Refactoring To Patterns

version 0.15

Joshua Kerievsky
joshua@industriallogic.com
Industrial Logic, Inc.
http://industriallogic.com

Table of Contents

Introduction

> Patterns are a cornerstone of object-oriented design, while test-first programming and merciless refactoring are cornerstones of evolutionary design. To stop over- or under-engineering, it's necessary to learn how patterns fit into the new, evolutionary rhythm of software development. – Joshua Kerievsky

The great thing about software patterns is that they convey many useful design ideas. It follows, therefore, that if you learn a bunch of these patterns, you'll be a pretty good software designer, right? I considered myself just that once I'd learned and used dozens of patterns. They helped me develop flexible frameworks and build robust and extensible software systems. After a couple of years, however, I discovered that my knowledge of patterns and the way I used them frequently led me to over-engineer my work.

Once my design skills had improved, I found myself using patterns in a different way: I began refactoring to patterns, instead of using them for up-front design or introducing them too early into my code. My new way of working with patterns emerged from my adoption of Extreme Programming design practices, which helped me avoid both over- and under-engineering.

Zapping Productivity

When you make your code more flexible or sophisticated than it needs to be, you over-engineer it. Some do this because they believe they know their system's future requirements. They reason that it's best to make a design more flexible or sophisticated today, so it can accommodate the needs of tomorrow. That sounds reasonable, if you happen to be a psychic.

But if your predictions are wrong, you waste precious time and money. It's not uncommon to spend days or weeks fine-tuning an overly flexible or unnecessarily sophisticated software design—--leaving you with less time to add new behavior or remove defects from a system.

What typically happens with code you produce in anticipation of needs that never materialize? It doesn't get removed, because it's inconvenient to do so, or because you expect that one day the code will be needed. Regardless of the reason, as overly flexible or unnecessarily sophisticated code accumulates, you and the rest of the programmers on your team, especially new members, must operate within a code base that's bigger and more complicated than it needs to be.

To compensate for this, folks decide to work in discrete areas of the system. This seems to make their jobs easier, but it has the unpleasant side effect of generating copious amounts of duplicate code, since everyone works in his or her own comfortable area of the system, rarely seeking elsewhere for code that already does what he or she needs.

Over-engineered code affects productivity because when someone inherits an over-engineered design, they must spend time learning the nuances of that design before they can comfortably extend or maintain it.

Over-engineering tends to happen quietly: Many architects and programmers aren't even aware they do it. And while their organizations may discern a decline in team productivity, few know that over-engineering is playing a role in the problem.

Perhaps the main reason programmers over-engineer is that they don't want to get stuck with a bad design. A bad design has a way of weaving its way so deeply into code that improving it

becomes an enormous challenge. I've been there, and that's why up-front design with patterns appealed to me so much.

The Patterns Panacea

When I first began learning patterns, they represented a flexible, sophisticated and even elegant way of doing object-oriented design that I very much wanted to master. After thoroughly studying the patterns, I used them to improve systems I'd already built and to formulate designs for systems I was about to build. Since the results of these efforts were promising, I was sure I was on the right path.

But over time, the power of patterns led me to lose sight of simpler ways of writing code. After learning that there were two or three different ways to do a calculation, I'd immediately race toward implementing the Strategy pattern, when, in fact, a simple conditional expression would have been simpler and faster to program—a perfectly sufficient solution.

On one occasion, my preoccupation with patterns became quite apparent. I was pair programming, and my pair and I had written a class that implemented Java's TreeModel interface in order to display a graph of Spec objects in a tree widget. Our code worked, but the tree widget was displaying each Spec by calling its toString() method, which didn't return the Spec information we wanted. We couldn't change Spec's toString() method since other parts of the system relied on its contents. So we reflected on how to proceed. As was my habit, I considered which patterns could help. The Decorator pattern came to mind, and I suggested that we use it to wrap Spec with an object that could override the toString() method. My partner's response to this suggestion surprised me. "Using a Decorator here would be like applying a sledgehammer to the problem when a few light taps with a small hammer would do." His solution was to create a small class called NodeDisplay, whose constructor took a Spec instance, and whose one public method, toString(), obtained the correct display information from the Spec instance. NodeDisplay took no time to program, since it was less than 10 simple lines of code. My Decorator solution would have involved creating over 50 lines of code, with many repetitive delegation calls to the Spec instance.

Experiences like this made me aware that I needed to stop thinking so much about patterns and refocus on writing small, simple, straightforward code. I was at a crossroads: I'd worked hard to learn patterns to become a better software designer, but now I needed to relax my reliance on them in order to become truly better.

Going Too Fast

Improving also meant learning to not under-engineer. Under-engineering is far more common than over-engineering. We under-engineer when we become exclusively focused on quickly adding more and more behavior to a system without regard for improving its design along the way. Many programmers work this way—I know I sure have. You get code working, move on to other tasks and never make time to improve the code you wrote. Of course, you'd love to have time to improve your code, but you either don't get around to it, or you listen to managers or customers who say we'll all be more competitive and successful if we simply don't fix what ain't broke.

That advice, unfortunately, doesn't work so well with respect to software. It leads to the "fast, slow, slower" rhythm of software development, which goes something like this:

1. You quickly deliver release 1.0 of a system, but with junky code.
2. You attempt to deliver release 2.0 of the system, but the junky code slows you down.
3. As you attempt to deliver future releases, you go slower and slower as the junky code multiplies, until people lose faith in the system, the programmers and even the process that got everyone into this position.

That kind of experience is far too common in our industry. It makes organizations less competitive than they could be. Fortunately, there is a better way.

Socratic Development

Test-first programming and merciless refactoring, two of the many excellent Extreme Programming practices, dramatically improved the way I build software. I found that these two practices have helped me and the organizations I've worked for spend less time over-engineering and under-engineering, and more time designing just what we need: well-built systems, produced on time.

Test-first programming enables the efficient evolution of working code by turning programming into what Kent Beck once likened to a Socratic dialogue: Write test code to ask your system a question, write system code to respond to the question and keep the dialogue going until you've programmed what you need. This rhythm of programming put my head in a different place. Instead of thinking about a design that would work for every nuance of a system, test-first programming enabled me to make a primitive piece of behavior work correctly before evolving it to the next necessary level of sophistication.

Merciless refactoring is an integral part of this evolutionary design process. A refactoring is a "behavior-preserving transformation," or, as Martin Fowler defined it, "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior." [Fowler, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999)].

Merciless refactoring resembles the way Socrates continually helped dialogue participants improve their answers to his questions by weeding out inessentials, clarifying ambiguities and consolidating ideas. When you mercilessly refactor, you relentlessly poke and prod your code to remove duplication, clarify and simplify.

The trick to merciless refactoring is to not schedule time to make small design improvements, but to make them *whenever* your code needs them. The resulting quality of your code will enable you to sustain a healthy pace of development. Martin Fowler et al.'s book, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999), documents a rich catalog of refactorings, each of which identifies a common need for an improvement and the steps for making that improvement.

Why Refactor To Patterns?

On various projects, I've observed what and how my colleagues and I refactor. While we use many of the refactorings described in Fowler's book, we also find places where patterns can help us improve our designs. At such times, we refactor to patterns, being careful not to produce overly flexible or unnecessarily sophisticated solutions.

When I explored the motivation for refactoring to patterns, I found that it was identical to the motivation for implementing non-patterns-based refactorings: to reduce or remove duplication, simplify the unsimple and make our code better at communicating its intention.

However, the motivation for refactoring to patterns is not the primary motivation for using patterns that is documented in the patterns literature. For example, let's look at the documented Intent and Applicability of the Decorator pattern and then examine Erich Gamma and Kent Beck's motivation for refactoring to Decorator in their excellent, patterns-dense testing framework, JUnit.

Decorator's Intent [*Design Patterns*, page 175]:
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator's Applicability (GoF, page 177):

- To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- For responsibilities that can be withdrawn.
- When extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and could produce an explosion of subclasses to support every combination, or a class definition may be hidden or otherwise unavailable for subclassing.

Motivation for Refactoring to Decorator in JUnit
Erich remembered the following reason for refactoring to Decorator:

> "Someone added TestSetup support as a subclass of TestSuite, and once we added RepeatedTestCase and ActiveTestCase, we saw that we could reduce code duplication by introducing the TestSetup , Decorator." [private email]

Can you see how the motivation for refactoring to Decorator (reducing code duplication) had very little connection with Decorator's Intent or Applicability (a dynamic alternative to subclassing)? I noticed similar disconnects when I looked at motivations for refactorings to other patterns. Consider these examples:

Pattern	Intent (GoF)	Refactoring Motivations
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations.	Simplify code Remove duplication Reduce creation errors
Factory Method	Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses.	Remove duplication Communicate intent
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.	Remove duplication

Based on these observations, I began to document a catalog of refactorings to patterns to illustrate when it makes sense to make design improvements with patterns. For this work, it's essential to show refactorings from real-world projects in order to accurately describe the kinds of forces that lead to justifiable transformations to a pattern.

My work on refactoring to patterns is a direct continuation of work that Martin Fowler began in his excellent catalog of refactorings, in which he included the following refactorings to patterns:

- Form Template Method (345)
- Introduce Null Object (260)
- Replace Constructor with Factory Method (304)
- Replace Type Code with State/Strategy (227)
- Duplicate Observed Data (189)

Fowler also noted the following:

> There is a natural relation between patterns and refactorings. Patterns are where you want to be; refactorings are ways to get there from somewhere else. Fowler, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999)

This idea agrees with the observation made by the four authors of the classic book, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994):

> Our design patterns capture many of the structures that result from refactoring. … Design patterns thus provide targets for your refactorings.

Evolutionary Design

Today, after having become quite familiar with patterns, the "structures that result from refactoring," I know that understanding good reasons to refactor to a pattern are more valuable than understanding the end result of a pattern or the nuances of implementing that end result.

If you'd like to become a better software designer, studying the evolution of great software designs will be more valuable than studying the great designs themselves. For it is in the evolution that the real wisdom lies. The structures that result from the evolution can help you, but without knowing why they were evolved into a design, you're more likely to misapply them or over-engineer with them on your next project.

To date, our software design literature has focused more on teaching great solutions than teaching evolutions to great solutions. We need to change that. As the great poet Goethe said, "That which thy fathers have bequeathed to thee, earn it anew if thou wouldst possess it." The refactoring literature is helping us reacquire a better understanding of good design solutions by revealing sensible evolutions to those solutions.

If we want to get the most out of patterns, we must do the same thing: See patterns in the context of refactorings, not just as reusable elements existing apart from the refactoring literature. This is perhaps my primary motivation for producing a catalog of refactorings to patterns.

By learning to evolve your designs, you can become a better software designer and reduce the amount of work you over- or under-engineer. Test-first programming and merciless refactoring are the key practices of evolutionary design. Instill refactoring to patterns in your knowledge of refactorings and you'll find yourself even better equipped to evolve great designs.

Writing Goals

At present, I've written more than a dozen refactorings and have many more in the works. My goal in writing this is to help you learn how to

- refactor *to Patterns* when appropriate and *away from Patterns* when something simpler is discovered
- use Patterns to communicate intention
- know and continue to learn a large body of Patterns
- understand how to implement Patterns in simple and sophisticated ways
- use Patterns to clean, condense, clarify and simplify code
- evolve designs

The form I am using in this work is nearly identical to the one used by Martin in his Refactoring book. I have added the following to this form:

- A section on Communication, Duplication and Simplicity
- Numbered steps in the Mechanics section that correspond to numbered steps in the Examples section.

This is a continuously evolving piece of work. Your feedback is welcome – please send thoughts, comments or questions to joshua@industriallogic.com. This work lives on the internet at the following address: http://industriallogic.com/xp/refactoring/

I've also started an email list – called refactoring@yahoogroup.com – which is a good place to discuss refactoring, refactoring to patterns and emerging tools and IDEs that enable automated refactorings.

Chain Constructors

You have multiple constructors
that contain duplicate code

Chain the constructors together
to obtain the least duplicate code

```
public class Loan {
  ...
  public Loan(float notional, float outstanding, int rating, Date expiry) {
    this.strategy = new TermROC();
    this.notional = notional;
    this.outstanding = outstanding;
    this.rating =rating;
    this.expiry = expiry;
  }
  public Loan(float notional, float outstanding, int rating, Date expiry, Date maturity) {
    this.strategy = new RevolvingTermROC();
    this.notional = notional;
    this.outstanding = outstanding;
    this.rating = rating;
    this.expiry = expiry;
    this.maturity = maturity;
  }
  public Loan(CapitalStrategy strategy, float notional, float outstanding,
                  int rating, Date expiry, Date maturity) {
    this.strategy = strategy;
    this.notional = notional;
    this.outstanding = outstanding;
    this.rating = rating;
    this.expiry = expiry;
    this.maturity = maturity;
  }
}
```

```
public class Loan {
  ...
  public Loan(float notional, float outstanding, int rating, Date expiry) {
    this(new TermROC(), notional, outstanding, rating, expiry, null);
  }
  public Loan(float notional, float outstanding, int rating, Date expiry, Date maturity) {
    this(new RevolvingTermROC(), notional, outstanding, rating, expiry, maturity);
  }
  public Loan(CapitalStrategy strategy, float notional, float outstanding,
                  int rating, Date expiry, Date maturity) {
    this.strategy = strategy;
    this.notional = notional;
    this.outstanding = outstanding;
    this.rating = rating;
    this.expiry = expiry;
    this.maturity = maturity;
  }
}
```

Motivation

Code that's duplicated across two or more of a class's constructors is an invitation for trouble. Someone adds a new variable to a class, updates a constructor to initialize the variable, but neglects to update the other constructors, and bang, say hello to your next bug. The more constructors you have in a class, the more duplication will hurt you. It's therefore a good idea to reduce or remove all duplication if possible, which has the added bonus of reducing your system's *code bloat*.

We often accomplish this refactoring with *constructor chaining*: specific constructors call more general-purpose constructors until a final constructor is reached. If you have one constructor at the end of every chain, I call that your *catch-all* constructor, since it handles every constructor call. This catch-all constructor often accepts more parameters than the other constructors, and may or may not be private or protected.

If you find that having many constructors on your class detracts from its usability, consider applying *Replace Multiple Constructors with Creation Methods* (15).

Communication	Duplication	Simplicity
When constructors in a class implement duplicate work, the code fails to communicate what is specific from what is general. Communicate this by having specific constructors forward calls to more general-purpose constructors and do unique work in each constructor.	Duplicate code in your constructors makes your classes more error-prone and harder to maintain. Find what is common, place it in general-purpose constructors, forward calls to these general constructors and implement what isn't general in each constructor.	If more than one constructor contains the same code, it's harder to see how each constructor is different. Simplify your constructors by making specific ones call more general purpose ones, in a chain.

Mechanics

1. Find two constructors (called A and B) that contain duplicate code. Determine if A can call B or if B can call A, such that the duplicate code can be safely (and hopefully easily) deleted from one of the two constructors.

2. Compile and test.

3. Repeat steps 1 and 2 for all constructors in the class, including ones you've already touched, in order to obtain as little duplication across all constructors as possible.

