Refactoring Workbook

William C. Wake
William.Wake@acm.org

Draft of 01-02-03

In memory of Booker Tyler Wake, 1913-2002
A truly great uncle

Refactoring Workbook (01-02-03)

Table of Contents

Table of Exercises

Preface

What's refactoring? What's this book?

Refactoring is the art of "improving the design of existing code." This book is a workbook designed to give you practice recognizing problems, give you practice in refactoring to fix those problems, and help you think more about how your code can become as effective as possible.

What are the goals of this book?

- Give practice identifying the most important "smells" (problems)
- Provide practice with the most important refactoring techniques
- Enable the reader to discover new refactorings
- Have fun!

Who is this book for?

Refactoring is a technique for code, and this book is especially intended for practicing programmers, who write and maintain code.

Students can also benefit from refactoring, though I'd expect they'd only see the value once they've had a chance to develop medium-sized or larger programs, or had to work in teams. (This probably applies to juniors, seniors, and graduate students.)

What background do you need?

It would be helpful to have Martin Fowler's book *Refactoring*, or access to his www.refactoring.com web site, for its catalog of refactorings. (This book could be read in parallel.) Martin has worked out step-by-step instructions for many refactorings, and those won't be repeated in this book. And he's provided a fully worked-out example, along with a lot of good discussion and background material. Someone determined to get through this book without that one could probably do it, but I wouldn't recommend doing it that way.

The examples in the book are Java. This is not because it's the easiest language to refactor, but because it's popular and the environments are providing refactoring support. A C# or C++ programmer probably has enough reading knowledge of Java to make sense of many of the questions. Later parts of the book ask readers to modify, test, and run larger programs, however.

The book *Design Patterns* described patterns as "targets for refactoring." It would be helpful to have some familiarity with ideas in that book. (I certainly feel free to refer to the patterns it mentions.) If you're not yet familiar with *Design Patterns*, let me

recommend Steve Metsker's *Design Patterns Java Workbook* as a companion for that study.

How to use this book?

Solving a problem is more challenging than recognizing a solution. There are answers to some problems in the back of the book, but you'll learn more if you try the problems before peeking at them. If you work through the problems, you'll probably even find that you disagree with me on some answers. That will be more fun for all of us than if you just look at my answer and nod.

I think it's more fun to work with others (either a partner or in a small group). I recognize that isn't always possible.

The later (longer) examples need to be done at a computer. Looking for problems, and figuring out how to solve them, is different when you're looking at a program in your environment.

Can I contact the author?

Sure: William.Wake@acm.org

I have a web site as well: http://www.xp123.com. Its focus is extreme programming (XP), and refactoring is an important part of that. Refactoring (and this book) has its own corner: http://www.xp123.com/rwb.

I'm interested in your experience with these exercises, and with refactoring in general, so please feel free to write.

Acknowledgments

[TBD]

Chapter 1. Introduction and Overview

What is refactoring?

The subtitle of Martin Fowler's book *Refactoring* is "Improving the Design of Existing Code," and that is a good starting point. For this text, we'll define refactoring as "a process of improving the design of existing code (without changing its behavior) via identification of problems and safe transformations that address them."

Existing code is important. Suppose you introduce a new design method that is easy to learn and produces great designs. The world still faces a problem: "What can we do about the billions of lines of code already in use?"

Refactoring tackles this problem. It says, "If we can identify problems and safe transformations, we can improve our code a little bit at a time."

Design versus Designing

The *design* (noun) of something is the structure and dynamics of what it is and how it works. *Designing* (verb) is the process of deciding what the design should be.

One approach to designing is to use tools like UML to plan a design in advance of its implementation. One of the skills you might develop is the ability to critique and manipulate UML diagrams, anticipating any problems likely to arise when the design is implemented. This approach is sometimes referred to as "up-front design," and is usually described as "analyze, design, code, and test."

The challenge of the up-front design is that experience tells us that our designs often don't survive contact with implementation: there are changes due to unforeseen problems, or misunderstandings about the intended design, and so on. It's easy for a paper design to look good, even if it isn't.

A second approach is to design as little as possible, move more directly into implementation, and clean up the result until it had a good design. This is known as "post-hoc design" or "emergent design." (Extreme Programming—XP—leans more toward this approach.) Ralph Johnson describes XP's test-driven development as "analyze, test, code, design." In effect, design decisions ("how will we organize this?") take place in the context of the code at hand.

The tradeoff between up-front and emergent design hinges on how well we can anticipate problems or assess them in code, and whether it's easier to "design then translate to code" or "code then improve."

Refactoring changes the balance point between up-front design and emergent design. Since refactoring improves our ability to recognize code in need of improvement, and

then safely improve the code, it lowers the cost and risk of the second approach. (Reasonable people can disagree on where the line is, but I think all will agree it shifts.)

Can refactoring lead to a design as good as a one created in advance of implementation? I don't know for sure. In my experience, refactoring has an advantage: it's an additional tool. Since it considers code that exists, it has more information available to it than up-front design, so it may have a better opportunity.

Is refactoring better than rewriting? Sven Gorts points out (private communication) that refactoring preserves the knowledge embedded in the existing code. There are times where it's better to start fresh, but refactoring changes the balance point, making it possible to improve code rather than take the risk of rewriting.

Types of Refactoring

One way to look at refactoring is by considering the sizes or scope of the changes. This is typically divided into "small refactorings" (takes a couple minutes and affects little of the system), "medium refactorings" (takes a whole session and can change a number of files), and "large refactorings" (takes place over days, weeks, or months, and can substantially affect the system). Small refactorings typically involve only a class or two. Larger refactorings are often composed of small refactorings. While a large refactoring is in progress, you may have to live with two ways of doing things, with the hassles that entails.

A second division between refactorings is by intent. Most of the refactorings we'll consider have the goal of improving the design in terms of its simplicity, lack of duplication, and clarity. A different goal may be "refactoring for performance," which might be willing to compromise the clarity of a design to improve its performance, scalability, etc. given demonstrated performance problems with the original design. (Better designs often perform better with less need for this performance tuning.)

Another division of refactorings is by the programming language in use. Some refactorings are for Java or C#; others can be defined even at the UML level.

Yet another approach might consider refactorings based on what they change: some are focused on names, others on straight-line code, others on object structure, and so on.

The Term "Refactoring"

Our definition of refactoring started from the idea "improving the design of existing code."

Some people misunderstand refactoring to mean "any change in a system" – including design improvements, but also including adding new functionality. We reject that interpretation. While refactoring can be *part* of the process used to create new code, it's not the part that adds new features. (For example, XP uses an approach known as test-

driven development, which consists of test-first programming to introduce new features, followed by refactoring to improve the design.)

Some people misunderstand refactoring to be any re-structuring intended to improve code. What distinguishes our definition from this is the idea that refactorings strive to be *safe* transformations. Even "big refactorings" that change large amounts of code are divided into smaller, safe refactorings. (In the best case, refactorings are so well defined that they can be automated.) We won't regard a change as refactoring if it leaves the code not working (that is, not passing its tests) for longer than a working session.

Some people misunderstand refactoring to include only large-scale changes. In fact, many refactorings are small. Ideally, the small refactorings are applied "mercilessly" enough that large refactorings are rarely needed. And even when large-scale refactorings are needed, the approach is not "no new features for six months while we refactor," but rather, "refactor as you go, and keep the system running at all times."

The Environment for Refactoring

Refactoring can be done with just a simple text editor, but refactoring is easier and safer with a supportive environment.

Team or Partner: For non-trivial decisions about code, it's helpful to have more than one person considering the problem. A team can often generate ideas better than one person alone: different people have different experiences and different exposure to different parts of the system.

Tests: Even though refactorings are designed to be safe, it's possible to make a mistake in applying them. By having a test suite you run before and after refactoring, you help ensure that you change the design of your code, not its effects.

What if you don't have tests? Then add them, at least to the areas affected by the refactoring. Sometimes this is tricky—you may be unable to test effectively without changing the design, yet it's unsafe to change the design without tests. And areas that are tricky to test often indicate other problems in the design.

> "If you want to refactor, the essential precondition is having solid tests."
>
> —Martin Fowler, *Refactoring*, p. 89

CRC Cards or UML Sketches: Refactoring doesn't mean eliminating design. Sometimes you may hold a CRC card session or draw UML sketches to compare alternatives for refactoring.

Configuration Management/Version Control: This can range from "undo" to a full-fledged configuration management system. If you make a mistake while refactoring, you'd like to be able to return to the last "known good" point. Alternatively, you may want to apply a refactoring but you're not sure if the result will be an improvement or not. It's helpful to be able to try it, and then decide whether to keep the result.

Sophisticated IDE (Integrated Development Environment): Simple-but-powerful languages such as Smalltalk and Lisp have had refactoring support available in their environments for a number of years. Java has only had such support for the last year or two, and refactoring support is just now beginning to show up for C#.

Consider the *Extract Method* refactoring. In a simple text editor, it can be a tedious process of copying code, working out parameters, adjusting variables, arranging for a return value, and so on. With tool support, this is faster and much simpler: "select text, extract method, enter new method name." The tool handles the mechanics. (You don't escape the challenge of deciding what to do, however.)

The Book Refactoring

The book *Refactoring: Improving the Design of Existing Code,* by Martin Fowler et al., is the key book in the refactoring literature. (When I say *Refactoring* (in italics), or mention "Martin" or "Fowler," this book is what I'm talking about.) This book gives an extended example of refactoring, it discusses most of the problems we'll consider, and it contains interesting background material. Its centerpiece is an extensive catalog of refactorings. This catalog describes the current state, the mechanics of a solution that can address it, and an example of its use.

The catalog is a great reference. Martin has worked out step-by-step instructions, including telling you when to compile and test. This will help you refactor more safely than you otherwise would.

This workbook owes a lot to *Refactoring*. I've tried to be a little more concrete in explaining how to identify particular smells, but you'll find "what to do" is often nearly identical to Martin's prescriptions. I hope the re-organization "by smell" is helpful in its own right, but the real focus of this book is the exercises and challenges.

Relationship to XP

Extreme Programming (XP) is an "agile" software development method, characterized by a particular planning style, a team focus, and programming through tests and refactoring. While refactoring has been around longer, there's a cluster of people including Kent Beck, Ward Cunningham, Martin Fowler, Ralph Johnson, and many others, who have been working in the areas of object-oriented development, patterns, refactoring, and other related ideas.

Programming in XP is built around three things: an ethos of simple design, test-first programming, and refactoring. We'll talk briefly about simple design later. Test-first programming is an approach that says, "write a test, see it fail, write code, see the test pass." When you put these things together, you get what is now called test driven development.

I'm a fan of the test-driven approach (and XP, for that matter), but the discipline of refactoring doesn't require it.

Plan for the Book

Chapter 2 examines how individual refactorings work, and how they work together.

Chapters 3 through 11 look at various problems ("smells") that you might see in your code, and suggest possible improvements. These chapters include a number of exercises.

Chapter 12 considers how you might create and document new refactorings.

Chapters 13 through 17 are "hands-on" examples, with moderate-sized programs. They cover several areas: a database, a game, a store catalog, and a graphical program.

Finally, chapter 18 takes a brief look at "where to go from here," with bits of advice, some exercises, and a tour of resources.

Sprinkled throughout the chapters are "interludes"—brief excursions into analyzing the refactoring catalog in *Refactoring* or the patterns in *Design Patterns*.

There are answers to selected problems at the end of the book.

Chapter 2. The Refactoring Cycle

"Step by step"

In this chapter, I'll assume you have already seen an example showing how a series of refactorings can improve a design. (If you haven't, there's an example near the beginning of *Refactoring*, or you can just peek ahead to one of the medium-sized examples in the last third of this book.)

"Smells" = Problems

"Smells" (especially "code smells") are warning signs about potential problems in code. Not all smells indicate a problem, but most are worthy of a look and a decision.

Some people dislike the term "smell," and prefer to talk about "potential problems" or "flaws." But I think "smell" is a good metaphor. Think about what happens when you open a fridge that has a few things going bad inside. Some smells will be strong, and it will be obvious what to do about them. Other smells will be subtler; you won't be sure if the leftover peas are the problem, or maybe that Kung Pao Chicken. Some things will be bad, but not have a particularly bad smell. Code smells seem like that to me: some are obvious, some aren't. Some mask other problems. Some go away unexpectedly when you fix something else.

Smells usually describe localized problems. It would be nice if people could easily find problems across a whole system. But humans aren't so good at that job; local smells work with our tendency to consider only the part we're looking at right now.

The Refactoring Cycle

There's a basic pattern for refactoring:

The Refactoring Cycle

Start with a working program
While smells remain
    Choose the worst smell
    Select a refactoring that will address the smell.
    Apply the refactoring.

Refactoring improves the design, and none of the steps change the program's observable behavior, so the program remains in a working state each trip through the cycle. If we select only refactorings that improve the code, the program improves on every cycle too.

Thus, the cycle improves code but retains behavior. The trickiest part of the whole process is identifying the smell, and that's why the bulk of this book covers that topic so deeply.

I think of refactoring as "crossing a stream." One way to cross a stream is to take a running leap, and hope for the best. The refactoring way is to find step-stones, and cross one stone at a time.

When Are We Done?

One approach is to seek the "simplest" design. Kent Beck (*Extreme Programming Explained*) has identified four rules for simple design; if your code violates these rules (which are in priority order), then you have a problem to address.

Simple Design

1. The code works.
2. The code communicates what it needs to.
3. The code has no duplication.
4. The code has as few classes and methods as possible.

A shorthand name for these rules is "OAOO": "Once And Only Once." The code has to state something once, so it can pass its tests and communicate the programmer's understanding and intent. And it should say things only once: with no duplication.

It's hard to clean up code that hasn't been kept clean; few teams can afford to turn the lights out for months on a quest for perfection. But we can learn to make our code better during development, and we can add a little energy each time we're working in an area.

Inside a Refactoring

One of the defining aspects of refactoring is the focus on safe transformations. We'll walk through a simple refactoring. Along the way we'll derive some guidelines that will help us understand better how refactorings work, so we'll better be able to do refactorings not yet cataloged.

Consider the refactoring *Encapsulate Field*. Its goal is to make clients of an object use methods to access a field rather than access the field directly:

```
public String name;
        ➔
private String name;
public String getName() {return name;}
public void setName(String newName) {name = newName;}
```

Martin Fowler's catalog tells us:

1. Create getters and setters for the field.
2. Locate all references; replace accesses with calls to the getter, and changes to the field with calls to the setter.
3. Compile and test after changing each reference.
4. Declare the field as private.
5. Compile and test

Refactoring is a step-by-step process. The steps are smaller than people typically expect. Most refactorings tend to take from a minute to an hour to apply; the average is probably five to ten minutes. So, if a refactoring takes a few minutes, the steps are even smaller.

Refactoring works in tiny steps.

Initially, we have:

```
public class Person {public String name;}
```

The test client looks like this:

```
      Person person;
      person.name = "Bob Smith";
      assertEquals("Bob Smith", person.name);
```

Step 1: Create getter and setter methods.

```
public class Person {
      public String name;
      public String getName() {return name;}
      public void setName(String newName) {name = newName;}
}
```

Note that the client is unchanged: nobody's using these new methods yet!

Step 2: Find all clients; replace references with calls. Do this one at a time.

One way to find those references is to temporarily make the field private; if you do this, put it back to public scope before changing the clients, so you don't break any clients.

In many refactorings, the compiler will tell you if things are going right. For example, suppose you're using *Hide Method* to make a public method be private. You change the visibility keyword, and when you re-compile you get a warning that some client is still using the method. Because the warning is at compile-time instead of an error at run-time, you're protected while you refactor.

The compiler talks to you.

Assignment -

```
      person.name = "Bob Smith";
```

becomes

```
person.setName("Bob Smith");
```

Step 3: Compile and test.

Even though refactorings have the goal of having an improved system at the end of the refactoring, many of them have "bases," "safe points," or "interruption points" along the way. (Think of bases in baseball or the children's game of tag: they may not be the ultimate destination, but at least you can't get tagged while you're on the base.) The system is not as clean as it will be in the end, since it embodies two different approaches mid-refactoring. But we can stop, run the tests, and make sure we're OK so far.

Most refactorings have built-in "bases."

I imagine holding my breath while the system is in an unsafe state, and then letting it go when the tests run correctly. This mild tension and release feels so much better than the feeling you get where you're halfway through one thing, and you realize you want to do something else before you finish, and so on until you're juggling five balls instead of one.

Large refactorings use this "base" idea as well. (It's even more important there.) If it will take months to clean out the remnants of some decision, we certainly want safe points along the way.

Step 2, at a different reference.

Reference -

```
assertEquals("Bob Smith", person.name);
```

becomes

```
assertEquals("Bob Smith", person.getName());
```

Step 3: Compile and test

If we absolutely had to, we could stop after either occurrence of step 3, and still be safe. The code would be worse than it started in one sense: it uses two inconsistent approaches to access the field. But, it is progress toward a better approach.

Step 4: Once all clients are changed, make the field private:

```
public class Person {
      private String name;
      public String getName() {return name;}
      public void setName(String newName) {name = newName;}
}
```

Step 5: Compile and test one last time.

Again, we're safe, and the field is completely encapsulated.

Challenges

CYCLE-1. Small steps. Pick a refactoring, and identify a place where the approach builds in small steps even though larger steps "could" work.

Solution ideas on page 173.

CYCLE-2. Simple design.

A. Justify each of Beck's rules for simple design.

B. Why are these rules in priority order? Can you find an example where communication overrides avoidance of duplication?

Solution ideas on page 173.

Conclusion

As you use refactorings, and as you design new ones, use them in such a way that the system moves from good state to good state. Prefer a "small steps but safer" approach to a "fast but not always safe" approach when you refactor. When you identify your own refactorings, keep the steps of the refactoring cycle in mind.

Chapter 3. Measured Smells

The smells in this chapter are similar. They're dead easy to detect. They're objective (agree on a way to count and a maximum acceptable score). They're odious.

And, they're common.

Extract Method is a very common refactoring, and you'll see it applied here to deal with several smells. There are a few other refactorings used in this chapter as well.

You can think of these smells as being caught by a software metric. Each metric tends to catch different aspects of why code isn't as good as it could be. Some metrics measure variants of code length, others try to measure the connections between methods or objects; others measure a distance from an ideal.

Most metrics seem to correlate with length, so I tend to worry about size first (usually noticeable as a Large Class or Long Method). But if a metric is easy to compute, I'll use it as an indicator that some section of code deserves a closer look.

Metrics are indicators, not absolutes. It's very easy to get into the trap of "making numbers" without addressing the total complexity. So don't refactor just for a better number; make sure it really improves your code.

Comments

Symptoms

Scan the text for "//" or "/*" (comment markers). (Some IDEs can help, by color-coding different types of comments.)

Causes

Usually, for the best of reasons. The author realizes that something isn't as clear as it could be, and adds a comment.

Some comments are particularly helpful:

- Those that tell "why" something is done a particular way (or why it wasn't)
- Those that cite non-obvious algorithms (although… can a simple algorithm take its place?)

Other comments can just as well be reflected in the code itself. For example, the goal of a routine can often be communicated just as well through the routine's name as through a comment.

What to Do

- When a comment explains a block of code, you can often use *Extract Method* to pull the block out into a separate method. The comment will often suggest a name for the new method.
- When a comment explains what a method does (better than the method's name!), use *Rename Method* using the comment as the basis of the new name.
- When a comment explains preconditions, consider using *Introduce Assertion* to replace the comment with code.

Payoff

Improves communication, may expose duplication.

Counter-Indications

Don't delete comments that are pulling their own weight.

MEASURE-1. Comments. Consider this code.

Matcher.java

```
public class Matcher {
    public Matcher() {}
    public boolean match(int[] expected, int[] actual,
                         int clipLimit, int delta) {

        // Clip "too-large" values
```

```
        for (int i = 0; i < actual.length; i++)
            if (actual[i] > clipLimit)
                actual[i] = clipLimit;

        // Check for length differences
        if (actual.length != expected.length)
            return false;

        // Check that each entry within expected +/- delta
        for (int i = 0; i < actual.length; i++)
            if (Math.abs(expected[i] - actual[i]) > delta)
                return false;

        return true;
    }
}
```

MatcherTest.java

```
import junit.framework.TestCase;

public class MatcherTest extends TestCase {
    public MatcherTest(String name) {super(name);}

    public void testMatch() {
        Matcher matcher = new Matcher();

        int[] expected = new int[] {10, 50, 30, 98};
        int clipLimit = 100;
        int delta = 5;

        int[] actual = new int[] {12, 55, 25, 110};

        assertTrue(matcher.match(expected, actual, clipLimit, delta));

        actual = new int[] {10, 60, 30, 98};
        assertTrue(!matcher.match(expected, actual, clipLimit, delta));

        actual = new int[] {10, 50, 30};
        assertTrue(!matcher.match(expected, actual, clipLimit, delta));
    }
}
```

A. Use *Extract Method* to make the comments in `match()` redundant.

B. Can everything important about the code be communicated using the code alone? Or do comments have a place?

C. Find some code you wrote recently. Odds are good you commented it. Can you eliminate the need for some of those comments by making the code reflect your intentions more directly?

Some solutions on page 173.

Long method

Symptoms

Large number of lines in the method. (I'm immediately suspicious of any method with more than five to ten lines.)

Causes

I think of it as the "Columbo syndrome." Columbo was the detective who always had "just one more thing." A method has started down a path, and rather than break the flow or identify the helper objects, the author adds "one more thing." Code is often easier to write than it is to read, so there's a temptation to write blocks that are too big.

What to Do

Use *Extract Method* to break up the method into smaller pieces. Look for comments or white space that delineate "interesting" blocks. You want to extract methods that are semantically meaningful, not just introduce a function call every seven lines.

You may find other refactorings (those that clean up straight-line code, conditionals, and variable usage) helpful before you even begin splitting up the method.

Payoff

Improves communication, may expose duplication.

Discussion

People are sometimes worried about the performance hit from increasing the number of method calls. Most of the time this is a non-issue. By getting the code as clean as possible before worrying about performance, you have the opportunity to gain "big insights" that can re-structure systems and algorithms in a way that dramatically increases performance.

MEASURE-2. Long Method. Consider this code: (available online)

Machine.java

```
public class Machine {
    String name;
    String location;
    String bin;

    public Machine(String name, String location) {
        this.name = name;
        this.location = location;
    }

    public String take() {
        String result = bin;
        bin = null;
        return result;
    }

    public String bin() {
        return bin;
    }

    public void put(String bin) {
        this.bin = bin;
    }

    public String name() {return name;}
}
```

Robot.java

```
public class Robot {
    Machine location;
    String bin;

    public Robot() {}

    public Machine location() {return location;}
    public void moveTo(Machine location) {this.location = location;}

    public void pick() {this.bin = location.take();}
    public String bin() {return bin;}

    public void release() {
        location.put(bin);
        bin = null;
    }
}
```

RobotTest.java

```
import junit.framework.*;

public class RobotTest extends TestCase{
    public RobotTest(String name) {super(name);}

    public void testRobot() {
        Machine sorter = new Machine("Sorter", "left");
        sorter.put("chips");
        Machine oven = new Machine("Oven", "middle");
        Robot robot = new Robot();

        assertEquals("chips", sorter.bin());
        assertNull(oven.bin());
        assertNull(robot.location());
        assertNull(robot.bin());

        robot.moveTo(sorter);
        robot.pick();
        robot.moveTo(oven);
        robot.release();

        assertNull(robot.bin());
        assertEquals(oven, robot.location());
        assertNull(sorter.bin());
        assertEquals("chips", oven.bin());
    }
}
```

Report.java

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

public class Report {
    public static void report(Writer out, List machines, Robot robot)
throws IOException {
        out.write("FACTORY REPORT\n");

        Iterator line = machines.iterator();
        while (line.hasNext()) {
            Machine machine = (Machine) line.next();
            out.write("Machine " + machine.name());

            if (machine.bin() != null)
                out.write(" bin=" + machine.bin());

            out.write("\n");
        }
        out.write("\n");

        out.write("Robot");
        if (robot.location() != null)
            out.write(" location=" + robot.location().name());
```

```
        if (robot.bin() != null)
            out.write(" bin=" + robot.bin());

        out.write("\n");

        out.write("========\n");
    }
}
```

ReportTest.java

```
import junit.framework.TestCase;

import java.util.ArrayList;
import java.io.PrintStream;
import java.io.StringWriter;
import java.io.IOException;

public class ReportTest extends TestCase {
    public ReportTest(String name) {super(name);}

    public void testReport() throws IOException {
        ArrayList line = new ArrayList();
        line.add(new Machine("mixer", "left"));

        Machine extruder = new Machine("extruder", "center");
        extruder.put("paste");
        line.add(extruder);

        Machine oven = new Machine("oven", "right");
        oven.put("chips");
        line.add(oven);

        Robot robot = new Robot();
        robot.moveTo(extruder);
        robot.pick();

        StringWriter out = new StringWriter();
        Report.report(out, line, robot);

        String expected =
            "FACTORY REPORT\n" +
            "Machine mixer\nMachine extruder\n" +
            "Machine oven bin=chips\n\n" +
            "Robot location=extruder bin=paste\n" +
            "========\n";

        assertEquals(expected, out.toString());
    }
}
```

A. In Report.java, circle four blocks of code above to show what functions you might extract in the process of refactoring this code.

B. Rewrite the `report()` method as four statements, as if you had done *Extract Method* for each block.

C. Does it make sense to extract a one-line method?

D. Long methods are trivially easy to spot, yet they seem to occur often in real code. Why?

Solution ideas on page 173.

Large Class

Symptoms

- Large number of instance variables
- Large number of methods
- Large number of lines

Causes

A little bit at a time. The author adds "one more capability" to a class, and eventually it grows too big. Sometimes the problem is a lack of insight into the parts that make up the whole class. In any case, the class represents too many responsibilities folded into one object.

What to Do

In general, you're trying to break up the class. If the class has Long Methods, address that smell first. To break up the class, three approaches are most common:

- *Extract Class*, if you can identify a new object that has part of this class's responsibilities
- *Extract Subclass,* if you can divide responsibilities between the class and a new subclass
- *Extract Interface,* if you can identify subsets of features that clients use.

Sometimes, the class is big because it's a GUI (user interface) class, and it's representing not only the display component, but the model as well. In this case, you can use *Duplicate Observed Data* to help extract a domain object.

Payoff

Improves communication, may expose duplication.

MEASURE-3. Large Class. Consider this declaration from the Java libraries:

```
public class JTable extends JComponent
  implements Accessible, CellEditorListener,
      ListSelectionListener, Scrollable,
      TableColumnModelListener, TableModelListener
{
// Constants
public static final int AUTO_RESIZE_ALL_COLUMNS
public static final int AUTO_RESIZE_LAST_COLUMN
public static final int AUTO_RESIZE_NEXT_COLUMN
public static final int AUTO_RESIZE_OFF
public static final int AUTO_RESIZE_SUBSEQUENT_COLUMNS

