Long method
- Variables could be local to method
- Useless method comment
- Could we read template once, instead of each time?
- The code is tied to using the file system
- Questionable use of "!=" for string compare
- Use of StringBuffer without append
- Reallocating StringBuffer in loop
- The `close()` methods are not in `catch` or `finally` clause
- Inconsistent/unclear exception handling
- Lots of string addition
- Almost-duplicate code in handling `"%CODE%"` and `"%ALTCODE%"`
- Lots of extraneous `new String()`s
- Magic numbers (6 and 9) and symbols
- Lots of temporary variables

The worst smell is that long `substitute()` method, so use *Extract Method* to break it up. Many of the refactorings we use are available online in the catalog at *http://www.refactoring.com*.

First, pull out `readTemplate()` as a new method:

```
String readTemplate() {
  String templateDir = System.getProperty(TEMPLATE_DIR, "");
  StringBuffer sb = new StringBuffer("");
  try {
    FileReader fr=new FileReader(templateDir+"template.html");
    BufferedReader br = new BufferedReader(fr);
    String line;
    while(((line=br.readLine())!="")&&line!=null)
      sb = new StringBuffer(sb + line + "\n");
    br.close();
    fr.close();
  } catch (Exception e) {
  }
  sourceTemplate = new String(sb);
  return sourceTemplate;
}
```

Even though the change is so simple it could not possibly fail, *run the test!* (And of course, the first time I ran the test, it failed because I forgot to call my new function, highlighting the importance of the mechanics.)

Notice also that we're not immediately chopping the routine into three or four pieces all at once; we're working one step at a time. We'll develop a steady rhythm: change some code, run the test, change some code, run the test. We never go far without verifying what we've done. If we make a mistake, it must be in the last thing we did.

Let's get the template name in one place (via *Introduce Explaining Variable*), replacing `templateDir` with

```
String templateName = System.getProperty(TEMPLATE_DIR,"")
    + "template.html";
```

(*Run the test.*) Then eliminate the `fr` variable (*Inline Temp*):

```
BufferedReader br = null;
    ...
br = new BufferedReader(newFileReader(templateName));
```

(and drop `fr.close()`.) (*Run the test.*)

Now we're in a position to fix one of the bugs we noticed: the stream is not properly closed in case of errors.

```
try {
  ...
} catch (Exception ex) {
} finally {
  if (br != null) try {br.close();}
      catch (IOException ioe_ignored) {}
}
```

(*Run the test.*)

Next look at another potential problem, the `!=` string test. We verify (by looking around and asking around) that the template reader was not intended to stop at blank lines or anything like that, so this condition is meaningless.

```
String line = br.readLine();
while (line != null) {
    sb = new StringBuffer(sb + line + "\n");
    line = br.readLine();
}
```

(*Run the test.*) Martin Fowler (1999) points out that it is safer to keep bug-fixing and refactoring separate. He would create a new test case to demonstrate the bug, complete refactoring, and only then go back to fix it. In this small example, we'll just proceed with the fixed code.

Consider the assignment to `sb`. It is redundant: there's no reason to create a new StringBuffer each time, when we can just add to the

one we already have. Also, we can use `append()` to eliminate the string addition (*Replace String with StringBuffer*).

```
sb.append(line);
sb.append('\n');
```

(*Run the test.*)

Instead of `sourceTemplate = new String(sb);` let's push the work of creating the string onto the StringBuffer: `sourceTemplate = sb.toString();`. (*Run the test.*)

The routine both assigns to "sourceTemplate" and returns it. Let's move the responsibility for the assignment to the caller and just `return sb.toString()` instead. Declare the return type as String. (*Reapportion Work between Caller and Callee.*) (*Run the test.*)

For exceptions, let's declare the routine as throwing `IOException` and delete the empty catch clause; the caller will have to deal with any exceptions. This causes a change (untested! hmm...) in behavior because no partial template will be returned in case of error. We'll confirm by looking and asking whether this is OK. (*Run the test.*)

Our routine now appears as follows:

```
String readTemplate() throws IOException {
    String templateName = System.getProperty(TEMPLATE_DIR, "")
                          + "template.html";
    StringBuffer sb = new StringBuffer("");
    BufferedReader br = null;
    try {
        br = new BufferedReader(new FileReader(templateName));
        String line = br.readLine();
        while (line != null) {
            sb.append(line);
            sb.append('\n');
            line = br.readLine();
        }
    } finally {
        if (br != null) try {br.close();}
                        catch (IOException ioe_ignored) {}
```

```
    }
    return sb.toString();
}
```

The next thing I don't like is that the routine decides both where to find the template and how to read it, leaving it coupled to the file system. This may be a problem in the future (if templates were to come from somewhere else), but it's also a problem now: our test has to use an external file. I'm not sure of the right approach, so we'll defer this problem.