4. Change the visibility of any constructors that may not need to be public.

5. Compile and test.

Example

1. We'll go with the example shown in the code sketch. We start with a single Loan class, which has three constructors to represent different types of loans and tons of bloated and ugly duplication:

```
public Loan(float notional, float outstanding, int rating, Date expiry) {
    this.strategy = new TermROC();
    this.notional = notional;
    this.outstanding = outstanding;
    this.rating = rating;
    this.expiry = expiry;
}
```

```
public Loan(float notional, float outstanding, int rating, Date expiry, Date maturity) {
    this.strategy = new RevolvingTermROC();
    this.notional = notional;
    this.outstanding = outstanding;
    this.rating = rating;
    this.expiry = expiry;
    this.maturity = maturity;
}

public Loan(CapitalStrategy strategy, float notional, float outstanding, int rating,
            Date expiry, Date maturity) {
    this.strategy = strategy;
    this.notional = notional;
    this.outstanding = outstanding;
    this.rating = rating;
    this.expiry = expiry;
    this.maturity = maturity;
}
```

I study the first two constructors. They do contain duplicate code, but so does that third constructor. I consider which constructor it would be easier for the first constructor to call. I see that it could call the third constructor, with a minimum about of work. So I change the first constructor to be:

```
public Loan(float notional, float outstanding, int rating, Date expiry) {
    this(new TermROC(), notional, outstanding, rating, expiry, null);
}
```

2. I compile and test to see that the change works.

3. I repeat steps 1 and 2, to remove as much duplication as possible. This leads me to the second constructor. It appears that it too can call the third constructor, as follows:

```
public Loan(float notional, float outstanding, int rating, Date expiry, Date maturity) {
    this(new RevolvingTermROC(), notional, outstanding, rating, expiry, maturity);
}
```

I'm now aware that constructor three is my class's catch-all constructor, since it handles all of the construction details.

4. I check all callers of the three constructors to determine if I can change the public visibility of any of them. In this case, I can't (take my word for it – you can't see the code that calls these methods).

5. I compile and test to complete the refactoring.

Chaining To An Init Method

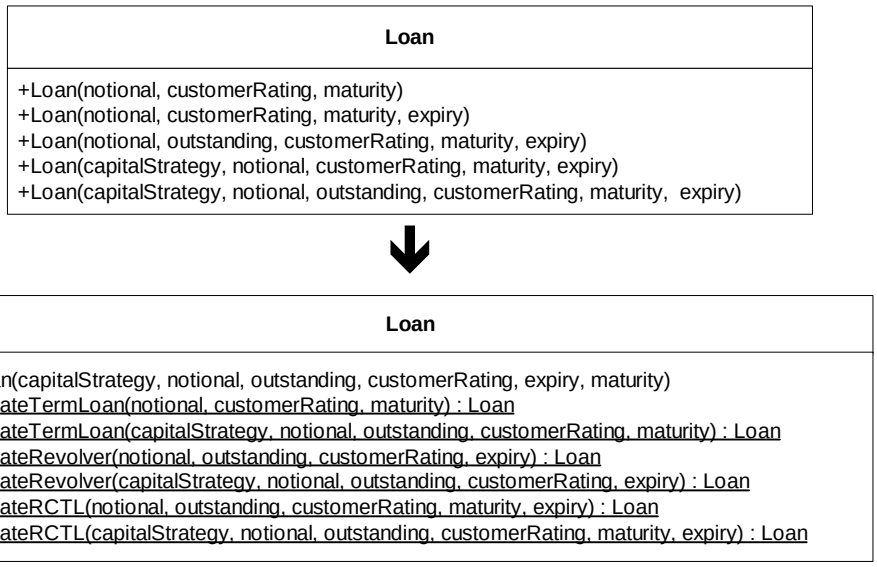
Sometimes your own logic will prevent you from chaining constructors the way you'd like to. [More to write]

[Init methods are sometimes necessary beause you are doing dynamic object loading – Class.forName.newInstance()]

Replace Multiple Constructors with Creation Methods

Constructors on a class make it hard to decide
which constructor to call during development

Replace the constructors with intention-revealing
Creation Methods that return object instances



Motivation

Some languages allow you to name your constructors any old way you like, regardless of the name of your class. Other languages, such as C++ and Java, don't allow this: each of your constructors must be named after your class name. If you have one simple constructor, this may not be problem. If you have multiple constructors, programmers will have to choose which constructor to call by studying which parameters are expected and/or poking around at the constructor code. What's wrong with that? A lot. Constructors simply don't communicate intention efficiently or effectively. The more constructors you have, the easier it is for programmers to mistakenly choose the wrong one. Having to choose which constructor to call slows down development and the code that does call one of the many constructors often fails to sufficiently communicate the nature of the object being constructed.

If you think that sounds bad, it gets worse. As systems mature, programmers often add more and more constructors to classes without checking to see if older constructors are still being used. Constructors that continue to live in a class when they aren't being used are dead weight, serving only to bloat the class and make it more complicated than it needs to be. Mature software systems are often filled with dead constructor code because programmers lack fast, easy ways to identify all callers to specific constructors: either their IDE doesn't help them with this or it is too much trouble to devise and execute search expressions that will identify the exact callers of a specific method. On the other hand, if the majority of object creation calls come through specifically-named methods, like `createTermLoan()` and `createRevolver()`, it is fairly trivial to find all callers to such explicitly-named methods.

Now, what does our industry call a method that creates objects? Many would answer "Factory Method," after the name given to a creational pattern in the classic book, *Design Pattern* [GoF]. But are all methods that create objects true Factory Methods? Given a broad definition of the term – i.e. a method that simply creates objects – the answer would be an emphatic "yes!" But given the way the authors of the creational pattern, Factory Method, wrote about it (in 1994),

it is clear that not every method that creates objects offers the kind of loose-coupling provided by a genuine Factory Method. So, to help us all be clearer when discussing designs or refactorings related to object creation, I'm using the term *Creation Method* to refer to a method that creates objects. This means that every Factory Method is a Creation Method but not necessarily the reverse. It also means that you can substitute the term *Creation Method* wherever Martin Fowler uses the term "factory method" in *Refactoring* [Fowler] and wherever Joshua Bloch uses the term "static factory method" in *Effective Java* [Bloch].

Communication	Duplication	Simplicity
Copious constructors don't communicate available types very well – communicate type availability clearly by offering access to instances via intention-revealing Creation Methods	There is no direct duplication here; just many nearly identical-looking constructors	Figuring out which constructor to call isn't simple – make it simple by offering up the various types through intention-revealing Creation Methods.

Mechanics

After identifying a class that has copious constructors, it's best to consider applying *Extract Class (149)* [Fowler] or *Extract Subclass (330)* [Fowler] *before* you decide to apply this refactoring. *Extract Class* is a good choice if the class in question is simply doing too much work – i.e. it has too many responsibilities. *Extract Subclass* is a good choice if instances of the class only use a small portion of the class's instance variables. If you apply *Extract Subclass*, also consider applying *Encapsulate Classes with Creation Methods (21)*.

1. Identify a class that has copious constructors, is not overburdened with responsibilities and which has instances that use most of its instance variables.

2. Identify the *catch-all* constructor or create one using *Chain Constructor (12)*.

   Strictly speaking, you can implement this refactoring without having a catch-all constructor, though it's a good idea to create one if doing so eliminates duplicate code.

3. Identify a constructor that clients call to create a *kind* of instance and produce a Creation Method for that kind of instance. Make the Creation Method call your catch-call constructor whenever possible, to enable the elimination of constructors (step 6).

   Give your Creation Method an intention-revealing name and make it accept the least number of parameters necessary to produce valid instances. Note that you may create more than one Creation Method for a given constructor.

4. Replace constructor calls that create the kind of instance choosen in step 3 with calls to your Creation Method.

5. Repeat steps 3 and 4, compiling and testing as you go.

6. Delete constructors that are no longer being called and compile.

7. If your class has no subclasses, declare its remaining constructor(s) private. If it has subclasses, declare its remaining constructor(s) protected.

8. Compile.

Example

1. I'll use the example shown in the code sketch. We start with a simple Loan class, which has copious constructors to represent some form of a Term Loan, Revolver or RCTL (a Revolver and Term Loan combination).

```
public class Loan …
    public Loan(double notional, int customerRating, Date maturity) {
        this(notional, 0.00, customerRating, maturity, null);
    }
    public Loan(double notional, int customerRating, Date maturity, Date expiry) {
        this(notional, 0.00, customerRating, maturity, expiry);
    }
    public Loan(double notional, double outstanding, int customerRating, Date maturity,
        Date expiry) {
        this(null, notional, outstanding, customerRating, maturity, expiry);
    }
    public Loan(CapitalStrategy capitalStrategy, double notional, int customerRating,
        Date maturity, Date expiry) {
        this(capitalStrategy, notional, 0.00, customerRating, maturity, expiry);
    }
    public Loan(CapitalStrategy capitalStrategy, double notional, double outstanding,
        int customerRating, Date maturity, Date expiry) {
        this.notional = notional;
        this.outstanding = outstanding;
        this.customerRating = customerRating;
        this.maturity = maturity;
        this.expiry = expiry;
        this.capitalStrategy = capitalStrategy;

        if (capitalStrategy == null) {
            if (expiry == null)
                this.capitalStrategy = new TermCapitalStrategy();
            else if (maturity == null)
                this.capitalStrategy = new RevolverCapitalStrategy();
            else
                this.capitalStrategy = new RCTLCapitalStrategy();
        }
    }
```

This class represents different types of loans that behave in similar ways and that share the same instance variables. The class has five constructors, the last of which is the catch-all constructor. If you look at these constructors, it isn't easy to know which ones create Term Loans, which ones create Revolvers, and which ones create RCTLs. I happen to know that an RCTL needs both an expiry date and a maturity date; so to create one, I must call a constructor that lets me pass in both dates. But did you know that? Do you think the next programmer who reads this code will know it?

What else is embedded as implicit knowledge in the above constructors? Plenty. If you call the first constructor, which takes three parameters, you'll get back a Term Loan. But if you want a Revolver, you'll need to call one of the constructors that take two dates, and supply null for the maturity date. Hmmm, I wonder if all users of this code will know this? Or will they just have to learn by encountering some ugly bugs?

2. The next task is to identify the catch-all constructor for the Loan class. This is easy – it is the constructor that takes the most parameters:

```
public Loan(CapitalStrategy capitalStrategy, double notional, double outstanding,
    int customerRating, Date maturity, Date expiry) {
    this.notional = notional;
    this.outstanding = outstanding;
    this.customerRating = customerRating;
    this.maturity = maturity;
    this.expiry = expiry;
    this.capitalStrategy = capitalStrategy;

    if (capitalStrategy == null) {
        if (expiry == null)
```

```
            this.capitalStrategy = new TermCapitalStrategy();
        else if (maturity == null)
            this.capitalStrategy = new RevolverCapitalStrategy();
        else
            this.capitalStrategy = new RCTLCapitalStrategy();
    }
}
```

3. Next, I identify a constructor that clients call to create a kind of instance:

```
public Loan(double notional, int customerRating, Date maturity) {
    this(notional, 0.00, customerRating, maturity, null);
}
```

This constructor is called to produce a Term Loan with a default TermCapitalStrategy. In order to produce a Creation Method for this kind of instance, I write a test first:

```
public void testTermLoanCreation() {
    Loan term1 = Loan.createTermLoan(NOTIONAL, CUSTOMER_RATING, MATURITY_DATE);
    assertTrue("type = term loan", term1.toString().indexOf("term loan") > -1);
}
```

This test doesn't compile, run or pass until I add the following public static method to `Loan`:

```
public static Loan createTermLoan(double notional, int customerRating, Date maturity) {
    return new Loan(null, notional, 0.00, customerRating, maturity, null);
}
```

I make this method call `Loan`'s catch-all constructor since doing so may allow me to delete, at a later step, the constructor I started with.

4. Now, I find all client calls to the constructor identified in the previous step. Since that constructor only creates Term Loans with a default TermCapitalStrategy, it is safe to replace all of the constructor calls with calls to the new Creation Method. So code that looked like:

```
Loan termLoan = new Loan(notional, customerRating, maturity);
```

is changed to:

```
Loan termLoan = Loan.createTermLoan(notional, customerRating, maturity);
```

5. Repeating steps 3 and 4 yields the following set of `Loan` Creation Methods:

```
public static Loan createTermLoan(double notional, int customerRating, Date maturity) {
    return new Loan(null, notional, 0.00, customerRating, maturity, null);
}
public static Loan createTermLoan(CapitalStrategy capitalStrategy, double notional,
    double outstanding, int customerRating, Date maturity) {
    return new Loan(capitalStrategy, notional, outstanding, customerRating, maturity,
         null);
}
public static Loan createRevolver(double notional, double outstanding,
    int customerRating, Date expiry) {
    return new Loan(null, notional, outstanding, customerRating, null, expiry);
}
public static Loan createRevolver(CapitalStrategy capitalStrategy, double notional,
    double outstanding, int customerRating, Date expiry) {
    return new Loan(capitalStrategy, notional, outstanding, customerRating, null, expiry);
}
public static Loan createRCTL(double notional, double outstanding, int customerRating,
    Date maturity, Date expiry) {
    return new Loan(null, notional, outstanding, customerRating, maturity, expiry);
}
public static Loan createRCTL(CapitalStrategy capitalStrategy, double notional,
    double outstanding, int customerRating, Date maturity, Date expiry) {
    return new Loan(capitalStrategy, notional, outstanding, customerRating, maturity,
         expiry);
```

```
}
```

6. The compiler is now my friend, as I attempt to delete constructors that are no longer being called. I'm able to delete all but the catch-all constructor, which is being called by all of the new Creation Methods.

7. The catch-all constructor can now be safely declared private:

```
private Loan(CapitalStrategy capitalStrategy, double notional, double outstanding,
    int customerRating, Date maturity, Date expiry)
```

8. The compiler agrees with my changes and I'm done.

It's now quite clear how to obtain the different kinds of Loan instances. The ambiguities have been revealed and the implicit knowledge has been made explicit. What's left to do? Well, the Creation Methods still do take a fairly large number of parameters, so I may consider applying *Introduce Parameter Object (295)* [Fowler].

Parameterized Creation Methods

As you consider implementing this refactoring on a class, you may calculate in your head that you'd need something on the order of 50 Creation Methods to account for every object configuration supported by your class. Since writing 50 methods doesn't sound like much fun, you may decide not to do this refactoring. However, there are ways to handle this situation. First, you need not produce a Creation Method for every object configuration: you can write Creation Methods for the most popular configurations and leave some public constructors around to handle the rest of the cases. In addition, it often makes sense to use parameters to cut down on the number of Creation Methods – we call these *parameterized* Creation Methods. For example, a single Apple class could be instantiated in a variety of ways:

- based on the family of apple
- based on the apple's country of origin
- based on the color of apple
- with or without seeds
- peeled or not peeled

These options present numerous kinds of Apples, even though they aren't defined as explicit Apple subclasses. To obtain the Apple instance you need, you must call the correct Apple constructor. But there can be many of these Apple constructors, corresponding with the many Apple types:

```
public Apple(AppleFamily family, Color color) {
    this(family, color, Country.USA, true, false);
}
public Apple(AppleFamily family, Color color, Country country) {
    this(family, color, country, true, false);
}
public Apple(AppleFamily family, Color color, boolean hasSeeds) {
    this(family, color, Country.USA, hasSeeds, false);
}
public Apple(AppleFamily family, Color color, Country country, boolean hasSeeds) {
    this(family, color, country, hasSeeds, false);
}
public Apple(AppleFamily family, Color color, Country country, boolean hasSeeds, boolean
isPeeled) {
    this.family = family;
    this.color = color;
    this.country = country;
    this.hasSeeds = hasSeeds;
    this.isPeeled = isPeeled;
```

```
}
```

As we've noted before, all of these constructors make the Apple class harder to use. To improve the usability of the Apple class, yet not write a large quantity of Creation Methods, we could identify the most popular kinds of Apples created and simply make Creation Methods for them:

```
public static Apple createSeedlessAmericanMacintosh();
public static Apple createSeedlessGrannySmith();
public static Apple createSeededAsianGoldenDelicious();
```

These Creation Methods would not altogether replace the public constructors, but would supplement them and perhaps reduce their number. However, because the above Creation Methods aren't parameterized, they could easily multiple over time, yielding many Creation Methods that would also make it hard to choose the kind of Apple someone needed. Therefore, when faced with so many possible combinations, it often makes sense to write parameterized Creation Methods:

```
public static Apple createSeedlessMacintosh(Country c);
public static Apple createGoldenDelicious(Country c);
```

Encapsulate Classes with Creation Methods

Clients directly instantiate classes that reside
in one package and implement a common interface

Make the class constructors non-public and let clients create instances of them using superclass Creation Methods

Motivation

A client's ability to directly instantiate classes is useful so long as the client needs to know about the very existence of those classes. But what if the client doesn't need that knowledge? What if the classes live in one package, implement one interface and those conditions aren't likely to change? In that case, the classes in the package could be hidden from clients outside the package using public, superclass Creation Methods, each of which would return an instance that satisfied some common interface.