// Constructors
```

```
public JTable()
public JTable(TableModel, TableColumnModel)
public JTable(TableModel, TableColumnModel, ListSelectionModel)
public JTable(int, int)
public JTable(Object[][], Object[][])
public JTable(java.util.Vector, java.util.Vector)

// Methods
public void addColumn(TableColumn column)
public void addColumnSelectionInterval(int start, int finish)
public void addNotify()
public void addRowSelectionInterval(int start, int finish)
public void clearSelection()
public void columnAdded(TableColumnModelEvent event)
public void columnAtPoint(Point p)
public void columnMarginChanged(ChangeEvent event)
public void columnMoved(TableColumnModelEvent event)
public void columnRemoved(TableColumnModelEvent event)
public void columnSelectionChanged(ListSelectionEvent event)
public void convertColumnIndexToModel(int viewColumn)
public void convertColumnIndexToView(int modelColumn)
public void createDefaultColumnsFromModel()
public boolean editCellAt(int row, int column)
public boolean editCellAt(int row, int column, EventObject event)
public void editingCanceled(ChangeEvent event)
public void editingStopped(ChangeEvent event)
public AccessibleContext getAccessibleContext()
public boolean getAutoCreateColumnsFromModel()
public int getAutoResizeMode()
public TableCellEditor getCellEditor()
public TableCellEditor getCellEditor(int row, int column)
public Rectangle getCellRect(int row, int column,
                             boolean includeSpacing)
public boolean getCellSelectionEnabled()
public TableColumn getColumn(Object object)
public Class getColumnClass(int column)
public int getColumnCount()
public TableColumnModel getColumnModel()
public String getColumnName(int column)
public Boolean getColumnSelectionAllowed()
public TableCellEditor getDefaultEditor(Class class)
public TableCellRenderer getDefaultRenderer(Class class)
public int getEditingColumn()
public int getEditingRow()
public Component getEditorComponent()
public Color getGridColor()
public Dimension getIntercellSpacing()
public TableModel getModel()
public Dimension getPreferredScrollableViewportSize()
public int getRowCount()
public int getRowHeight()
public int getRowMargin()
public Boolean getRowSelectionAllowed()
public int getScrollableBlockIncrement(
                  Rectangle visible, int orientation, int direction)
```

```
public Boolean getScrollableTracksViewportHeight()
public Boolean getScrollableTracksViewportWidth()
public int getScrollableUnitIncrement(
                  Rectangle visible, int orientation, int direction)
public int getSelectedColumn()
public int getSelectedColumnCount()
public int[] getSelectedColumns()
public int getSelectedRow()
public int getSelectedRowCount()
public int[] getSelectedRows()
public Color getSelectionBackground()
public Color getSelectionForeground()
public ListSelectionModel getSelectionModel()
public Boolean getShowHorizontalLines()
public Boolean getShowVerticalLines()
public JTableHeader getTableHeader()
public String getToolTipText(MouseEvent event)
public TableUI getUI()
public String getUIClassID()
public Object getValueAt(int row, int column)
public Boolean isCellEditable(int row, int column)
public Boolean isCellSelected(int row, int column)
public Boolean isColumnSelected(int column)
public Boolean isEditing()
public boolean isManagingFocus()
public Boolean isRowSelected(int row)
public void moveColumn(int column, int newColumn)
public Component prepareEditor(TableCellEditor editor,
                               int row, int column)
public Component prepareRenderer(TableCellRenderer renderer,
                               int row, int column)
public void removeColumn(TableColumn column)
public void removeColumnSelectionInterval(int column1, int column2)
public void removeEditor()
public void removeRowSelectionInterval(int row1, int row2)
public void reshape(int x, int y, int width, int height)
public int rowAtPoint(Point point)
public void selectAll()
public void setAutoCreateColumnsFromModel(Boolean doAutoCreate)
public void setAutoResizeModel(int mode)
public void setCellEditor(TableCellEditor editor)
public void setCellSelectionEnabled(Boolean maySelect)
public void setColumnModel(TableColumnModel model)
public void setColumnSelectionAllowed(Boolean maySelect)
public void setColumnSelectionInterval(int column1, int column2)
public void setDefaultEditor(Class class, TableCellEditor editor)
public void setDefaultRenderer(Class class, TableCellRenderer renderer)
public void setEditingColumn(int column)
public void setEditingRow(int row)
public void setGridColor(Color color)
public void setIntercellSpacing(Dimention dim)
public void setModel(TableModel model)
public void setPreferredScrollableViewportSize(Dimension dim)
public void setRowHeight(int height)
public void setRowMargin(int margin)
```

```
public void setRowSelectionAllowed(Boolean maySelect)
public void setSelectionBackground(Color background)
public void setSelectionForeground(Color foreground)
public void setSelectionMode(int mode)
public void setSelectionModel(ListSelectionModel model)
public void setShowGrid(Boolean showing)
public void setShowHorizontalLines(Boolean b)
public void setShowVerticalLines(Boolean b)
public void setTableHeader(JTableHeader header)
public void setUI(TableUI ui)
public void setValueAt(Object value, int row, int column)
public void sizeColumnsToFit(int resizingColumn)
public void tableChanged(TableModelEvent event)
public void updateUI()
public void valueChanged(ListSelectionEvent)

// plus 22 protected variables

// plus 10 protected methods

// plus 97 methods inherited from JComponent (and fields etc.)

// plus about 35 methods from Container

// plus about 85 methods from Component

// plus 11 methods from Object
}
```

A. Why does this class have so many methods?

B. Go through the methods listed, and categorize them into five to ten major areas of responsibility.

C. In what ways could the library writers have eliminated some of these methods?

D. In Java, Object has eleven methods. In Smalltalk, it has more than one hundred. Why the difference? Talk to a Smalltalk person and find out why, and if this is a smell.

Solution ideas on page 174.

Long Parameter List

Symptoms

Count the number of parameters to a method.

(Even three or four might be too many.)

Causes

An author often tries to minimize coupling between objects. Instead of the called object being aware of relationships between objects, you let the caller locate everything; then the method concentrates on what it is being asked to do with the pieces.

Or, the author generalizes the routine to deal with multiple variations: there's a general algorithm and a lot of "control" parameters.

What to Do

- If the parameter value can be obtained from another object this one already knows, *Replace Parameter with Method*.
- If the parameters come from a single object, try *Preserve Whole Object*.
- If the data is not from one logical object, you still might group them via *Introduce Parameter Object*.

Payoff

Improves communication, may expose duplication, often reduces size.

Counter-Indications

Sometimes, you *want* to avoid a dependency between this object and others. For example, the caller may have the dependency, but you don't want to propagate it. Ensure that your changes don't upset this balance.

MEASURE-4. Long parameter list. Consider these methods declared in the Java libraries:

From java.swing.CellRendererPane:

```
public void paintComponent(Graphics gr, Component renderer,
   Container parent, int x, int y, int width, int height,
   Boolean shouldValidate)
```

From java.awt.Graphics:

```
public Boolean drawImage(Image image,
    int x1Dest, int y1Dest, int x2Dest, int y2Dest,
    int x1Source, int y1Source, int x2Source, int y2Source,
    Color color, ImageObserver obs)
```

From java.swing.DefaultBoundedRangeModel:

```
public void setRangeProperties(
    int newValue, int newExtent,
    int newMin, int newMax,
    boolean isAdjusting)
```

From java.swing.JOptionPane:

```
public static int showConfirmDialog(Component parent, Object message,
String title, int optionType, int messageType, Icon icon)
```

A. For each declaration above, is there any cluster of parameters you might reasonably group into a new object?

B. Why might those signatures have so many parameters?

C. Look back at the JTable declaration (Measure-3. page 22). Do you see any clusters of parameters there?

Solution ideas on page 174.

More Challenges

MEASURE-5. Smells and refactorings.
These are the "smells" we discussed:

A. Comments
B. Large Class
C. Long Method
D. Long Parameter List

For each refactoring below, write in the letter for the smell(s) it might help cure:

___ Duplicate Observed Data
___ Extract Class
___ Extract Interface
___ Extract Method
___ Extract Subclass
___ Introduce Assertion
___ Introduce Parameter Object
___ Preserve Whole Object
___ Rename Method
___ Replace Parameter with Method

Solution on page 174.

MEASURE-6. Triggers. Consider the smells described in this chapter: Comments, Large Class, Long Method, Long Parameter List.

A. Which of these do you find most often? Which do you create most often?

B. To stop children from sucking their thumb, some parents put a bad-tasting or spicy solution on the child's thumb. This serves as a trigger that reminds the child not to do that. What triggers can you give yourself, to help you recognize when you're just beginning to create one of these smells?

Solution ideas on page 175.

Conclusion

The smells in this chapter are the easiest to identify. They're not necessarily the easiest to fix.

There are other metrics that have been applied to software. Many of them are just refinements of "code length." Pay attention when things feel like they're getting too big.

Refactorings don't fit one-to-one with smells; we'll run into some of these again. For example, *Extract Method* is a tool that can fix many problems.

Finally, remember a smell is an indication of a potential problem, not a guarantee of an actual problem. You will occasionally find "false positives": things that smell to you, but are actually better than the alternatives. But most code has plenty of real smells that can keep you busy.

Interlude 1 – Smells and Refactorings

The refactorings in Martin Fowler's book *Refactoring* are listed on the next page, and the smells on the page after that.

I1-1. Put a checkmark. Put a checkmark by each refactoring for each time a smell references it.

I1-2. Fix the most. Which refactorings fix the most smells?

Solution on page 175.

I1-3. Not mentioned. Which refactorings aren't mentioned by any of the smells? Why not?

Solution ideas on page 175.

I1-4. Other smells. Does this list suggest any other smells you might want to be aware of?

Solution ideas on page 175.

The list of refactorings, from inside the front cover of *Refactoring*:

Add Parameter	Pull Up Constructor Body
Change Bidirectional Association to Unidirectional	Pull Up Field
	Pull Up Method
Change Reference to Value	Push Down Field
Change Unidirectional Association to Bidirectional	Push Down Method
	Remove Assignment to Parameters
Change Value to Reference	Remove Control Flag
Collapse Hierarchy	Remove Middle Man
Consolidate Conditional Expression	Remove Parameter
Consolidate Duplicate Conditional Expression	Remove Setting Method
	Rename Method
Convert Procedural Design to Objects	Replace Array with Object
Decompose Conditional	Replace Conditional with Polymorphism
Duplicate Observed Data	Replace Constructor with Factory Method
Encapsulate Collection	Replace Data Value with Object
Encapsulate Downcast	Replace Delegation with Inheritance
Encapsulate Field	Replace Error Code with Exception
Extract Class	Replace Exception with Test
Extract Hierarchy	Replace Inheritance with Delegation
Extract Interface	Replace Magic Number with Symbolic Constant
Extract Method	
Extract Subclass	Replace Method with Method Object
Extract Superclass	Replace Nested Conditional with Guard Clause
Form Template Method	
Hide Delegate	Replace Parameter with Explicit Methods
Hide Method	Replace Parameter with Method
Inline Class	Replace Record with Data Class
Inline Method	Replace Subclass with Fields
Inline Temp	Replace Temp with Query
Introduce Assertion	Replace Type Code with Class
Introduce Explaining Variable	Replace Type Code with State/Strategy
Introduce Foreign Method	Replace Type Code with Subclasses
Introduce Local Extension	Self Encapsulate Field
Introduce Null Object	Separate Domain from Presentation
Introduce Parameter Object	Split Temporary Variable
Move Field	Substitute Algorithm
Move Method	Tease Apart Inheritance
Parameterize Method	
Preserve Whole Object	

A list of smells and associated refactorings, from the back cover of *Refactoring*:

Smell	Common Refactorings
Alternative Classes with Different Interfaces	Rename Method, Move Method
Comments	Extract Method, Introduce Assertion
Data Class	Move Method, Encapsulate Field, Encapsulate Collection
Data Clumps	Extract Class, Introduce Parameter Object, Preserve Whole Object
Divergent Change	Extract Class
Duplicated Code	Extract Method, Extract Class, Pull Up Method, Form Template Method
Feature Envy	Move Method, Move Field, Extract Method
Inappropriate Intimacy	Move Method, Move Field, Change Bidirectional Association to Unidirectional, Replace Inheritance with Delegation, Hide Delegate
Incomplete Library Class	Introduce Foreign Method, Introduce Local Extension
Large Class	Extract Class, Extract Subclass, Extract Interface, Replace Data Value with Object
Lazy Class	Inline Class, Collapse Hierarchy
Long Method	Extract Method, Replace Temp with Query, Replace Method with Method Object, Decompose Conditional
Long Parameter List	Replace Parameter with Method, Introduce Parameter Object, Preserve Whole Object
Message Chains	Hide Delegate
Middle Man	Remove Middle Man, Inline Method, Replace Delegation with Inheritance
Parallel Inheritance Hierarchies	Move Method, Move Field
Primitive Obsession	Replace Data Value with Object, Extract Class, Introduce Parameter Object, Replace Array with Object, Replace Type Code with Class, Replace Type Code with Subclasses, Replace Type Code with State/Strategy
Refused Bequest	Replace Inheritance with Delegation
Shotgun Surgery	Move Method, Move Field, Inline Class
Speculative Generality	Collapse Hierarchy, Inline Class, Remove Parameter, Rename Method
Switch Statements	Replace Conditional with Polymorphism, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Parameter with Explicit Methods, Introduce Null Object
Temporary Field	Extract Class, Introduce Null Object

Chapter 4. Duplication

"You can say that again!"

Duplication has been recognized for more than twenty-five years as the bane of the programmer's lot.

How does duplication cause problems?

- It means more code to maintain (a conceptual and physical burden)
- The parts that vary are buried in the parts that stay the same (a perceptual problem–it's hard to see the important stuff)
- Code variations often hide deeper similarities – it will be hard to see the deeper solution among all the similar code.
- There's a tendency to fix a bug in one place but not all "identical" ones. When you see two variations of something, it's hard to know which one is the right pattern or if there's a good reason for the difference.

David Parnas introduced the idea of information hiding: a good module has a secret. By ensuring that a module keeps its secret, we usually reduce duplication.

Duplication

Symptoms

The easy version:
Two fragments of code look nearly identical.

The hard version:
Two fragments of code have nearly identical effects (at any conceptual level).

Causes

Some duplication occurs because programmers were working independently in different parts of the system, and they didn't realize that they were creating almost identical code. Sometimes people realize there's duplication, but they're too lazy to remove it. Other times, duplication will be hidden by other smells; once they're fixed, the duplication becomes more obvious.

A worse case (but perhaps the most common) occurs when the programmers intentionally duplicate code. They find some code that is "almost" right, so they copy-and-paste it into the new spot with some slight alterations.

What to Do

- If the duplication occurs because a special number, string, or other value recurs, use *Replace Magic Number with Symbolic Constant*.
- If the duplication is within a method or in two different methods in the same class: use *Extract Method* to pull the common part out into a separate method.
- If the duplication is within two sibling classes: use *Extract Method* to create a single routine, then *Pull Up Field* and/or *Pull Up Method* to bring the common parts together. Then you may be able to use *Form Template Method* to create a common algorithm in the parent, and unique steps in the children.
- If the duplication is in two unrelated classes: either extract the common part into a new class via *Extract Class*, or decide that the smell is Feature Envy so the common code really belongs on only one class or the other.
- In any of these cases, you may find that the two places aren't literally identical but have the same effect. Then you may do a *Substitute Algorithm* so that only one copy is involved.

Payoff

Reduces duplication, lowers size. Can lead to better abstractions and more flexible code.

Counter-Indications

Very rarely, you might decide that the duplication communicates significantly better, and leave it in place. Or you may have duplication that is only coincidental; folding the two places together would only confuse the reader.

Alternative Classes with Different Interfaces

Symptoms

Find two classes that seem to be doing the same thing but use different method names.

Causes

People created similar code to handle a similar situation, but didn't realize the other code existed.

What to Do

Harmonize classes so you can eliminate one of them.

1. *Rename Method* to make method names similar.
2. *Move Method, Add Parameter,* and *Parameterize Method* to make protocols (method signatures and approach) similar.
3. If the two classes are similar but not identical, use *Extract Superclass* once you have them reasonably well harmonized.
4. Remove the "extra" class if possible.

Payoff

Reduces duplication, may reduce size. Improves communication (removing ambiguity about which approach to use).

Counter-Indications

Sometimes the two classes can't be changed (e.g., if both are in different libraries). Each library may have its own vision for the same object, but leave you no good way to unify them.

DUP-1. Two libraries.

You're trying to integrate two modules from two different sources. Each module has its own logging approach.

System A:

```
package com.fubar.log;
public final class Log {
      public int INFO=1, WARN=2, ERROR=3, FATAL=4;
      public static void setLog(File f) {…}
      public static void log(int level, String msg) {…}
}
```

Calls to `Log.log` are sprinkled throughout the code.

System B:

```
package com.barf.logger;
public class Logger {
      public void informational(String msg) {…}
      public void informational(String msg, Exception e) {…}
      public void warning(String msg) {…}
      public void warning(String msg, Exception e) {…}
      public void fatal(String msg) {…}
      public void fatal(String msg, Exception e) {…}
}

public class LogFacility {
      public Logger makeLogger(String id) {…}
      public static void setOutput(OutputStream out) {…}
}
```

Objects that may need to log hold a Logger object.

Your long-term goal is to move to the new standard logging facility in JDK 1.4, but your environment doesn't support that yet.

A. What overall approach would you use to harmonize these classes with where you want to go? (Make sure to address the JDK 1.4 concern.)

B. Create a simple test for each logger, and implement the logger with the simplest approach you can.

C. Harmonize the classes so you can eliminate one of them. (Don't worry about the JDK 1.4 future yet.)

Solution ideas on page 176.

Challenges

DUP-2. Properties

```
public void getTimes(Properties props)
      throws Exception {
  String valueString;
  int value;

  valueString = props.getProperty("interval");
  if (valueString == null) {
    throw new MissingPropertiesException("monitor interval");
  }
  value = Integer.parseInt(valueString);

  if (value <= 0) {
    throw new MissingPropertiesException("monitor interval > 0");
  }
  checkInterval = value;

  valueString = props.getProperty("duration");
  if (valueString == null) {
    throw new MissingPropertiesException("duration");
  }
  value = Integer.parseInt(valueString);
  if (value <= 0) {
    throw new MissingPropertiesException("duration > 0");
  }
  if ((value % checkInterval) != 0) {
    throw new MissingPropertiesException("duration % checkInterval");
  }
  monitorTime = value;

  valueString = props.getProperty("departure");
  if (valueString == null) {
    throw new MissingPropertiesException("departure offset");
  }
  value = Integer.parseInt(valueString);
  if (value <= 0) {
    throw new MissingPropertiesException("departure > 0");
  }
  if ((value % checkInterval) != 0) {
    throw new MissingPropertiesException("departure % checkInterval”);
  }
  departureOffset = value;
}
```

A. How would you handle the duplication? Notice that determination of the `checkInterval` doesn't involve "%".

Solution ideas on page 176.

DUP-3. Template example (originally in *Extreme Programming Explored*)

```
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()");
}
```

A. What duplication do you see?

B. What would you do to remove the duplication?

C. One piece that repeats is a structure of the form

`new String(`some other string`)`

What does this code do? What does this have to do with the intern() **method on class String? Does it apply here?**

Solution on page 176.

DUP-4. Duplicate Observed Data

The refactoring *Duplicate Observed Data* works like this:

> If you have domain data in a widget-type object, move the domain data to a new domain object, and set up an observer so that the widget is notified of any changes in the domain object.

Thus, we've taken a situation where data was in one place, and not only have we duplicated it, we've added a need for synchronization between two objects.

A. Why is this duplication considered acceptable (desirable, even)?
(Hint: your answer should touch on the Observer or Model-View-Controller pattern.)

B. What are the performance implications of this approach?

Solution ideas on page 177.

DUP-5. Java Libraries

A. The Java libraries have several places where there is duplication. Describe some examples of this. They might be at a low, medium, or high level.

B. Why does this duplication exist? Is it worth it?

Solution ideas on page 177.

DUP-6. Points. Suppose you see these two classes:

```
public class Bird {
```

```
      // …
      public void move(Point vector) {
            x += vector.x % maxX;
            y += vector.y % maxY;
      }
}

public class Button {
      // …
      public void setPosition(Point p) {
            x = p.x;
            while (x >= maxX) x -= maxX;
            while (x < 0) x += maxX;

            y = p.y;
            while (y >= maxY) y -= maxY;
            while (y < 0) y += maxY;
      }

}
```

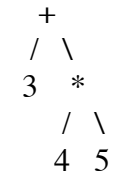
A. What is the duplication?

B. What could you do to eliminate duplication in these two classes?

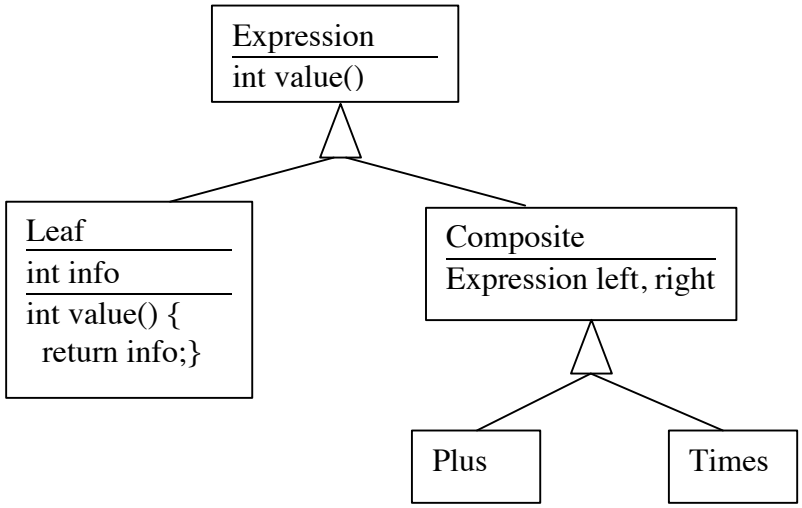
C. Sometimes, two versions of duplicated code are similar, but one has fixed a bug and the other hasn't. How can refactoring help you in this situation?

Solution ideas on page 178.

DUP-7. Expression. Suppose we have an expression that is structured in tree form. So the tree for "3+4*5" might look like this:



The classes might be structured like this:



The code for Plus.value():

```
public int value() {
      int v1 = left.value();
      int v2 = right.value();
      return v1 + v2;
}
```

The code for Times.value():

```
public int value() {
      int v1 = left.value();
      int v2 = right.value();
      return v1 * v2;
}
```

A. What steps would you take to reduce the duplication?

Solution ideas on page 178.

Conclusion

Duplication may not be the most obvious smell, but it's one of the most critical to address. Strive to make your code express each idea "once and only once."

Chapter 5. Data

Data are simple facts, divorced from information about what to do with them. “Data” has a dusty whiff about it, the old-fashioned ring of “data processing” or “data structures.” But data is a natural starting point for thinking about things. For example, we know we have a first name, middle name, and last name, so we create a Person class with that information. But objects are about data and behavior together.

Data-oriented objects are an opportunity. If they represent a good clustering, we’ll usually be able to find behavior that belongs on that object.

Primitive Obsession

Symptoms

Look for:

- uses of the primitive or near-primitive types (int, float, String, etc.)
- constants and enumerations representing small integers
- string constants representing field names

Causes

Several things can cause over-use of primitives:

- *Missing class:* Since almost all data must be in a primitive somewhere, it's easy to start with a primitive, and miss an opportunity to introduce a new object.
- *Simulated types:* Some primitives act as type codes, in effect simulating objects. A class may have started with one behavior, then picked up a boolean to say which of two behaviors are used, and then an enumeration of small values to say which of several behaviors to use. But by then, it may well be hiding a need for multiple classes.
- *Simulated field accessors:* Sometimes, a primitive is used as an index, to provide access to pseudo-fields in an array. Strings are occasionally used this way with HashTables or Maps.

What to Do

For missing objects:

- See Data Clump (this chapter), because the primitives can often be encapsulated by addressing that problem.
- *Replace Data Value with Object* to make data values "first-class."

For simulated types: an integer type code stands in for a class.

- If no behavior is conditional on the type code, then it is more like an enumeration, so *Replace Type Code with Class*.
- If the type code is immutable, and the class is not subclassed already, *Replace Type Code with Subclass*.
- If the type code changes or the class is subclassed already, *Replace Type Code with State/Strategy*. In *Refactoring*, there's an example of this transformation in the description of *Replace Conditional with Polymorphism*.

For simulated field accessors:

- If the primitive is used to treat certain array elements specially, *Replace Array with Object*.

Payoff

Improves communication, may expose duplication. Improves flexibility. Often exposes the need for other refactorings.

Counter-Indications

- Primitives that are really missing objects are so common that I hesitate to provide any excuses for not addressing it. But as for Data Clump, there are occasionally dependency or performance issues that stop you from fixing these.
- A Map can sometimes be used instead of an object with fixed fields, by using the names of fields as indices. This can reduce coupling to the structure of a simple object, at some cost in performance, type-safety, and clarity cost.

Notes

- A close relative of this problem occurs when ArrayList (or some other generic structure) is over-used.

DATA-1. Alternative Representations. Suggest two or three alternative representations for each of these types.

Money –

Position (in a list) –

Range –

Social Security Number (government identification number: "123-45-6789") –

Telephone number –

Street Address ("123 E. Main Street") –

ZIP (postal) code –

Solution on page 178.

DATA-2. A counter-argument. Consider a business application where a user enters a ZIP code (among other things), and it gets stored in a relational database. Someone argues: "It's not worth the bother of turning it into an object: when it gets written, it will just have to be turned into a primitive again." Why might it be worth creating the object in spite of the need for two conversions?

Solution ideas on page 179.

DATA-3. Iterator. How does an Iterator or Enumerator reduce primitive obsession?

Solution on page 179.

DATA-4. Consider this interface to an editor:

```
public class Editor {
   public void insert(String text) {…}
   public String contents(int n) {…}   // fetch n characters
   public void moveTo(int position) {…}
   public int position() {…}
   // more, omitted
}
```

and this sequence of calls:

```
   editor.insert("ba(nana)");
   int firstParendPosition = 2;
   editor.moveTo(firstParendPosition);
   assertEquals("(", editor.contents(1));

   editor.moveTo(1);
   editor.insert("x");  // Now: bxa(nana)
   editor.moveTo(firstParendPosition);
   assertEquals(___, editor.contents(1));
```

A. Given the interface provided, what string would you expect to use in place of the ___?

B. Based on the variable name (firstParendPosition), what string might you like instead? Of what use would this be?

C. The crux of the problem is the use of "int" as a position index. Suggest an alternative approach.

D. Relate your solution to the Memento design pattern (from the book *Design Patterns,* by Gamma et al.)

Solution on page 179.

Data Class

A class with public members exposes its implementation. This lets clients depend on the mutability and representation of the class.

Symptoms

The class consists only of public data members, or of simple getting and setting methods.

Causes

It's common for classes to begin like this: you realize that some data is part of an independent object, so you extract it out. But objects are about the commonality of *behavior*, and these objects aren't developed enough to have much behavior yet.

What to Do

1. *Encapsulate Field* to block direct access to the fields (allowing access only through getters and setters).
2. *Remove Setting Methods* for any methods you can.
3. *Encapsulate Collection* to remove direct access to any collection-type fields.
4. Look at each client of the object. Almost invariably, you'll find clients accessing the fields and manipulating the results, when the class could do it for them. (This is often a source of duplication, as many callers will tend to do the same things with the data.) On the client, use *Extract Method* to pull out the class-related code, then *Move Method* to put it over on the class.
5. After doing this a while, you may find that you have several similar methods on the class. Use *Rename Method, Extract Method, Add Parameter, Remove Parameter,* etc. to harmonize signatures and remove duplication.
6. Most access to the fields shouldn't be needed any more, as the moved methods cover the real use. So apply *Hide Method* to eliminate access to the getters and setters. (You may decide to keep them with private access, and have all internal access go through them.)

Payoff

Improves communication. May expose duplication (as you'll often find clients manipulating the fields in similar ways).

Counter-Indications

- There are times when the encapsulation of fields can have a performance cost. For example, consider a point with x and y coordinates. The interface (probably) isn't going to change, and people may deal with *lots* of points. So in Java, the Point class makes its fields public. (This saves the cost of a procedure call for each access.)

- Some persistence mechanisms rely on reflection to see fields or getter/setter methods, so they can decide what should be loaded or stored. For these classes, you may never be able to eliminate their "data class" nature. If I'm stuck with this, I try to treat these classes as Mementos (see *Design Patterns*). Sometimes I'll use another class as a layer over top of these persistence-only classes; that new class can benefit from all the changes described above, and it will hide the low-level classes.

DATA-5. Library classes. Compare these library classes. What do they have in common?