Substitute for "%CODE%"

The routine is still too long. We also still have the near-duplicate code for substitutions.

So next we'll *Extract Method* for replacing `"%CODE%"`.

```
String substituteForCode(String template, String reqId) {
  int templateSplitBegin = template.indexOf("%CODE%");
  int templateSplitEnd = templateSplitBegin + 6;
  String templatePartOne = new String(
    template.substring(0, templateSplitBegin));
  String templatePartTwo = new String(
     template.substring(templateSplitEnd, template.length()));
  code = new String(reqId);
  template = new String(templatePartOne+code+templatePartTwo);
  return template;
}
```

Then we'll adjust the variable declarations left in `substitute()`. (*Run the test.*)

The first thing I notice is the string `"%CODE%"` and the value `6` (the length of the pattern). Pull out the pattern and use it (*Replace Magic Number with Calculation*).

```
  String pattern = "%CODE%";
  int templateSplitBegin = template.indexOf(pattern);
  int templateSplitEnd = templateSplitBegin+pattern.length();
```

(*Run the test.*)

We create a lot of new Strings too: all those `new String()` constructions are redundant, because their arguments are Strings already (*Remove Redundant Constructor Calls*).

```
String templatePartOne =
      template.substring(0,templateSplitBegin);
  String templatePartTwo =
      template.substring(templateSplitEnd, template.length());
  code = reqId;
  return templatePartOne + code + templatePartTwo;
```

(*Run the test.*)

We'll eventually want to address the remaining string addition (on the return statement), but let's take care of `"%ALTCODE%"` first.

Substitute for "%ALTCODE%"

Let's do the same *Extract Method* and simplification for the other case, and see where we are.

```
void substituteForAltcode(String template, String code,
                          PrintWriter out) {
  String pattern = "%ALTCODE%";
  int templateSplitBegin = template.indexOf(pattern);
  int templateSplitEnd = templateSplitBegin+pattern.length();

  String templatePartOne = template.substring(
               0, templateSplitBegin);
  String templatePartTwo = template.substring(
              templateSplitEnd, template.length());
  altcode = code.substring(0,5) + "-" + code.substring(5,8);
  out.print(templatePartOne + altcode + templatePartTwo);
  }
```

This code is *so* similar to `substituteForCode()`, it's clear we should be able to unify the two routines. But there are three differences: they look for different patterns, they substitute different values, and they write to different streams. Drive them toward

duplication by passing in the patterns and replacements as arguments (*Parameterize Method*).

```
void substituteForAltCode(String template, String pattern,
  String replacement, PrintWriter out) {
  int templateSplitBegin = template.indexOf(pattern);

  int templateSplitEnd = templateSplitBegin+pattern.length();

  String templatePartOne = template.substring(
        0, templateSplitBegin);

  String templatePartTwo = template.substring(
        templateSplitEnd, template.length());
  out.print(templatePartOne + replacement + templatePartTwo);
}
        ...
  altcode = reqId.substring(0,5) + "-" + reqId.substring(5,8);
    substituteForAltCode(template, "%ALTCODE%", altcode, out);
```

(*Run the test.*)

We can address the excessive string addition by using a series of `print()` calls (*Replace String Addition with Output*), and we'll pull the buffer flush up as well.

```
    out.print(templatePartOne);
    out.print(replacement);
    out.print(templatePartTwo);
    out.flush();
```

(*Run the test.*)

The big difference now is that `"%CODE%"` results in a String, and `"%ALTCODE%"` is written to a PrintWriter. These can be reconciled via the class `java.io.StringWriter`. So make `substituteForCode()` take an argument `PrintWriter out`, and create and pass in a `new PrintWriter(new StringWriter())` as its stream (*Unify String and I/O*). (*Run the test.*)

How did I know to look for a class such as StringWriter? First, I could have talked to a co-worker and found it; there's usually someone

who knows the odd corners you don't. (In XP, you start with your pair partner; if your partner doesn't know, the rest of the team is in the same room.) Second, I've used enough systems to know that many have a way to let you treat strings as I/O, and vice versa. Third, I could have learned it when I read once through the core APIs when learning the language. Finally, it *is* documented and can be found when needed.