There are several motivations for doing this. First, it provides a way to rigorously apply the mantra, *separate interface from implementation* [GoF], by ensuring that clients interact with classes via their common interface. Second, it provides a way to reduce the "conceptual weight" [Bloch] of a package by hiding classes that don't need to be publicly visible outside their package. And third, it simplifies the construction of available *kinds* of instances by making the set available through intention-revealing Creation Methods.

Despite these good things, some folks have reservations about applying this refactoring. I address and respond to their concerns below:

1. They don't like giving a superclass knowledge of its subclasses, since it causes a dependency cycle - i.e. you have to add new Creation Method to a superclass just because you create a new subclass or add/modify a subclass constructor. When I point out that this refactoring happens within the context of one package with subclasses that implement one interface, they usually quiet down.

2. They don't like mixing Creation Methods with implementation methods on a superclass. I don't have a problem doing this, unless the Creation Methods just make it too hard to see what the superclass does, in which case I apply *Extract Creation Class (27)*.

3. They don't like this refactoring when code is handed off as object code, since programmers who must use the object code won't be able to add or modify the non-public classes or the Creation Methods. I'm more sympathetic to this reservation. If extensibility within a package is necessary and users don't have source code, I would not encapsulate the classes, but would provide a Creation Class for common instances.

The sketch at the start of this refactoring gives you a glimpse of some object to relational database mapping code. Before the refactoring was applied, programmers (including myself) occasionally instantiated the wrong subclass or the right subclass with incorrect arguments (for example, we called a constructor that took a primitive Java `int` when we really needed to call the constructor that took Java's `Integer` object). The refactoring reduced bug creation by encapsulated the knowledge about the subclasses and producing a single place to get a variety of well-named subclass instances.

Communication	Duplication	Simplicity
When you expect client code to communicate with classes via one interface, your code needs to communicate this. Public constructors don't help, since they allow clients to couple themselves to class types. Communicate your intentions by protecting class constructors, producing instances via superclass Creation Methods and making the return type for the instances a common interface or abstract class type.	Duplication isn't an issue with this refactoring.	Making classes publicly visible when you want clients to interact with them via one interface isn't simple: it invites programmers to instantiate and couple themselves to class types and it communicates that it is ok to extend the public interface of an individual class. Simplify by making it impossible to instantiate these classes and by offering instances via superclass Creation Methods.

Forces

- Your classes share a common public interface.

  This is essential because after the refactoring, all client code will interact with class instances via their common interface.

- Your classes reside in the same package.

Mechanics

1. Write an intention-revealing Creation Method on the superclass for a *kind* of instance that a class's constructor produces. Make the method's return type be the common interface type and make the method's body be a call to the class's constructor.

2. For the kind of instance chosen, replace all calls to the class's constructor with calls to the superclass Creation Method.

3. Compile and test.

4. Repeat steps 1 and 2 for any other kinds of instances that may be created by the class's constructor.

5. Declare the class's constructor to be non-public (i.e. protected or package-protected).

6. Compile.

7. Repeat the above steps until every constructor on the class is non-public and all available class instances may be obtainted via superclass Creation Methods.

Example

1. We begin with a small hierarchy of classes that reside in a package called `descriptors`. The classes assist in the object-to-relation database mapping of database attributes to instance variables:

```
package descriptors;

public abstract class AttributeDescriptor {
    protected AttributeDescriptor(…)

public class BooleanDescriptor extends AttributeDescriptor {
    public BooleanDescriptor(…) {
        super(…);
    }

public class DefaultDescriptor extends AttributeDescriptor {
    public DefaultDescriptor(…) {
        super(…);
    }

public class ReferenceDescriptor extends AttributeDescriptor {
    public ReferenceDescriptor(…) {
        super(…);
    }
```

The abstract `AttributeDescriptor` constructor is protected, and the constructors for the three subclasses are public. Let's focus on the `DefaultDescriptor` subclass. The first step is to identify a kind of instance that can be created by the `DefaultDescriptor` constructor. To do that, I look at some client code:

```
protected List createAttributeDescriptors() {
    List result = new ArrayList();
    result.add(new DefaultDescriptor("remoteId", getClass(), Integer.TYPE));
    result.add(new DefaultDescriptor("createdDate", getClass(), Date.class));
    result.add(new DefaultDescriptor("lastChangedDate", getClass(), Date.class));
    result.add(new ReferenceDescriptor("createdBy", getClass(), User.class,
        RemoteUser.class));
    result.add(new ReferenceDescriptor("lastChangedBy", getClass(), User.class,
        RemoteUser.class));
    result.add(new DefaultDescriptor("optimisticLockVersion", getClass(), Integer.TYPE));
    return result;
}
```

Here I see that `DefaultDescriptor` is being used to represent mappings for Integers and Dates. It may also be used to map other types, but I must focus on one kind of instance at a time. So I decide to write a Creation Method to produce attribute descriptors for Integers:

```
public abstract class AttributeDescriptor {
    public static AttributeDescriptor forInteger(...) {
        return new DefaultDescriptor(...);
    }
```

I make the return type for the Creation Method an `AttributeDescriptor` because I want clients to interact with all `AttributeDescriptor` subclasses via the `AttributeDescriptor` interface and because I want to hide the very existence of `AttributeDescriptor` subclasses from anyone outside the `descriptors` package.

If you do test-first programming, you would begin this refactoring by writing a test to obtain the `AttributeDescriptor` instance you want from the superclass Creation Method.

2. Now client calls to create an Integer version of a `DefaultDescriptor` must be replaced with calls to the superclass Creation Method:

```
protected List createAttributeDescriptors() {
    List result = new ArrayList();
    result.add(AttributeDescriptor.forInteger("remoteId", getClass()));
    result.add(new DefaultDescriptor("createdDate", getClass(), Date.class));
    result.add(new DefaultDescriptor("lastChangedDate", getClass(), Date.class));
    result.add(new ReferenceDescriptor("createdBy", getClass(), User.class,
        RemoteUser.class));
    result.add(new ReferenceDescriptor("lastChangedBy", getClass(), User.class,
        RemoteUser.class));
    result.add(AttributeDescriptor.forInteger("optimisticLockVersion", getClass()));
    return result;
}
```

3. I compile and test that the new code works.

4. Now I continue to write Creation Methods for the remaining kinds of instances that the `DefaultDescriptor` constructor can create. This leads to 2 more Creation Methods:

```
public abstract class AttributeDescriptor {
    public static AttributeDescriptor forInteger(...) {
        return new DefaultDescriptor(...);
    }
    public static AttributeDescriptor forDate(...) {
        return new DefaultDescriptor(...);
    }
    public static AttributeDescriptor forString(...) {
        return new DefaultDescriptor(...);
```

```
    }
```

5. I now declare the DefaultDescriptor constructor protected:

```
public class DefaultDescriptor extends AttributeDescriptor {
    protected DefaultDescriptor(…) {
        super(…);
    }
```

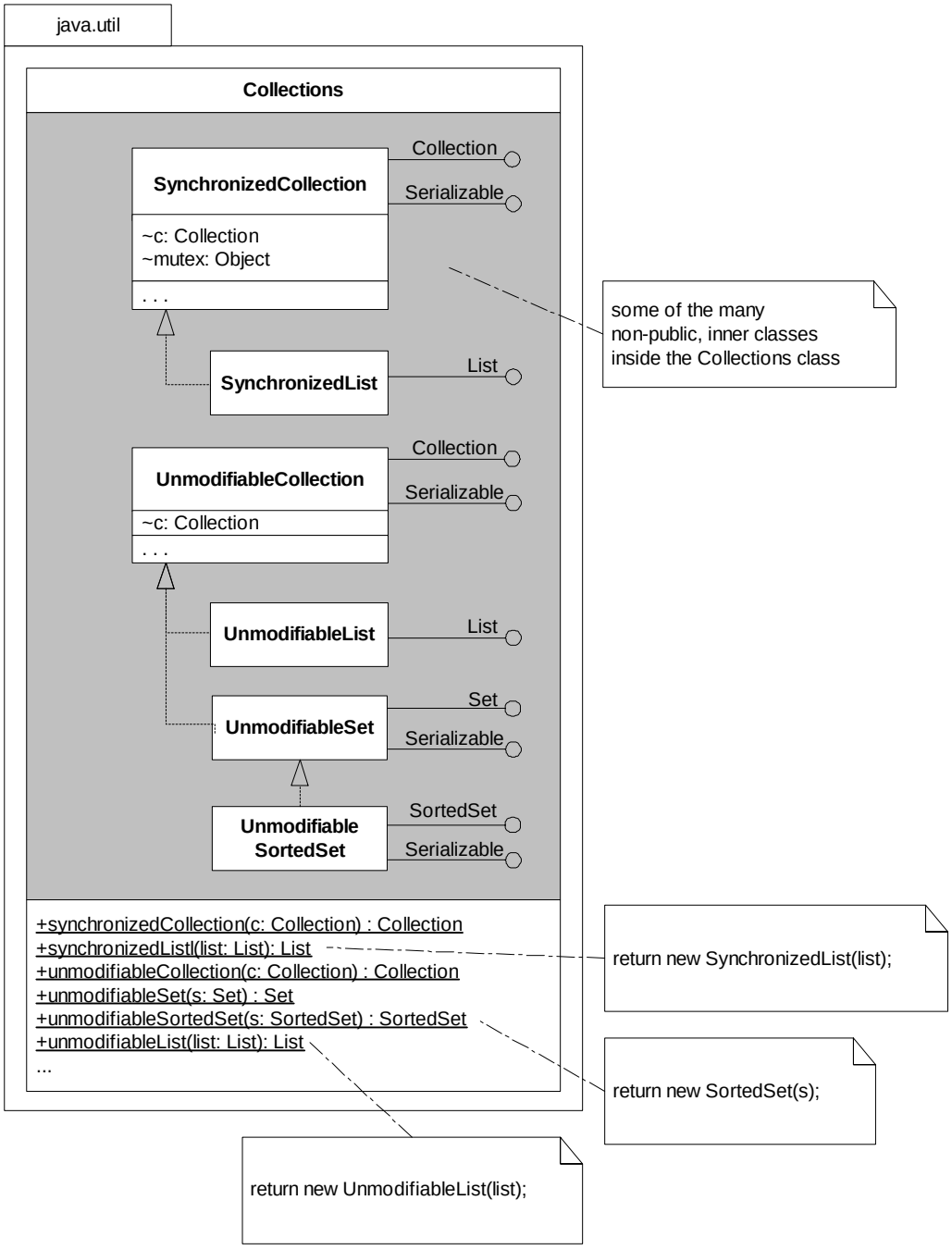
6. I compile and everything goes according to plan.

7. Now I repeat the above steps for the other `AttributeDescriptor` subclasses. When I'm done, the new code:

- gives access to `AttributeDescriptor` subclasses via their superclass
- ensures that clients obtain subclass instances via the `AttributeDescriptor` interface
- prevents clients from directly instantiating `AttributeDescriptor` subclasses
- communicates to other programmers that `AttributreDescriptor` subclasses are not meant to be public – the convention is to offer up access to them via the superclass and a common interface.

Encapsulating Inner Classes

The JDK's `java.util.Collections` class is a remarkable example of what encapsulating classes with Creation Methods is all about. The class's author, Joshua Bloch, needed to give programmers a way to make Collections, Lists, Sets and Maps unmodifiable and/or synchronized. He wisely chose to implement this behavior using the Decorator pattern. However, instead of creating public, `java.util` Decorator classes (for handling synchronization and unmodifiabilty) and then expecting programmers to decorate their own collections, he defined the Decorators in the Collections class as non-public inner classes and then gave Collections a set of Creation Methods from which programmers could obtain the kinds of decorated collections they needed. Below is a sketch of a few of the inner classes and Creation Methods that are specified by the Collections class:



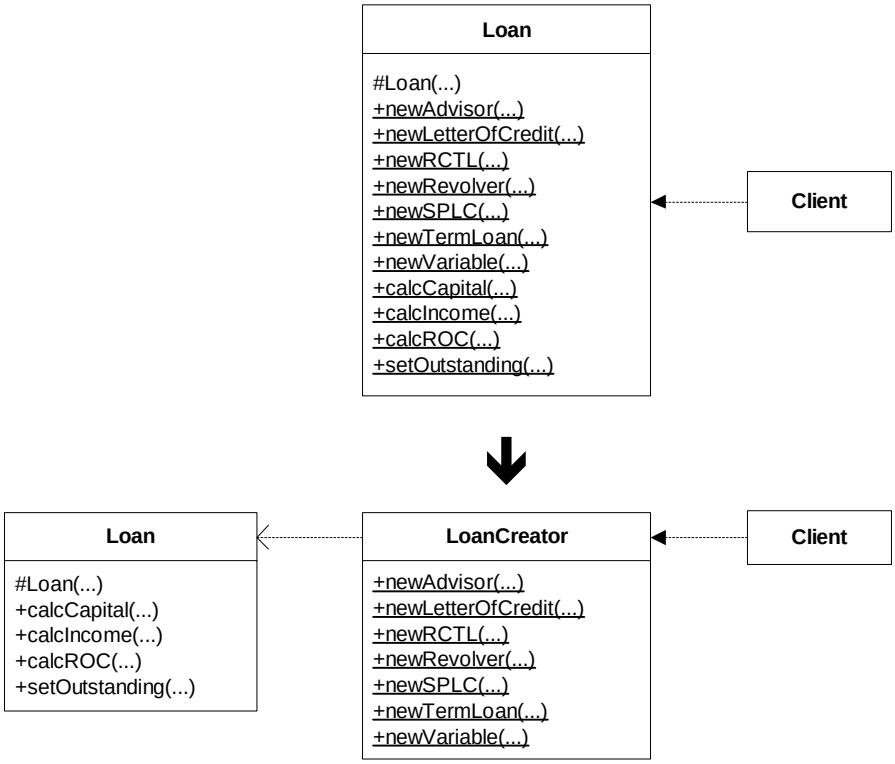
4. Notice that `java.util.Collections` even contains small hierarchies of inner classes, all of which are non-public. Each inner class has a corresponding method that receives a collection, decorates it and then returns the decorated instance, using a commonly defined interface type (such as List or Set). This solution reduced the number of classes programmers needed to know about, while providing the necessary funcationality. `java.util.Collections` is an example of a Creation Class (see *Extract Creation Class (27)).*

).

Extract Creation Class

Too many Creation Methods on a class
obscure it's primary responsibility

Move the Creation Methods for a related set of classes to one Creation Class



Motivation

This refactoring is essentially *Extract Class* [Fowler], only it's done on a class's Creation Methods. There's nothing wrong with a few Creation Methods on a class, but as the number of them grows, a class's own primary responsibilities – it's main purpose in life – may begin to feel obscured or overshadowed by creational logic. When that happens, it's better to restore the class's identity by moving its Creation Methods to a Creation Class.

Creation Classes and Abstract Factories [GoF] are similar in that they create families of objects, but they are quite different, as the following table illustrates:

	Creation Class	Abstract Factory
Substitutable at runtime	No	Yes
Instantiates a family of products	Yes	Yes
Supports creation of new products easily	Yes	No
Separates interface from implementation	May or may not	Yes
Is implemented with static methods	Usually	No
Is implemented as a Singleton	No	Often

In general, Creation Classes are good if you have one and only one class for creating a family of products, you don't need to substitut for another object, you can safely go with a Creation

Class. anothereven though they are booin that they often create a related set of objects, but are most unlike Abstract Factories in that you don't substitute one Creation Class for another at runtime, because you're not concerned with swapping out one family of products for another. Creation Classes are usually implemented as classes that contain static methods, each of which instantiates and returns an object instance.