```
java.awt.Event
java.awt.GridBagConstraints
java.awt.Insets
java.awt.Point
java.awt.Polygon
java.awt.Rectangle
```

Solution ideas on page 180.

DATA-6. Color and Date. The classes java.awt.Color and java.util.Date are examples of classes encapsulated such that access to members is only through methods.

A. Propose at least two representations for each.

B. How does "no direct access to members" promote the ability of a class to be immutable?

Solution on page 180.

DATA-7. Proper names.
A. Clean up this "data class."

Person.java

```
public class Person {
    public String last;
    public String first;
    public String middle;

    public Person(String last, String first, String middle) {
        this.last = last;
        this.first = first;
        this.middle = middle;
    }
}
```

PersonClient.java

```
// The clients are in one file for convenience;
// imagine them as separate files.

import junit.framework.TestCase;

import java.io.*

public class PersonClient extends TestCase {
    public PersonClient(String name) {super(name);}

    public void client1(Writer out, Person person) throws IOException {
        out.write(person.first);
        out.write(" ");
        if (person.middle != null) {
            out.write(person.middle);
            out.write(" ");
        }
        out.write(person.last);
    }

    public String client2(Person person) {
        String result = person.last + ", " + person.first;
        if (person.middle != null)
            result += " " + person.middle;
        return result;
    }

    public void client3(Writer out, Person person) throws IOException {
        out.write(person.last);
        out.write(", ");
        out.write(person.first);
        if (person.middle != null) {
            out.write(" ");
            out.write(person.middle);
```

```
        }
    }

    public String client4(Person person) {
        return person.last + ", " +
               person.first +
               ((person.middle == null) ? "" : " " + person.middle);
    }

    public void testClients() throws IOException {
        Person bobSmith = new Person("Smith", "Bob", null);
        Person jennyJJones = new Person("Jones", "Jenny", "J");

        StringWriter out = new StringWriter();
        client1(out, bobSmith);
        assertEquals("Bob Smith", out.toString());

        out = new StringWriter();
        client1(out, jennyJJones);
        assertEquals("Jenny J Jones", out.toString());

        assertEquals("Smith, Bob", client2(bobSmith));
        assertEquals("Jones, Jenny J", client2(jennyJJones));

        out = new StringWriter();
        client3(out, bobSmith);
        assertEquals("Smith, Bob", out.toString());

        out = new StringWriter();
        client3(out, jennyJJones);
        assertEquals("Jones, Jenny J", out.toString());

        assertEquals("Smith, Bob", client4(bobSmith));
        assertEquals("Jones, Jenny J", client4(jennyJJones));
    }
}
```

B. There's a new requirement: to support people only one name (say, Cher or Madonna), or someone with several last names (Oscar de los Santos)? Compare the difficulty of this change before and after your refactoring in the previous part.

Data Clump

Symptoms

The same two or three items frequently appear together in classes and parameter lists.

You also get a whiff of this when people declare some fields, then methods that work with those fields, then more fields and more methods, etc. (That is, there are groups of fields and methods together within the class.)

You may see groups of field names that start or end with similar substrings.

Causes

The values are typically part of some other entity, but no one has yet had the insight to realize that there's a missing class. Or sometimes, people know the object is missing, but think it's too small or unimportant to stand alone.

(Identifying these classes is often a major step toward simplifying objects, and it often helps you generalize classes more easily.)

What to Do

- If the values are fields in a class, use *Extract Class* to pull them into a new class.
- If the values are together in method calls, *Introduce Parameter Object* to extract the new object.
- Look at calls that pass around the values from the new object, and see if they can *Preserve Whole Object* instead.
- Look at uses of the values; there are often opportunities to *Move Method* etc. to move those uses into the new object (as you would to address the Data Class smell).

Payoff

Improves communication, may expose duplication, usually reduces size.

Counter-Indications

- Occasionally, passing the whole object will introduce a dependency you don't want (as lower-level classes get exposed to the whole new object instead of just its values). So you may pass in the pieces to prevent this dependency.

- Very rarely, there is a measured performance problem solved by passing in the "parts" of the object instead of the object itself. Recognize that this is a compromise in the object model for performance. Such code is worth commenting!

Interlude 2 – Opposites

When we refactor, we're trying to respond to the forces affecting code. Sometimes, what was a good change today no longer looks so good tomorrow, and we find ourselves reversing a refactoring.

I2-1. Opposites. The next two pages present a list of refactorings. Match each refactoring to the one that undoes it. (Mark a "*" next to any refactoring with no opposite present.)

___ Add Parameter	1. Add Parameter
___ Change Bidirectional Association to Unidirectional	2. Change Bidirectional Association to Unidirectional
___ Change Reference to Value	3. Change Reference to Value
___ Change Unidirectional Association to Bidirectional	4. Change Unidirectional Association to Bidirectional
___ Change Value to Reference	5. Change Value to Reference
___ Collapse Hierarchy	6. Collapse Hierarchy
___ Consolidate Conditional Expression	7. Consolidate Conditional Expression
___ Consolidate Duplicate Conditional Fragments	8. Consolidate Duplicate Conditional Fragments
___ Convert Procedural Design to Objects	9. Convert Procedural Design to Objects
___ Decompose Conditional	10. Decompose Conditional
___ Duplicate Observed Data	11. Duplicate Observed Data
___ Encapsulate Collection	12. Encapsulate Collection
___ Encapsulate Downcast	13. Encapsulate Downcast
___ Encapsulate Field	14. Encapsulate Field
___ Extract Class	15. Extract Class
___ Extract Hierarchy	16. Extract Hierarchy
___ Extract Interface	17. Extract Interface
___ Extract Method	18. Extract Method
___ Extract Subclass	19. Extract Subclass
___ Extract Superclass	20. Extract Superclass
___ Form Template Method	21. Form Template Method
___ Hide Delegate	22. Hide Delegate
___ Hide Method	23. Hide Method
___ Inline Class	24. Inline Class
___ Inline Method	25. Inline Method
___ Inline Temp	26. Inline Temp
___ Introduce Assertion	27. Introduce Assertion
___ Introduce Explaining Variable	28. Introduce Explaining Variable
___ Introduce Foreign Method	29. Introduce Foreign Method
___ Introduce Local Extension	30. Introduce Local Extension
___ Introduce Null Object	31. Introduce Null Object
___ Introduce Parameter Object	32. Introduce Parameter Object
___ Move Field	33. Move Field
___ Move Method	34. Move Method
___ Parameterize Method	35. Parameterize Method
___ Preserve Whole Object	36. Preserve Whole Object
___ Pull Up Constructor Body	37. Pull Up Constructor Body
___ Pull Up Field	38. Pull Up Field
___ Pull Up Method	39. Pull Up Method
___ Push Down Field	40. Push Down Field
___ Push Down Method	

___ Remove Assignment to Parameters ___ Remove Control Flag ___ Remove Middle Man ___ Remove Parameter ___ Remove Setting Method ___ Rename Method ___ Replace Array with Object ___ Replace Conditional with Polymorphism ___ Replace Constructor with Factory Method ___ Replace Data Value with Object ___ Replace Delegation with Inheritance ___ Replace Error Code with Exception ___ Replace Exception with Test ___ Replace Inheritance with Delegation ___ Replace Magic Number with Symbolic Constant ___ Replace Method with Method Object ___ Replace Nested Conditional with Guard Clause ___ Replace Parameter with Explicit Methods ___ Replace Parameter with Method ___ Replace Record with Data Class ___ Replace Subclass with Fields ___ Replace Temp with Query ___ Replace Type Code with Class ___ Replace Type Code with State/Strategy ___ Replace Type Code with Subclasses ___ Self Encapsulate Field ___ Separate Domain from Presentation ___ Split Temporary Variable ___ Substitute Algorithm ___ Tease Apart Inheritance	41. Push Down Method 42. Remove Assignment to Parameters 43. Remove Control Flag 44. Remove Middle Man 45. Remove Parameter 46. Remove Setting Method 47. Rename Method 48. Replace Array with Object 49. Replace Conditional with Polymorphism 50. Replace Constructor with Factory Method 51. Replace Data Value with Object 52. Replace Delegation with Inheritance 53. Replace Error Code with Exception 54. Replace Exception with Test 55. Replace Inheritance with Delegation 56. Replace Magic Number with Symbolic Constant 57. Replace Method with Method Object 58. Replace Nested Conditional with Guard Clause 59. Replace Parameter with Explicit Methods 60. Replace Parameter with Method 61. Replace Record with Data Class 62. Replace Subclass with Fields 63. Replace Temp with Query 64. Replace Type Code with Class 65. Replace Type Code with State/Strategy 66. Replace Type Code with Subclasses 67. Self Encapsulate Field 68. Separate Domain from Presentation 69. Split Temporary Variable 70. Substitute Algorithm 71. Tease Apart Inheritance

Chapter 6. Interfaces

“Any way you want it, that’s the way you need it.”—Journey

A well-designed and consistent interface (protocol) makes it easier for a client to use a class well.

Incomplete Library Class

Symptoms

You're using a library class, and there's a feature you wish were on that class, but it's not. If it were a "normal" class, you'd modify it, but since somebody else owns the library, you can't change it.

Causes

The author of the library class didn't anticipate your need (or declined to support it due to other tradeoffs).

What to Do

- See if the owner of the class or library will consider adding the support you want.
- If it's just one or two methods, *Introduce Foreign Method* on a client of the library class. (This is still Feature Envy, but what can you do?)
- If you have several methods, *Introduce Local Extension* (adding a new pseudo-library class).
- You may decide to introduce a layer "covering" the library.

Payoff

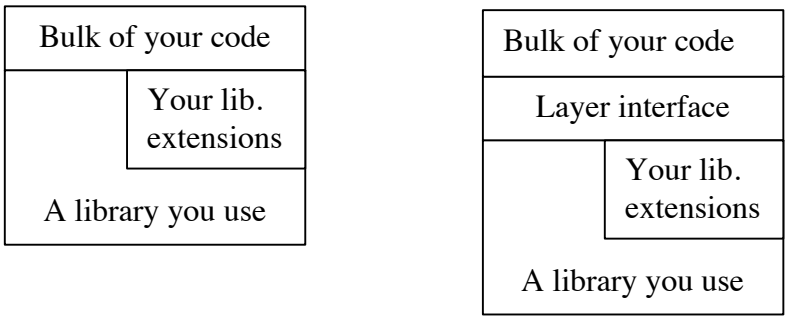
Reduces duplication (when you can re-use library code instead of implementing it completely from scratch).

Counter-Indications

If several projects each use incompatible ways to extend a library, this can lead to extra work if the library changes.

INT-1. Layers.

One way to deal with libraries is to put them beneath a layer. This lets you isolate the bulk of your code from direct dependency on other libraries. Consider these two alternatives:



A. Redraw this as a UML package diagram showing dependencies.

B. Explain how the "bulk of your code" does or does not depend on the library code in each of these situations.

C. What effects does this layering have in terms of:

- **Performance?**
- **Conceptual integrity?**
- **Portability?**
- **Testing?**

D. What mechanisms do you have available to enforce the layering? (That is, what stops someone from turning the second approach into the first one?)

INT-2. Trees.

The Swing tree library (see javax.swing.tree.*) has an interesting flaw. This library has the idea of a DefaultTreeModel (class) built out of MutableTreeNodes (interface) objects. Although there is a DefaultMutableTreeNode object, you might not want to use it if you already have node-like objects. (You might prefer to implement the MutableTreeNode interface rather than copy your information to a DefaultMutableTreeNode.)

The flaw in the library is a gap in the MutableTreeNode interface: it supports a method `setUserObject()`, but no method `getUserObject()`.

Suppose you'd like to have this method.

A. Does Introduce Foreign Method or Introduce Local Extension seem more appropriate? Why?

Solution on page 180.

INT-3. String

A. The String class in Java is declared "final." What effect does this have on *Introduce Local Extension*?

B. Why would the creators of Java make String final?

Solution ideas on page 180.

Refused Bequest

Symptoms

- A class inherits from a parent, but just throws an exception instead of supporting a method (honest refusal), or
- A class inherits from a parent, but an inherited routine just doesn't work when called on the class (implicit refusal), or
- Clients tend to access the class through a handle to the subclass, rather than a handle to the parent, or
- Inheritance doesn't really make sense; the subclass just isn't an example of the parent.

Causes

Perhaps someone inherited from a class just for implementation convenience without really intending the class to be substitutable for the parent. Or, there may have been a conscious decision to let subclasses deny use of some features, to prevent an explosion of types for all feature combinations.

What to Do

- If it's not confusing, you might decide to leave as is.
- If there's no reason to share a class relationship, then *Replace Inheritance with Delegation*.
- If the parent-child relationship does make sense, you can create a new child class via *Extract Subclass, Push Down Field,* and *Push Down Method;* let this class holds the non-refused behavior, and change clients of the parent to be clients of the new class. (Then the parent need not mention the feature.) You may be able to eliminate the refused methods from the original class and the parent.

Payoff

Improves communication. Improves testability (as you don't have to worry about different capabilities up and down the class hierarchy).

Counter-Indications

Sometimes this is used to prevent an explosion of new types. For example, in Java, Stack inherits from Vector. You really shouldn't change the contents of the middle of a stack, but since clients mostly use it as a stack and not a vector, it's not a big problem. (Stack doesn't explicitly refuse the vector-changing operations, but even if it did, clients that use only the stack methods (most clients?) would be unaffected.)

INT-4. Collection classes.

The Collection classes in Java don't have a separate hierarchy for read-only collections; they just provide a "wrapper" that will refuse to make changes to its underlying collection.

A. Which approach to spotting this applies here?

B. Explain why a class that uses a read-only collection doesn't have to catch or declare an exception.

C. Do you agree with the library designers' approach? What other approach could they have used?

INT-5. Refused Bequest.

Find an example of a Refused Bequest in code you have access to.

More Challenges

INT-6. Filter.

The Collections class in java.util defines a number of utility methods, including one to produce an enumeration for a collection. But the provided enumerator just gives a list of all elements, without providing a way to filter them.

A. Write a class to wrap Enumerator, extending it with the ability to return objects that meet some criteria. (Include tests.)

B. What parent class or interface does your class have (if any)?

C. What arguments does the constructor take (if any)?

D. What methods does it define?

Solution ideas on page 180.

INT-7. Diagrams.

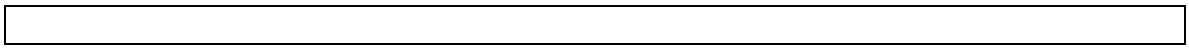
Draw class or other UML diagrams to help explain and show the difference between *Introduce Foreign Method*, *Introduce Local Extension*, layering, and wrapping.

INT-8. A missing function.

Consider the Zumbacker Z function, at the core of your application. (In fact, it's such a commonly used function in your domain that you're a little surprised it's not in the Java math libraries already.) It's defined:

$$Z(x) = \text{abs}(\cos(x) + \sin(x) - \exp(x))$$

How could you handle the problem of java.lang.Math being an incomplete library?



Chapter 7. Responsibility

"I do perceive her a divided duty."—*Othello*, Act I, Scene 3.

Many problems are fairly easy to find and fix: long methods, overused primitives, and obvious duplication. One thing you're left with is deciding what should be the balance of responsibility between objects.

This is a tough challenge. Tools such as design patterns or CRC cards can help (see http://www.c2.com/cgi/wiki?CrcCards), but in the end, the challenge remains.

Feature Envy

Symptoms

You see a method in a class that seems to be focused on manipulating the data of another class rather than itself. (You may notice this because of duplication—several clients do the same manipulation.)

Causes

This is very common among clients of current and former data classes, but you can see it for any class and its clients.

What to Do

- *Move Method* to put the actions on the correct class. (You may have to *Extract Method* first to isolate the misplaced part.)

Payoff

Reduces duplication, and often improves communication (since code is where people would expect it).

Counter-Indications

Sometimes, behavior is intentionally put on the "wrong" class. For example, some design patterns, such as Strategy or Visitor, pull behavior to a separate class so it can be independently changed. If you *Move Method* to put it back, you can end up putting things together that should change separately.

RESP-1. Feature Envy.

Look back at the MEASURE-2 exercise (page 18). In `Report.report()`, notice how the information being printed is obtained by looking inside a Machine or Robot's fields. Fix these two examples of feature envy.

Inappropriate Intimacy

Symptoms

One class accesses internal ("should-be-private") parts of another class. This can happen between independent classes, or between a subclass and its parent.

Causes

The two classes probably became intertwined a little at a time. By the time you realize there's a problem, they're coupled. There may be a missing class that should mediate between them.

What to Do

- If two independent classes are entangled, use *Move Method* and *Move Field* to put the right pieces on the right class.
- If the tangled part seems to be a missing class, use *Extract Class* and *Hide Delegate* to introduce the new class.
- If classes point to each other, use *Change Bidirectional Reference to Unidirectional* to turn it into a one-way dependency.
- If a subclass is too coupled, *Replace Inheritance with Delegation*.

Payoff

Reduces duplication, often improves communication, may reduce size.

Counter-Indications

None identified.

RESP-2. Inappropriate Intimacy.
Consider the code used in RESP-1 (page 68) and MEASURE-2 (page 18).

A. How is walking the list of machines a case of inappropriate intimacy?

B. Address this by extracting a new class. Make sure it has a `report()` method.

Divergent Change

Symptoms

You find yourself changing the same class for different reasons. (For contrast, see Shotgun Surgery, next.)

Causes

The class picked up more responsibilities as it evolved, with no one noticing that two different types of decisions were involved. [Ref. Parnas TBD]

What to Do

- If the class "finds an object" and "does something with it," let the caller find the object and pass it in, or let the class return a value that the caller uses.
- *Extract Class* to pull out separate classes for the separate decisions.
- If several classes are sharing the same type of decisions, you may be able to consolidate those new classes (e.g., by *Extract Superclass* or *Extract Subclass*). In the limit, these classes can form a layer (e.g., a persistence layer).

Payoff

Reduces duplication. Improves communication (by expressing intent better).

Counter-Indications

None identified.

RESP-3. CsvWriter. Consider this code:

CsvWriter.java

```
public class CsvWriter {
    public CsvWriter() {}

    public void write(String[][] lines) {
        for (int i = 0; i < lines.length; i++)
            writeLine(lines[i]);
    }

    private void writeLine(String[] fields) {
        if (fields.length == 0)
            System.out.println();
        else {
            writeField(fields[0]);

            for (int i = 1; i < fields.length; i++) {
                System.out.print(",");
                writeField(fields[i]);
            }
```

```
            System.out.println();
        }
    }

    private void writeField(String field) {
        if (field.indexOf(',') != -1 || field.indexOf('\"') != -1)
            writeQuoted(field);
        else
            System.out.print(field);
    }

    private void writeQuoted(String field) {
        System.out.print('\"');
        for (int i = 0; i < field.length(); i++) {
            char c = field.charAt(i);
            if (c == '\"')
                System.out.print("\"\"");
            else
                System.out.print(c);
        }
        System.out.print('\"');
    }
}
```

CsvWriterTest.java (A Manual Test)

```
import junit.framework.TestCase;

public class CsvWriterTest extends TestCase {
    public CsvWriterTest (String name) {
        super(name);
    }

    public void testWriter() {
        CsvWriter writer = new CsvWriter();
        String[] [] lines = new String[] [] {
  new String[] {},
  new String[] {"only one field"},
  new String[] {"two", "fields"},
  new String[] {"", "contents", "several words included"},
  new String[] {",", "embedded , commas, included", "trailing comma,"},
  new String[] {"\"", "embedded \" quotes",
                                  "multiple \"\"\" quotes\"\""},
  new String[] {"mixed commas, and \"quotes\"", "simple field"}
        };

        // Expected:
        //                       -- (empty line)
        // only one field
        // two,fields
        // ,contents,several words included
        // ",","embedded , commas, included","trailing comma,"
        // """","embedded "" quotes","multiple """""" quotes"""""
        // "mixed commas, and ""quotes""",simple field
```

```
        writer.write(lines);
    }
}
```

A. How is this code an example of divergent change? (What decisions does it embody?)

B. Modify this code to write to a stream passed in as an argument.

C. Starting from the original code, modify the functions to return a string value corresponding to what the functions would have written.

D. Which version seems better, and why? Which is easier to test?

Solution ideas on page 180.

RESP-4. CheckingAccount. Consider these class fragments:

```
public class CheckingAccount {
      public int balance() {…}
      public void add(Transaction transaction) {…}
}