We see that the two routines are now identical: eliminate `substituteForAltcode()`, and just call `substituteForCode()` twice (*Merge Identical Routines*). (*Run the test.*)

Back to readTemplate()

We were able to verify (by asking our customer) that it would be acceptable to read the template once at startup, rather than once per call to `substitute()`. We can declare the constructor to throw IOException, and make the call to `readTemplate()` there.

```
sourceTemplate = readTemplate(
        System.getProperty(TEMPLATE_DIR, ""));
```

(*Run the test.*)

This helps eliminate some strings we would otherwise create.

At last, we can address that old thorn: direct reading of a file to load the template. The `readTemplate()` routine currently takes a directory name and constructs a file. Instead, we'll pass in a Reader and let that do the work. First pull up construction of the FileReader into the constructor (*Reapportion Work between Caller and Callee*).

```
public CodeReplacer() throws IOException {
    String templateName =
System.getProperty(TEMPLATE_DIR, "") + "template.html";
    sourceTemplate = readTemplate(
                        new FileReader(templateName));
}
```

```
public String readTemplate(Reader reader)
                     throws IOException {
    ...
}
```

(*Run the test.*)

Next introduce a constructor that takes a Reader, which the caller is responsible for forming. (They will probably use the `getProperty()` code that was there before.)

```
public CodeReplacer(Reader reader) throws IOException {
  sourceTemplate = readTemplate(reader);
}
```

(*Run the test.*)

I'll put a testing hat back on and modify my test. We can simplify `testTemplateLoadedProperly()`, because we no longer need to do a substitution to see the template just to load it. Also, instead of setting up with the file `template.html`, we'll test using a StringReader. This helps decouple our tests from the environment.

```
final static String templateContents =
                      "xxx%CODE%yyy%ALTCODE%zzz\n";
        ...
  replacer = new CodeReplacer(new StringReader(
                               templateContents));
        ...
public void testTemplateLoadedProperly() {
  assertEquals(templateContents, replacer.sourceTemplate);
}
```

(*Run the test.*)

Finally, the `close()` call in `substitute()` is a little out of place. Because the caller of the object opens the stream, we'd like the caller to be responsible for closing the stream as well. We'll modify `substitute()` and its callers (*Reapportion Work between Caller and Callee*).

(*Run the test one last time.*)

Final Result

Here's the complete new version of CodeReplacer.java:

```
import java.io.*;
import java.util.*;
/** Replace %CODE% with requested id, and %ALTCODE% w/"dashed"
version of id.*/

public class CodeReplacer {
    String sourceTemplate;

    public CodeReplacer(Reader reader) throws IOException {
        sourceTemplate = readTemplate(reader);
    }

    String readTemplate(Reader reader) throws IOException {
        BufferedReader br = new BufferedReader(reader);
        StringBuffer sb = new StringBuffer();
        try {
            String line = br.readLine();
            while (line!=null) {
                sb.append(line);
                sb.append("\n");
                line = br.readLine();
            }
        } finally {
            try {if (br != null) br.close();}
           catch (IOException ioe_ignored) {}
        }
        return sb.toString();
    }

    void substituteCode (String template, String pattern,
                         String replacement, Writer out)
                           throws IOException {
        int templateSplitBegin = template.indexOf(pattern);
        int templateSplitEnd =
             templateSplitBegin + pattern.length();
        out.write(template.substring(0, templateSplitBegin));
        out.write(replacement);
```

```
        out.write(template.substring(
                  templateSplitEnd, template.length()));
        out.flush();
    }

    public void substitute(String reqId, PrintWriter out)
                  throws IOException {
        StringWriter templateOut = new StringWriter();
        substituteCode(sourceTemplate, "%CODE%",
                       reqId, templateOut);

        String altId = reqId.substring(0,5) + "-" +
                          reqId.substring(5,8);
        substituteCode(templateOut.toString(), "%ALTCODE%",
                       altId, out);
    }
}
```