Communication	Duplication	Simplicity
When object creation begins to dominate the public interface of the class, the class no longer strongly communicates its main purpose. Communicate the act of object creation by creating a special class just to create object instances.	Duplication is not an issue with respect to this refactoring.	When creational responsibilities mix too much with a class's main responsibilities, the class isn't simple. Simplify it by extracting the creational code into a Creation Class.

Mechanics

1. Identify a class (which we'll call "A") that is overrun with Creation Methods.

2. Create a class that will become your Creation Class. Name it after it's purpose in life, which will be to create various objects from a set of related classes.

3. Move all Creational Methods from A to your new class, making sure that all protection privledges are accounted for.

4. Change all callers to obtain object references from your new Creation Class.

Example

Though I use different example code from Martin Fowler, I do tend to repeat it as I am intrinsically lazy. So if you don't mind, we'll work with the same brainless Loan example, outlined in the code sketch above. Assume that there is test code for the example code below –I didn't include it the text since this refactoring is fairly trivial.

1. We begin with a Loan class that has lots of code for handling the responsibilities of a Loan and being a creator of Loan objects:

```
public class Loan {
   private double notional;
   private double outstanding;
   private int rating;
   private Date start;
   private CapitalStrategy capitalStrategy;
   private Date expiry;
   private Date maturity;
   // . . . more instances variables not shown

   protected Loan(double notional, Date start, Date expiry,
      Date maturity, int riskRating, CapitalStrategy strategy) {
      this.notional = notional;
      this.start = start;
      this.expiry = expiry;
      this.maturity = maturity;
      this.rating = riskRating;
      this.capitalStrategy = strategy;
   }
   public double calcCapital() {
      return capitalStrategy.calc(this);
   }
```

```
   public void setOutstanding(double newOutstanding) {
      outstanding = newOutstanding;
   }

   // ... more methods for dealing with the primary responsibilities of a Loan, not shown

   public static Loan newAdvisor(double notional, Date start,
                                 Date maturity, int rating)
      return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
   }
   public static Loan newLetterOfCredit(double notional, Date start,
                                        Date maturity, int rating) {
       return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
   }
   public static Loan newRCTL(double notional, Date start,
                              Date expiry, Date maturity, int rating) {
      return new Loan(notional, start, expiry, maturity, rating, new RCTLCapital());
   }
   public static Loan newRevolver(double notional, Date start,
                                  Date expiry, int rating) {
      return new Loan(notional, start, expiry, null, rating, new RevolverCapital());
   }
   public static Loan newSPLC(double notional, Date start,
                              Date maturity, int rating) {
      return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
   }
   public static Loan newTermLoan(double notional, Date start,
                                  Date maturity, int rating) {
      return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
   }
   public static Loan newVariableLoan(double notional, Date start,
                                      Date expiry, Date maturity, int rating) {
      return new Loan(notional, start, expiry, maturity, rating, new RCTLCapital());
   }
}
```

2. Next, I create a class called `LoanCreator`, since it's sole purpose in life is to be a place where clients can obtain `Loan` instances:

```
public class LoanCreator {
}
```

3. Now I move all of the Creation Methods from `Loan` to `LoanCreator`, placing `LoanCreator` in the same package as `Loan` (and it's Capital stategies) so it has the protection level it needs to instantiate `Loans`:

```
public class LoanCreator {
    public static Loan newAdvisor(double notional, Date start,
                                  Date maturity, int rating)
      return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
   }
   public static Loan newLetterOfCredit(double notional, Date start,
                                        Date maturity, int rating) {
       return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
   }
   public static Loan newRCTL(double notional, Date start,
                              Date expiry, Date maturity, int rating) {
      return new Loan(notional, start, expiry, maturity, rating, new RCTLCapital());
   }
   public static Loan newRevolver(double notional, Date start,
                                  Date expiry, int rating) {
      return new Loan(notional, start, expiry, null, rating, new RevolverCapital());
   }
   public static Loan newSPLC(double notional, Date start,
                              Date maturity, int rating) {
      return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
   }
   public static Loan newTermLoan(double notional, Date start,
                                  Date maturity, int rating) {
      return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
```

```
   }
   public static Loan newVariableLoan(double notional, Date start,
                                      Date expiry, Date maturity, int rating) {
      return new Loan(notional, start, expiry, maturity, rating, new RCTLCapital());
   }
}
```

4. To finish, I simply change calls of the form:

```
Loan termLoan = Loan.newTermLoan(…)
```

to

```
Loan termLoan = LoanCreator.newTermLoan(…)
```

Move Object Composition to Creation Method

Client code is responsible for wrapping objects together
to obtain one instance with the desired behavior

Move the object composition responsibility
to an intention-revealing Creation Method

Motivation

Prerequisites

Mechanics

Example

Replace Multiple Instances with Singleton

You create multiple instances of an object that
consumes too much memory and/or takes a while to create

Replace the multiple instances with a Singleton

Example about Zip Code, City, State Object

Replace Singleton with Object Reference

A class is a Singleton but has no business being a Singleton

Replace the Singleton with a plain old, non-global instance and pass this instance to objects that need it.

```
public void someMethod() ...
        Profile.getInstance().getUserLevel()
```

```
public void someMethod(Profile profile) ...
        profile.getUserLevel()
```

Motivation

Replace Singleton with Registry

By J. B. Rainsberger

Motivation

You have a package or library that lives within an application and relies on global objects (singletons) provided by a part of the application. Your package is therefore coupled with the current application, but you would like your package to be used somewhere else.

You prefer not to apply Replace Global with Object Reference, because it will cause an unknown ripple effect throughout the application. This ripple effect is not something you can afford to handle at the present moment, so you are looking for a refactoring to help you get part of the way towards fixing the overall design issues.

After applying Replace Singleton with Registry, your package has access to the same data it had before, but that access is made local to the package in the form of a Registry. It then suffices to change the application so that it registers its data with the package's Registry. You can then use the package in other applications, as long as the application places the data your package needs in the prescribed, well-known location.

What is a Registry?

Briefly, a Registry is a namespace for objects. Clients can store information within a Registry so that other clients can retrieve that data without binding these clients to each other. We usually implement a Registry as a singleton, so the Registry is a well-known, global location for objects that allows providers and consumers of the data to operate independently of one another.

Forces

You have an application with (usually many) singletons that individual packages use to perform their work. Usually this is configuration information or widely-used resources like databases and external servers.

You would like to use one of your packages in a different application, or simply improve its design to make it application-independent.

You can refactor the application to register its singleton instances with well-known objects within the package. If you cannot do this, consider creating a simple application facade [insert reference] to help during the refactoring.

Mechanics

Identify the application-level objects your package needs to operate. Create a class called PackageConfiguration that aggregates all these objects.

Make PackageConfiguration a singleton, but add setX() methods for each object your package wants the application to register.

Within your application, as each object becomes available, call the corresponding setX() method on the PackageConfiguration object to "register" the object.

Within your package, replace each reference to the application's globally-accessible objects with the corresponding getX() method on the PackageConfiguration to retrieve the registered object.

When you have finished, you will have inserted a Registry, acting as a mediator [insert reference] between the application and your package. By registering application-level objects with the Registry, your application is one step closer to being decoupled from your package. It is possible now to have multiple packages retrieve objects from the Registry in order to perform their work. By retrieving application-level objects from the Registry, your package is one step closer to being decoupled from the application. It is possible now to have any application (although only one per virtual machine) register its global objects with the Registry without the package knowing the source of the objects.

Why did this happen?

You may be wondering how this abuse of singletons would arise in the first place. Put simply, the singleton is an easy way to make data available "from the application down" -- that is, storing data within the application and making it available to the components that need them. In many cases, the application itself only requires an attribute for each of these objects; however, in order to make the various components more "independent", programmers often create singletons in the hopes of pulling information from the application, rather than having the application push that information to the components.

Unfortuntately, as often happens, the component programmer requires something to be configured at the application level, and not at the component level. The programmer may, under the constraints of time and patience, give in to the temptation of simply "grabbing the data from the application," rather than providing a means for the application to configure the component.

[...]

Introduce Polymorphic Creation with Factory Method

Classes in a hierarchy implement a method similarly,
except for an object creation step

Make a single superclass version of the method that calls a Factory Method to handle the instantiation

Motivation

What is a Factory Method [GoF]? It is a polymorphic method for creating and returning a Product. The method is declared in a superclass or interface. A superclass may implement the method and a subclass may override it, in order to make local decisions about the creation, including whether to instantiate a subclass of Product and/or how to initialize an instance.

Factory Method is a specialization of Creation Method: both provide for object creation using a method instead of a constructor, but Factory Method adds the ability to do polymorphic object creation within a hierarchy. The following table illustrates primary differences:

	Creation Method	Factory Method
May be implemented as abstract in a superclass	No	Yes
Subclasses may override the method	No	Yes
Is implemented with static or non-static methods	Yes	No

Why would you refactor to a Factory Method? One motivation involves duplicate code: you find a method either in a superclass and overridden by a subclass or in several subclasses and this method is implemented nearly identically, except for an object creation step. You see how you could replace all versions of this method with a single superclass Template Method [GoF], provided that it could issue the object creation call, while letting the superclass and/or subclasses do the actual object creation work. No pattern is better suited to that task than Factory Method.

In his refactoring, *Form Template Method (345)* [Fowler], Martin Fowler observes that, "inheritance is a powerful tool for eliminating duplicate behavior." Inheritance is also what enables us to implement a Factory Method's polymorphic object creation, since subclasses may control the class of object that gets instantiated. Template Methods often call Factory Methods [GoF, page 330], and many programmers refactor to both patterns to reduce duplication in class hierarchies.

Communication	Duplication	Simplicity
A well-chosen name for a Factory Method communicates intention better than a direct constructor call. Factory Methods also serve to communicate that the instances they return all implement a common interface.	Duplication of a method often results from a need to create an object instance in different ways. Remove the duplication by making a single method that obtains the instance it needs via a call to a Factory Method.	It's usually simpler to read code that issues a call to a Factory Method than it is to read code that performs the actual instantiation. However, for those who aren't comfortable with polymorphism, Factory Methods can seem to be more complex than direct instantiation calls.

Forces

- Near-duplicate versions of a method exist in a class hierarchy, and the only difference between them is that they perform object instantiation differently.

- The classes of the objects being instantiated implement a common interface. If they do not, consider applying *Extract Superclass (336)* [Fowler] to give them a common interface.

Mechanics

Choose from the following two sets of mechanics:

A. When a method is duplicated because a superclass and subclass instantiate a type of object differently, refactor as follows:

1. In the superclass method, apply *Extract Method (110)* [Fowler] on the object instantiation code to produce a Factory Method.

   Make sure the return type for the Factory Method is a generic type, not the type of the concrete product being instantiated.

2. On each subclass that overrides the superclass method to do custom object creation, extract the instantiation logic (using *Extract Method (110)* [Fowler]) to produce a Factory Method with the same signature as the superclass Factory Method

3. Remove subclass versions of the method that are no longer needed, compile and test.

   If you don't expect subclasses to ever override this method, declare it as final.

B. When a method is duplicated across several subclasses because they instantiate a type of object differently, refactor as follows:

1. Create a Factory Method on the superclass. Declare it abstract if it does not make sense to have a default implementation, otherwise make it instantiate and return a default instance.

2. On each subclass that duplicates the method to do custom object creation, extract the instantiation logic (using *Extract Method (110)* [Fowler]) to produce a Factory Method with the same signature as the superclass Factory Method.

3. Apply *Form Template Method (345)* [Fowler], compile and test.

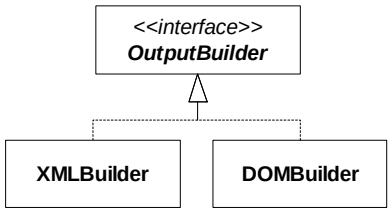
   When you finish this step, the once duplicated method will now be a Template Method on the superclass, and this Template Method will call your new Factory Method.If you don't expect subclasses to ever override this method, declare it as final.

Example

Writing data in the form of XML or HTML is a pretty common task these days. The code sketch at the start of this refactoring comes from a system that outputs XML data using a hierarchy of PageWriter classes. Let's begin by looking at code from the `AbstractPageWriter` class:

```
public abstract class AbstractPageWriter…
    public String pageText() {
        OutputBuilder outputBuilder = new XMLBuilder();
        writeHeaderOn(outputBuilder);
        writeBodyOn(outputBuilder);
        writeFooterOn(outputBuilder);
        return outputBuilder.toString();
    }
    protected abstract void writeBodyOn(OutputBuilder builder);
    protected abstract void writeFooterOn(OutputBuilder builder);
    protected abstract void writeHeaderOn(OutputBuilder builder);
```

The method, `pageText()`, is a Template Method [GoF]. By default, it creates an `OutputBuilder` of type `XMLBuilder` and passes it to three methods, after which it returns the `OutputBuilder`'s output. Subclasses override the three methods to customize what they output. Before we look at an example subclass, let's look at `OutputBuilders`:



XMLBuilder is a class that can build simple XML documents. It is usually sufficient for producing output in a system. On some occasions, however, code that builds output needs something a little more sophisticated, such as a DOMBuilder, which gives access to the Document Object Model.

A subclass of `AbstractPageWriter`, called `PrimaryInsurerPageWriter`, needed a `DOMBuilder`, so a programmer overrode the `pageText()` method as follows:

```
public class PrimaryInsurerPageWriter extends AbstractPageWriter…
    public String pageText() {
        OutputBuilder outputBuilder = new DOMBuilder();
        writeHeaderOn(outputBuilder);
        writeBodyOn(outputBuilder);
        writeFooterOn(outputBuilder);
        return outputBuilder.toString();
    }
```

As you can see, this is nearly a replica of the superclass `pageText()` method, the only difference being what kind of `OutputBuilder` is instantiated. Such duplication is a "breeding ground for bugs," as Martin Fowler likes to call it. The duplication can be removed by refactoring to a Factory Method [GoF], as the steps below will show. Note: the refactoring mechanics labeled as "A" will be used in this example.