public class Transaction {
      public int cost() {…}
}
```

A. What decisions are embedded in these classes?

B. Extract a simple Money class.

Solution ideas on page 180.

Shotgun Surgery

Symptoms

You want to make a "simple" change, but you have to touch several classes to do so.

Causes

One responsibility has been split among several classes. There may be a missing class that would understand the whole responsibility (and which would get a cluster of changes). Or, this can happen through an over-zealous attempt to eliminate Divergent Change.

What to Do

- Identify the class that should own the group of changes. It may be an existing class, or you may need to *Extract Class* to create a new one.
- Use *Move Field* and *Move Method* to put the functionality onto the chosen class. Once the "other" class is simple enough, you may be able to *Inline Class* to eliminate that class.

Payoff

Reduces duplication. Improves communication. Improves maintainability (as future changes will be more localized).

Counter-Indications

None identified.

RESP-5. Shotgun Surgery.

In code you have access to, find examples of this problem. Some frequent candidates:

- **configuration information**
- **logging**
- **persistence**
- **sometimes it takes two calls on an object to get something common done, and this "two-step" is used in several places.**

Parallel Inheritance Hierarchies

Symptoms

- You make a new subclass in one hierarchy, and find yourself required to create a related subclass in another hierarchy.
- You find two hierarchies, where the subclasses have the same prefix. (The naming reflects the requirement to coordinate hierarchies.)

Causes

The hierarchies probably grew in parallel, a class and its pair being needed at the same time. As usual, it probably wasn't bad at first, but after two or more pairs get introduced, it becomes too complicated to change one thing. (Often both classes embody different aspects of the same decision.)

What to Do

- Use *Move Field* and *Move Method* to re-distribute the features in such a way that you can eliminate one of the hierarchies.

Payoff

Reduces duplication, may improve communication, may reduce size.

Counter-Indications

None identified. (This smell may happen along the way from improving a particularly tangled situation.)

RESP-6. Duplicate Observed Data.

Duplicate Observed Data **splits a class in two, one part model, the other part view. (For example, it might turn Card into CardModel and CardView.) It is often natural for the model classes to form a hierarchy (they have similar notification needs), and it's natural for the views to form a hierarchy (they all display). This sounds like a parallel inheritance hierarchy. Is it a problem?**

Solution on page 181.

Combinatorial Explosion

This is a relative of Parallel Inheritance Hierarchies, but everything has been folded into one hierarchy.

Symptoms

- You want to introduce a single new class, but it turns out that you have to introduce multiple versions in various points of the hierarchy. –or–
- You notice that each layer of the hierarchy uses a common set of words (e.g., one level adds "style" information, and the next adds mutability).

Causes

What should be independent decisions instead get implemented via a hierarchy.

What to Do

- If things aren't too far gone, you may be able to *Replace Inheritance with Delegation*. (By keeping the same interface for the variants, you can create an example of the Decorator design pattern.)
- If the situation has grown too complex, you're in "big refactoring" territory, and can *Tease Apart Inheritance*.

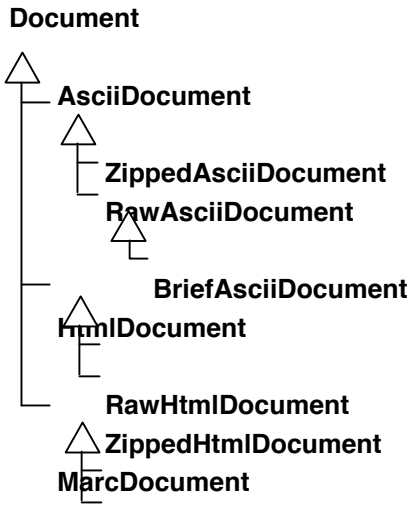
Payoff

Reduces duplication. Reduces size.

Counter-Indications

None identified.

RESP-7. Documents. Consider this hierarchy:



BriefMarcDocument

FullMarcDocument

A. What's the impact of adding a new compression type that all document types will support?

B. Rearrange the hierarchy so it's based first on compression (or none), then brief/full, then document type. Is this an improvement?

C. Turn this hierarchy into an instance of the Decorator pattern (with Document as the basic element).

Solution ideas on page 181.

Chapter 8. Unnecessary Complexity

"You aren't gonna need it."–Ron Jeffries

Sometimes, code has gotten complicated for "historical reasons" but no longer needs the complexity. Remove these problems when you run into them.

Dead Code

Symptoms

- A variable, parameter, field, method, or class is not used anywhere (perhaps other than tests).

(This can be hard to detect without tool support. Once your suspicions are alerted, you can do things like global search for "new classname.")

Causes

- Complicated logic resulted in some combinations of conditions that can't actually happen; you'll see this when simplifying conditionals.
- Requirements have changed, or new approaches were introduced, without adequate cleanup.

What to Do

Delete the unused code and any associated tests.

- For an unnecessary class:
    - If parents or children of the class seem like the right place for its behavior, fold it into one of them via *Collapse Hierarchy*.
    - Otherwise, fold its behavior into its caller via *Inline Class*.
- For an unnecessary method: Ensure there are no references, and remove it.
- For an unnecessary field: Ensure there are no references, and remove it.
- For an unnecessary parameter: *Remove Parameter*.
- For an unused variable: Remove it.

Payoff

Reduces size. Improves communication. Improves simplicity.

Counter-Indications

- If your application is a framework, you may have elements present to support clients' needs that strictly speaking aren't needed by the framework itself. For example, a class may have an empty "hook" method that will be called by (not-yet-existing) subclasses.

Lazy Class

Symptoms

- A class isn't doing much: its parents, children, or callers seem to be doing all the associated work, and there isn't enough behavior left in the class to justify its continued existence.

Causes

Typically, all the class' responsibilities were moved to other places in the course of refactoring.

What to Do

- If parents or children of the class seem like the right place for its behavior, fold it into one of them via *Collapse Hierarchy*.
- Otherwise, fold its behavior into its caller via *Inline Class*.

Payoff

Reduces size. Improves communication. Improves simplicity.

Counter-Indications

- Sometimes a lazy class is present to communicate intent. You may have to balance communication versus simplicity.

Speculative Generality

Symptoms

- You find unused classes, methods, methods, fields, parameters, etc. They may have no users or only tests.

Causes

The class may have build built with the expectation that it would become more useful, but never did. When people try to out-guess the needs of the code, they often add things "for generality" or "for completeness" that never end up being used. Sometimes, the code was used before, but is no longer needed because of new or revised ways of doing things. ("Speculative Generality" is "Dead Code" created on purpose.)

What to Do

- For an unnecessary class:
    - If parents or children of the class seem like the right place for its behavior, fold it into one of them via *Collapse Hierarchy*.
    - Otherwise, fold its behavior into its caller via *Inline Class*.
- For an unnecessary method: *Inline Method* or *Remove Method*.
- For an unnecessary field: Ensure there are no references, and remove it.
- For an unnecessary parameter: *Remove Parameter*.

Payoff

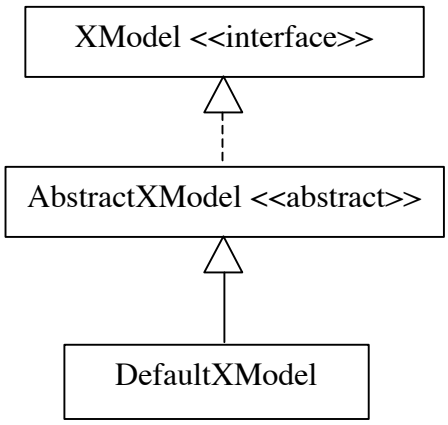
Reduces size. Improves communication. Improves simplicity.

Counter-Indications

- If your application is a framework, you may have elements present to support clients' needs that strictly speaking aren't needed by the framework itself. For example, a class may have an empty "hook" method that will be called by (not-yet-existing) subclasses.
- Some elements are used by test methods, but they're exposed as "test probe points" to allow a test to have privileged information about the class.

Challenges

YAGNI-1. Swing libraries. The Swing libraries have a pattern:



The AbstractXModel is the only implementation of the interface, and the DefaultXModel is the only subclass of the AbstractXModel. (Typically the abstract model has the notification handling built in, but may support other things.)

A. What advantages are there from having the model be an interface, separate from the AbstractXModel class?

B. Why is this not an example of a lazy class?

C. Should you adopt this three-class structure for your objects?

D. You'll sometimes see the idea of introducing an interface to "break dependencies." How does it do that?

E. In Swing, JTree is in the javax.swing package, and TreeModel, AbstractTreeModel, and DefaultTreeModel are in the javax.swing.tree package. (The table classes have a similar organization.) The interface does break dependencies between the classes, but there are still dependencies at the package level. Show a dependency diagram between the packages, and show a way to reassign the classes to packages so that concrete things depend more on abstract things than vice versa.

Solution ideas on page 181.

Chapter 9. Message Calls

"Call me any, any time."—Blondie

It's easy to think of refactoring as "this is what you do to fix this problem." But it's subtler than that: most refactorings are reversible, and they often trade off two or more good things.

A nice example of this is Message Chains vs. Middle Man. Sometimes there's a way to improve both at the same time, but many times it's a balancing act between them.

Message Chains

Symptoms

You see calls of the form:

```
a.b().c().d()
```

(This may happen directly, or through intermediate results.)

Causes

An object cooperates with other objects to get things done. That part is OK; the problem is that we're coupled both to the objects *and* the path to get to them.

This sort of coupling goes against two maxims of object-oriented programming: "Tell, Don't Ask" and the Law of Demeter. "Tell, Don't Ask," says that instead of *asking* for objects so you can manipulate them, *tell* them to just do the manipulation for you. It's phrased even more clearly in the Law of Demeter: a method shouldn't talk to strangers. That is, it should only talk to itself, its arguments, its own fields, or objects it creates. (Hunt and Thomas' *The Pragmatic Programmer* describes both these rules in more detail.)

What to Do

- If the manipulations actually belong on the target object (the one at the end of the chain), use *Extract Method* and *Move Method* to put them there.
- Use *Hide Delegate* to make the method depend on one object only. (So, rather than a.b().c().d(), put a d() method on the a object.)

Payoff

May reduce or expose duplication.

Counter-Indications

This is a "tradeoff" refactoring. If you apply *Hide Delegate* too much, you get to the point where everything's so busy delegating that nothing seems to be doing any actual work. Sometimes, it's just easier and less confusing to call a small chain.

Middle Man

Symptoms

Most methods of a class call the same or a similar method on another object:

```
f() {delegate.f();}
```

Causes

One way this happens is from applying Hide Delegate to address Message Chains. Perhaps other features have moved out since then, and you're left with mostly delegating methods.

What to Do

- In general, *Remove Middle Man* by having the client call the delegate directly.
- If the delegate is owned by the middle man or is immutable, and the middle man has behavior to add, and the middle man can be seen as an example of the delegate, you might *Replace Delegation with Inheritance*.

Payoff

Reduces size, may improve communication.

Counter-Indications

- Some patterns (e.g., Proxy, Decorator) intentionally create delegates. Don't remove a middle man that's there for a reason.
- Middle Man and Message Chain trade off against each other.
- Delegates provide a sort of Façade, letting a caller remain unaware of details of messages and structures. Removing a middle man can expose clients to more information than they should know.

MESSAGE-1. Middle Man. Consider these classes:

(This code has been greatly simplified; the queue wasn't originally storing strings.)

Queue.java

```
import java.util.ArrayList;

public class Queue {
  ArrayList delegate = new ArrayList();
  public Queue() {}
  public void addRear(String s) {delegate.add(s);}
  public int getSize() {return delegate.size();}

  public String removeFront() {
      String result = delegate.get(0).toString();
      delegate.remove(0);
```

```
            return result;
        }
    }
```

QueueTest.java (selection)

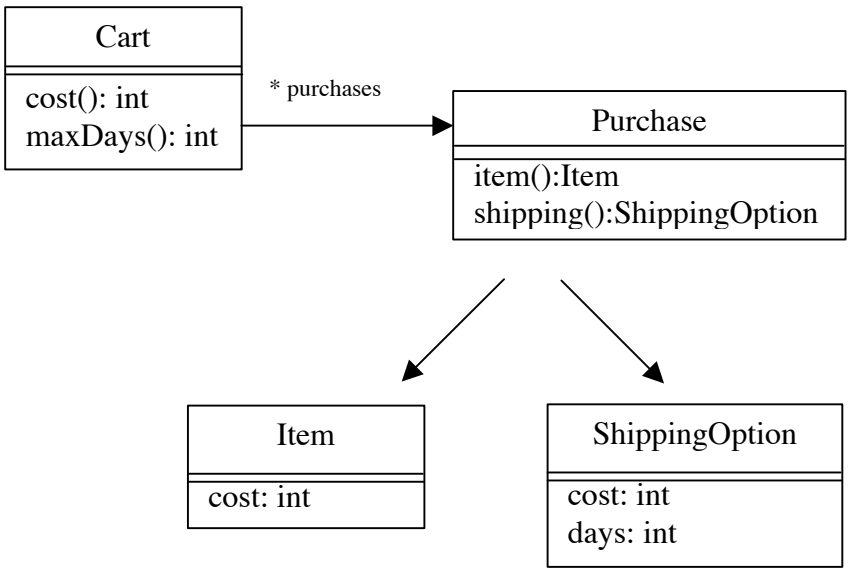
```
    public void testQ() {
      Queue q = new Queue();
      q.addRear("E1");
      q.addRear("E2");
      assertEquals("E1", q.removeFront());
      assertEquals("E2", q.removeFront());
      assertEquals(0, q.getSize());
    }
```

A. *Remove Middle Man* so that the queue is no longer a middle man for the ArrayList. Is this an improvement?

B. Put the middle man back in via *Hide Delegate*.

Solution ideas on page 181.

MESSAGE-2. Cart. Consider these classes:



Here is Cart.cost():

```
public int cost() {
      int total = 0;
      for (int i=0; i < purchases.size(); i++) {
            Purchase p = (Purchase) purchases.elementAt(i);
            total += p.item().cost + p.shipping().cost;
      }
      return total;
}
```

A. Write the implied classes (and tests). The `maxDays()` method computes the largest number of days for any ShippingOption in the purchase.

B. Apply *Hide Delegate* so Cart accesses only Purchase.

C. *Hide Delegate* causes the middle man (Purchase) class to have a wider interface; that is, it exposes more methods. But applying that refactoring can open up a way to make the interface narrower. Explain this apparent contradiction.

D. Use this line of reasoning to narrow the Purchase interface.

E. Notice that primitive the "int" type is used to represent money. Would it be easier to introduce a Money class before or after the delegate changes?

Interlude 3 – Design Patterns

I3-1. Patterns.

Following is a list of design patterns described in Gamma et al.'s *Design Patterns.* What refactorings might you use to evolve to some of these patterns?

Creational patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Solution ideas on page 181.

Chapter 10. Conditional Logic

"If wishes were horses, beggars would ride."

Independently of the object structure, code has opportunities to be good or bad. Conditional logic seems to be the most challenging part: we have to worry about the condition, what it does on this branch, and what it does on that branch. Furthermore, the condition tends to grow toward complexity as we come up with more and more exceptions to a rule.

Sometimes we find conditional logic used as a replacement for object structure.

Null Check

Symptoms

There are repeated occurrences of code like this:

```
if (xxx == null) …
```

Causes

Someone decided, "We'll use null to mean the default." It may have let them avoid the trouble of initializing certain fields, or of creating certain objects, or it may have been an afterthought for an unexpected case. It may have been introduced to work around a null-pointer bug (without addressing the underlying cause).

What to Do

Introduce Null Object to create a "default" object that you explicitly use.

Payoff

Reduces duplication. Reduces logic errors and exceptions.

Counter-Indications

- If the null check occurs in only one place (e.g., in a factory method), it may not be worth the effort to create a separate Null Object.
- Null objects need to act like "identity" objects (as 0 does relative to addition). If you can't define a safe behavior or value for each method, you may not be able to use a Null Object.
- Watch out for a case where null means two or more different things in different contexts. (You may be able to support this with different Null Objects).

COND-1. Null object.

A. Look back at the example in MEASURE-2 (page 18). Introduce Null Object where you can.

B. Some of the null checks are checks for null strings. One approach would be to use empty strings instead. What are the down sides of this approach (taking into account all the other client classes you don't see here)?

C. What's another approach to this problem?

Solution ideas on page 182.

Complicated Boolean Expressions

Symptoms

Code has complex conditions involving "and," "or," and "not."

Causes

The code may have started out complicated, or it may have picked up additional conditions along the way.

What to Do

- Apply *DeMorgan's Law*:
  ! (a && b) => (!a) || (!b)
  and
  !(a || b) => (!a) && (!b)

You may find that some variables will communicate better if they change names to reflect their flipped sense.

- *Introduce Explaining Variable* to make each clause clearer.
- Use guard clauses to "peel off" certain conditions; the remaining clauses get simpler.

Payoff

Improves communication.

Counter-Indications

You may be able to find other ways to simplify the expressions, or you may find that the re-written expression communicates less well.

COND-2. Conditional Expression.
Consider this code fragment:

```
if (!((score > 700) ||
   ((income >= 40000) && (income <= 100000)
  && authorized && (score > 500)) ||
   (income > 100000)))
   reject();
else
   accept();
```

A. Apply DeMorgan's Law to simplify this as much as possible.

B. Starting from the original, rewrite the condition by introducing explaining variables.

C. Starting from the original, flip the "if" and "else" clauses, then break it into several "if" clauses. (You'll call `accept()` in three different places.)

D. Which approach was the simplest? The clearest? Can you combine the techniques?

E. Describe the conditions in table form. The rows and columns should be based on three variables: one for the three score ranges, one for the income ranges, and one for the authorized flag. The cells should say either "accept" or "reject."

Solution ideas on page 182.

Special Case

Symptoms

- Complex "if" statements
- Checks for particular values before doing work (especially comparisons to constants or enumerations)

Causes

Someone realized a special case was needed.

What to Do

- If the conditionals are taking the place of polymorphism, *Replace Conditional with Polymorphism*.
- If the "if" and "then" clauses are similar, you may be able to rewrite them so they have the proper value in both cases; then the conditional can be eliminated.

Payoff

Improves communication, may expose duplication.

Counter-Indications

- In a recursive definition, there is always a "base case" that will stop the recursion; you can't expect to eliminate these.
- Sometimes an "if" is just the simplest way to do something.

Switch Statement

Symptoms

- Locate uses of the `switch` statement.
- Locate methods with several `if` statements in a row

Causes

It's often caused by laziness in introducing new classes. The first time conditional behavior is needed, a `switch` statement is used rather than a new class. It's not a big problem at this point, because it only occurs once. Then you need another condition based on the same type code, and introduce a second `switch`, instead of fixing the lack of polymorphism.

Sometimes the lack of polymorphism is hidden behind a series of `if` statements instead of an explicit switch statement, but the root problem is the same.

What to Do

"Don't simulate inheritance" – use mechanisms built in to the programming language.

If the `switch` occurs in several places, it is often acting like a "type code"; replace this with the polymorphism built into objects. It takes a series of refactorings to make this change:

1. *Extract Method* – Pull out the code for each "branch"
2. *Move Method* – Move related code onto the right object
3. *Replace Type Code with Subclass* –or– *Replace Type Code with State/Strategy* – set up the inheritance structure
4. *Replace Conditional with Polymorphism* – eliminate the conditionals

If the conditions occur within a single object, you might be able to replace the conditional logic via *Replace Parameter with Explicit Methods* or *Introduce Null Object*.

Payoff

Improves communication, may expose duplication.

Counter-Indications

Sometimes a switch statement is just a switch statement. If it's doing something simple, you may not feel the need for a separate class. This may be especially common for places that are interfacing with non-object-oriented parts of the system.

A single `switch` statement is sometimes used in a factory or abstract factory. (See *Design Patterns*, by Gamma et al. for more information.) This one place decides how to configure the whole factory. You can sometimes replace it with `Class.forName()` (i.e.,

dynamic class loading), but that is sometimes not attractive for security or performance reasons.

Sometimes a switch statement is used in several related places to control a state machine. Decide whether it makes more sense as is, or whether the State pattern (*Design Patterns,* Gamma et al.) is more appropriate.

COND-3. Switch Statement. Consider this code:

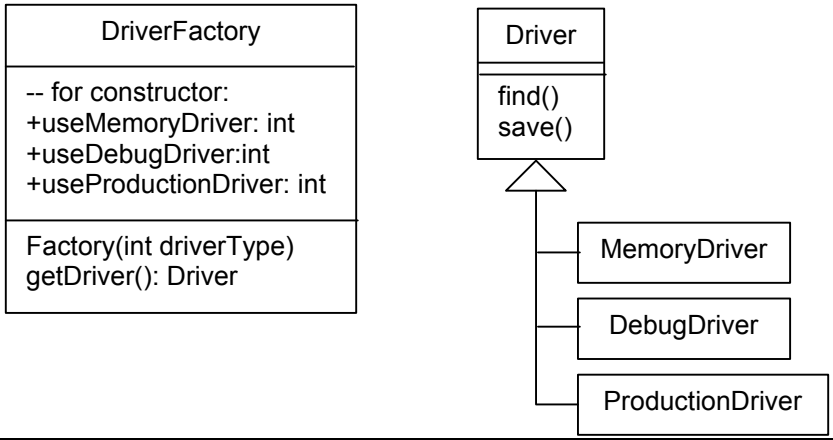
```
public void printIt(Operation op) {
    String out = "?";
    switch (op.type) {
    case '+': out = "push"; break;
    case '-': out = "pop"; break;
    case '@': out = "top"; break;
    default: out = "unknown";
    }
    System.out.println("operation=" + out);
}

public void doIt(Operation op, Stack s, Object item) {
    switch (op.type) {
    case '+': s.push(item); break;
    case '-': s.pop(); break;
    }
}
```

A. What would you do?

Solution ideas on page 182.

COND-4. Factory method. Consider this class structure:



A. Write code for the factory according to the implied design. (One of the three constants is passed to the constructor; this determines what type driver will be returned by `getDriver()`.)

B. Your code probably smells of a switch statement (even if it's implemented using "`if`"). Is this conditional logic justified?

C. Some factories use dynamic class loading (via `Class.forName()`) to load the correct class on demand. Modify your factory to accept a string argument (the name of the driver class), and load the instance dynamically.

Your code no longer mentions the types explicitly. What are some advantages to that? [Hint: think of the impact on building the system]

D. What are some disadvantages to the new arrangement?

Solution ideas on page 182.

Chapter 11. Names

Good names:

- support subtle expectations
- thesaurus
- communicate intent
- metaphor
- common vocabulary
- system of names

Name smells:

- Hungarian
- Compound words
- Confusing words

One of the challenges of software is finding the right mental model. The XP "metaphor" idea also captures this struggle. Sometimes you can find a metaphor that really clarifies what's happening. If your metaphor is good enough, it will become so familiar that people no longer think of it as a metaphor. (Think of data structures such as list, stack, tree, or window.)

Picking good names is hard; it's worth the effort to pick good ones. Ward Cunningham talks about using a thesaurus to suggest good ones; I think of this as the struggle to pick names that "support subtle expectations" about what the system is and how it should work.

What about domain knowledge? This is important too. The key is that everybody must agree on what terms mean.

Names with Embedded Types (including Hungarian)

Symptoms

You'll see:

- Names that are compound words, usually consisting of a word plus the type of the argument(s). For example, you might see a method "`addCourse(Course c)`".
- "Hungarian notation," where the type of an object is encoded into the name, e.g., `iCount` as an integer member variable.
- Variable names reflect their type rather than their purpose or role.

How It Got This Way

In the name of communication, the type may have been added. So `schedule.addCourse(course)` might be regarded as more readable than `schedule.add(course)`. (I don't think it is, but some do.)

The embedded type name represents duplication: both the argument and the name mention the same type. The embedded name can create unnecessary troubles later: suppose we introduce a parent class for Course, to cover both courses and series of courses. Now, all the places that refer to `addCourse()` have a name that's not quite appropriate. We either change the name at every call site, or live with a poor name. Finally, by naming things for the operation alone, we make it easier to see duplication.

Hungarian notation is often introduced as part of a coding standard. In pointer-based languages (like C), it was useful to know that `**ppc` is in fact a character, but in object-oriented languages it over-couples a name to its type.

Some programmers or teams use a convention where a prefix indicates that something is a member variable (`_count` or `m_count`), or that something is a constant (`ALL_UPPER_CASE`). Again, this causes complications as we change whether something is a local variable, a member, etc. (Aren't there times when we need to know which is which? Sure – and if it's not easy to tell, then it may be a sign that a class is too big.)

What to Do

Use *Rename Method* (or field or constant) to get a more "neutral" name.

Payoff

Improves communication. May make it easier to spot duplication.

Counter Indications

You might rarely have a class that wants to do the same sort of operation to two different but related types. For example, we might have a Graph object with `addPoint()` and

`addLink()` methods. Sometimes we can overload the method (`add()`), but sometimes the behavior for the two cases is not the same.

Sometimes you're working in the context of a coding standard that uses typographical conventions to distinguish things. You may then value the team's readability of code above the flexibility of "untyped" names, and follow those conventions. (This smell is not universally agreed to; even the code in *Refactoring* uses `_member` variables.)

Uncommunicative Names

Symptoms

A name doesn't communicate its intent well enough. (Sometimes names are too short, other times they're misleading.)

Causes

When you first implement something, you have to name things somehow. You give the best name you can think of at the time, and move on. Later, you may have an insight that lets you pick a better name.

What to Do

Use *Rename Method* (or field, constant, etc.) to give it a better name.

Payoff

Improves communication.

NAME-1. Names.
Classify these names as "embedded type," "uncommunicative," or "OK."

```
addItem(item)

doIt()

getNodesArrayList()

getData()

makeIt()

multiplyIntInt(int1, int2)

processItem()

sort()

spin()
```

Inconsistent Names

Symptoms

You find one term for something in one place, and a different name for the same thing somewhere else. For example, you might see `add()`, `store()`, `put()`, `place()`, etc. for the same basic feature.

Causes

Different people may have created the classes at different times. (People may have forgotten to explore the existing classes before adding more.) Occasionally, you'll find people doing this intentionally (but misguided) so they can distinguish the names.

What to Do

Pick the "best" name, and use *Rename Method* (etc.) to give the same name to the same things. Once you've done this, you may find that some classes appear to be more similar than they did before. Look for a "duplication" smell, and eliminate it.

Payoff

Improves communication, may expose duplication.

Discussion

The Eiffel language uses a common pool of words for the names of its library features. You can use this technique as inspiration: look to existing library names for the vocabulary you use.

There's a convention common enough to be worth mentioning: attributes of an object tend to have noun or perhaps adjective names; operations tend to use verbs for names.

NAME-2. Critique the names. Which name would you expect to use?

A. To empty a window (on-screen) –

```
clear()

wash()

erase()

deleteAll()
```

B. For a stack –

```
add()

insert()
```

```
push()

addToFront()
```

C. For an editor (to get rid of the selected text) –

```
cut()

delete()

clear()

erase()
```

D. As part of a file comparison program –

```
line1.compare(line2)

line1.equals(line2)

line1.identicalTo(line2)

line1.matches(line2)
```

Solution ideas on page 182.

NAME-3. XmlEditor.

You have a class XmlEditor. You want to introduce a parent class.

A. What do you call it?

B. You want to introduce an interface above that (that the parent will implement). What's a good name?

Solution ideas on page 183.

Chapter 12. Identifying New Refactorings

How do you discover and document a new refactoring? Just as patterns arise from repeated occurrence, so do refactorings. The first step is, "see it used several times." Not all new refactorings will recur, though. You'll sometimes have a medium-large change you'd like to make safely. So you may work it out abstractly before you apply it.

When I begin to document a new refactoring, I look at several questions:

- *What's the trigger? What's the benefit?* What made you realize that you wanted to change things? What will be better because of it? What will be worse? When is it not good to make the change?
- *What does the code look like before and after?* I usually use UML diagrams, CRC cards, or sample code fragments to show these.
- *What are the steps in the transformation?* Recall that refactorings often tolerate some duplication or some extra work in the middle to ensure nothing breaks partway through the refactoring. Can you move steps around to improve safety?
- *What else is helpful to know about this refactoring?* You may want extra explanation, a larger example, a list of special conditions, etc. You may be able to relate this to other refactorings.

If you're planning to share this refactoring with others, you can use the template Martin Fowler's uses in his catalog:

Refactoring Name

Sentence describing problem.
Sentence describing solution.

UML diagram or code

Motivation
Background material

Mechanics

- list of steps, almost always ending:
- compile and test

Example
Optionally, an example of the transformation

Other Notes
Another optional section. Mention any related refactorings.

You can adopt this format or another one.

Challenges

NEW-1. One exit.

Suppose you want to eliminate return statements from the middle of a loop:

```
for (int i = 0; i < n; i++) {
      // stuff
      if (something)
            return expression;
      // more stuff
}
```

becomes