Here's CodeReplacerTest.java:

```
import java.io.*;
import junit.framework.*;

public class CodeReplacerTest extends TestCase {
    final static String templateContents =
                  "xxx%CODE%yyy%ALTCODE%zzz\n";

    CodeReplacer replacer;

    public CodeReplacerTest(String testName)
    {super(testName);}

    protected void setUp() {
        try {
            replacer = new CodeReplacer(new StringReader(
                                        templateContents));
        } catch (Exception ex) {
            fail("CodeReplacer couldn't load");
        }
    }
```

```
    public void testTemplateLoadedProperly() {
        assertEquals(templateContents,
                     replacer.sourceTemplate);
    }

    public void testSubstitution() {
        StringWriter stringOut = new StringWriter();
        PrintWriter testOut = new PrintWriter (stringOut);
        String trackingId = "01234567";

        try {
            replacer.substitute(trackingId, testOut);
            testOut.close();
        } catch (IOException ex) {
            fail ("testSubstitution exception - " + ex);
        }

        assertEquals("xxx01234567yyy01234-567zzz\n",
                     stringOut.toString());
    }

}
```

Analysis

We now have a much better routine. We've fixed a few bugs and ambiguities on the way. It's clearly much easier to read, because we've squeezed out the duplicate code. We replaced string addition with cheaper operations in several places. We've isolated our templates from the file system; instead, they use Readers to load themselves. We still have some cleanup to do: the variable names could be better (e.g., `sb` and `br`); also, the way the alternate identifier is formed is not obvious.

The new code embodies three basic things:

1. How a template is represented (currently a string)
2. How substitution is made
3. What codes and values to substitute (`"%CODE%"` and `"%ALTCODE%"`)

The first two items are part of how a template works; the third is the "business logic" that tells how it's used. A stand-alone Template class would address the first two points. That was not obvious from the monster routine with which we started. This shows how our sense of the code smells can change over time: I'd now argue that an overdependence on the String type keeps us working at too low a level.

Pulling the template into a separate class would allow us to address performance even more, because we would be free to change the representation of templates. The current implementation scans the template (twice) to locate the patterns that need substitution. We might be able to preprocess a template to identify the potential substitutions. We could change the interface to the substitution process to take a list or mapping of substitutions, so we could do them all in one pass.

Summary

What have we seen?

- We can make very small, incremental improvements to the code. At each point, the latest version is better than the one before. We don't have to take the risk inherent in "Just scrap it and start over."
- The unit tests act as a constant safety net. We never go more than a few minutes without the reassurance they provide.
- Some improvements allow further improvements. The need for a Template class was far less obvious in the initial code.
- Although it's possible that some refactorings may interfere with performance, they often will open up possibilities for dramatic improvements. Extracting the `readTemplate()` method allowed us to notice it was called for every substitution when it only needed to be called once; a future version of the Template class might be able to do all substitutions in one

pass. These possibilities are a much bigger factor than the "extra" procedure call that *Extract Method* originally introduced.

- In a handful of moves (about 20), we've substantially improved our code.

Make refactoring (including testing!) a part of your normal programming practice. Sensitize yourself to code smells, and learn refactorings that can address them. It will pay off in the simplicity and flexibility your system will have.

Resources

Beck, Kent. 2000. *Extreme Programming Explained*. Boston: Addison-Wesley.

Bentley, Jon L. 1982. *Writing Efficient Programs*. Englewood Cliffs, NJ: Prentice-Hall.

Bentley, Jon L. 1988. *More Programming Pearls: Confessions of a Coder*. Reading, MA: Addison-Wesley.

Bentley, Jon L. 2000. *Programming Pearls, Second Edition*. Boston: Addison-Wesley.

Brown, William H., Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. 1988. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. New York: John Wiley and Sons.

Fowler, Martin, et al. 1999. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley.
A catalog of code smells and refactorings.

Fowler, Martin. 2001. Online catalog. Available from *http://www.refactoring.com*. INTERNET.

Hunt, Andrew, and David Thomas. 2000. *The Pragmatic Programmer: From Journeyman to Master*. Boston: Addison-Wesley.

JUnit. Available from *http://www.junit.org*. INTERNET.

Lee, Stan. ed. 1996. *The Ultimate Spider-Man*. New York: Boulevard (a subsidiary of The Berkeley Publishing Group).