1. On the superclass, `AbstractPageWriter`, we apply *Extract Method (110)* [Fowler] to produce a Factory Method [GoF], called `createOutputBuilder()`:

```
public abstract class AbstractPageWriter…
    public String pageText() {
        OutputBuilder outputBuilder = createOutputBuilder();
        writeHeaderOn(outputBuilder);
        writeBodyOn(outputBuilder);
        writeFooterOn(outputBuilder);
        return outputBuilder.toString();
    }
    protected OutputBuilder createOutputBuilder() {
        return new XMLBuilder();
    }
```

2. We peform a similar step on the subclass, `PrimaryInsurerPageWriter`:

```
public class PrimaryInsurerPageWriter extends AbstractPageWriter…
    public String pageText() {
        OutputBuilder outputBuilder = createOutputBuilder();
        writeHeaderOn(outputBuilder);
        writeBodyOn(outputBuilder);
        writeFooterOn(outputBuilder);
        return outputBuilder.toString();
    }
    protected OutputBuilder createOutputBuilder() {
        return new DOMBuilder();
    }
```

3. Now the `pageText()` method from `PrimaryInsurerPageWriter` can be deleted:

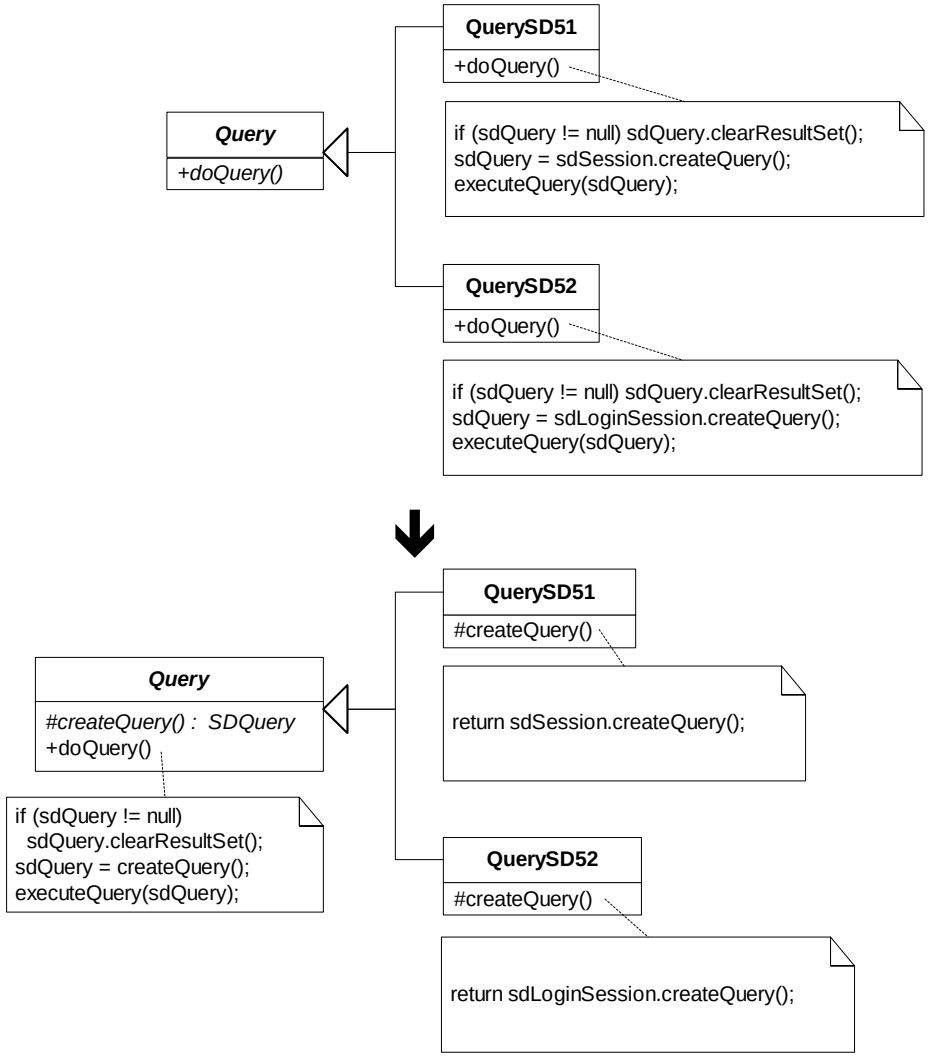
```
public class PrimaryInsurerPageWriter extends AbstractPageWriter…
    public String pageText() {
```

```
        OutputBuilder outputBuilder = createOutputBuilder();
        writeHeaderOn(outputBuilder);
        writeBodyOn(outputBuilder);
        writeFooterOn(outputBuilder);
        return outputBuilder.toString();
    }
```

We compile and run tests, such as `testPrimaryInsurerPageOutput()`, to confirm that everything still works:

```
public void testPrimaryInsurerPageOutput() {
    String primaryInsurerOutput = getPrimaryInsurerPageWriter().pageText();
    assertTrue(primaryInsurerOutput.indexOf(KIM_NAME) > -1);
    assertTrue(primaryInsurerOutput.indexOf(KIM_ADDRESS) > -1);
    assertTrue(primaryInsurerOutput.indexOf(KIM_OCCUPATION) > -1);
    ...
}
```

Duplication Across Subclasses



This example is similar to the previous one, only this time we begin with duplication in two subclasses. We can remove this duplication by introducing a Factory Method and a Template Method. I'll use the mechanics labeled as "B" to demonstrate how this refactoring works.

1. We start with some classes that handle doing database queries:

```
abstract class Query…
    public abstract void doQuery() throws QueryException;

class QuerySD51 extends Query …
    public void doQuery() throws QueryException {
        if (sdQuery != null) sdQuery.clearResultSet();
          sdQuery = sdSession.createQuery(SDQuery.OPEN_FOR_QUERY);
          executeQuery(sdQuery);
    }

class QuerySD52  extends Query …
    public void doQuery() throws QueryException {
        if (sdQuery != null) sdQuery.clearResultSet();
        sdQuery = sdLoginSession.createQuery(SDQuery.OPEN_FOR_QUERY);
        executeQuery(sdQuery);
    }
```

I add a Factory Method to the superclass, `Query`, and declare it abstract so that subclasses must implement it:

```
abstract class Query…
    protected abstract SDQuery createQuery() throws QueryException;
```

2. Now I'll create a Factory Method in each subclass by extracting the instantiation logic from the subclass implementations of `doQuery()`:

```
class QuerySD51 extends Query …
    protected SDQuery createQuery() {
        return sdSession.createQuery(SDQuery.OPEN_FOR_QUERY);
    }
    public void doQuery() throws QueryException {
        if (sdQuery != null) sdQuery.clearResultSet();
        sdQuery = createQuery();
        executeQuery(sdQuery);
    }

class QuerySD52  extends Query …
    protected SDQuery createQuery() {
        return sdLoginSession.createQuery(SDQuery.OPEN_FOR_QUERY);
    }
    public void doQuery() throws QueryException {
        if (sdQuery != null) sdQuery.clearResultSet();
        sdQuery = createQuery();
        executeQuery(sdQuery);
    }
```

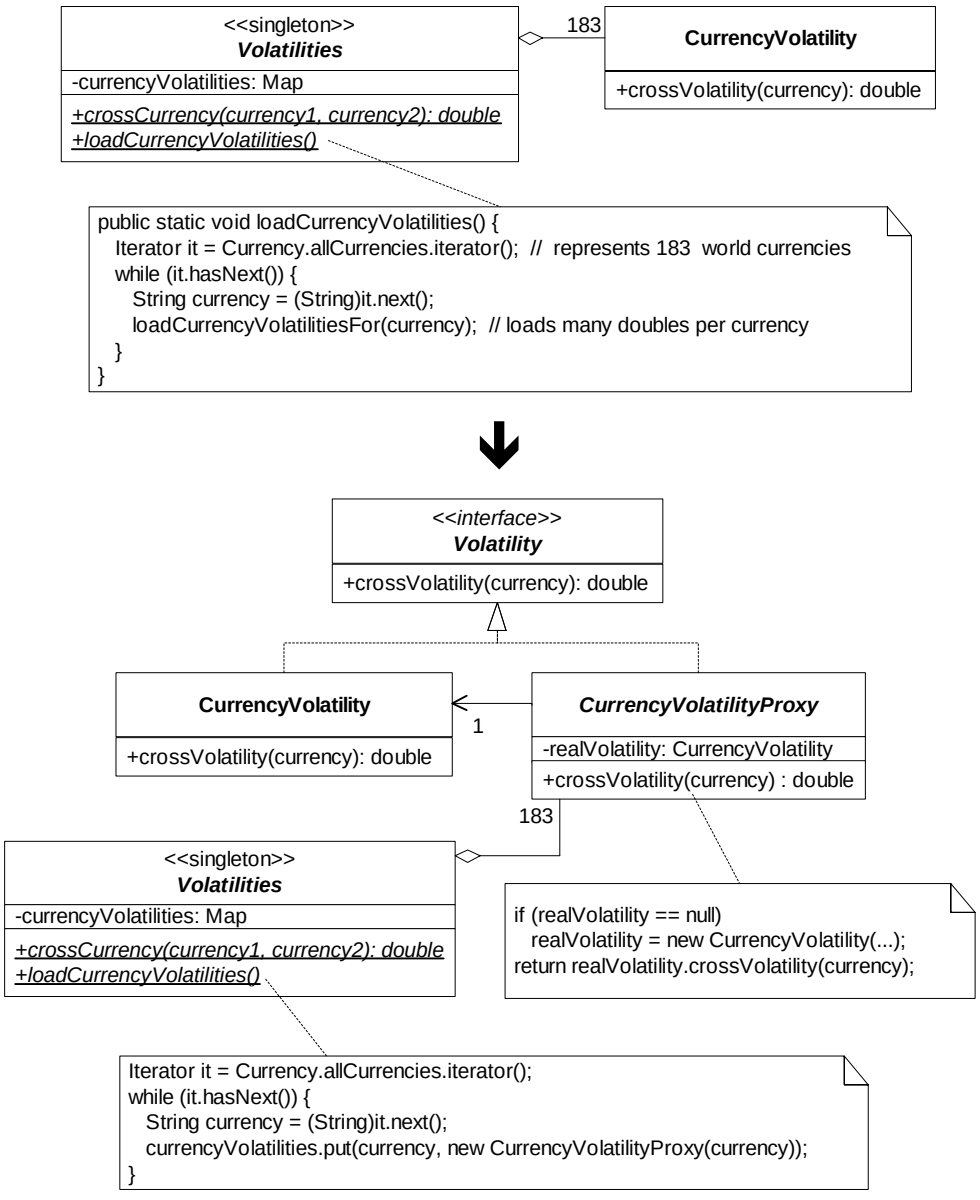
3. Finally, I apply *Form Template Method (345)* [Fowler], to produce a single, superclass doQuery() method:

```
abstract class Query…
    protected abstract SDQuery createQuery() throws QueryException;
    public void doQuery() throws QueryException {
        if (sdQuery != null) sdQuery.clearResultSet();
        sdQuery = createQuery();
        executeQuery(sdQuery);
    }

class QuerySD51 extends Query …
    protected SDQuery createQuery() {
        return sdSession.createQuery(SDQuery.OPEN_FOR_QUERY);
    }

class QuerySD52  extends Query …
    protected SDQuery createQuery() {
        return sdLoginSession.createQuery(SDQuery.OPEN_FOR_QUERY);
    }
```

Defer Slow Creation with Virtual Proxy

One or many objects take a while to instantiate or load,
but your system may not use them during execution.

Create a Virtual Proxy that can instantiate
and delegate to the real object, when necessary



Motivation

ne one or more methods that return a context instance, properly outfitted with the appropriate Strategy instance.

Conditional Logic or Slow Code.

Communication	Duplication	Simplicity
Code readability is often sacrificed when deferred creation logic is mixed together with primary logic. Let the primary logic communicate clearly by placing the deferred creation logic into a Virtual Proxy, where it will be invisible to client code.	Conditional logic that checks whether an expensive object has been loaded tends to get duplicated in client programs. Remove the duplication by centralizing the conditional logic in a Virtual Proxy.	The simplicity of a system is slightly reduced when a Virtual Proxy is implemented, since it adds a minor amount of sophistication around the act of object creation. However, since the interface of a proxy and real subject are identical, it's just as simple for a client program to use one or the other.

Mechanics

Example

Many custom banking applications calculate risk on financial products. The calculations often require access to large amounts of numerical data, such as cross-currency volatilities. The trouble is, it can take a while to instantiate (or create) all of the data that may be used by the calculations, and meanwhile, the users want their numbers to be computed quickly. Virtual Proxies offer a good solution here. Data that may not be necessary during program execution doesn't need to be loaded, but can instead be represented by a lightweight delegate, which looks and acts like the real thing.

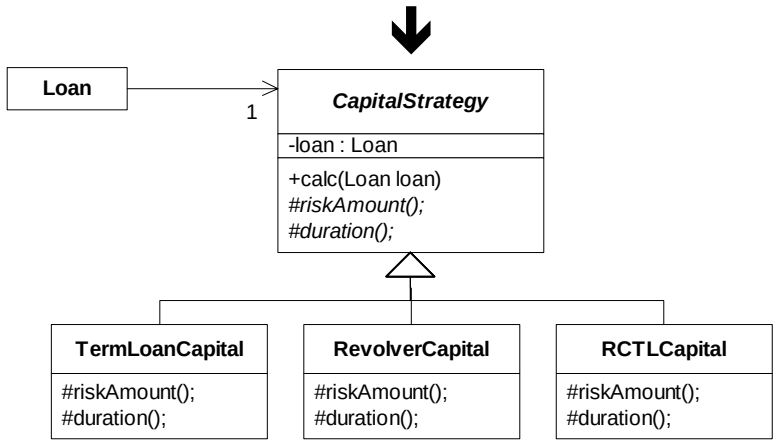
Our example deals with cross-currency volatility data. Let's begin by see how this data is loaded into objects:

Replace Conditional Calculations with Strategy

You use a lot of conditional logic in a calculation

Delegate the calculation to a Strategy object

```
public class Loan ...
  public double calcCapital() {
    return riskAmount() * duration() * RiskFactor.forRiskRating(rating);
  }
  private double riskAmount() {
    if (unusedPercentage != 1.00)
      return outstanding + calcUnusedRiskAmount();
    else return outstanding;
  }
  private double calcUnusedRiskAmount() {
    return (notional - outstanding) * unusedPercentage;
  }
  private double duration() {
    if (expiry == null)
      return ((maturity.getTime() - start.getTime())/MILLIS_PER_DAY)/365;
    else if (maturity == null)
      return ((expiry.getTime() - start.getTime())/MILLIS_PER_DAY)/365;
    else {
      long millisToExpiry = expiry.getTime() - start.getTime();
      long millisFromExpiryToMaturity = maturity.getTime() - expiry.getTime();
      double revolverDuration = (millisToExpiry/MILLIS_PER_DAY)/365;
      double termDuration = (millisFromExpiryToMaturity/MILLIS_PER_DAY)/365;
      return revolverDuration + termDuration;
    }
  }
  private void setUnusedPercentage() {
    if (expiry != null && maturity != null) {
      if (rating > 4) unusedPercentage = 0.95;
      else unusedPercentage = 0.50;
    } else if (maturity != null) {
        unusedPercentage = 1.00;
    } else if (expiry != null) {
        if (rating > 4) unusedPercentage = 0.75;
        else unusedPercentage = 0.25;
    }
  }
```



Motivation

A lot of condition logic can obscure any calculation, even a simple one. When that happens, your calculation can be misunderstood by others and harder to maintain, debug and extend. Strategy is a pattern that deals well with calculations. A context object obtains a Strategy object and then delegates a calculation (or calculations) to that Strategy. This lightens the context class by moving the conditional calculation logic to a small collection of independent calculation objects (*strategies*), each of which can handle one of the various ways of doing the calculation.

Does this sound like a pattern you'd refactor to a lot? It may, but in my experience, I don't refactor to Strategy that often. I certainly have refactored to it, but I find that a lot of calculation logic I either write or come across isn't sufficiently complicated to justify using Strategy. In addition, when there is enough conditional logic to merit using the pattern, I have to consider whether a Template Method would be a better choice. But using a Template Method assumes that you can place the skeleton of your calculation in a base class, and have subclasses supply some or all of the calculation details. That may or may not be possible given your situation. For example, if you already have subclasses and the various ways of calculating something won't easily fit into those subclasses, you may not be able to *Form Template Method* [Fowler]. Or, you may find that by placing calculations in separate subclasses, you limit your ability to swap one calculation type for another at runtime, since it would mean changing the type of object a client is working with, rather than simply substituting one Strategy object for another.

Once you do decide to refactor to Strategy, you have to consider how the calculation embedded in each strategy class will get access to the variables it needs to do its calculation. To accomplish that, I usually pass the context class as a reference to the Strategy object, and make whatever variables are needed by each Strategy accessible via public methods on the context class.

The final thing to consider is how your context class will obtain its Strategy. Whenever possible, I like to shield client code from having to worry about both instantiating a Strategy instance and passing it to a context's constructor. Creation Methods can help with this: just define one or more methods that return a context instance, properly outfitted with the appropriate Strategy instance.