```
for (…) {…}
return some-expression;
```

A. Why might you want to do this?

B. What is the code going to look like before and after? There are two issues: exiting early and the return value. How can you address these separately? Under what circumstances can you combine them?

C. Any special concerns?

D. When would you not apply this refactoring?

Solution ideas on page 183.

NEW-2. Your refactorings.

What are some other refactorings you've done? Write them up informally.

NEW-3. Strings and StringBuffers.

A. You have code that uses string addition. Why might it be better off using a StringBuffer?

B. You have code that uses a StringBuffer to append strings. Why might it be better off using string addition?

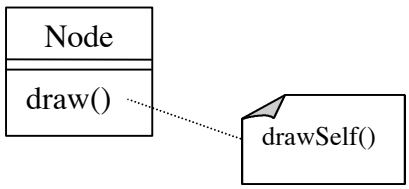
C. Pick one refactoring or the other, and write it up.

Solution ideas on page 184.

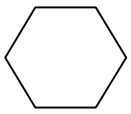
Refactoring at the UML Level

When identifying new refactorings, I'll often use UML diagrams or note cards to help me decide how to best move features around. We'll work through an example to demonstrate the technique.

Imagine a program that does some drawings. It originally had a Node object with a draw method. When asked to draw, it would put its shape on the screen. (We'll ignore how it knows *where* to draw.)

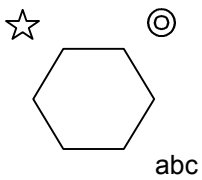


UML

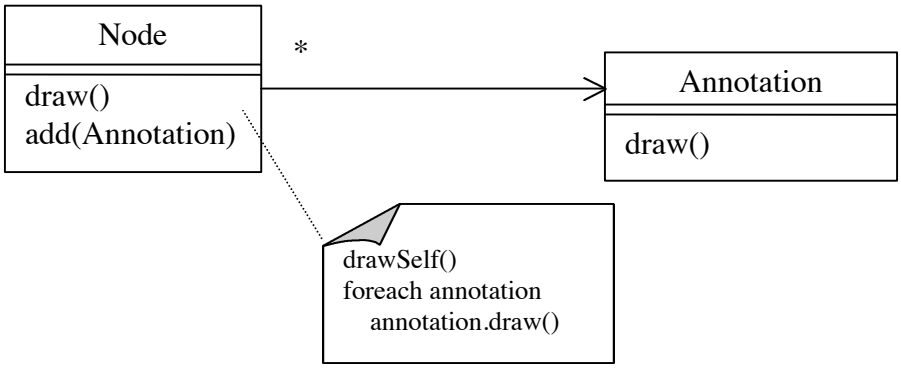


On-screen

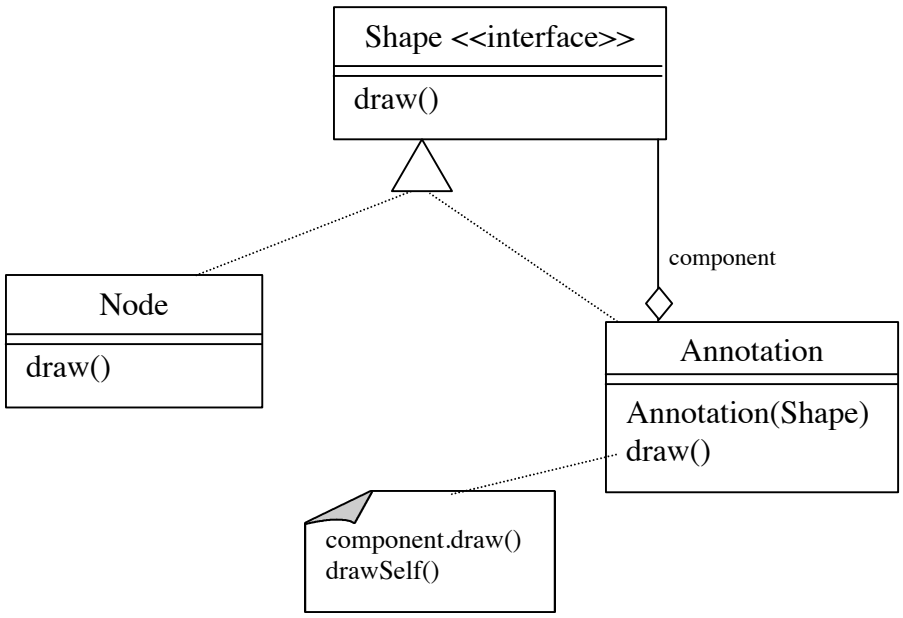
As so often happens, the requirements changed. Now, the node has annotations (satellites?) around it:



In the approach used, each annotation knows how to draw itself relative to the main node. The modified code worked like this:



The node holds its annotations. Suppose we think that a simple Decorator design pattern would work better, so we decide to refactor toward it:



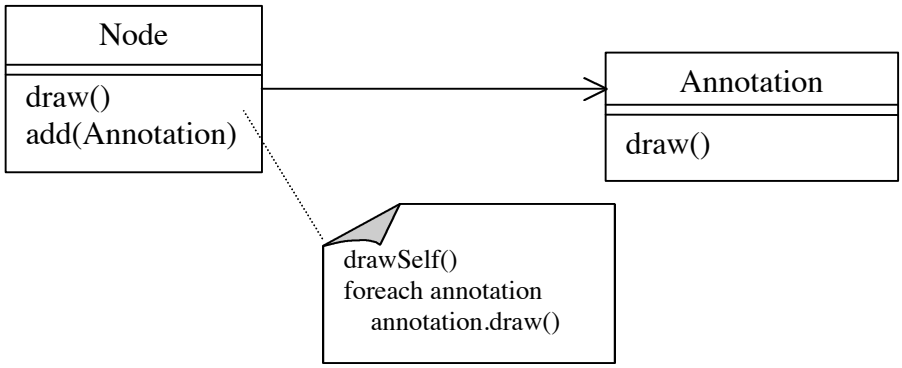
NEW-4. Decorator.
Why might Decorator be an improvement?

I think of designing this new refactoring as an instance of the "transformation game." The rules are:

- Make one change at a time
- Justify each move as being (or setting up) an improvement in communication, a reduction in duplication, a decrease in code volume, or a generalization.

NEW-5. Revised drawing.
Fill in a revised drawing and a justification.

Start:



A. Pass Node into a new Annotation constructor rather than to the `draw()` routine.

Justification: _____

B. Extract a Shape interface.

Justification: _____

C. Annotate any shape:

Justification: _____

We're at a challenging point. Consider the two draw() routines:

Node.draw():
```
drawSelf()
foreach annotation
     annotation.draw()
```

Annotation.draw():
```
drawSelf()
```

We'd like to do a "flip":

Node.draw():
```
drawSelf()
```

Annotation.draw():
```
component.draw()
drawSelf()
```

Extract the "foreach" to be a new method `drawAnnotations()`. Justification: communication.

We want to drop the `add()` mechanism, and the annotations link from Node to Annotation. Rather than adding annotations to the node, we want a chain of annotations with node at the bottom.

The trouble is, this affects the client. Currently, clients look something like this:

```
Node node = new Node();
node.add(new Annotation(node));
node.add(new Annotation(node));
node.draw();
```

We want them to become like this:

```
Shape shape = new Node();
shape = new Annotation(shape);
shape = new Annotation(shape);
shape.draw();
```

The first part of our refactoring focused on the structural aspects. Now we'll look at the code level. We'll still keep the transformation game rules, though.

Client	Node.draw()	Annotation.draw()
`node.draw()`	`drawSelf()`	`drawSelf()`
	`drawAnnotations()`	

Add a new method to Node, `drawNew()`. This represents the drawing routine we want to end up with. All it does is call `drawSelf()`. Work one client at a time, in a variant of *Inline Method*. Change the client from "`node.draw()`" to "`node.drawNew(); node.drawAnnotations();`". (Note that things should keep working after each one of these substitutions, no matter how many clients.) Justification: setup for future steps.

At this point, all clients are calling `node.drawNew()`, and none are calling `node.draw()`. Delete `node.draw()`, and rename `drawNew()` to `draw()`. Justification: dead code (simplifying).

Let's look at the clients in a little broader scope:
Client:

```
Node node = new Node();
node.add(new Annotation(node));
node.add(new Annotation(node));
node.draw();
node.drawAnnotations();
```

Our next goal is to get rid of the annotations on Node.

Start by creating a routine Annotation.drawNew(), which has the code we want to end up with:

```
component.draw();
drawSelf();
```

(Justification: setup for future moves.)

One client at a time, change to:
Client:

```
Node node = new Node();
Shape shape = new Annotation(node);
shape = new Annotation(shape);
shape.draw();
```

Justification: simplification.

Once this is done, nobody is calling `Node.drawAnnotations()`; delete it. Nobody is calling `Annotation.draw()`; delete it as well. (Justification: less code.)

Finally, rename `Annotation.drawNew()` to `draw()`. (Justification: simplicity.)

- - -

This feels like a lot of steps. But once you have a description, you may find ways to simplify it.

Notice how we handled changes that affect an arbitrary number of classes. (In this example, this was mostly where we changed how the client worked.) Where so many things need changing, it's good to ensure that we can work one client at a time. This is why we introduced the "`drawNew()`" functions. It would have been easy to say "make `draw()` call only `drawSelf()`, and make its callers call `drawAnnotations()`," but the slower way is better: we can run tests after changing each client. This helps reassure us along the way that our transformations are safe.

NEW-6. Analysis.
Is this a good refactoring?

Caveats

Refactorings try to create safe transformations, but that's not always easy. Ideally (and certainly for tool support), you'd like to be able to *prove* that a refactoring is safe.

A refactoring just being written up for the first time will tend not to be as mature as a more long-standing one.

I know of several areas that particularly cause problems. (This is not an exhaustive list; there are certainly lots of areas that I haven't run into yet.)

1. Inheritance. The rules for inheritance are more complicated than you might think. When you rearrange classes that are in an inheritance hierarchy, you need to make sure you've considered the effects on parents, children, and clients.

2. Renaming. One of the lessons logicians learned before most of us could program is that it's tricky to rename things. You need to consider the possibility that a name might already be in use in some context.

3. Threads. One of the explicit disclaimers Martin Fowler makes in *Refactoring* is that he didn't work through the consequences of refactoring thread-based code.

Interlude 4 – Gen-A-Refactoring

Several refactorings have the form "Verb Noun."

Consider these verbs: Extract, Inline; Move; Rename; Pull Up, Push Down; Hide, Expose

Consider these nouns: Field, Method; Class, Interface, Subclass, Superclass; Hierarchy

I4-1. Verbs and nouns.

In the table below, put a "-" in combinations that don't make sense, a "+" in ones that are in Fowler's catalog, and a "*" in ones that make sense but aren't in the catalog.

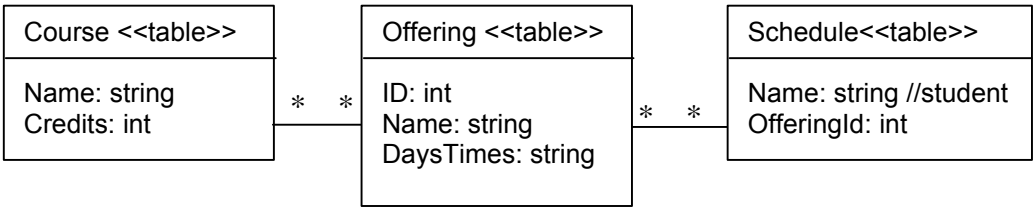
Solution on page 184.

	Extract	Inline	Move	Rename	Pull Up	Push Down	Hide	Expose
Field								
Method								
Class								
Interface								
Subclass								
Superclass								
Hierarchy								

Chapter 13: A Database Example

"Reggie" is a system to handle course registration for a small school.

The developers are using a database to maintain information about students and classes. The first version of the database is simple:



A Course is a class that could be offered. For now, we'll use simple names ("Econ101" for "Introduction to Economics.") Later, this will expand to include a full title, description, and other information about the course.

An Offering is a version of a class, taught on some schedule. DaysTimes is a comma-separated string of days and times ("M10,T11,F10").Again, there will be more information added in development.

A Schedule is a particular set of offerings that a student has chosen. There are rules about how schedules can be formed:

- At least 12 credits
- At most 18 credits, unless an overload is authorized
- No conflicting times
- No duplicate courses

These rules aren't enforced by the database, but by code that checks schedules.

DB-1. Data Smells.

Refactoring **mostly deals with code smells. But there are "data smells" too: the database community has notions of what constitutes a good data design.**

A. What potential problems do you see in this database structure?

B. What changes to the database might address them? (Don't make the changes yet.)

Solution on page 184.

A good introduction to database normalization is "A Simple Guide to Five Normal Forms in Relational Theory," by William Kent, in *Communications of the ACM*, V26 #2, February, 1983, pp. 120-125.

In the code, there are three classes corresponding to the tables. They started out small, and perhaps a little too focused on the data, but they're much bigger with database code added. For an introduction to database programming in Java, see

Course.java (available online)

```
import java.sql.*;

public class Course {
    private String name;
    private int credits;
    static String url = "jdbc:odbc:Reggie";

    static { try { Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); }
catch (Exception ignored) {} }

    public static Course create(String name, int credits) throws
Exception {
      Connection conn = null;

      try {
          conn = DriverManager.getConnection(url, "", "");
          Statement statement = conn.createStatement();
          statement.executeUpdate(
            "DELETE FROM course WHERE name = '" + name + "';");
          statement.executeUpdate(
            "INSERT INTO course VALUES ('" + name
            + "', '" + credits + "');");
          return new Course(name, credits);
      } finally {
          try { conn.close(); } catch (Exception ignored) {}
      }
    }

    public static Course find(String name) {
        Connection conn = null;
        try {
            conn = DriverManager.getConnection(url, "", "");
            Statement statement = conn.createStatement();
            ResultSet result = statement.executeQuery(
                "SELECT * FROM course WHERE Name = '" + name + "';");
            if (!result.next()) return null;

            int credits = result.getInt("Credits");
            return new Course(name, credits);
        } catch (Exception ex) {
```

```
            return null;
        } finally {
            try { conn.close(); } catch (Exception ignored) {}
        }
    }

    public void update() throws Exception {
        Connection conn = null;

        try {
            conn = DriverManager.getConnection(url, "", "");
            Statement statement = conn.createStatement();

            statement.executeUpdate(
                  "DELETE FROM COURSE WHERE name = '" + name + "';");
            statement.executeUpdate(
                  "INSERT INTO course VALUES('" +
                  name + "','" + credits + "');");
        } finally {
            try { conn.close(); } catch (Exception ignored) {}
        }
    }

    Course(String name, int credits) {
        this.name = name;
        this.credits = credits;
    }

    public int getCredits() {
        return credits;
    }

    public String getName() {
        return name;
    }
}
```

Offering.java

```
import java.sql.*;

public class Offering {
    private int id;
    private Course course;
    private String daysTimes;

    static String url = "jdbc:odbc:Reggie";

    static { try { Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); }
catch (Exception ignored) {} }

    public static Offering create(Course course, String daysTimesCsv)
throws Exception {
        Connection conn = null;
```

```
    try {
        conn = DriverManager.getConnection(url, "", "");
        Statement statement = conn.createStatement();

        ResultSet result = statement.executeQuery(
              "SELECT MAX(ID) FROM offering;");
        result.next();
        int newId = 1 + result.getInt(1);

        statement.executeUpdate("INSERT INTO offering VALUES ('"
                + newId + "','" + course.getName()
                + "','" + daysTimesCsv + "');");
        return new Offering(newId, course, daysTimesCsv);
    } finally {
        try { conn.close(); } catch (Exception ignored) {}
    }
}

public static Offering find(int id) {
    Connection conn = null;

    try {
        conn = DriverManager.getConnection(url, "", "");
        Statement statement = conn.createStatement();
        ResultSet result = statement.executeQuery(
              "SELECT * FROM offering WHERE ID =" + id + ";");
        if (result.next() == false)
            return null;

        String courseName = result.getString("Course");
        Course course = Course.find(courseName);
        String dateTime = result.getString("DateTime");
        conn.close();

        return new Offering(id, course, dateTime);
    } catch (Exception ex) {
        try { conn.close(); } catch (Exception ignored) {}
        return null;
    }
}

public void update() throws Exception {
    Connection conn = null;

    try {
        conn = DriverManager.getConnection(url, "", "");
        Statement statement = conn.createStatement();

        statement.executeUpdate(
              "DELETE FROM Offering WHERE ID=" + id + ";");
        statement.executeUpdate(
                "INSERT INTO Offering VALUES('" + id + "','" +
                  course.getName() + "','" + daysTimes + "');");
    } finally {
```

```
            try { conn.close(); } catch (Exception ignored) {}
        }
    }

    public Offering(int id, Course course, String daysTimesCsv) {
        this.id = id;
        this.course = course;
        this.daysTimes = daysTimesCsv;
    }

    public int getId() {
        return id;
    }

    public Course getCourse() {
        return course;
    }

    public String getDaysTimes() {
        return daysTimes;
    }

    public String toString() {
        return "Offering " + getId() + ": "
      + getCourse() + " meeting " + getDaysTimes();
    }
}
```

Schedule.java

```
import java.util.*;
import java.sql.*;

public class Schedule {
    String name;
    int credits = 0;
    static final int minCredits = 12;
    static final int maxCredits = 18;
    boolean overloadAuthorized = false;
    ArrayList schedule = new ArrayList();

    static String url = "jdbc:odbc:Reggie";
    static { try { Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); }
catch (Exception ignored) {} }

    public static void deleteAll() throws Exception {
        Connection conn = null;

        try {
            conn = DriverManager.getConnection(url, "", "");
            Statement statement = conn.createStatement();

            statement.executeUpdate("DELETE * FROM schedule;");
        } finally {
```

```
        try { conn.close(); } catch (Exception ignored) {}
    }
}

public static Schedule create(String name) throws Exception {
    Connection conn = null;

    try {
        conn = DriverManager.getConnection(url, "", "");
        Statement statement = conn.createStatement();

        statement.executeUpdate(
              "DELETE FROM schedule WHERE name = '" + name + "';");
        return new Schedule(name);
    } finally {
        try { conn.close(); } catch (Exception ignored) {}
    }
}

public static Schedule find(String name) {
    Connection conn = null;

    try {
        conn = DriverManager.getConnection(url, "", "");
        Statement statement = conn.createStatement();
        ResultSet result = statement.executeQuery(
            "SELECT * FROM schedule WHERE Name= '" + name + "';");

        Schedule schedule = new Schedule(name);

        while (result.next()) {
            int offeringId = result.getInt("OfferingId");
            Offering offering = Offering.find(offeringId);
            schedule.add(offering);
        }

        return schedule;
    } catch (Exception ex) {
        return null;
    } finally {
        try { conn.close(); } catch (Exception ignored) {}
    }
}

public static Collection all() throws Exception {
    ArrayList result = new ArrayList();
    Connection conn = null;

    try {
        conn = DriverManager.getConnection(url, "", "");
        Statement statement = conn.createStatement();
        ResultSet results = statement.executeQuery(
              "SELECT DISTINCT Name FROM schedule;");

        while (results.next())
```

```
            result.add(Schedule.find(results.getString("Name")));
        } finally {
            try { conn.close(); } catch (Exception ignored) {}
        }

        return result;
    }

    public void update() throws Exception {
        Connection conn = null;

        try {
            conn = DriverManager.getConnection(url, "", "");
            Statement statement = conn.createStatement();

            statement.executeUpdate(
                  "DELETE FROM schedule WHERE name = '" + name + "';");

            for (int i = 0; i < schedule.size(); i++) {
                Offering offering = (Offering) schedule.get(i);
                statement.executeUpdate(
                  "INSERT INTO schedule VALUES('" + name + "','"
                  + offering.getId() + "');");
            }
        } finally {
            try { conn.close(); } catch (Exception ignored) {}
        }
    }

    public Schedule(String name) {
        this.name = name;
    }

    public void add(Offering offering) {
        credits += offering.getCourse().getCredits();
        schedule.add(offering);
    }

    public void authorizeOverload(boolean authorized) {
        overloadAuthorized = authorized;
    }

    public List analysis() {
        ArrayList result = new ArrayList();

        if (credits < minCredits)
            result.add("Too few credits");

        if (credits > maxCredits && !overloadAuthorized)
            result.add("Too many credits");

        checkDuplicateCourses(result);

        checkOverlap(result);
```

```
        return result;
    }

    public void checkDuplicateCourses(ArrayList analysis) {
        HashSet courses = new HashSet();
        for (int i = 0; i < schedule.size(); i++) {
            Course course = ((Offering) schedule.get(i)).getCourse();
            if (courses.contains(course))
               analysis.add("Same course twice - " + course.getName());
            courses.add(course);
        }
    }

    public void checkOverlap(ArrayList analysis) {
      HashSet times = new HashSet();

      for (Iterator iterator = schedule.iterator();
iterator.hasNext();) {
          Offering offering = (Offering) iterator.next();
          String daysTimes = offering.getDaysTimes();
          StringTokenizer tokens = new StringTokenizer(daysTimes, ",");
          while (tokens.hasMoreTokens()) {
              String dayTime = tokens.nextToken();
              if (times.contains(dayTime))
                  analysis.add("Course overlap - " + dayTime);
              times.add(dayTime);
          }
      }
    }

    public String toString() {
        return "Schedule " + name + ": " + schedule;
    }
}
```

Report.java

```
import java.util.*;

public class Report {
    public Report() {
    }

    Hashtable offeringToName = new Hashtable();

    public void populateMap() throws Exception {
        Collection schedules = Schedule.all();
        for (Iterator eachSchedule = schedules.iterator();
eachSchedule.hasNext();) {
            Schedule schedule = (Schedule) eachSchedule.next();

            for (Iterator each = schedule.schedule.iterator();
each.hasNext(); ) {
                Offering offering = (Offering) each.next();
```

```
                populateMapFor(schedule, offering);
            }
        }
    }

    private void populateMapFor(Schedule schedule, Offering offering) {
        ArrayList list = (ArrayList) offeringToName.get(
            new Integer(offering.getId()));
        if (list == null) {
            list = new ArrayList();
            offeringToName.put(new Integer(offering.getId()), list);
        }
        list.add(schedule.name);
    }

    public void writeOffering(StringBuffer buffer, ArrayList list,
Offering offering) {
        buffer.append(offering.getCourse().getName() + " "
                    + offering.getDaysTimes() + "\n");

        for (Iterator iterator = list.iterator(); iterator.hasNext();)
        {
            String s = (String) iterator.next();
            buffer.append("\t" + s + "\n");
        }
    }

    public void write(StringBuffer buffer) throws Exception {
        populateMap();

        Enumeration enumeration = offeringToName.keys();
        while (enumeration.hasMoreElements()) {
           Integer offeringId = (Integer) enumeration.nextElement();
           ArrayList list = (ArrayList) offeringToName.get(offeringId);
           writeOffering(buffer, list,
               Offering.find(offeringId.intValue()));
        }

        buffer.append("Number of scheduled offerings: ");
        buffer.append(offeringToName.size());
        buffer.append("\n");
    }
}
```

TestSchedule.java

```
import junit.framework.TestCase;

import java.util.List;
import java.util.Collection;

public class TestSchedule extends TestCase {
    public TestSchedule(String name) {
        super(name);
```

```
}

public void testMinCredits() {
    Schedule schedule = new Schedule("name");
    Collection analysis = schedule.analysis();
    assertEquals(1, analysis.size());
    assertTrue(analysis.contains("Too few credits"));
}

public void testJustEnoughCredits() {
    Course cs110 = new Course("CS110", 11);
    Offering mwf10 = new Offering(1, cs110, "M10,W10,F10");
    Schedule schedule = new Schedule("name");
    schedule.add(mwf10);
    List analysis = schedule.analysis();
    assertEquals(1, analysis.size());
    assertTrue(analysis.contains("Too few credits"));

    schedule = new Schedule("name");
    Course cs101 = new Course("CS101", 12);
    Offering th11 = new Offering(1, cs101, "T11,H11");
    schedule.add(th11);
    analysis = schedule.analysis();
    assertEquals(0, analysis.size());
}

public void testMaxCredits() {
    Course cs110 = new Course("CS110", 20);
    Offering mwf10 = new Offering(1, cs110, "M10,W10,F10");
    Schedule schedule = new Schedule("name");
    schedule.add(mwf10);
    List analysis = schedule.analysis();
    assertEquals(1, analysis.size());
    assertTrue(analysis.contains("Too many credits"));

    schedule.authorizeOverload(true);
    analysis = schedule.analysis();
    assertEquals(0, analysis.size());
}

public void testJustBelowMax() {
    Course cs110 = new Course("CS110", 19);
    Offering mwf10 = new Offering(1, cs110, "M10,W10,F10");
    Schedule schedule = new Schedule("name");
    schedule.add(mwf10);
    List analysis = schedule.analysis();
    assertEquals(1, analysis.size());
    assertTrue(analysis.contains("Too many credits"));

    schedule = new Schedule("name");
    Course cs101 = new Course("CS101", 18);
    Offering th11 = new Offering(1, cs101, "T11,H11");
    schedule.add(th11);
    analysis = schedule.analysis();
    assertEquals(0, analysis.size());
```

```
}

public void testDupCourses() {
    Course cs110 = new Course("CS110", 6);
    Offering mwf10 = new Offering(1, cs110, "M10,W10,F10");
    Offering th11 = new Offering(1, cs110, "T11,H11");
    Schedule schedule = new Schedule("name");
    schedule.add(mwf10);
    schedule.add(th11);
    List analysis = schedule.analysis();
    assertEquals(1, analysis.size());
    assertTrue(analysis.contains("Same course twice - CS110"));
}

public void testOverlap() {
    Schedule schedule = new Schedule("name");

    Course cs110 = new Course("CS110", 6);
    Offering mwf10 = new Offering(1, cs110, "M10,W10,F10");
    schedule.add(mwf10);

    Course cs101 = new Course("CS101", 6);
    Offering mixed = new Offering(1, cs101, "M10,W11,F11");
    schedule.add(mixed);

    List analysis = schedule.analysis();
    assertEquals(1, analysis.size());
    assertTrue(analysis.contains("Course overlap - M10"));

    Course cs102 = new Course("CS102", 1);
    Offering mixed2 = new Offering(1, cs102, "M9,W10,F11");
    schedule.add(mixed2);

    analysis = schedule.analysis();
    assertEquals(3, analysis.size());
    assertTrue(analysis.contains("Course overlap - M10"));
    assertTrue(analysis.contains("Course overlap - W10"));
    assertTrue(analysis.contains("Course overlap - F11"));
}

public void testCourseCreate() throws Exception {
    Course c = Course.create("CS202", 1);

    Course c2 = Course.find("CS202");
    assertEquals("CS202", c2.getName());

    Course c3 = Course.find("Nonexistent");
    assertNull(c3);
}

public void testOfferingCreate() throws Exception {
    Course c = Course.create("CS202", 2);
    Offering offering = Offering.create(c, "M10");
    assertNotNull(offering);
}
```

```
    public void testPersistentSchedule() throws Exception {
        Schedule s = Schedule.create("Bob");
        assertNotNull(s);
    }

    public void testScheduleUpdate() throws Exception {
        Course cs101 = Course.create("CS101", 3);
        cs101.update();
        Offering off1 = Offering.create(cs101, "M10");
        off1.update();
        Offering off2 = Offering.create(cs101, "T9");
        off2.update();

        Schedule s = Schedule.create("Bob");
        s.add(off1);
        s.add(off2);
        s.update();

        Schedule s2 = Schedule.create("Alice");
        s2.add(off1);
        s2.update();

        Schedule s3 = Schedule.find("Bob");
        assertEquals(2, s3.schedule.size());

        Schedule s4 = Schedule.find("Alice");
        assertEquals(1, s4.schedule.size());
    }
}
```

TestReport.java

```
import junit.framework.TestCase;

import java.util.List;
import java.util.Collection;

public class TestReport extends TestCase {
    public TestReport(String name) { super(name); }

    public void testEmptyReport() throws Exception {
        Schedule.deleteAll();
        Report report = new Report();

        StringBuffer buffer = new StringBuffer();

        report.write(buffer);

        assertEquals(
            "Number of scheduled offerings: 0\n",
            buffer.toString());
    }
```

```
    public void testReport() throws Exception {
        Schedule.deleteAll();

        Course cs101 = Course.create("CS101", 3);
        cs101.update();
        Offering off1 = Offering.create(cs101, "M10");
        off1.update();
        Offering off2 = Offering.create(cs101, "T9");
        off2.update();

        Schedule s = Schedule.create("Bob");
        s.add(off1);
        s.add(off2);
        s.update();

        Schedule s2 = Schedule.create("Alice");
        s2.add(off1);
        s2.update();

        Report report = new Report();

        StringBuffer buffer = new StringBuffer();

        report.write(buffer);

        String result = buffer.toString();
        String valid1 = "CS101 M10\n\tAlice\n\tBob\n" +
                        "CS101 T9\n\tBob\n" +
                        "Number of scheduled offerings: 2\n";