Communication	Duplication	Simplicity
Copious conditional logic obscures the steps of a calculation. Communicate the steps clearly by separating each calculation variety into its own Strategy class. Then clarify which variety of calculation your object uses by writing code to pass the appropriate Strategy to the object for its use in performing the calculation.	Conditional calculation code can often contain duplicate conditional statements that are used to calculate various variables in an algorithm. Replace all of the conditional logic by encapsulating each variety of the calculation in its own Strategy class.	Classes that contain a lot of conditional logic are never simple. But if a class contains lots of conditional logic for calculating something in a variety of ways, it may also be more complex than it needs to be, as it knows too much. Simplify the class by extracting each variety of the calculation into its own Strategy class and then delegate to one of these classes to obtain a calculation.

Mechanics

1. On a class (which we'll call "A") identify a calculation method, or helper methods to such a method, that contain a lot of conditional logic. This class will be known as your context class as it will be the context for a Strategy object.

2. Create a concrete class and name it based on the behavior performed by the chosen calculation method. This will be your Strategy.

You can append the word "Strategy" to the class name if you find it helps communicate the purpose of this new type.

3. Apply *Move Method* [Fowler] to move the primary calculation method and any helper methods to your Strategy class. If the code you move needs to obtain information from A, pass A as a parameter to the primary calculation method or as a parameter to the Strategy class's constructor and make sure the information on A is publicly available.

   You can alternatively pass the necessary information from A to the Strategy, without passing a reference of A to the Strategy. This will result in less coupling between A and your Strategy, but may require you to pass a lot of information. See *Design Patterns* [GoF] for an in-depth discussion about communication between the context, A, and the Strategy.

4. Create a field (which we'll call "S") in A for the Strategy and instantiate it.

5. Update the primary calculation method in A to delegate the calculation to S.

6. Compile and test

7. On the Strategy class, apply *Replace Conditional with Polymorphism* [Fowler] on the primary calculation method and any helper methods you moved from A. It is best to do this step slowly, by focusing on extracting one subclass at a time, then performing steps 8 and 9 below and then repeating this step. When finished, you will have substantially reduced the conditional logic in your Strategy class and you will have defined concrete Strategy classes for each variety of the calculation you started with.

   Consider applying *Form Template Method* [Fowler] for your Strategy's primary calculation method. You may also make your original Strategy class abstract.

8. Add code to A to either use its internal logic to set the value of S or to allow an external client to pass in a value for S.

If you go with the latter approach, let clients pass in a value for S via constructor calls if clients won't need to change S's value at runtime. Otherwise, supply a setter method to let clients set the value of S at runtime. For convenience, you can also do both. If clients will be able to pass in a value of S to A, you'll need to update the code for every existing client of A.

9. Compile and test.

Example

The example in the code sketch above deals with calculating capital for bank loans. It shows a fair amout of conditional logic that's used in performing this calculation, although it is even less conditional logic than was contained in the original code, which had to handle capital calculations for 7 different loan profiles.

In the example, the context class is called `Loan`. We'll be seeing how `Loan`'s method for calculating capital can be *strategized*, i.e. delegated to a Strategy object. As you study the example, you may wonder why `Loan` wasn't just subclassed to support the three different styles of capital calculations. That *was* an option, however, because the application that uses `Loan` needed to change a `Loan`'s capital calculation at runtime, without changing the class type of the `Loan`, it was better to use the Strategy pattern.

1. We'll begin by looking at the `Loan` class's `calcCapital()` method and its helper methods (note: I show a few tests for `calcCapital()` in step 6 below):

```
public class Loan ...
   private double notional;
   private double outstanding;
   private int rating;
   private double unusedPercentage;
   private Date start;
   private Date expiry;
   private Date maturity;
   private static final int MILLIS_PER_DAY = 86400000;

   public double calcCapital() {
      return riskAmount() * duration() * RiskFactor.forRiskRating(rating);
   }
   private double calcUnusedRiskAmount() {
      return (notional - outstanding) * unusedPercentage;
   }
   private double duration() {
      if (expiry == null)
         return ((maturity.getTime() - start.getTime()) / MILLIS_PER_DAY) / 365;
      else if (maturity == null)
         return ((expiry.getTime() - start.getTime()) / MILLIS_PER_DAY) / 365;
      else {
         long millisToExpiry = expiry.getTime() - start.getTime();
         long millisFromExpiryToMaturity = maturity.getTime() - expiry.getTime();
         double revolverDuration = (millisToExpiry / MILLIS_PER_DAY) / 365;
         double termDuration = (millisFromExpiryToMaturity / MILLIS_PER_DAY) / 365;
         return revolverDuration + termDuration;
      }
   }
   private double riskAmount() {
      if (unusedPercentage != 1.00)
         return outstanding + calcUnusedRiskAmount();
      else
         return outstanding;
   }
   public void setOutstanding(double newOutstanding) {
      outstanding = newOutstanding;
   }
   private void setUnusedPercentage() {
      if (expiry != null && maturity != null) {
         if (rating > 4)
            unusedPercentage = 0.95;
         else
            unusedPercentage = 0.50;
      } else if (maturity != null) {
         unusedPercentage = 1.00;
      } else if (expiry != null) {
         if (rating > 4)
            unusedPercentage = 0.75;
         else
            unusedPercentage = 0.25;
      }
   }
```

2. The Strategy I'd like to define will handle the `calcCapital()` calculation. So I create a class called `CapitalStrategy`.

```
public class CapitalStrategy {
}
```

3. Now I'm up to the hardest step: I need to move methods from `Loan` to `CapitalStrategy`. I begin with the `calcCapital()` method. In this case, I don't want to move this method, but rather, copy it to `CapitalStrategy`:

```
public class CapitalStrategy {
   public double calc() {
      return riskAmount() * duration() * RiskFactor.forRiskRating(rating);
```

```
   }
}
```

That code won't even compile, because `CapitalStrategy` doesn't contain the methods it is calling. No problem. I pass `calc()` a `Loan` parameter and update the code as follows:

```
public double calc(Loan loan) {
    return loan.riskAmount() * loan.duration() * RiskFactor.forRiskRating(loan.rating);
}
```

That gets us closer, but the compiler still complains that the methods and variable I'm accessing on `Loan` aren't visible (i.e. they are private, not public). I change the visibility to public and finally the compiler is happy. Later, I'll be moving some of these public methods/fields to `CapitalStrategy` or making them accessible via `Loan` getter methods.

Now I focus on moving each piece of the calculation from `Loan` to `CapitalStrategy`. The method, `riskAmount()` (which is now public) is first on my radar screen.

```
public double riskAmount() {
   if (unusedPercentage != 1.00)
      return outstanding + calcUnusedRiskAmount();
   else
      return outstanding;
}
```

This method relies on other fields and methods within `Loan`. I study the code and see that the field, `outstanding`, is used extensively in the `Loan` class, but the field, `unusedPercentage`, along with the methods, `setUnusedPercentage()` and `calcUnusedRiskAmout()` are only there to help the `calcCapital()` method. So I decide to move all of this code, with the exception of the field, `outstanding`, to `CapitalStrategy`:

```
public class CapitalStrategy {
   private Loan loan;
   public double calc(Loan loan) {
      this.loan = loan;
      return riskAmount() * loan.duration() * RiskFactor.forRiskRating(loan.rating);
   }
   private double calcUnusedPercentage() {
      if (loan.expiry != null && loan.maturity != null) {
         if (loan.rating > 4)
            return 0.95;
         else
            return 0.50;
      } else if (loan.maturity != null) {
         return 1.00;
      } else if (loan.expiry != null) {
         if (loan.rating > 4)
            return 0.75;
         else
            return 0.25;
      }
      return 0.0;
   }
   private double calcUnusedRiskAmount() {
      return (loan.notional - loan.outstanding) * calcUnusedPercentage();
   }
   public double riskAmount() {
      if (calcUnusedPercentage() != 1.00)
         return loan.outstanding + calcUnusedRiskAmount();
      else
         return loan.outstanding;
   }
}
```

To make this compile, I need to make more fields on `Loan` public:

```
public class Loan ...
```

```
   public double notional;
   public double outstanding;
   public int rating;
   private double unusedPercentage; //replaced with calculation method on CapitalStrategy
   public Date start;
   public Date expiry;
   public Date maturity;
```

By now I'm not happy having all these public fields. So I make getter methods for them and update the `CapitalStrategy` code accordingly. After this, all I do is move the `duration()` calculation over to `CapitalStrategy` and this step of the refactoring is done. `CapitalStrategy` now looks like this:

```
public class CapitalStrategy {
   private Loan loan;
   private static final int MILLIS_PER_DAY = 86400000;
   public double calc(Loan loan) {
      this.loan = loan;
      return riskAmount() * duration() * RiskFactor.forRiskRating(loan.getRating());
   }
   private double calcUnusedPercentage() {
      if (loan.getExpiry() != null && loan.getMaturity() != null) {
         if (loan.getRating() > 4) return 0.95;
         else return 0.50;
      } else if (loan.getMaturity() != null) {
         return 1.00;
      } else if (loan.getExpiry() != null) {
         if (loan.getRating() > 4) return 0.75;
         else return 0.25;
      }
      return 0.0;
   }
   private double calcUnusedRiskAmount() {
      return (loan.getNotional() - loan.getOutstanding()) * calcUnusedPercentage();
   }
   public double duration() {
      if (loan.getExpiry() == null)
         return (
            (loan.getMaturity().getTime() - loan.getStart().getTime()) / MILLIS_PER_DAY)
            / 365;
      else if (loan.getMaturity() == null)
         return (
            (loan.getExpiry().getTime() - loan.getStart().getTime()) / MILLIS_PER_DAY)
            / 365;
      else {
         long millisToExpiry = loan.getExpiry().getTime() - loan.getStart().getTime();
         long millisFromExpiryToMaturity =
            loan.getMaturity().getTime() - loan.getExpiry().getTime();
         double revolverDuration = (millisToExpiry / MILLIS_PER_DAY) / 365;
         double termDuration = (millisFromExpiryToMaturity / MILLIS_PER_DAY) / 365;
         return revolverDuration + termDuration;
      }
   }
   public double riskAmount() {
      if (calcUnusedPercentage() != 1.00)
         return loan.getOutstanding() + calcUnusedRiskAmount();
      else
         return loan.getOutstanding();
   }
}
```

4. Now I need to make a field in the `Loan` class for the `CapitalStrategy` class:

```
public class Loan…
   private CapitalStrategy capitalStrategy = new CapitalStrategy();
```

5. And I'm finally ready to have Loan delegate its calculation of capital to `CapitalStrategy`'s `calc()` method:

```
public double calcCapital() {
```

```
   return capitalStrategy.calc(this);
}
```

6. I can now compile and run my tests. Here are a few of the tests that ensure whether the capital calculation works for various types of loan profiles:

```
public void testTermLoanCapital() {
   Loan termLoan = Loan.newTermLoan(10000.00, startOfLoan(), maturity(), RISK_RATING);
   termLoan.setOutstanding(10000.00);
   assertEquals("Capital for Term Loan", 37500.00, termLoan.calcCapital(), penny);
}
public void testRevolverROC() {
   Loan revolver = Loan.newRevolver(10000.00, startOfLoan(), expiry(), RISK_RATING);
   revolver.setOutstanding(2000.00);
   assertEquals("Capital for Revolver", 6000.00, revolver.calcCapital(), penny);
}
public void testRevolverTermROC() {
   Loan rctl = Loan.newRCTL(10000.00, startOfLoan(), expiry(), maturity(),RISK_RATING);
   rctl.setOutstanding(5000.00);
   assertEquals("Capital for RCTL", 28125.00, rctl.calcCapital(), penny);
}
```

These tests, and similar ones, all run successfully.

7. At this point I've moved a lot of code out of the `Loan` class and into the `CapitalStrategy` class, which now encapsulates the bulky conditional calculation logic. I want to tame this logic by decomposing `CapitalStrategy` into several subclasses, one for each way we calculate capital. I do this by applying *Replace Conditional with Polymorphism* [Fowler].

First, I identify a total of three different ways of doing the capital calculation, each of which corresponds to a specific Loan profile: Term loan, Revolver or RCTL (a combination of a Revolver, which converts to a Term Loan on an expiry date). I decide to start by creating a subclass of `CapitalStrategy` that is capable of calculating capital for a Term Loan:

```
public class TermLoanCapital extends CapitalStrategy {
}
```

Now, I find the specific calculation code that applies to a Term Loan and push it down into the new subclass:

```
public class TermLoanCapital extends CapitalStrategy {
   protected double duration() {
      return (
         (loan.getMaturity().getTime() - loan.getStart().getTime()) / MILLIS_PER_DAY)
         / 365;
   }
   protected double riskAmount() {
      return loan.getOutstanding();
   }
}
```

I now push on to steps 8 and 9 of the refactoring, after which I'll circle back to define, configure and test two more concrete Strategy classes: `RevolverCapital` and `RCTLCapital`.

8. Now I need to configure the Loan class with the `TermLoanCapital` strategy when it is applicable, so that I can test whether it works. To do this, I make the following modifications:

```
public class Loan...
   private CapitalStrategy capitalStrategy;

   protected Loan(double notional, Date start, Date expiry,
      Date maturity, int riskRating, CapitalStrategy strategy) {
      this.notional = notional;
      this.start = start;
```

```
      this.expiry = expiry;
      this.maturity = maturity;
      this.rating = riskRating;
      this.capitalStrategy = strategy;
   }
   public static Loan newRCTL(double notional, Date start, Date expiry,
      Date maturity, int rating) {
      return new Loan(notional, start, expiry, maturity, rating, new CapitalStrategy());
   }
   public static Loan newRevolver(double notional, Date start, Date expiry,
      int rating) {
      return new Loan(notional, start, expiry, null, rating, new CapitalStrategy());
   }
   public static Loan newTermLoan(double notional, Date start, Date maturity,
      int rating) {
      return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
   }
```

9. I compile and test and all goes well. Now I circle back to step 7, to define the additional concrete Strategy classes, configure the Loan class to work with them and test everything. When I'm done, almost all of the original conditional calculation logic is gone and I have three Strategies for calculating capital:

```
public class Loan...
   public static Loan newRCTL(double notional, Date start, Date expiry,
      Date maturity, int rating) {
      return new Loan(notional, start, expiry, maturity, rating, new RCTLCapital());
   }
   public static Loan newRevolver(double notional, Date start, Date expiry,
      int rating) {
      return new Loan(notional, start, expiry, null, rating, new RevolverCapital());
   }
   public static Loan newTermLoan(double notional, Date start, Date maturity,
      int rating) {
      return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
   }

public abstract class CapitalStrategy {
   protected Loan loan;
   protected static final int MILLIS_PER_DAY = 86400000;
   public double calc(Loan loan) {
      this.loan = loan;
      return riskAmount() * duration() * RiskFactor.forRiskRating(loan.getRating());
   }
   protected abstract double duration();
   protected abstract double riskAmount();
}

public class TermLoanCapital extends CapitalStrategy {
   protected double duration() {
      return (
         (loan.getMaturity().getTime() - loan.getStart().getTime()) / MILLIS_PER_DAY)
         / 365;
   }
   protected double riskAmount() {
      return loan.getOutstanding();
   }
}

public class RevolverCapital extends CapitalStrategy {
   private double calcUnusedPercentage() {
      if (loan.getRating() > 4) return 0.75;
      else return 0.25;
   }
   private double calcUnusedRiskAmount() {
      return (loan.getNotional() - loan.getOutstanding()) * calcUnusedPercentage();
   }
   protected double duration() {
      return (
         (loan.getExpiry().getTime() - loan.getStart().getTime()) / MILLIS_PER_DAY)
         / 365;
   }
   protected double riskAmount() {
```

```
      return loan.getOutstanding() + calcUnusedRiskAmount();
   }
}

public class RCTLCapital extends CapitalStrategy {
   private double calcUnusedPercentage() {
      if (loan.getRating() > 4) return 0.95;
      else return 0.50;
   }
   private double calcUnusedRiskAmount() {
      return (loan.getNotional() - loan.getOutstanding()) * calcUnusedPercentage();
   }
   protected double duration() {
      long millisToExpiry = loan.getExpiry().getTime() - loan.getStart().getTime();
      long millisFromExpiryToMaturity =
         loan.getMaturity().getTime() - loan.getExpiry().getTime();
      double revolverDuration = (millisToExpiry / MILLIS_PER_DAY) / 365;
      double termDuration = (millisFromExpiryToMaturity / MILLIS_PER_DAY) / 365;
      return revolverDuration + termDuration;
   }
   protected double riskAmount() {
      return loan.getOutstanding() + calcUnusedRiskAmount();
   }
}
```

Thinking I'm now done, I inspect the results of the refactoring. I wonder, "Is there anything left to simplify or communicate better?" "Is there any duplication to remove?" The duration calculations for the three strategies execute a similar formula: find the difference in time between two dates, divide them by the number of milliseconds in a day, and divide that by 365. That formula is being duplicated! To remove the duplication, I apply *Pull Up Method* [Fowler]:

```
public abstract class CapitalStrategy…
   private static final int DAYS_PER_YEAR = 365;
   protected double calcDuration(Date start, Date end) {
      return ((end.getTime() - start.getTime()) / MILLIS_PER_DAY) / DAYS_PER_YEAR;
   }

public class TermLoanCapital extends CapitalStrategy…
   protected double duration() {
      return calcDuration(loan.getStart(), loan.getMaturity());
   }

public class RevolverCapital extends CapitalStrategy {
   protected double duration() {
      return calcDuration(loan.getStart(), loan.getExpiry());
   }

public class RCTLCapital extends CapitalStrategy…
   protected double duration() {
      double revolverDuration = calcDuration(loan.getStart(), loan.getExpiry());
      double termDuration = calcDuration(loan.getExpiry(), loan.getMaturity());
      return revolverDuration + termDuration;
   }
```

I compile, run the tests and everything is good. Now, for the moment, I'm done.

Replace Implicit Tree with Composite

You implicitly form a tree structure, using a primitive representation, such as a String

Replace your primitive tree representation with Composite

```
String orders = "<orders>";
orders += "<order number='123'>";
orders += "<item number='x1786'>";
orders += "carDoor";
orders += "</item>";
orders += "</order>";
orders += "</orders>";
```

```
TagNode orders = new TagNode("orders");
    TagNode order  = new TagNode("order");
    order.addAttribute("number", "123");
    orders.add(order);
        TagNode item = new TagNode("item");
        item.addAttribute("number", "x1786");
        item.addValue("carDoor");
        order.add(item);
String xml = orders.toString();
```

Motivation

One problem with implicit tree construction is the tight coupling between the code that builds the tree and how the tree is represented. Consider the example above, in which an XML document is built using a String. The nodes on the built XML tree and the way that they are formatted are combined in one place. While that may seem simple, it actually makes it harder to change the tree's representation and forces every programmer to remember every tree representation rule: like using single quotes for attributes or closing all open tags. I've seen programmers fight many bugs that originated in primitive tree formatting mistakes.

A Composite encapsulates how a tree is represented. This means that a client only needs to tell a Composite what to add to a tree and where to add it. When a client needs a representation of the tree, it can ask the Composite to render it. This simpler arrangement leads to less error-prone code.

But this doesn't mean that you should always avoid using primitive tree construction. What if your system doesn't create many trees? In that case, why go to the trouble of creating a Composite when some primitive tree construction code would do? If you later find that you or others are creating more trees, you can refactor to a solution that simplifies the tree construction perhaps by decoupling the tree-building code from the tree-representation code.

The choice may also involve your development speed. On a recent project, I was tasked with generating an HTML page from XML data using an XSLT processor. For this task, I needed to generate an XML tree that would be used in the XSLT transformation. I knew I could use a Composite to build that tree, but I instead choose to build it with a String. Why? Because I was more interested in going fast and facing every technical hurdle involved in doing the XSLT transformation than I was in producing refined XML tree construction code. When I completed the XSLT transformation, I went back to refactor the primitive tree construction code to use a Composite, since that code was going to be emulated in many areas of the system.

Communication	Duplication	Simplicity
The best tree-construction code communicates the structure of a tree without overwhelming readers with unnecessary tree-construction details. Primitive tree-construction code exposes too many details. Trees composed using Composite communicate better by hiding tedious and repetitive tree-construction tasks.	Code that manually builds a tree often repeats the same set of steps: format a node, add the node to the tree and balance the node with a corresponding node or some such thing. Composite-constructed trees minimize duplication by encapsulating repetitive instructions, like formatting nodes and tree-construction mechanics.	It's easier to make mistakes building trees manually than it is to build trees using Composite. Manually-constructed trees must ensure that child nodes are added correctly – for example, a tag in an XML tree must have a corresponding end tag. By knowing how to construct themselves, Composite-constructed trees are simpler.

Mechanics

1. Identify the primitive tree-construction code you'd like to refactor.

2. Identify node types for your new Composite. Keep it simple: test-first design one or more concrete node types and don't worry about creating an abstract node type (you may not need one). Create a method to validate the contents of your budding Composite.

3. Give your nodes the ability to have children. Do not give nodes the ability to remove children if your application only adds nodes and never removes them. Compile and test.

4. If needed, give clients a way to set attributes on nodes. Compile and test.

5. Replace the original tree-construction code with calls to your new Composite. Compile and test.

Example

1. We'll begin with the XML example from the code sketch above:

```
String orders = "<orders>";
orders += "<order number='123'>";
orders += "<item number='x1786'>";
orders += "carDoor";
orders += "</item>";
orders += "</order>";
orders += "</orders>";
```

2. In this case, every node in the tree has an open tag ("<orders>") and close tag ("</orders>"). While some of the nodes have attributes and values, I identify just one node type that we need to produce a Composite version of this tree. I test-first design a node type called TagNode, give this class a way to set its name and create a toString() method to return the resulting XML:

```
public void testOneNodeTree() {
    String expectedResult =
    "<orders>" +
    "</orders>";
    TagNode orders = new TagNode("orders");
    assertXMLEquals("xml comparison", expectedResult, orders.toString());
}

public class TagNode {
    private String tagName;
```

```
    public TagNode(String name) {
        tagName = name;
    }
    public String toString() {
        String result = new String();
        result += "<" + tagName + ">";
        result += "</" + tagName + ">";
        return result;
     }
}
```

3. Next, I give TagNode the ability to have children.

```
public void testAddingChildrenToTree() {
    String expectedResult =
    "<orders>" +
        "<order>" +
            "<item>" +
            "</item>" +
        "</order>" +
    "</orders>";
    TagNode orders = new TagNode("orders");
    TagNode order = new TagNode("order");
    TagNode item = new TagNode("item");
    orders.add(order);
    order.add(item);
    assertXMLEquals("adding children", expectedResult, orders.toString());
}

public class TagNode {
    private String tagName;
    private List children = new ArrayList();
    public TagNode(String name) {
        tagName = name;
    }
    public void add(TagNode childNode) {
        children.add(childNode);
    }
    public String toString() {
        String result = new String();
        result += "<" + tagName + ">";
        Iterator it = children.iterator();
        while (it.hasNext()) {
            TagNode node = (TagNode)it.next();
            result += node.toString();
        }
        result += "</" + tagName + ">";
        return result;
     }
}
```

4. Now the Composite must be extended to support XML attributes or values or both. Again, I do this by letting my test code drive the development process:

```
 public void testTreeWithAttributesAndValues() {
    String expectedResult =
    "<orders>" +
        "<order>" +
            "<item number='12660' quantity='1'>" +
            "Dog House" +
            "</item>" +
            "<item number='54678' quantity='1'>" +
            "Bird Feeder" +
            "</item>" +
        "</order>" +
    "</orders>";
    TagNode orders = new TagNode("orders");
    TagNode order = new TagNode("order");
    TagNode item1 = new TagNode("item");
    item1.addAttribute("number", "12660");
    item1.addAttribute("quantity", "1");
    item1.setValue("Dog House");
```

```
    TagNode item2 = new TagNode("item");
    item2.addAttribute("number", "54678");
    item2.addAttribute("quantity", "1");
    item2.setValue("Bird Feeder");
    orders.add(order);
    order.add(item1);
    order.add(item2);
    assertXMLEquals("attributes&values", expectedResult, orders.toString());
}

public class TagNode {
    private String tagName;
    private String tagValue = "";
    private String attributes = "";
    private List children = new ArrayList();
    public TagNode(String name) {
        tagName = name;
    }
    public void add(TagNode childNode) {
        children.add(childNode);
    }
    public void addAttribute(String name, String value) {
        attributes += (" " + name + "='" + value + "'");
    }
    public void addValue(String value) {
        tagValue = value;
    }
    public String toString() {
        String result = new String();
        result += "<" + tagName + attributes + ">";
        Iterator it = children.iterator();
        while (it.hasNext()) {
            TagNode node = (TagNode)it.next();
            result += node.toString();
        }
        if (!tagValue.equals(""))
            result += tagValue;
        result += "</" + tagName + ">";
        return result;
     }
}
```

5. In the final step, I replace the original primitive tree-construction code with the Composite code, compile and test:

```
TagNode orders = new TagNode("orders");
     TagNode order  = new TagNode("order");
     order.addAttribute("number", "123");
     orders.add(order);
          TagNode item = new TagNode("item");
          item.addAttribute("number", "x1786");
          item.addValue("carDoor");
          order.add(item);
```

Encapsulate Composite with Builder

Your Composite code exposes too many details, forcing clients to create, format, add and remove nodes and handle validation logic

Encapsulate the Composite with a simpler, more intention-revealing Builder

```
TagNode orders = new TagNode("orders");
    TagNode order  = new TagNode("order");
    order.addAttribute("number", "123");
    orders.add(order);
        TagNode item = new TagNode("item");
        item.addAttribute("number", "x1786");
        item.addValue("carDoor");
        order.add(item);
String xml = orders.toString();
```

```
XMLBuilder orders = new  XMLBuilder("orders");
    orders.addBelow("order");
    orders.addAttribute("number", "123");
        orders.addBelow("item");
        orders.addAttribute("number", "x1786");
        orders.addValue("carDoor");
String xml = orders.toString();
```

Motivation

I'm always interested in simplifying client code: I want it to read as clearly as English. So when it comes to creating really simple tree-construction code, I like the Builder pattern even better than the Composite pattern. Builders give clients a clean and easy-to-use interface while hiding details about how the nodes of a Composite are hooked together and what accompanying steps must take place during construction.

If you study a typical piece of client code that creates some tree structure, you'll often find node creation and setup logic mixed together with tree creation and validation logic. A Builder-based alternative can simplify such code by taking on the burden of node creation and tree validation logic and let client code concentrate on what is important: building the tree. The result of refactoring to Builder is often simpler, more intention-revealing client code.

I use Builders a lot with XML. XML documents represent trees, so they work well with both the Composite and Builder patterns. But Composite-only solutions for creating an XML tree expose too many details. XML Builders, by contrast, offer a nice way to have your cake and eat it too: clients talk to a simple XML Builder interface, while the XML Builder itself relies on a Composite for representing the XML tree. The example below will show you how this is done. In addition, I've included an extended example which shows how an XML Builder was updated to implement and encapsulate performance logic used in rendering a Composite of XML nodes to a string.

Communication	Duplication	Simplicity
Client code that creates a tree needs to communicate the essence of the activity: what is added to the tree, and where it is added. A Composite solution doesn't communicate this clearly because it exposes too many details. By handling the tree-construction details, Builders enable client code to communicate clearly.	Composite-based tree-construction code is filled with calls to create new nodes and add them to trees. Builder code removes this duplication by handling node creation and simplifying how nodes are added to a tree.	With a Composite, a client must know *what*, *where* and *how* to add items to a tree. With a Builder, a client needs to know only *what* and *where* to add to the tree; the Builder takes care of the rest. Builders often simplify client code by handling the mechanics of tree construction.

Mechanics

1. Identify the Composite that you would like to encapsulate.

2. Create a new Builder class:
   - Give the new class a private instance variable for the encapsulated Composite.
   - Initialize the Composite in a constructor.
   - Create a method to return the results of doing a build.

3. Create intention-revealing methods on your Builder for every type of node that gets added to your Composite. These methods will add new nodes to an inner Composite and keep track of the state of the tree.

   You may create additional methods to let users set attributes on nodes, or you can let users add new nodes and set attributes on them using one convenient method call.

4. Replace the tree-construction Composite calls with calls to the Builder. Compile and test.

Example

1. We'll begin with the Composite code that was shown in the code sketch above. As I study this code, I realize that it contains more detail than it needs to:

```
TagNode orders = new TagNode("orders");
     TagNode order  = new TagNode("order");
     order.addAttribute("number", "123");
     orders.add(order);
          TagNode item = new TagNode("item");
          item.addAttribute("number", "x1786");
          item.addValue("carDoor");
          order.add(item);
```

2. I define an XMLBuilder class, encapsulate the original Composite, initialize it and write a toString() method to obtain the results of a build. I do this all from test code, which helps me confirm that my new class produces correct XML.

```
public void testOneElementTree() {
    String expected =
    "<orders>" +
    "</orders>";
    XMLBuilder builder = new XMLBuilder("orders");
    assertXMLEquals("one element tree", expected, builder.toString());
}
```

Now, my Builder looks like this:

```
public class XMLBuilder {
```

```
    private TagNode root;
    public XMLBuilder(String rootName) {
        root = new TagNode(rootName);
    }
    public String toString() {
        return root.toString();
    }
}
```

3. Next, I create methods for every type of node that gets added to the Composite. In this case it's trivial: there are only `TagNodes`. But I still have to consider the different ways in which clients will add nodes to the inner Composite. I begin with the case of adding nodes as children of parent nodes:

```
public void testAddBelow() {
    String expected =
    "<orders>" +
        "<order>" +
            "<item>" +
            "</item>" +
        "</order>" +
    "</orders>";
    XMLBuilder builder = new XMLBuilder("orders");
    builder.addBelow("order");
    builder.addBelow("item");
    assertXMLEquals("adding below", expected, builder.toString());
}
```

This leads to the creation of the `addBelow()` method, along with a few changes to the `XMLBuilder` class:

```
public class XMLBuilder {
    private TagNode root;
    private TagNode current;
    public XMLBuilder(String rootName) {
        root = new TagNode(rootName);
        current = root;
    }
    public void addBelow(String child) {
        TagNode childNode = new TagNode(child);
        current.add(childNode);
        current = childNode;
    }
    public String toString() {
        return root.toString();
    }
}
```

Next I must enable the `XMLBuilder` to add a node at the same level as an existing node (i.e., not as a child, but as a sibling). This leads to more test and `XMLBuilder` code:

```
public void testAddBeside() {
    String expected =
    "<orders>" +
        "<order>" +
            "<item>" +
            "</item>" +
            "<item>" +
            "</item>" +
        "</order>" +
    "</orders>";
    XMLBuilder builder = new XMLBuilder("orders");
    builder.addBelow("order");
    builder.addBelow("item");
    builder.addBeside("item");
    assertXMLEquals("adding beside", expected, builder.toString());
}

public class XMLBuilder {
    private TagNode root;
```

```
    private TagNode current;
    private TagNode parent;
    public XMLBuilder(String rootName) {
        root = new TagNode(rootName);
        current = root;
        parent = root;
    }
    public void addBelow(String child) {
        TagNode childNode = new TagNode(child);
        current.add(childNode);
        parent = current;
        current = childNode;
    }
    public void addBeside(String sibling) {
        TagNode siblingNode = new TagNode(sibling);
        parent.add(siblingNode);
        current = siblingNode;
    }
    public String toString() {
        return root.toString();
    }
}
```

I continue on this approach until I have a working Builder that satisfies all of my tests. In some cases, adding new behavior to the `XMLBuilder` is trivial, since it merely requires delegating calls to the inner Composite. For example, here is how XML attributes are implemented:

```
public void testAddBelowWithAttribute() {
    String expected =
    "<orders>" +
        "<order number='12345' quantity='2'>" +
        "</order>" +
    "</orders>";
    builder = createBuilder("orders");
    builder.addBelow("order");
    builder.addAttribute("number", "12345");
    builder.addAttribute("quantity", "2");
    assertXMLEquals("built xml", expected, builder.toString());
}

public class XMLBuilder. . .
    public void addAttribute(String name, String value) {
        current.addAttribute(name, value);
    }
}
```

4. Now it is time to replace the original client code that used the Composite with the `XMLBuilder`. I do this one line at a time, removing some lines and rewriting others. The final code makes no references to the now encapsulated Composite, `TagNode`.

```
XMLBuilder orders = new  XMLBuilder("orders");
     orders.addBelow("order");
     orders.addAttribute("number", "123");
          orders.addBelow("item");
          orders.addAttribute("number", "x1786");
          orders.addValue("carDoor");
```

Notice how the calls to the `XMLBuilder` are generic – the methods and data passed to them reveal nothing about the underlying structure of the tree. Should we need to work with a variety of Builders, we won't have to change very much client code.