        String valid2 = "CS101 T9\n\tBob\n" +
                        "CS101 M10\n\tAlice\n\tBob\n" +
                        "Number of scheduled offerings: 2\n";
        assertTrue(result.equals(valid1) || result.equals(valid2));
    }
}
```

DB-2. Duplication.

A. Identify smells, especially the duplication (and the differences!) between these classes.

B. What is the pattern of access for persistence? (That is, trace the life of an object from creation to destruction.)

DB-3. Application.

A. The database-related routines are tantalizingly similar. Apply refactorings that will reduce this duplication.

B. Create a class that provides the bulk of the database support. Should this be a parent of these objects, or a separate class?

DB-4. Database layer.

One effect of the refactorings you just did is to move the code toward a database layer.

A. What extra work would you have to do to create an in-memory version of the database classes?

B. Why might you want to do this?

C. How much of the JDBC would be exposed by your new layer?

Solution ideas on page 185.

For more information on creating a database layer, look at: J2EE (tool support for automatic mapping of EJBs), WebGain TopLink ™ (see http://www.webgain.com/products/toplink/), or Scott Ambler's description (http://www.ambysoft.com/mappingObjects.html). (One thing these will show you is that you may not want to lightly undertake development of a full-fledged layer.)

DB-5. Find.

The `find()` routines create a new instance of an object even if it's already been "found" before.

A. Make `find()` cache its objects, returning a previously found object if possible.

B. Is this refactoring or development? Did you add tests to verify the new behavior? (You should.)

C. The Report makes it a point to store "keys" of offerings. Use the improved nature of `find()` to simplify the report's code.

Solution ideas on page 185.

DB-6. Multiple open queries.

Some databases have a restriction that a connection can't have multiple queries "open" at the same time. (An example occurs in Schedule.find(): while building a Schedule, it queries Offerings.find(), which also queries the database.)

A. How much code (and in how many places) would be affected by a change to enforce this discipline?

B. The "naïve" approach used in this code instantiates all related objects for each row that is loaded. How else might you do it?

DB-7. Counter.

The counter in Offering is manipulated in a two-step process: get the current max value of the counter in any row, and create a new row with an ID one bigger. When will this strategy be inadequate? What could you do about it?

Solution ideas on page 185.

DB-8. Report.

The Report creates a list of offerings and the students in each.

A. (If you know SQL…) Write a SQL query to produce the information in this report.

B. What are the tradeoffs between the two approaches?

DB-9. Database refactorings.

Earlier, we described some possible database changes: introducing IDs (rather than using string keys), extracting a new table, and so on. We can't just make these changes in the database: our code depends on the database structure.

A. Describe "Rename Column" as a refactoring. (Tell what to change in the database or the code, when to change it, when to run tests, and so on.)

B. Describe "Extract Table" as a refactoring too. Make sure you account for any data migration.

C. Are there intermediate steps that could temporarily leave the database in a slightly worse structure, but make the overall refactoring be safer? (Think of how Extract Method copies and adjusts the new method before deleting the old one, or how some refactorings change things one at a time, running tests after each change.)

DB-10. Domain class independence.

Suggest ways in which the domain classes could be made more independent of the database access part of these classes.

Chapter 14: A Recursive Example

This example involves refactoring, test-driven design, and recursion.

Suppose we've decided to develop a system to play games in the tic-tac-toe family: squares occupied by different markers. In tic-tac-toe, you have a 3x3 grid, and try to get three in a row. In Connect Four (trademark Hasbro), you have a rectangular grid, and try to get four in a row, but columns have to be filled from bottom to top. We'll start with a simplified version of tic-tac-toe, and work our way up to the general case.

Here are some tests and the first version of the code:

```
import junit.framework.*;

public class GameTest extends TestCase {

      public GameTest(String s) {super(s);}

      public void testDefaultMove() {
            Game game = new Game("XOXOX-OXO");
            assertEquals(5, game.move('X'));

            game = new Game("XOXOXOOX-");
            assertEquals(8, game.move('O'));

            game = new Game("---------");
            assertEquals(0, game.move('X'));

            game = new Game("XXXXXXXXX");
            assertEquals(-1, game.move('X'));
      }

      public void testFindWinningMove() {
            Game game = new Game("XO-XX-OOX");
            assertEquals(5, game.move('X'));
      }

      public void testWinConditions() {
            Game game = new Game("---XXX---");
            assertEquals('X', game.winner());
      }
}

public class Game {
      public StringBuffer board;

      public Game(String s) {board = new StringBuffer(s);}

      public Game(StringBuffer s, int position, char player) {
            board = new StringBuffer();
            board.append(s);
```

```
            board.setCharAt(position, player);
      }

      public int move(char player) {
            for (int i = 0; i < 9; i++) {
                  if (board.charAt(i) == '-') {
                        Game game = play(i, player);
                        if (game.winner() == player)
                              return i;
                  }
            }

            for (int i = 0; i < 9; i++) {
                  if (board.charAt(i) == '-')
                        return i;
            }
            return -1;
      }

      public Game play(int i, char player) {
            return new Game(this.board, i, player);
      }

      public char winner() {
            if (board.charAt(0) != '-'
&& board.charAt(0) == board.charAt(1)
&& board.charAt(1) == board.charAt(2))
                  return board.charAt(0);
            if (board.charAt(3) != '-'
&& board.charAt(3) == board.charAt(4)
&& board.charAt(4) == board.charAt(5))
                  return board.charAt(3);
            if (board.charAt(6) != '-'
&& board.charAt(6) == board.charAt(7)
&& board.charAt(7) == board.charAt(8))
                  return board.charAt(6);
            return '-';
      }
}
```

Notice that the `winner()` routine is simplified: you win by getting three in a row horizontally. Notice also that the heuristics for what to play are primitive: win if you can, play anything otherwise. We'll migrate toward something capable of more sophisticated strategies.

REC-1. Smells.
Go through this code and identify smells.

REC-2. Easy changes.
It's not always easy to know what to do with code. Let's fix some of the easy things first.

Fix them one at a time.

- **The name `move()` isn't descriptive enough. Change it to `bestMoveFor()`.**
- **The variable `i` doesn't explain much either. Change it to move.**
- **The value `-1` is a flag value; create a constant `NoMove` to represent it.**
- **The `winner()` function has a lot of duplication. Eliminate the duplication.**
- **The check for a board character being a '-' is really a check that the square is unoccupied. Extract a method to do this, and name it appropriately.**

We have two "for" loops: one to find a winning move, the other to find a default move. One way to handle this would be to extract each one into a method. As we add more strategies, we could see each strategy getting its own method. An alternative would be to merge the two loops and handle things more in one pass through the possible moves. We'll take the latter approach.

REC-3. Fuse Loops.

"Fuse Loops" means combine two loops into one. Do so in small steps, in such a way that you maintain safety as you do it. When is it safe to merge two loops? ("Fuse Loops" is a standard technique compilers use; it's not in Martin's catalog.)

One step along the way might be to assign a value to a temporary variable rather than return it right away. You might have made the second loop look like this:

```
defaultMove = NoMove;
for (int i = 0; i < 9; i++) {
      if (board[i] == '-')
            if (defaultMove == NoMove)
                  defaultMove = i;
}
```

That was the safest approach, used because we did not want our refactoring to change behavior. To be equivalent to the original, we need the guard clause to make sure we haven't assigned a `defaultMove` yet. But let's put on a development hat: we don't really care which default move we make, so we could delete the "`defaultMove==NoMove`" test. It's not necessary to stop when we find a viable move. (That is, there's no harm in trying each possible move provided we prefer wins to defaults.) So you can delete the "break" tests that exit early. Run the tests again and be sure you haven't changed anything important. (You may have to change the tests. What does this tell you?)

Now we have a single loop, but the condition to decide what to return is still a little complicated:

```
if (winningMove != NoMove)
      return winningMove;
if (defaultMove != NoMove)
      return defaultMove;
return NoMove;
```

REC-4. Result.
How would you simplify this?

REC-5. Next.
What refactorings would you tackle next?

There are still a lot of "magic numbers" floating around. The `winner()` routine is full of them, and we still have a "9" in the main loop of `bestMoveFor()`.

REC-6. Constants.
What is "9"? Name some constants and rewrite it.

I struggled over the names, and ended up using "rows" and "columns." (I'm also struggling over naming conventions; some styles might use ROWS and COLUMNS.)

REC-7. Duplication in winner().
Here's the `winner()` routine as it is now.

```
      public char winner() {
            if (!canPlay(0)
                && board.charAt(0) == board.charAt(1)
                && board.charAt(1) == board.charAt(2))
                  return board.charAt(0);
            if (!canPlay(3)
                && board.charAt(3) == board.charAt(4)
                && board.charAt(4) == board.charAt(5))
                  return board.charAt(3);
            if (!canPlay(6)
                && board.charAt(6) == board.charAt(7)
```

```
                && board.charAt(7) == board.charAt(8))
                  return board.charAt(6);
            return '-';
      }
```

Eliminate the duplication in this routine.

I imagine you used a loop over the rows, and extracted some sort of method that decided if there was a winning combination.

To assess your refactoring of this, switch back to a development hat. There are two changes we might make, given our vision. The first is that we're not playing tic-tac-toe yet because we're only allowing horizontal 3-in-a-rows. Extend the routine to allow vertical and diagonal wins. Was it easy to change given your refactoring?

There's another hidden constant: the number in a row that it takes to win. (Recall we mentioned Connect Four as one of the variations we wanted to eventually support.) Suppose we changed to a 5x5 grid and wanted four in a row to win. How easy is that to put in? (You needn't add this feature yet; this is more of a thought question.)

(I gave myself a B on this one: I defined a `winningCombination()` routine that took three positions as arguments, and I created an array that explicitly called out the winning squares. This works fine for tic-tac-toe but doesn't cover n-in-a-row.)

Until now, most of the refactorings we've applied have been obvious improvements. Now I want to grow and improve my program through a combination of refactoring and new implementation. But I'm not sure what's best to do next.

I think of this as "subjunctive programming." The subjunctive tense is the one you use to talk about possible worlds ("If I were a rich man…"). My stance is that I'll try some ideas and see where they lead, but if they don't work out, that's OK.

Two things make subjunctive programming bearable: a partner, so you can kick around ideas, and a source control system, so you can back out anything you don't like.

The general direction is that I want to allow more sophisticated strategies than "win if you can and play anything otherwise." My thought is to create a Move object and let it think about how good it is.

REC-8. Iterator.

We're running a for loop over the integers representing possible moves. Turn this into an iterator over the moves. This is not a refactoring in Martin's catalog, so let's take it in small steps.

1. Convert from int to Integer first; that will get us into the domain of objects.
2. Make an Iterator, where the next() method returns an Integer. Remove all references to int from the bestMoveFor() method.
3. Introduce a Move type.

Now, the main loop looks something like this:

```
for (Iterator iter = new MoveIterator(); iter.hasNext(); ) {
      Move move = (Move)iter.next();
      if (!canPlay(move)) continue;
```

REC-9. Legal moves only.

Our iterator delivers all moves, legal or not. Move the "canPlay" test into the iterator so it only delivers legal moves.

Currently, we're just looping through possible moves, trying to select the best one. Right now, we have a simple rule: wins are best, anything else is acceptable. But wins are rare; we'd like to pick a good intermediate move. (Some moves are better than others.) We can think of each move as having a score: how good is it. Just to have something to work with, we'll call a win worth 100 points, and any other move 0 points. (We can imagine wins by the other player being worth -100 points, but we don't check for those—yet.)

Note that we're out of the domain of refactoring; we're making a semantic change to our program. That's the way development works. It's often the case that as refactoring makes things cleaner, we can see better ways to do things.

Development Episodes

REC-10. Scores.

Modify the program to calculate scores for moves, and return the move with the best score.

Notice how a score is associated with a particular move. Perhaps it should be part of the Move object. Doing this might let us eliminate tracking of the integer score from the "main" loop.

REC-11. Comparing moves.

Add the score to the Move object. Add some sort of "max" or comparison operator between Moves.

Is it better this way? It seems so to me, so I'd tentatively accept this change. But my commitment to it is not unswerving; for now it helps, but if it gets in the way in the future, away it goes.

- - - - -

The program calculates every possible move and response. This is feasible for tic-tac-toe, and perhaps okay if we were to convert it to Hasbro Connect-Four™ , but certainly not feasible for a game like chess or Go. Eventually we would have to develop a new strategy.

One way to handle this is to limit the depth to which we search. Suppose we establish a depth cutoff value: searches deeper than this will just return "don't know." We will pass an additional parameter representing the current depth.

REC-12. Depth.

Use *Add Parameter* to add a depth parameter, and maintain its value properly. Once you have the depth parameter, add an early check that returns when things are too deep. What move will you return?

REC-13. Caching.

There is a type of refactoring that I call "refactoring for performance." It's on the line between development, refactoring, and performance tuning. If we think of the program as exploring the game tree of possible moves, we might see the same board via different paths. Could you cache the moves, so you could recognize boards you've already rated?

REC-14. Balance.

Do we have the right balance in our objects? Are there any missing objects? Should Game calculate the score or should Move? Try shifting it around and see the consequences. Do some of these decisions make caching easier or harder?

REC-15. New features.

Add some new features, test-first; make sure to refactor along the way.

A. Score a win by the opponent at -100.

B. Add another feature: use the min-max algorithm. Instead of just saying "non-wins are all the same," you say "Choose my best move, assuming the opponent makes the move that's worst for me." The opponent uses the same rule.

C. Generalize to mxn tic-tac-toe if you haven't yet. Does your move routine still work properly? You'll almost certainly have to add tests. What other refactoring do you need?

REC-16. Min-max.

In artificial intelligence circles, the approach we're using is known as min-max. When it's our turn, we try to maximize our score. When it's the opponent's turn, they try to minimize our score. How is this reflected in this code? Is it a "trick"?

There's an extension to the approach, called alpha-beta pruning. It says that we can avoid searching parts of the tree by establishing cutoff values. Find an AI book, and see if you can implement this approach. Is this a refactoring, new development, or what?

Chapter 15. More on Responsibility

"Who's in charge here?"

This is a moderate-sized example involving both refactoring and development.

Introduction

Imagine a store selling a variety of items. During a brief design session, we decide there are three objects of interest: Item (something for sale), Catalog (the set of all items), and Query (to find a specific set of items).

Catalog and Query must collaborate to search.

Imagine a fourth object, Interrogator. It has a method evaluate(catalog, query), returning a list of items. But this method must make a decision about how it handles things:

MORE-1. Evaluate.
Should the "core" of `evaluate()`'s implementation be

(a) `catalog.itemsMatching(query)`, or
(b) `query.matchesIn(catalog)`, or
(c) `process(catalog.data, query.data)` (that is, this module uses information from each to decide what to do)

Argue for and against each choice, and suggest others if you can.

We'll work with option (a) for now. For our first cut, we'll start with a Catalog, and just use strings for queries and items. Here's a test:

```
Catalog catalog;
public void setUp() {
      catalog = new Catalog();
      catalog.add("Hammer – 10 lb");
      catalog.add("shirt – XL – blue");
      catalog.add("shirt – L – green");
      catalog.add("Halloween candle – orange");
      catalog.add("Halloween candy – gum");
}

public void testSimpleQuery() {
      List result = catalog.itemsMatching("shirt");
      AssertEquals(2, list.size());
```

```
}
```

MORE-2. Catalog.
Write a simple Catalog class with the two implied methods.

MORE-3. Query.
Extract the current query string into a Query class.

Your catalog class probably has a line something like

```
if (item.indexOf(query) != -1)
```

modify this to

```
if (item.indexOf(query.toString()) != -1)
```

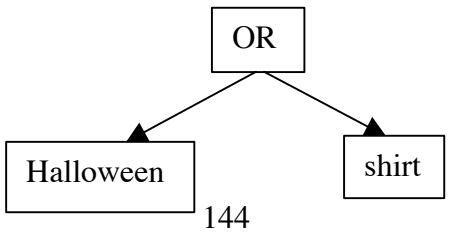
as being the least change.

MORE-4. Trading off smells.
Explain how this adds a small case of Feature Envy or Inappropriate Intimacy in the process of cleaning up a little Primitive Obsession.

Solution ideas on page 185.

MORE-5. Move to Query.
Extract Method **on "`item.contains(query.toString())`", then** ***Move Method*** **to put the matching behavior on the query. (In the end, you'll call "`query.matches(item)`".)**

We'd like to expand our queries to support more complex queries. "Halloween OR shirt" should find several items. In this exercise, we won't worry about parsing; instead assume our queries will be composites like this:



We'll call the top node an OrQuery, and the others will be StringQuery. Both these will be subclasses of Query.

MORE-6. StringQuery and Query.

***Extract Superclass* (or subclass, depending how you look at it:), so StringQuery becomes a subclass of Query.**

Sometimes you'll hear people speaking of a refactoring "making room" for new code. This is an example of that: we're making room for an OrQuery subclass. You could come at it the other way, introducing the OrQuery and then removing the duplication.

MORE-7. OrQuery.

Create an OrQuery class that meets this test:

```
Query query = new OrQuery (
      new StringQuery("shirt"),
      new StringQuery("Halloween"));
list = catalog.itemsMatching(query);
assertEquals(4, list.size());
```

One way to think of this approach is as an example of the Strategy pattern. The catalog is the context, and the different queries each implement different algorithms for matching. (You might also detect a Composite pattern in the way the queries are constructed.)

MORE-8. Performance.

Suppose a profile showed that our queries are too slow. Furthermore, we find that queries are used thousands of times per day, while additions are done only once per day, in a batch. We also find that any given word is unlikely to appear in more than a small fraction of the items.

What ways can you suggest to speed up performance?

Solution ideas on page 185.

The road not (yet) taken

Suppose we use the same `setup()` routine, but start with a different test:

```
Query query = new Query("shirt");
List list = query.matchesIn(catalog);
AssertEquals(2, list.size());
```

MORE-9. A different test.
Implement Catalog and Query for this test.

(You'll probably loop over items, and call `query.matches(item)` somewhere inside.)

MORE-10. StringQuery and Query again.
Refactor StringQuery to be a subclass of Query (as before).

MORE-11. OrQuery again.
Implement OrQuery. It should use `matchesIn(catalog)` on each of its subqueries.

You probably found that List wasn't the most convenient type; you'd really prefer a Set (so "or" doesn't produce duplicates). You can change the List types to Sets, but you might also take this as a warning: perhaps the result wants to be its own object, or perhaps the result of a query is another catalog.

MORE-12. Items versus sets.
At some level, this seems more complicated than the earlier implementation. I think it's because the earlier one compared individual items, and this one deals in sets.

A. With this scheme, what sort of queries will go to the catalog?

B. What opportunities are there for performance improvement?

C. Describe this version as an implementation of the Interpreter pattern.

Solution ideas on page 185.

The first optimization exposed information about the query (words to filter by) to the catalog. The second optimization exposed information about the catalog (sets of items) to the query.

A Third Way

The first alternative passed the query to the catalog. The second passed the catalog to the query. This couples these classes to each other: one must know about the other. A third approach might be to let `Interrogator.evaluate()` take some information from the catalog, and combine it with some information from the query, so that only it knows both classes.

MORE-13. FilterEnumerator.

A. Make Catalog provide an enumerator of its items. Make Query determine whether an item is acceptable. (Both classes know items, but neither knows the other.) Create a FilterEnumerator class: it's an enumerator that takes another enumerator and a query, and only returns items that match the query. Interrogator will set up the objects and use the new enumerator.

B. This is an example of a pattern; which one?

C. Does this solution seem more similar to the first or second approach?

D. What are the performance implications?

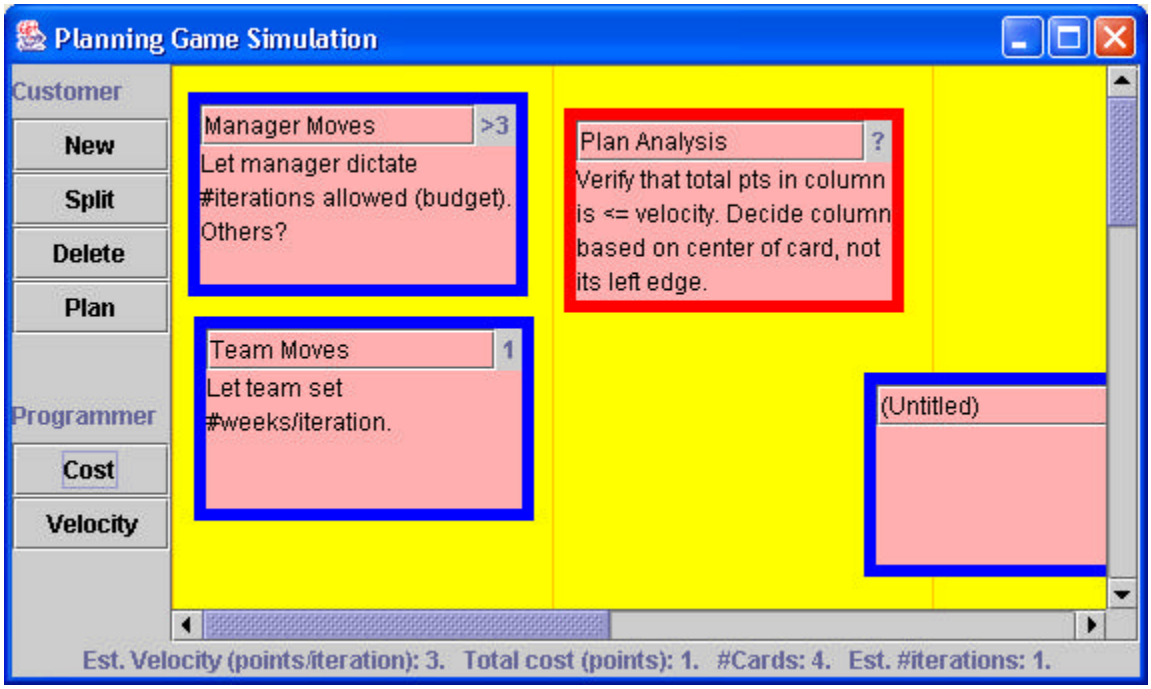
MORE-14. Other approaches.
Are there other approaches that keep Query and Catalog from knowing about each other?

Conclusion

We started with two objects that needed to collaborate, and looked at three approaches to their core interaction. I don't have a general rule to tell which approach is better. Note that they pointed toward different systems in the end. Choosing the balance of responsibility between objects is still an art.

Chapter 16: Planning Game Simulator

We have a GUI program we'd like to clean up. It's a simple simulation of the planning game, and it looks like this:



The customer can create new cards, split existing cards, delete old cards, and get a simple analysis of the plan. The programmer can put a cost estimate on a card, and update the velocity estimate. In addition, anyone can type on a card or move it around by dragging its border. Notice also that there is summary information on the bottom of the screen, updated after each action.

The code for this program is built around three objects: Table (the overall application, including the buttons), Background (where cards are put), and Card (representing an index card).

Part 1. Original Objects

Code (available online)

Table.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.JTextComponent;
```

```
public class Table extends JFrame {
    public static void main(String[] args)
    {
        new Table();
    }

    private JComponent body;
    private Card selection;
    private JLabel summary;
    int velocity = 1;
    int budget = 4;
    JTabbedPane tabs;

    public Table()
    {
        super("Planning Game Simulation");
        JComponent buttons = makeButtons();
        this.getContentPane().add(buttons, "West");

        body = new Background(1000, 1000, 190);
        body.addContainerListener(new ContainerListener() {
            // Listen for container changes so we know when
            // to update selection highlight
            public void componentAdded(ContainerEvent e)
            {
                resetSelection();
            }

            public void componentRemoved(ContainerEvent e)
            {
                resetSelection();
            }

            private void resetSelection()
            {
                if (selection != null)
                    selection.setBorder(
                        BorderFactory.createLineBorder(Color.blue, 6));

                if (body.getComponentCount() == 0) {
                    selection = null;
                } else {
                    selection = (Card) body.getComponent(0);
                    selection.setBorder(
                        BorderFactory.createLineBorder(Color.red, 6));
                }
            }
        });
        JScrollPane scroll = new JScrollPane(body);
        scroll.setPreferredSize(new Dimension(100, 100));
        this.getContentPane().add(scroll, "Center");

        summary = new JLabel("", SwingConstants.CENTER);
        summary.setText(summary());
```

```
    this.getContentPane().add(summary, "South");

    this.pack();
    this.setSize(800, 600);
    this.show();
}

private JComponent makeButtons()
{
    JPanel panel = new JPanel(new GridLayout(0, 1));

    panel.add(new JLabel("Customer"));
    makeCustomerButtons(panel);

    panel.add(new JLabel(" "));

    panel.add(new JLabel("Programmer"));
    makeProgrammerButtons(panel);

    JPanel outer = new JPanel(new BorderLayout());
    outer.add(panel, "North");
    outer.add(new JLabel(""), "Center");
    return outer;
}

private void makeCustomerButtons(JPanel panel)
{
    JButton button;
    button = new JButton("New");
    panel.add(button);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e)
        {
            Card card = new Card();
            body.add(card, 0);
            selection = card;
            updateCost();
            repaint();
        }
    });

    button = new JButton("Split");
    panel.add(button);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e)
        {
            if (selection == null) return;
            Card card = new Card(selection);
            body.add(card, 0);
            selection = card;
            updateCost();
            body.repaint();
        }
    });
```

```
    button = new JButton("Delete");
    panel.add(button);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e)
        {
            if (body.getComponentCount() == 0) return;

            body.remove(0);
            selection = null;
            if (body.getComponentCount() != 0)
                selection = (Card) body.getComponent(0);
            updateCost();
            body.repaint();
        }
    });

    button = new JButton("Plan");
    panel.add(button);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e)
        {
            StringBuffer report = new StringBuffer();
            // Check for cards that need est. or splitting
            for (int i = 0; i < body.getComponentCount(); i++) {
                Card card = (Card) body.getComponent(i);
                if (card.needsEstimate())
                    report.append(
                      "Needs estimate: " + card.title() + "\n");
                else if (card.needsSplit())
                    report.append(
                      "Needs to be split: " + card.title() + "\n");
            }

            if (report.length() == 0)
                JOptionPane.showMessageDialog(
                  body,
                  "Plan OK; no cards need estimates or splitting",
                  "Issues in plan",
                  JOptionPane.OK_OPTION);
            else
                JOptionPane.showMessageDialog(
                  body, report.toString(),
                  "Issues in plan", JOptionPane.OK_OPTION);
        }
    });
}

private void makeProgrammerButtons(JPanel panel)
{
    JButton button;

    button = new JButton("Cost");
    panel.add(button);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e)
```

```
            {
                if (selection == null) return;
                selection.rotateCost();
                updateCost();
            }
        });

        button = new JButton("Velocity");
        panel.add(button);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                velocity++;
                if (velocity >= 10) velocity = 1;
                updateCost();

            }
        });
    }

    private void updateCost()
    {
        summary.setText(summary());
    }

    private String summary()
    {
        StringBuffer result = new StringBuffer();
        result.append(
            "Est. Velocity (points/iteration): " + velocity + ".   ");
        result.append("Total cost (points): " + cost() + ".   ");
        result.append("#Cards: " + body.getComponentCount() + ".   ");
        result.append(
            "Est. #iterations: "
            + (cost() + velocity - 1)/ velocity + ".   ");
        return result.toString();
    }

    // Total cost of the set of cards
    private int cost()
    {
        int total = 0;
        for (int i = 0; i < body.getComponentCount(); i++) {
            Card card = (Card) body.getComponent(i);
            total += card.cost();
        }
        return total;
    }
}
```

Background.java

```
import java.awt.*;
import javax.swing.*;

public class Background extends JLayeredPane {
    int lineDistance;

    public Background(int width, int height, int lineDistance)
    {
        super();
        setPreferredSize(new Dimension(width, height));
        this.lineDistance = lineDistance;

        setOpaque(true);
        setBackground(Color.yellow);
        setForeground(Color.orange);
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int height = getHeight();
        for (int i = 0; i < this.getWidth(); i+= lineDistance)
            g.drawLine(i, 0, i, height);
    }
}
```

Card.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class Card extends JPanel {
    static int cards = 0;

    final static int scale = 34;            // card size factor
    final static int height = 3 * scale;    // 3x5
    final static int width = 5 * scale;

    final static int rollover = 12; // New cards go in difft. pos'ns
    final static int offset = scale;// How far apart to put new cards

    JTextField title;
    JTextArea body;
    JLabel costLabel;
    int cost;

    public Card()
    {
        this("(Untitled)", "");
    }

    public Card(Card from)
```

```
{
    this(from.title.getText(), from.body.getText());
}

private Card(String titleText, String bodyText)
{
    super();
    setBorder(BorderFactory.createLineBorder(Color.blue, 6));
    cards++;

    setLayout(new BorderLayout());
    JPanel top = new JPanel(new BorderLayout());

    title = new JTextField(titleText);
    title.setBackground(Color.pink);
    title.addFocusListener(new FocusAdapter() {
        public void focusGained(FocusEvent e)
        {
            moveToFront();
        }
    });

    top.add(title, "Center");
    costLabel = new JLabel(" ? ");
    cost = 0;
    top.add(costLabel, "East");
    add(top, "North");

    body = new JTextArea(bodyText);
    body.setLineWrap(true);
    body.setWrapStyleWord(true);
    body.setBackground(Color.pink);
    body.addFocusListener(new FocusAdapter() {
        public void focusGained(FocusEvent e)
        {
            moveToFront();
        }
    });

    add(body, "Center");

    setLocation(
        (cards % rollover) * offset,
        (cards % rollover) * offset);
    setSize(width, height);

    // Turn on mouse events so we can detect being dragged
    enableEvents(
        AWTEvent.MOUSE_EVENT_MASK
        | AWTEvent.MOUSE_MOTION_EVENT_MASK);
}

private void moveToFront()
{
    JLayeredPane background = (JLayeredPane) this.getParent();
```

```
    background.moveToFront(this);
}

private int lastx, lasty;

public void processMouseEvent(MouseEvent e)
{
    if (e.getID() == MouseEvent.MOUSE_PRESSED) {
        lastx = e.getX();
        lasty = e.getY();
        moveToFront();
    } else
        super.processMouseEvent(e);
}

public void processMouseMotionEvent(MouseEvent e)
{
    if (e.getID() == MouseEvent.MOUSE_DRAGGED) {
        Point here = getLocation();
        setLocation((int) (here.getX() + e.getX() - lastx),
                    (int) (here.getY() + e.getY() - lasty));
    } else
        super.processMouseMotionEvent(e);
}

public void rotateCost()
{
    String label = costLabel.getText();
    if (label.equals(" ? ")) {
        costLabel.setText(" 1 ");
        cost = 1;
    } else if (label.equals(" 1 ")) {
        costLabel.setText(" 2 ");
        cost = 2;
    } else if (label.equals(" 2 ")) {
        costLabel.setText(" 3 ");
        cost = 3;
    } else if (label.equals(" 3 ")) {
        costLabel.setText(" >3 ");
        cost = 0;
    } else if (label.equals(" >3 ")) {
        costLabel.setText(" ? ");
        cost = 0;
    } else {  // shouldn't happen
        costLabel.setText(" ? ");
        cost = 0;
    }
}

public int cost()
{
    return cost;
}

public String title()
```

```
    {
        return title.getText();
    }

    public boolean needsSplit()
    {
        return costLabel.getText().equals(" >3 ");
    }

    public boolean needsEstimate()
    {
        return costLabel.getText().equals(" ? ");
    }
}
```

Challenges

PG-1. Smells.

A. What smells do you sense in this code? Use the list in the *Refactoring* catalog for inspiration; you'll probably find other things as well.

B. What strategy (order of refactorings) would you use to improve the code?

Solution on page 185.

PG-2. Tests.

What about tests? The author claimed, "It's basically all GUI, so I couldn't test it." Identify two or three strategies by which we could test this code, with little alteration to the current structure.

Solution ideas on page 187.

PG-3. Write tests.

Write at least one test for each object, using the simplest strategy you identified.

Solution ideas on page 187.

When faced with a moderately big refactoring task like this, I like to start easy, by first fixing smells like "poor names" or "long methods". When code has many things that can be improved, it can be helpful to work with it on its own terms for a while before trying dramatic improvements.

PG-4. Dead code.

There's some unused code in the Table class (in the form of unreferenced variables). Remove the dead code.

PG-5. Table => PlanningGame.

"Table" stinks as the name for the top-level class. ("What table?!?") Even "Main" would be a better name; it wouldn't communicate much, but at least it wouldn't *mis*-communicate. Let's try "PlanningGame" as the name for this class: rename the class appropriately.

PG-6. Anonymous inner classes.

The PlanningGame class has a number of places that use an anonymous inner class to set up event listeners. (Anonymous inner classes are the ones where the routines are defined in the middle of code; you can spot them by seeing "});" somewhere.) "Convert Anonymous Inner Class to Inner Class" by extracting each anonymous inner class to its own inner class.

PG-7. Magic numbers.

There are many "magic numbers" floating around. (Some of them might be better called "magic colors.") Extract each to the top of the class as a final static variable. (Focus on numbers and colors; we'll deal with strings later.)

Part 2: Redistributing Features

The story so far: we have a planning game simulator built around three classes: PlanningGame (managing the controls and the background), Background (a playing surface), and Card (for the index cards). We identified a number of places that could be improved, created a few simple tests, and did some simple refactorings. The starting-point code for this part is available at http://www.xp123.com/rwb/.

There were several places in PlanningGame (formerly Table) that accessed fields inside Background. This is a case of "Feature Envy": a class spends time manipulating data that is somebody else's responsibility. The cure is to move data and methods around.

PG-8. Test the buttons.

Before we move features around, we want to make sure our tests will warn us if we've done so incorrectly. My tests so far only called publicly available methods, and this didn't include the ability to simulate button clicks. If your tests don't have this ability, add it in. (You can expose the buttons "one by one," or you might find a more generic approach; once you have a button, you can call `doClick()` to make it do its thing.)

Solution ideas on page 187.

Notice how the Plan button causes trouble: it wants to create a dialog, which is a pain to test programmatically. I'll just leave it for now, and be extra careful when working on it. (This is one of those places where you want to refactor to enable you to create a test, but you want a test to help you safely refactor.)

PG-9. Move features to Background.

- **Extract the "string computation" part of the "Plan" button and move it to Background. (You may have to test it manually, but you should be able to automate a test after the refactoring.)**
- **Move `cost()` to Background.**
- **Create a `count()` method on Background that looks at `getComponentCount()`; adjust the callers.**
- **Create a `top()` method on Background that returns `getComponent(0)` and adjust the callers. (I'd try returning something of type Card for now because the callers all want to cast it anyways.)**

The latter two changes are more in the spirit of improving communication than decoupling.

PG-10. Move features to Card.

The Card doesn't know its own selection status. Change that by pulling over the code for a new method `setSelected(boolean)`. I had expected to need a method `isSelected()` but nobody looks at the selection status!

Solution ideas on page 188.

I'm not fully happy with the selection handling, nor with the fact that the Card knows about its background. But I'm not quite sure how to handle it, and there is still a lot of duplication around the "cost" in a Card, and the button-making in PlanningGame. Since I have ideas for those, I'll tackle them next.

PG-11. Clean up "cost."

- **Get rid of those annoying spaces surrounding each label. (It's pretty clear they're just for padding.)**
- **Extract a Cost class. Adjust the test classes.**

Pulling this class out made me wonder where `Card.needsEstimate()` and `Card.needsSplit()` are called. Each is called by `Background.planAnalysis()` and the tests. But `planAnalysis()` is asking the Card for those checks; why not let the card compute its own analysis? (Later, there might be other analyses not at the card level, but we'll let that day take care of itself.) So eliminate this feature envy by putting a `planAnalysis()` method on Card. This is a problem I hadn't noticed earlier, but the refactoring made it clear.

Five Whys

While we're looking at `planAnalysis()`, notice the last argument of `JOptionPane.showMessageDialog()`. Edmund Schweppe pointed out that OK_OPTION is not the right constant for method. That argument is intended to tell whether it's a warning, informational, etc. message; if we talk to our customer, we find out that it should be a warning (if there are any problems) or an informational (otherwise). It's not unusual to find a bug during refactoring.

Lean manufacturing has the notion of asking "why" five times to get at the root causes of problems. The outermost symptom was, "It always shows the stop sign—is that really what the user wants?" One level back, we see that the wrong constant was used. But why wasn't it detected? One reason is that we don't have any tests for dialog types. (Yes, they're a pain to test.) Note also how the code was formatted: one very long line, with the wrong constant at the far right end. In my editor, this means I would never have a chance

to see the constant unless I explicitly went to the end of the line. (Personal Resolution: update my personal coding style to format one argument per line on long lines.)

Why was it written wrong in the first place? One reason is that I didn't really have documentation at hand; I relied on the editor to suggest constants and grabbed one that sounded right. (Personal Resolution: Get a new JFC manual and don't rely on memory so much.) In this case, I was coding solo (my partner could easily have been looking up the arguments while I was typing).

And why is it possible to make this kind of error? Partly, it's a result of a decision the library designers made: to just use integer constants for all the option types and message types. The library designers could have created separate objects for these options (as a sort of enumeration); or, they could have used non-overlapping values or different methods for the different message types. (Personal Resolution: watch out for "primitive obsession.")

Even a small mistake has the seeds of many lessons.

Removing Duplication, Selection Troubles, and a Few Burrs

So where are we?

- Cost – I'm reasonably happy with this class. There are other ways to do it, but I'll take it as fine for now.
- Card – I'm still not thrilled about the cards/rollover stuff, but I'll leave it. Selection could be improved – should the card notify its focus listeners when it's clicked? This would remove its dependency on Background.
- Background – Other than changing Card and selection, I'm happy with it.
- PlanningGame – There's still duplication in the setup. Ron Crocker pointed out that we could let `makeCustomerButtons()` and `makeProgrammerButtons()` take the responsibility for putting in the proper label. The whole handling of containers and selection is still troublesome. And we're still calling `updateCost()` in a bunch of places.

PG-12. Button creation.

Clean up button creation in PlanningGame.

- **Move the labels into the "make" routines.**
- **Eliminate the duplication in constructing buttons.**

Solution ideas on page 188.

I've been trying to understand why the selection handling bothers me so much. I think it's based on the responsibilities of the classes: I want the planning game screen to be

responsible for holding the buttons, the Background, and the summary. Currently, it tracks the selection because that's what the action listeners work with. But I think we should instead make the Background track all the selections; some of the action listener work belongs over there too.

This change is a bit tentative, so I would certainly "checkpoint" my code so I could return to it if I didn't like where things end up. (I think of checkpointing as saving a version for easy recovery without releasing it to the whole team; this may or may not be easy in your environment.)

PG-13. Move selection handling to Background.

- **Make Background hold the selection.**
- **Move appropriate listener behavior over to Background (at least from New, Split, and Delete).**
- **Move the ContainerListener.**

Solution ideas on page 188.

Updating the cost is done by any button press that might change it. Each command must be careful to include a call to `updateCost()` after any changes it causes. In the current code, this is done by putting the call in each listener. One way to reduce this duplication would be to subclass listeners from a common parent, and make the parent responsible for the call.

An alternative would be to introduce listener-style notification; the summary would listen for changes in the underlying Background or velocity, and update itself accordingly. PropertyChangeEvent fits our need. On Background, the two things we typically monitor are the count and the cost. Rather than be specific about which property has changed, I'll use the standard bean convention that "null" for a property change means anything might have changed.

This change is a refactoring, but it's into the realm of "large refactoring" (or would be if this program were of a substantial size). I don't feel like my tests will adequately cover it, so I'll introduce new tests to make sure Background sends events. Since this is a relatively big change, checkpoint before you start.

PG-14. Property changes.

Transform PlanningGame so cost updates are triggered by property changes from Background rather than explicit calls.

Solution ideas on page 188.

The action listeners still have duplication: several of them need to check whether there are any cards before they do their work. This suggests a change: why not disable them unless they apply? This is a user interface change, but our user likes the idea.

PG-15. Enable buttons.
Make buttons enabled only when appropriate. (Test-first, of course.)

Solution ideas on page 188.

Look back at Card. It still depends on Background: when a card is clicked, it looks at the background and tells the background to move that card to the front. Since Background knows about Card, and Card about Background, this sets up an undesirable dependency.

PG-16. Card and Background.
Remove Card's dependency on Background.

- **Make the Card send a property notification when it is clicked. (You could make the case for a different notification if you wanted.)**
- **Make the Background listen for the new notification.**
- **Remove the ContainerListener aspect of Background. (Since Background now controls the `moveToFront()` call, it already knows when the contents are changing.)**

Solution ideas on page 188.

PG-17. Cleanup.
Go through your tests and code one more time, doing any simple cleanup you can.

Solution ideas on page 189.

Compare your code now to the original. Does it communicate better? Is it simpler? Better structured?

Part 3: Exercises for the Reader

Part 3 contains some ideas for pushing the code further, left as exercises for the reader.

PG-18. Remaining smells.
What smells remain in your code?

I've noticed that velocity updates are handled differently than other ones; I still have that "rollover" calculation in Card that I never liked; there are still some bits of duplication around; and I still think a separate model layer would help. In some ways, the current design is near a "local maximum": it's becoming reasonable for a design that keeps the model and view together, but we might have to go "off this hill" a little to go up a bigger peak.

If you're not familiar with the model/view or model/view/controller idea, you may want to do some background reading: the Observer pattern in *Design Patterns,* the TableModel (or other models) in the Java libraries, or the HotDraw framework (described at http://c2.com/cgi?CrcCards and http://www.c2.com/doc/crc/draw.html).

We might start by creating a Card model. Right away we'll face a decision: should it contain the model for the title, or should we use the one Java provides in the TextField? If we let the Card have its own model, we'll have to be careful not to cause a notification loop (where the Card notifies the TextField of a change, so it tells the Card, which notifies the TextField, and so on). If we use the Java-provided model, our hookup will be a little trickier.

PG-19: Separate Model.

A. Make sure to save the old version before you start these changes.

B. Apply *Duplicate Observed Data* to change Card to Card and CardView. (Note that this is not a trivial refactoring. Don't be surprised if this exercise takes a whole session.)

C. Apply *Duplicate Observed Data* to Background.

- **Feel free to make use of the DefaultListModel, which builds in notification.**
- **You can get very fine-grained notification about list changes, but start by just re-building the whole list when told it has changed.**
- **Note that the card's location and selection status are stored in the view. (Is this the right place?) If you try to regenerate views, you must make sure to track this information.**

D. Apply *Duplicate Observed Data* to Plan.

E. Divide the program into two separate packages: one for all the models, the other for views. Let view classes depend on the model classes, but not the other way around. (You may find you need to create separate test classes for models and views.)

F. This was a lot of work. Is it an improvement? What future changes will be easier because we've done this? (If it's not an improvement, restore your old version.)

PG-20. An optimization.

The simplest form of notification looks like this:

```
public void setTitle(String title) {
    this.title = title;
    notify();
}
```

A more sophisticated implementation is:

```
public void setTitle(String title) {
    if (this.title.equals(title)) return;
    this.title = title;
    notify();
}
```

A. When is this a performance optimization?

B. What other purpose does it serve?

Solution ideas on page 189.

PG-21. New features.

Consider new features that might be added. How robust is our implementation in the face of these?

- **New cards should be placed top-to-bottom, left-to-right instead of diagonally.**
- **We'd like buttons for the manager or team roles**
- **All colors and sizes should be made into preferences**
- **Plan analysis should detect "too many points in an iteration"**

- **Persistence – so you can save and restore your work**
- **Distributed support – so people on multiple sites can simultaneously work with it**

What other features might you add?

PG-22. Test-Driven Development. Re-implement this code from scratch, test-first. Don't look at the old version while you develop the new one. What do you see?

The code for this exercise was not originally written test-first. The experiences of people who do test-driven development indicate that a different design often emerges than the one they expected. Did that happen for you? Is the code better? Are the tests better? How much did the original design influence you?

PG-23. Lessons from test-driven.

Assuming your test-driven code is different than the code you were working with before, could you refactor the old code until it matches the new code. Are there refactorings not "in the book" that you need to transform your code? What "smells" could guide you so that you would naturally refactor in that direction? Does this teach you anything about refactoring, or about test-driven development?

Chapter 17: Where to Go From Here

One of the premises of this book is that refactoring is a skill, and benefits from practice. Look for opportunities to practice and use this skill.

Books

All the books in the bibliography will repay their study. But if you haven't yet acquired Martin Fowler's *Refactoring,* you should seriously consider doing so. The exercises in this book touch on perhaps half of the refactorings he catalogs. Tools are getting better at the mechanics of refactoring support, but it will be a long time before they effectively cover every aspect of this book.

Admonitions

Build refactoring into your practice

Knowing how to refactor isn't worth much… unless it's applied. Resolve to make your code "lean and clean." On an XP team, this is part of everyday life. But even heavily design-driven approaches expect programmers to implement the design well.

Build testing into your practice

There's an old adage (as so many are), "If it ain't broke, don't fix it." (How many times has the last "simple change" caused an unexpected bug?) In programming, the downside of applying this adage is that the code just gets uglier and uglier.

Refactoring is willing to go against this rule, through two mechanisms: safe refactorings, and a supply of tests to verify that the transformations have been done correctly. Don't neglect your tests.

Get help from others

Get other peoples' opinions about your code, whether through pair programming, design/code reviews, or simply bugging your neighbor. One of the things that really got hammered home to me in writing this book is that almost any code can be improved (and sometimes we get to take advantage of a whole internet's worth of help!).

Exercises to Try

Smell Scavenger Hunt/Smell of the Week

Pick a smell, and find as many occurrences of it as you can. Every week, search for a new smell.

Re-Refactor

Pick a good-sized piece of code (either your own, or one of the larger examples in the back of this book would work). Each day, start from the initial version, and refactor as far as you can in ten minutes.

Do you sense the same things each day? Do you get farther?

Just Refactor

Pick or develop a project. Spend ten minutes refactoring. (Each day, start where you left off the day before.)

Inhale/Exhale

Find code demonstrating some smell. Apply a refactoring that addresses it. Then apply the refactoring that reverses that one. Repeat this two more times.

Defactoring/Malfactoring

"Defactoring" and "malfactoring" are names I use for malicious refactoring: *worsening* the design of existing code. Take some code, and "refactor" it to make it as smelly as possible. (It's harder than it sounds.)

In addition to providing practice at refactoring, this may also help you realize when you're unintentionally malfactoring during development.

Be sure to restore the original after you've had your fun.

Refactoring Kata

A kata is a martial arts exercise that you repeat every day, for practice and to help get into the rhythm of the art. (A traditional series might be a defense against four opponents.) Develop a kata for refactoring: a program where you'll apply a fixed series of refactorings. Pick a series of smells and refactorings that you see or use often; for me, that might include some primitive obsession, some long methods, some observed data to duplicate, and some responsibilities to re-balance.

This will give you a chance to hone your editing skills and your understanding of your environment, as well as practice "smelling" and refactoring.

Web Sites

- http://groups.yahoo.com/group/refactoring - A group for discussing refactoring.
- http://groups.yahoo.com/group/extremeprogramming - A group for discussing XP. There's a lot of discussion unrelated to refactoring, but refactoring is one of the key practices in XP.
- www.refactoring.com - Martin Fowler's site associated with *Refactoring*. He has most of that book's catalog online, along with contributions from others.
- www.industriallogic.com - Joshua Kerievsky's corporate site. He's working on a book, *Refactoring to Patterns,* and has other interesting articles and games as well.
- www.xp123.com/rwb - In general, my site is focused on XP, and the rwb area is devoted to this book. You'll find source code for the larger examples, refactoring-related articles, and pointers to refactoring challenges others have proposed.

Bibliography

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

Bentley, Jon. *More Programming Pearls*. Addison-Wesley, 1988.

Bentley, Jon. *Programming Pearls*. Addison-Wesley, 1986.

Bentley, Jon. *Writing Efficient Programs*. Prentice-Hall, 1982.

Cunningham, Ward. http://c2.com/cgi/wiki?CrcCards, and http://www.c2.com/doc/crc/draw.html; Internet.

Flanagan, David. *Java in a Nutshell, 4/e*. O'Reilly: 2002.

Fowler, Martin, and Kendall Scott. *UML Distilled*. Addison-Wesley, 1997.

Gamma, Erich, et al. *Design Patterns*. Addison-Wesley, 1995.

Hunt, Andrew, and David Thomas. *The Pragmatic Programmer*. Addison-Wesley, 2000.

Metsker, Steve. *Design Patterns Java Workbook*. Addison-Wesley, 2002.

Wake, William C. *Extreme Programming Explored*. Addison-Wesley, 2002.

Selected Answers

Chapter 2 – The Refactoring Cycle

CYCLE-1. (page 11)

Many refactorings reflect this attitude (safety even in mid-refactoring). Consider our "Encapsulate Field" example. Even this simple program compiled three times. Others to look at include [TBD]

CYCLE-2. Simple design. (page 11)

A.

a) Passes all tests. "If it doesn't have to work, I can give it to you right now."
b) Communicates. This makes an appeal to our intuition about future readers of our code (including ourselves).
c) No duplication. Duplicate code is asking for trouble; it's too vulnerable to changes in one place but not the other.
d) Fewest classes and methods. All things being equal, we prefer "smaller" code.

B.

For me, the bottom line is that there's an appeal to the reader's ability to understand; we'll tolerate duplication to achieve better understanding.

Robert Wenner (personal communication) notes that JNI (Java Native Interface) code, linking Java to C, will have duplication between methods, but each method is just different enough to not make it worth pursuing. Each method has to deal with different arguments and different return values for each function it accesses.

I find that test code will sometimes have duplication, for communication reasons. For example, it may be easier to repeat an expected value rather than assign it to a variable and use the variable. Then, when you read the code, you know exactly what it was looking for, and you don't have to skim code to find the variable and make sure nothing else changed it along the way.

Chapter 3 – Measured Smells

MEASURE-1. (page 14)

A. Which smell? Commented code

B. Can everything important about the code be communicated using the code alone? Or do comments have a place?

Code can usually communicate "how" fairly well; it's not always so well able to communicate "why"; and it's almost impossible to communicate "why not."

When code becomes published for others, some people find it important to include JavaDoc comments to provide extra explanation.

MEASURE-2. Long method. (page 18)

A.

Block 1 – `out.println("FACTORY REPORT\n");`

Block 2 – From there until `out.print ("Robot");`
Block 3 – From `out.print ("Robot");` until `out.println();`
Block 4 – `out.println("=======\n");`

You might have chosen a slightly different set of blocks.

B.
```
public static void report (PrintStream out, List machines, Robot robot)
{
    reportHeader(out);
    reportMachines(out, machines);
    reportRobot(out, robot);
    reportFooter(out);
}
```

We wouldn't stop here, but this would be a good first step. (We could either move toward a Report class, or toward putting `report()` methods on the Machines and Robot classes.)

C. Does it make sense to extract a one-line method? Yes, if it communicates better.

MEASURE-3. Large class. (page 22)

A. It's doing a lot of things: some are inherited, but it seems to have a variety of responsibilities.

B. Go through the methods listed, and categorize them into five to ten major areas of responsibility.
My list (yours will vary) –

Columns	Editing	Rendering	Model	Selection
Appearance	Notification	Other		

C. It might be possible to generalize addresses, so there wouldn't be so much need to have symmetrical row and column functions. It's possible to pull out helper objects that would own some corner of the responsibility. Some of the methods appear to simply consult the table or column model for their answer.

MEASURE-4. Long parameter list. (page 26)

A.
`x/y/width/height` sounds like a Rectangle.
`value/extent` sounds like a range. (Notice that the limit parameters use min and max rather than a second range; the whole thing might be cleaner if there were a separate Range object.)
`x1/y1/x2/y2` sounds like another Rectangle.

B. In some ways, it's a reflection of an attempt to make a class more generic – pass in everything it could work with. Things like graphics tend to want to be "stateless," and using lots of parameters can help them do that.

C. There are a lot of occurrences of row/column together; that makes me wonder if the object would be simplified using some sort of "locator" instead. It would probably simplify selection, and it might address challenges such as cells that span two rows.

MEASURE-5. (page 28)

A. Comments

B. Large Class
C. Long Method
D. Long Parameter List

___ Duplicate Observed Data	[Key: B]
___ Extract Class	[Key: B]
___ Extract Interface	[Key: B]
___ Extract Method	[Key: A C]
___ Extract Subclass	[Key: B]
___ Introduce Assertion	[Key: A]
___ Introduce Parameter Object	[Key: D]
___ Preserve Whole Object	[Key: D]
___ Rename Method	[Key: A]
___ Replace Parameter with Method	[Key: D]

MEASURE-6. (page 28)
A. Which do you see or create most? Everybody's list will be different. Long Method and (Unhelpful) Comment are the two I see most. Of that list, Long Method is probably the one I self-inflict the most.

B. For these "measured" smells, you can give yourself a cutoff number that tells you to review what you're doing. For example, I check twice if a method exceeds about seven lines, and I question any comments in the body of a method. Define your own triggers.

Interlude 1 - Smells and Refactorings

I1-1. Put a checkmark by each refactoring for each time a smell references it. (page 31)

I1-2. Which refactorings fix the most smells? (page 31)
Move Method, Extract Class, Move Field, and Extract Method

I1-3. Which refactorings aren't mentioned by any of the smells? Why not? (page 31)
It's a long list. Some are just code manipulation; it's not that either way smells, but rather the refactoring provides a safe way to move between two valid alternatives. Others are a bit specialized (especially the "big" refactorings). Others are used as steps in applying another refactoring; the smell for the other refactoring triggers this one.

I1-4. Does this list suggest any other smells you might want to be aware of? (page 31)
Everybody's list will be different. I added these:

- Intertwined Model and UI – *Duplicate Observed Data*, *Separate Domain from Presentation*
- Cast – *Encapsulate Downcast*
- Unclear Communication – *Remove Assignment to Parameter*, *Replace Error Code with Exception, Replace Exception with Test, Replace Magic Number with Symbolic Constant, Split Temporary Variable*
- Conditional Logic – *Consolidate Conditional Expression, Consolidate Duplicate Conditional Expression, Introduce Null Object, Replace Error Code with Exception, Replace Exception with Test, Replace Nested Conditional with Guard Clause, Replace Conditional with Polymorphism*.

Chapter 4 – Duplication

DUP-1. Two libraries. (page 37)

A. One strategy:

- Define a new logger whose interface is compatible with the JDK 1.4 logger. It could be a simplified "layer" interface, or a class with a compatible interface (that in the future would be a subclass), or it might be a straightforward implementation of the new classes.
- Make the old loggers call the new logger.
- Modify Log and its callers to become like the new logger, so you can delete the Log class.
- Modify Logger to become like the new logger, so you can delete the Logger too.

There will be a temptation to do this relatively slowly: "for now, use the new logger for new and changed code." Note that this adds to our conceptual burden. You might be able to use automated support to make it easier.

DUP-2. Properties. (page 39)

A. Use *Extract Method* to pull out a routine that looks up the property, converts it to an integer, and validates it as positive. (Do this in steps: first, second, and third copies.)

You might determine that it's OK to set `monitorTime` and `departureOffset` even if the exception will be thrown. This will reduce the need for temps.

The end result might look like this:

```
checkInterval = getProperty("interval");
monitorTime = getProperty("duration");
departureOffset = getProperty("departure");

if ((monitorTime % checkInterval) != 0) {
  throw new MissingPropertiesException("duration % checkInterval");
}

if ((departureOffset % checkInterval) != 0) {
  throw new MissingPropertiesException("departure % checkInterval");
}
```

You might then extract a separate method to enforce the "%" restriction.

DUP-3. Template. (page 40)

A. Duplication

- The whole thing is two nearly identical copies, one for %CODE% and one for %ALTCODE%. One case writes to a string, the other to an output stream, though.
- The string literal "%CODE%" and the numeric literal 6 are aspects of the same thing (magic "number"). Likewise %ALTCODE% and 9.
- The construction of the resulting final string for each part is similar: appending a prefix, body, and suffix.

B. Remove duplication

- *Replace Magic Number with Symbolic Constant* (and with functions)
- *Extract Method*
- Look into "unifying" string and I/O so we can get the methods even more similar.

C. "`new String()`"

- What does it do? The new causes a new string with identical contents to be related. The new string `equals()` the old but doesn't == it.
- The intern() method returns a unique instance of the string, that will test == to all other intern'd strings equal to it. (By default, Strings are not required to be shared.)
- Does it apply here? No. Since we never compare intern'd strings, the "`new String`" part is redundant.

DUP-4. Duplicate Observed Data. (page 41)

A. The duplication is often not as dramatic as it first appears. Often, the domain object has its own "object" representation of itself, and the widget ends up holding a string or other display representation.

- The user interface is usually one of the most volatile parts of a program, while the domain objects tend to be modified less often (during development).
- Putting the domain information in the widget ties them together. A domain object should be able to change its value independently of whether the value is displayed on the screen.
- Mixing domain and screen objects makes the domain depend on its presentation; this is backwards. It's better to have them separate so the domain objects can be used with an entirely different presentation.

B. The performance can go either way. When they're in one object, the domain object updates its value using widget methods. This is typically slower as it must take into account buffering, screen updating, etc.

On the other hand, the synchronization can become relatively costly. On some occasions, you have to find a way to make this notification cheaper. In some situations, splitting the domain object and the widget will let the widget avoid being displayed

DUP-5. Java libraries. (page 41)

Examples

- AWT vs. Swing. There are two whole widget libraries included.
- Collections vs. Vector/HashTable. There are two collection libraries.
- Event listeners. There are many closely related variations.
- Duplicate methods in GUI classes (e.g., show() vs. setVisible())
- Others

B. Why?

- The most common reason seems to be that old chestnut, "historical reasons." Sun is understandably reluctant to change published interfaces that many people depend on. Instead of changing things, they add more, even if it overlaps in intent or code.

  Java has a "deprecation" flag that can be set, and is for some of these duplications. This warns clients that they're depending on the "old" way of doing things, and that there's a better way.

- In something as big as Java's libraries, there are many people working on them, and they don't always coordinate well enough to realize that they've duplicated work.

- Java's libraries are very public (with many books and articles describing them in detail). This means they're subjected to more scrutiny than most efforts. That is, their duplication may be no worse than others, it's just that they're more visible.

DUP-6. Points. (page 41)
A. Both are using points that "wrap" around the maxX and maxY values.

B. *Substitute Algorithm* to make both classes calculate wrapping the same way. Then *Extract Class* to pull out a "WrappingPoint" class.

The search for duplication can help you identify these situations. You can create a test that reveals the bug in the "bad" code. While you fix it, you can drive towards similarity to the "good" code, and then use the refactorings that address duplication to clean up the duplication.

DUP-7. Expressions. (page 42)
A. You could use Extract Method on the return statements, so "`return v1 + v2;`" becomes "`return value(v1,v2);`" and similarly for "*". This makes the `value()` routines identical, so you can *Pull Up Method* to bring them to the Composite node.

Chapter 5 – Data

DATA-1. Alternative Representations. (page 47)

Money – (Based on U.S. money, where 100 cents = 1 dollar, and a cent (AKA penny) is the smallest coin.)
- Integer count of cents
- A pair of integers managed as a "long long"
- Use Decimal type
- You may have to track fractions of pennies. (Some money is managed in terms of 1/10 cent.)
- String

Position (in a list) –
- Integer
- If there's only one position of interest, you might manage "the" list (as seen from outside) via two lists, containing those before and after the position.
- Pointer to the indicated item

Range –
- First and last index
- First index and length

Social Security Number (government identification number: "123-45-6789")
- String
- Integer
- Three integers

Telephone number –
- String
- Integer
- Two numbers, area code and local number

- Three numbers: area code, exchange, and last 4 digits

That only considers US phone numbers; it will be more complicated if you add international phone number support.

Street Address ("123 E. Main Street") –

- String
- Multiple fields
- Physical coordinates
- Standardized address (standard abbreviations)
- Index in a standard list of addresses

ZIP (postal) code –

- String
- Integer
- Two integers (U.S. post codes now use "ZIP+4" – "12345-6789")
- Index in a standard list of codes

DATA-2. A counter-argument. (page 48)

It depends on what's happening between the screen and the database. If it's truly a form-filling application, to get this field from the screen into that field on the database, we might not use an object-oriented approach. But as more functions are added that concern ZIP codes (validation, computing shipping distances, mapping routes, etc.), we'll expect more benefit from the object-oriented approach.

DATA-3. How does an Iterator or Enumerator reduce primitive obsession? (page 48)

By using the iterator, code relies less on the "for loop" approach. So there will be less use of integers as position counters.

DATA-4. Editor. (page 48)

```
assertEquals(___, editor.contents(1));
```

A. Given the interface provided, what string would you expect to use in place of the ___?

"a"

B. Based on the variable name (firstParendPosition), what string might you like instead? Of what use would this be?

" ("

We might like positions that remember where they are, even if text is inserted in front of them. For example, a programming editor might track the position of each method declaration.

C. The crux of the problem is the use of "int" as a position index. Suggest an alternative approach.

Instead of handing out "dead" integers, hand out Position objects. But let the editor own them. When text changes, the editor updates the Positions. The holders of the objects aren't aware of that; they just know that they can get one, or hand it back to move to a prior position.

D. Relate your solution to the Memento design pattern.

Memento uses an "opaque" object: in this case, the editor may know what's inside but clients definitely don't. The client can't manipulate the Memento directly, but must hand it back to the main object to use it.

DATA-5. What the classes have in common. (page 51)

- All have public data members
- All are subclasses of Object.
- Most are very stable.
- Most have a well-understood meaning outside of their use in Java (with the exception of GridBagConstraints).
- There may be a lot of them.

DATA-6. Date and Color. (page 51)

A. The Color constructor shows three different representations: an rgb triple of integers, a single integer holding all the values, or an rgb triple of floats. The color value could be an HSB triple as well.

Date could be stored as a set of values (year, month, day, etc.), or as an integer count (seconds or microseconds since some event), or it could even be stored as text.

B. Because clients have no direct access to the fields, they can't change an instance behind that object's "back" (without going through its methods).

Chapter 6 – Interfaces

INT-2. Trees. (page 61)

A. A foreign method seems easier. Since we already have to modify our node to implement the interface, we can put it there. A local extension might be a new subinterface of MutableTreeNode; this would seem better only if the tree clients could use the userObject() without caring about our node type.

INT-3. String. (page 61)

A. Declaring a class final prevents creation of a subclass. This makes it impossible to *Introduce Local Extension*.

B. There may be other reasons, but one reason Strings are final is from a security concern. In Java, Strings are immutable: once set, they cannot be changed. Some security checks rely on this fact. A subclass might subvert immutability. (There may be a performance benefit as well; substring operations can point to part of a string known not to change.)

INT-6. Filter. (page 65)

B. Implements Enumerator.
C. Takes an enumerator (the one to be filtered).
D. Defines the required methods (`hasMoreElements()` and `nextElement()`), and a new method `isValid()` for its subclasses to override.

Chapter 7 – Responsbility

RESP-3. CsvWriter. (page 68)

A. One decision is *where* to write; the other decision is *how* to write.

D. Which version seems better? They're probably not different enough to matter. Either approach could have evolved from a test-driven perspective.

RESP-4. CheckingAccount. (page 72)

A. One decision is the relationship between CheckingAccount and Transaction; another decision is the representation for money.

RESP-6. *Duplicate Observed Data.* (page 74)

This might or might not be a problem. The model/view split often arises out of a real need: pressures to change models and views independently. While the hierarchies may start out in parallel, they needn't remain that way. (You might have multiple views for one model, and none for another.)

You may find that generic models emerge. (Think of how TableModel serves as a generic model for many table-like things.)

RESP-7. Documents. (page 75)

A. It affects places all over the tree.
C. The brief/full and compression/none distinctions will become the "wrapping" types.

Chapter 8 – Unnecessary Complexity

YAGNI-1. (page 81)

A. Java supports multiple inheritance of interfaces, but only single inheritance for classes. By keeping the interface separate, classes can add support for the interface, but stay in the same class hierarchy otherwise.

B. While the framework has only one implementation of the class, it's expected that users of the framework will introduce other classes, and the abstract class forms a handy root for that. If there were no other classes coming, it's conceivable that you still might decide to split out the abstract class, just to make the code communicate better.

C. Should you adopt this three-class structure for your objects? Probably not – it's probably overkill.

D. You'll sometimes see the idea of introducing an interface to "break dependencies." How does it do that?

E.

Current: javax.swing -> javax.swing.tree

One improvement: pull the TreeModel into its own package (depending on nothing). Let javax.swing depend on the package, and javax.swing.tree depend on the package, but not on each other.

An alternative, using the original packages: move the TreeModel into the package with JTree. This leaves javax.swing.tree dependent on javax.swing, but not vice versa. That's an improvement, because if you implemented your own tree model class, you wouldn't need any of the javax.swing.tree package to be present.

Class loading doesn't actually load a package at a time, so package dependency is something of an intellectual construct. But I find it helpful to consider these dependencies, as they can reveal places where things that should be stable rely on things that change, rather than vice versa. (Look at Robert Martin's dependency rules for more thoughts on these lines.)

Chapter 9 – Message Calls

MESSAGE-1. Middle Man. (page 84)

Is a removed middle man an improvement? Not to my eyes – we've reduced the clarity of our code if nothing else. If the event handler is given an ArrayList, it implies a lot more freedom to change it. With the Queue class, we know it's a queue: we can see exactly where the delegate gets touched. Without the Queue class, we have to look across every event handler to be sure that non-queue operations aren't used.

Interlude 3 – Design Patterns

I3-1. What refactorings might you use to evolve to some of these patterns? (page 87)

- Abstract Factory: *Replace Constructor with Factory Method* (several times)
- Factory Method: *Replace Constructor with Factory Method*
- Flyweight: *Remove Setting Method, Change Value to Reference*
- Interpreter: *Extract Method and Move Method* (from Composite)
- Observer: *Separate Domain from Presentation*
- State: *Replace Type Code with State/Strategy*
- Strategy: *Replace Type Code with State/Strategy*
- Template Method: *Form Template Method*

Chapter 10 – Conditional

COND-1. (page 90)

B. An empty string may not be the right choice for a default value in every context.
C. It's possible that extracting a new class for Bin might give you the needed flexibility.

COND-3. (page 96)

A. If this were all there were to it, you probably wouldn't bother eliminating it in this instance.

COND-4. (page 96)

B. You can argue it either way. It's starting to get kind of big for an enumeration of integers though.

C. Some factories use dynamic class loading […] What are some advantages to that?

- The code is simpler (no conditional logic, a single place where class is created)
- The code has fewer direct dependencies (doesn't name the actual driver classes)
- The delivered code can be smaller (no longer necessary to deliver the debugging driver class – nothing depends on it directly

D. What are some disadvantages to the new arrangement?

- Performance is potentially a little worse.
- A simple text scan no longer reveals all the dependencies of the code.
- The configuration is a little trickier; an incorrect name or a bad CLASSPATH can leave the system unable to run.

E. If you've used switch statements as a substitute for polymorphic objects, it can require several refactoring steps to untangle and re-weave the object structure. What four refactorings were described as the sequence that will do this job?

1. *Extract Method* – Pull out the code for each "branch"
2. *Move Method* – Move related code onto the right object
3. *Replace Type Code with Subclass –or– Replace Type Code with State/Strategy* – set up the inheritance structure
4. *Replace Conditional with Polymorphism* – eliminate the conditionals

Chapter 11 – Names

NAME-2. (page 103)

If there's an area of personal taste, it's probably in names. These are just my thoughts.

A. `Clear()` or `erase()` both sound OK (depending on whatever the library or other code uses). `DeleteAll()` seems clunky. `Wash()` might be OK for a pane-of-glass simulation, but seems strained for this purpose.

B. `Push()` is traditional; `add()` is probably OK if that's what everything else in the collection library is using. `Insert()` seems misleading, since stacks don't put items in the middle. `AddToFront()` is odd as well; we think of queues having fronts but stacks having tops.

C. `Cut()` implies that the text is saved somewhere for pasting. `Delete()` is probably best; `clear()` and `erase()` sound like they might apply to the whole document.

D. `Equals()` is the "out-of-the-box" Java word. `IdenticalTo()` might work if `equals()` were inappropriate for some reason. `Matches()` could work, but carries a little baggage suggesting it might be a pattern match. `Compare()` is the worst of the lot; the other terms let us know "returns true if they're equal"; `compare()` doesn't tell us which way the answer will come out.

NAME-3. (page 104)
"Editor" would be my choice for each.

Chapter 12 – Identifying New Refactorings

NEW-1. Remove mid-loop returns. (page 106)
A. Why?
Sometimes it's handy to have routines with a single exit point. Later, we'll see an example where we want to merge two loops. It's trickier to do this if there are internal return statements.

B. Before and after
You might introduce a flag "exited" that tells you to exit, and a variable "result" to hold the computed-but-not-returned result.

```
result = something;
exited = false;
for (int i = 0; !exited && i < n; i++) {
      // stuff
      if (something) {
            result = expression;
            exited = true;
            continue;
      }
      // more stuff
}
if (exited)
      return result;
```

Notice that we used "continue" to force our way to the top of the loop; sometimes this won't be necessary.

Sometimes the result will have a special value (e.g., -1 or null) that can be set; then the test can see whether that's been changed. This lets us avoid a separate flag.

C. Special concerns

- We can apply this to all loop types.
- We can do something similar to eliminate break statements.
- There may be times when you can eliminate the continue statement, but be sure nothing extra happens after the pseudo-return.

D. When doesn't it apply?

If a routine just had a few lines consisting of a loop with a return statement in the middle, followed by a default return, I'd probably think that communicated well enough, and leave it alone. I'd tend to apply this when I had multiple loops or special conditions to worry about.

NEW-3. StringBuffer and String Addition. (page 107)

A. I can think of two reasons to prefer StringBuffer:

- Uniformity: String addition uses the overloaded "+" operator, but your classes can't. Changing to a StringBuffer lets you change to a normal object calling style. This might let further simplifications follow.
- Performance: On some systems, s1+s2+s3 generates slower code than the natural StringBuffer implementation. (Addition is defined as using temporary buffers.) Like all performance quests, let measured problems be your guide.

B. String addition is often more readable, and almost always more concise.

Interlude 4 – Gen-A-Refactoring

I4-1. In the table below, put a "-" in combinations that don't make sense, a "+" in ones that are in Fowler's catalog, and a "*" in ones that make sense but aren't in the catalog. (page 115)

	Extract	Inline	Move	Rename	Pull Up	Push Down	Hide	Expose
Field	-	-	+	*	+	+	*	*
Method	+	+	+	+	+	+	+	*
Class	+	+	*	*	-	-	*	*
Interface	+	*	*	*	-	-	*	*
Subclass	+	?	-	-	-	-	-	-
Superclass	+	?	-	-	-	-	-	-
Hierarchy	+	*	-	-	-	-	-	-

Some are arguable, especially if you want to consider inner classes.

Chapter 13 – A Database Example

DB-1. Database smells. (page 117)

A. Potential problems?

- There's no indication of what are the keys in each table
- Inconsistent use of IDs for keys
- Overloaded terms ("Name")
- Students have no existence apart from their presence in a schedule.
- DaysTimes is not atomic: it can be broken into further parts

A good rule in table design is "Each row depends on the key, the whole key, and nothing but the key."

B. Fixes?

- Identify keys: Course: Name; Offering: ID; Schedule: Name + OfferingID
- Give ID fields to Course and Student
- Extract a Student table
- Turn each session time into its own row (perhaps in another table)

DB-4. Database layer. (page 130)
B. A memory-based database could improve the performance of your system. This could be useful for tests, as not all tests need the "slow but safe" disk-based database.

DB-5. Alternative approaches. (page 131)
B. One approach would be to load only the *keys* of related objects. When a client does a "get" on the corresponding field, then look up based on the stored key. You could optionally cache the reference.

DB-7. Counter. (page 131)
This can be problematic if there's more than one process or thread simultaneously accessing the database. The usual solution is to introduce transactions. (Some databases may have a simple solution for the special case of a counter, but in general we may need transactions.)

Chapter 15 – More on Responsibility

MORE-4. (page 144)
It was Primitive Obsession because our Query was originally just a string. It's Feature Envy or Inappropriate Intimacy because the `toString()` method is exposing internal details of the query (its string), and its caller makes a decision, instead of the query deciding for itself.

MORE-8. (page 145)
- Since search outnumbers add, anything we do to speed up search at add's expense will speed up the overall system. This suggests that we could process the catalog after the last add, in such a way as to speed up searches.
- The way we've set up the interaction, the catalog has no idea what the search will do. One way to speed up search would be to couple them more. Suppose queries were willing to tell the catalog "one of these words must appear in any item I'd potentially be willing to select." Then we could cache a map of words to candidate items, and search many fewer items.

MORE-12. (page 146)
A. Only StringQueries go to the catalog; others are built out of the results of queries.
B. We definitely want to move to a Set type. Again, the trick will be a good preparation after the adds are done. We could pre-build a map from words (i.e., the StringQuery's query string) to sets of items; this would turn simple queries into a lookup instead of a scan.

Chapter 17 – Planning Game

Part 1 – Initial Objects

PG-1. Smells and strategy. (page 157)
A. Smells
Long Methods
- Several methods in Table are too long, especially with the "action" code embedded. It's hard to see where layout ends and action begins.
- Card has similar problems. The constructor in particular seems long.

Long Class
- Both Table and Card seem to be doing too much. (There's no separate model, so these classes act as model, view, and controller.)

Duplicate Code

- Code for setting up all the buttons is similar.
- Each action seems to duplicate some of the work in maintaining the selection.
- Both Table and Card set the border color.
- In Card, the title and the text have similar setup.
- String names of costs appear several times.
- Spaces surround each cost name. (Perhaps the label could be designed to have the space.)
- I'm sure I'll find more once I clean some things up.

Divergent Change

- You can't change the GUI and the actions separately; you couldn't run easily without a GUI.

Shotgun Surgery (one change requires touching several areas)

- The highlighting code (the card border) is in two separate objects.

Feature Envy

- Again, highlighting belongs to the card but the table seems to want in on it.
- Cost information is computed in Table but it does this by pulling apart what's in the Background.
- The summary and the plan report also pull information out of the Background. Several of the actions do that too.

Data Clumps

- The `costLabel` and the `cost` variable seem to go together.
- `Lastx/lasty, x/y, getX()/getY()` deal with paired values.

Primitive Obsession

- Int for x/y.
- Int for velocity?
- Strings for BorderLayout (there are constants defined).
- Strings for the cost label.

Switch Statements

- The `rotateCost()` method uses an "if" statement that is practically a "switch."

Lazy Class

- The Background is a pretty small class. I could see making some of the setup be the caller's responsibility. I'm not ready to label this a lazy class though; the feature envy of Table for Background's components makes me expect some methods will move over here.

Speculative Generality

- There are a couple unused variables: `budget` and `tabs`.

Temporary Field

- The selection (in table) has a little of this feel. (Sometimes it's set, sometimes it's null.)

Inappropriate Intimacy

- Table looks into Background's component list.
- Card knows a bit about its Background (it uses this information in `moveToFront()`).

What Else?

- The names need some work. "Table" is especially bad. "`UpdateCost()`" is updating a summary of more than just cost.

- The ContainerListener business seems awkward (as does all of selection).
- Each action calls `updateCost()` to update the summary section. (Could the event model handle this more cleanly?)
- Counting the cards and the rollover count (to set the initial location) is also awkward. Again – how much configuration should be inside a component, and how much outside?
- There's no separate model. At some level, I can accept that – this is trying to be a simple implementation. But we're already to the point where that seems troublesome (e.g., the summary updates).
- Magic numbers (and colors, and strings) in various places.
- The buttons could go into a Box instead of the combination of panels it uses now.

When looking for various smells, we keep returning to the same issues. Partly, this is the result of the smells not being completely orthogonal, but also it's that the same problem can give off different smells.

B. My planned strategy for fixing this:

- Do trivial things first: fix names, long methods, magic numbers, and such.
- Move things related to Background over to that class.
- See what duplication we can eliminate.
- Straighten out selection. This will involve moving features between objects, and might introduce new objects.
- Assess whether cost and velocity (?) should be their own objects.
- See where things are, but I'd expect to split the model out.

PG-2. Testing. (page 157)

From easy (low impact) on up:

- Test the classes "as is." For example, you could call `rotateCost()` and verify the resulting `cost()`.
- Develop methods that "walk" the components tree to find particular components. For example, you could find the "New" button, call its `doClick()` method, and then verify that the body had one more Card.
- Expose the instance variables of the classes so the tests can more directly access components.
- Restructure the application for better testability. For example, have a separate, easily tested model, along with a thin, "trivial" GUI layer.

PG-3. Tests. (page 158)

Using the simplest approach, which was just to create objects "as is" and call public methods, the test class tests at least something for each object. This test is in the part 2 code.

PG-4. Delete unused variables. (page 158)

PG-5. Rename "Table" to "PlanningGame". (page 158)

PG-6. Remove magic numbers and colors. (page 158)

The result is the starting point for part 2, the code is online.

PG-8. (page 159)

Before we move features around, we want to make sure our tests will warn us if we've done so incorrectly. My tests so far only called publicly available methods, and this didn't include the ability to simulate button clicks. If your tests don't have this ability, add it in. (You can expose the buttons "one by one," or you might find a more generic approach; then call `doClick()`.)

Add the extra tests. I made the buttons have package-level access so the test could get to them. I didn't handle the dialog for the Plan button.

PG-9. Move features over to Background. (page 159)

- Extract the "string computation" part of the "Plan" button and move it to Background. (You may have to test it manually, but you should be able to automate a test after the refactoring.)
- Move `cost()` to Background.
- Create a `count()` method on Background that looks at `getComponentCount()`; adjust the callers.
- Create a `top()` method on Background that returns `getComponent(0)` and adjust the callers. (I'd try returning something of type Card for now because the callers all want to cast it anyways.)

PG-10. Move features over to Card. (page 160)

The Card doesn't know its own selection status. Change that by pulling over the code for a new method `setSelected(boolean)`.

I had expected to need a method `isSelected()` but nobody looks at the selection status!

PG-11. Clean up "cost." (page 160)

I made Cost have a list of its possible values, with each value showing its name, cost, whether it needs an estimate, and whether it needs to be split.

PG-12. Clean up button creation in PlanningGame. (page 161)

- Move the labels into the "make" routines.
- Eliminate the duplication in constructing buttons.

It's easy to extract a method that creates each button and adds it to a panel.

PG-13. Move selection handling to Background. (page 162)

- Make Background hold the selection.
- Move appropriate listener behavior over to Background (at least from New, Split, and Delete).
- Move the ContainerListener.

I used Encapsulate Fields on the selection in PlanningGame, so the setter and getter were the only things accessing the selection. Then it was easy to Move Field over to Background. In moving the listener over, I left the "guard" part on the button, and made the actions assume they were called appropriately. For the ContainerListener, I made Background become a ContainerListener rather than maintain a separate class. (This could have gone either way.) Finally, I went through and made methods private where I could.

PG-14. Transform PlanningGame so cost updates are triggered by property changes from Background rather than explicit calls. (page 162)

This requires PlanningGame to have a listener, but lets us eliminate the `updateCost()` calls.

PG-15. Write tests and make buttons enabled only when appropriate. (page 163)

I added a parameter to `makeButton()` to tell whether the button was sensitive to the number of cards.

PG-16. Remove Card's dependency on Background. (page 163)

This was a little trickier than I expected—there are already other property change notifications when the border is changed. I used a new property name, and made the Background listen for those notifications.

PG-17. Go through your tests and code one more time, doing any simple cleanup you can. (page 163)

I restructured the `resetSelection()` handling, reduced some duplication in the actions for new card and split card, and did a few other bits of "polishing."

PG-20. Duplicate Observed Data. (page 165)

A. When is it cheaper to check before notifying?
When the cost of the string compare is less than the cost of the notification

B. What else?
It can help prevent subtle loops where setting the model notifies a listener, which updates a field, which notifies a listener that updates the model, and so on, in an infinite loop.