Extended Example

I could not resist telling you about a performance improvement that was made to the above-mentioned `XMLBuilder` class, since it reveals the elegance and simplicity of the Builder pattern. Some of my colleagues at a company called Evant had done some profiling of our system and they'd found that a `StringBuffer` used by the `XMLBuilder`'s encapsulated composite

(TagNode) was causing performance problems. This StringBuffer is used as a Collecting Parameter – it is created and then passed to every node in a composite of TagNodes in order to produce the results returned from calling TagNode's toString(). To see how this works, see the example in *Move Accumulation to Collecting Parameter (78).*

The StringBuffer that was being used in this operation was not instantiated with any particular size, which means that as more and more XML is added to the StringBuffer, it must automatically grow when it can no longer hold all its data. That's fine, since the StringBuffer class was written to be able to automatically grow. But there is a performance penalty one pays when you allow a StringBuffer to automatically grow: i.e. when it has to grow, it has work to do to transparently increase its size. That performance penalty in the Evant system was not acceptable and so the team needed to make an improvement.

The solution was to know what size the StringBuffer needed to be before instantiating it, and then to instantiate it with the proper size so that it would not need to grow. How could we compute this size? Easy. As each node gets added to an XML tree via an XMLBuilder, the builder increments a buffer size based on the size of the strings in the node. Then the final computed buffer size could be used when instantiating the StringBuffer. Let's see how this was implemented.

As usual, we start by writing a test. The test below will build an XML tree by making calls to an XMLBuilder, then it will obtain the size of the resulting XML string returned by the builder and finally, it will compare the size of the string with the computed buffer size for use by a StringBuffer:

```
public void testToStringBufferSize() {
    String expected =
    "<orders>" +
        "<order number='123'>" +
        "</order>" +
    "</orders>";
    builder = createBuilder("orders");
    builder.addBelow("order");
    builder.addAttribute("number", "123");

    int stringSize = builder.toString().length();
    int computedSize = ((XMLBuilder)builder).bufferSize();
    assertEquals("buffer size", stringSize, computedSize);
}
```

To pass this test and others like it, the following XMLBuilder attributes and methods were added or updated:

```
public class XMLBuilder {
    private int outputBufferSize;
    private static int TAG_CHARS_SIZE = 5;
    private static int ATTRIBUTE_CHARS_SIZE = 4;

    public void addAttribute(String name, String value) {
        // logic for adding an attribute to a tag
        incrementBufferSizeByAttributeLength(name, value);
    }
    public void addBelow(String child) {
        // logic for adding a Tag below another Tag
        incrementBufferSizeByTagLength(child);
    }
    public void addBeside(String sibling) {
        // logic for adding a Tag beside another Tag
        incrementBufferSizeByTagLength(sibling);
    }
    public void addBesideParent(String uncle) {
        // logic for adding a Tag beside the current Tag's parent
        incrementBufferSizeByTagLength(uncle);
    }
    public void addValue(String value) {
        // logic for adding a value to a node
        incrementBufferSizeByValueLength(value);
```

```
    }
    public int bufferSize() {
        return outputBufferSize;
    }
    private void incrementBufferSizeByAttributeLength(String name, String value) {
        outputBufferSize += (name.length() + value.length() + ATTRIBUTE_CHARS_SIZE);
    }
    private void incrementBufferSizeByTagLength(String tag) {
        int sizeOfOpenAndCloseTags = tag.length() * 2;
        outputBufferSize += (sizeOfOpenAndCloseTags + TAG_CHARS_SIZE);
    }
    private void incrementBufferSizeByValueLength(String value) {
        outputBufferSize += value.length();
    }
    protected void init(String rootName) {
        // logic for initializing the builder and root node
        outputBufferSize = 0;
        incrementBufferSizeByTagLength(rootName);
    }
}
```

The changes made to the `XMLBuilder` are transparent to the users of the builder, as it encapsulates this new performance logic. The only additional change must be made to the `XMLBuilder's toString()` method, so that it can instantiate a `StringBuffer` of the correct size, and pass it on to the root `TagNode`, which will accumulate the contents of the XML tree. To make that happen, the `toString()` method was changed from

```
    public String toString() {
        return root.toString();
    }
```

to:

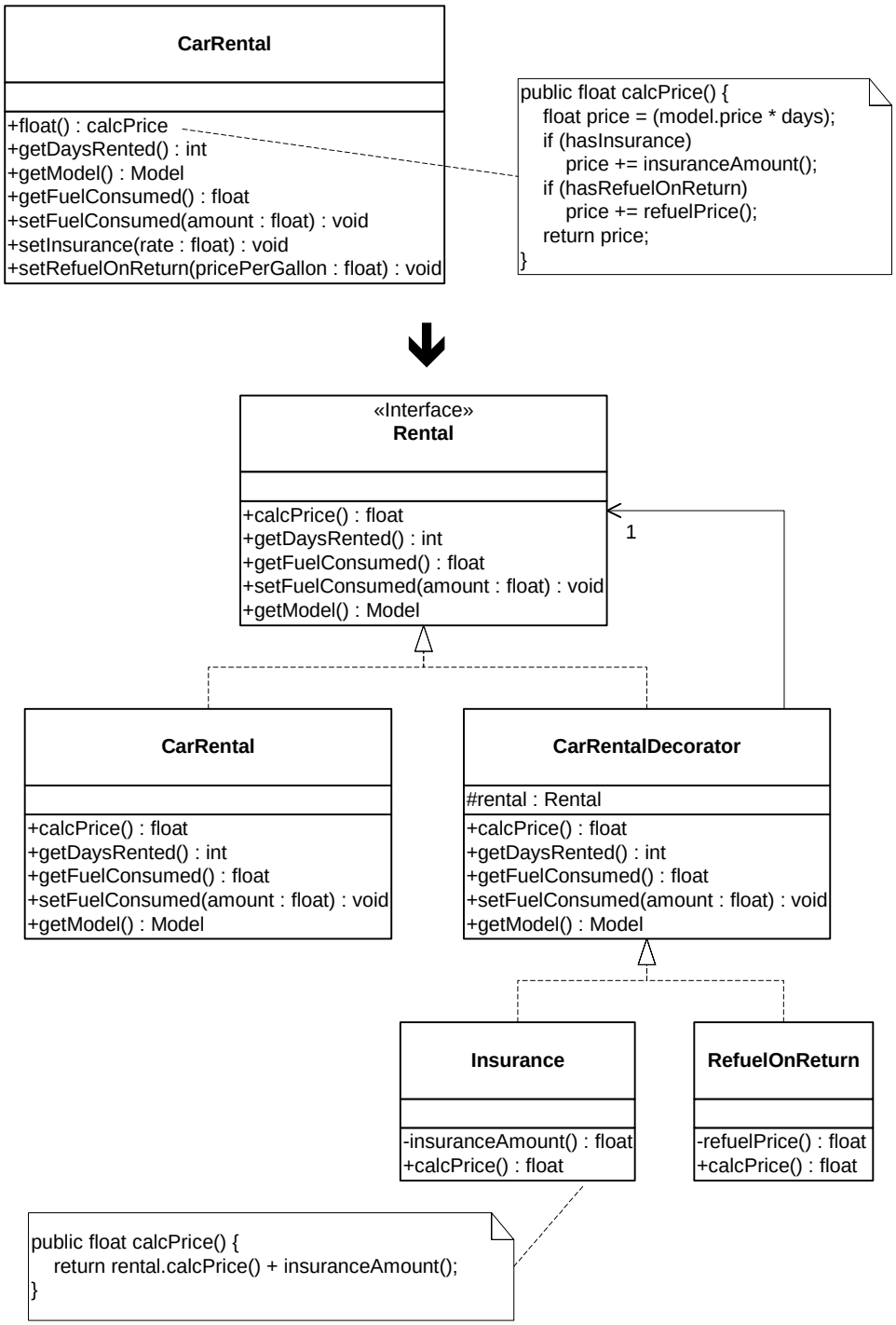
```
    public String toString() {
        return root.toStringHelper(new StringBuffer(outputBufferSize));
    }
```

And that was it. The tests passed and the `XMLBuilder` was now significantly faster.

Extract Special-Case Behavior into Decorators

Your classes or methods contain special-case behavior

Retain the core behavior but extract the optional or special-case behavior into Decorators



Motivation

Decorator is one of my favorite Patterns. It is simple and elegant, but I have to resist overusing it. The fact is, many problem chunks of code simply don't need to be refactored to use Decorator. Simpler solutions are often better. However, there is a time and place for this refactoring, and when you do use it to solve the right problems, it can add a great deal of clarity and simplicity to your design.

So what are the types of problems that merit this refactoring? Glad you asked. Let's look at an example. Consider an Invoice class that is responsible for keeping track of payment information for a customer invoice. Most invoices are simple - some dollar amount is owed, and all the Invoice object has to do is calculate the amount owed. But what happens when the amount owed is overdue or if a special discount must be applied because the customer is a preferred customer? Those are two special conditions that the Invoice's `calcAmountOwed()` method will have to deal with. No big deal – we probably still don't need a fancy Decorator to clean up the small amount of conditional logic in Invoice's `calcAmountOwed()` method.

But what happens when we add more special conditions to `calcAmountOwed()`? As more special conditions are added, the Invoice class gets more complex: it holds onto more instance variables, it supports more getter and setter methods for handling special conditions and its calculation logic gets longer and more involved.

So now we have a more complex Invoice class. Do we need it? What happens if you observe that most of the time the system needs to work with the simplest of Invoice objects – no special conditions, just a simple dollar amount that some customer owes. There are a few places in the system where the special conditions are needed, but not many. So why mix this some-of-the-time logic with your core logic? Keeping this logic together just makes your class more heavyweight, harder to understand and harder to maintain. This is good reason to refactor to Decorator.

What are other conditions under which this refactoring makes sense? Say your code is calling special methods on related objects, but you'd really like to have your code talk to one method on a common interface and handle the special stuff behind the scenes. Essentially, you are trying to make your processing logic polymorphic. So this may be a good place to refactor to Decorator, but maybe not. If you can remove all of the client calls to special methods and replace them with a single intention-revealing method, your code will be simpler and easier to understand. But what will you have to implement to make this possible?

There is some work involved in implementing this refactoring. In Java, refactoring to Decorator involves creating a Decorator class and special-purpose concrete Decorator subclasses as well as producing instantiation code that will wrap objects with the appropriate Decorator(s). This is a fair amount of work. It will make sense to do this only if you have more than one or two chunks of special behavior and/or you can really simplify your design with this refactoring.

Communication	Duplication	Simplicity
Some code just doesn't have to be run very often. But if you lump that code in with code that *does* have to be run often, you don't communicate what is and what is not important. Decorators give you a way to communicate what is core code from what is optional.	As logic gets more complicated, you often see code that tries to accommodate many combinations of behavior. This can lead to a lot of duplicate code. Decorators offer a better way to handle diverse combinations of behavior without duplicating code.	Code that mixes together the essential with the optional isn't as simple as code that contains solely what is essential. On the other hand, Decorators aren't always simple to use when you have to worry about the order in which you add them.

Mechanics

1. On some class (we'll call it "A") find an algorithm that is bulky with optional or special-case processing logic. Choose a piece of logic to extract.

2. Create an interface (we'll call it "IA") composed of all of A's public methods and make A implement that interface.

3. Create a class that implements the IA interface and name this class after the optional or special-case logic you chose. This will be your first concrete Decorator.

   Don't worry about creating an abstract Decorator at this point. Abstract Decorators are only needed when you have multiple concrete Decorators that need to share part of their implementation.

4. In your new Decorator, create an instance variable of type IA (we'll call it "delegate") and let users set it from a constructor argument.

5. For each method defined by your Decorator, forward each method call to the same method on delegate.

6. Test that your Decorator works: create a new instance of A, decorate it with an instance of your new Decorator and assert that it works just like an instance of A.

7. Now move the piece of logic you chose in step 1 to your new Decorator. This step may require you to make changes to IA and A that let the moved logic function without duplication of state or behavior.

8. Test that your Decorator still works: create an instance of A, decorator it with an instance of your Decorator and assert that it works just like an instance of A.

9. Repeat for any other Decorators you would like to create. As you do this, it is best to factor out common Decorator code into an abstract Decorator class. As soon as you have created more than one Decorator, test that decorating objects with multiple Decorators work.

   You have to be very careful with supporting multiple Decorators. It is best to have Decorators be so independent of each other that they can be added to objects in any combination. In practice, however, that may not be possible, in which case you can write Creation Methods to give access to objects decorated in various ways.

10. Adjust client code to refer to IA instead of A, and decorate instances of A where necessary.

Example

If you've ever rented a car, you know that you can rent different types of cars with different rental options, such as an insurance or no-insurance option, a refuel or no-refuel option, one driver or additional drivers, limited miles or unlimited miles and so forth.

We'll be looking at a `CarRental` class that can handle just two rental options: insurance and the refuel option. We'll be refactoring this code to use Decorator to show how this refactoring is done. However, if you carefully study the Before Code, you may wonder if the code is complicated enough to justify this refactoring. In my opinion, it isn't. I'd prefer if the Before

Code were harder to follow, perhaps having to handle three or more rental options, which could be combined in different ways. But if the example contained all of that code, it might span five pages of code. So please use your imagination and consider that CarRental is more complex than it is in this example.

1. We begin with the `CarRental` class and it's `calcPrice()` method. The optional or special-case logic from `calcPrice()` is highlighted in bold:

```
class CarRental {
    protected float fuelConsumed;
    protected int days;
    protected Model model;
    protected float insuranceRate;
    protected boolean hasInsurance;
    protected boolean hasRefuelOnReturn;
    protected float refuelPrice;

    public CarRental(Model m, int rentalDays) {
        model = m;
        days = rentalDays;
        hasInsurance = false;
        hasRefuelOnReturn = false;
    }
    public float calcPrice() {
        float price = (model.price * days);
        if (hasInsurance)
            price += insuranceAmount();
        if (hasRefuelOnReturn)
            price += refuelPrice();
        return price;
    }
    public int getDaysRented() {
        return days;
    }
    public Model getModel() {
        return model;
    }
    public float getFuelConsumed() {
        return fuelConsumed;
    }
    public void setFuelConsumed(float amount) {
        fuelConsumed = amount;
    }
    private float insuranceAmount() {
        return insuranceRate * getDaysRented();
    }
    public void setInsurance(float rate) {
        insuranceRate = rate;
        hasInsurance = true;
    }
    private float refuelPrice() {
        return(getModel().fuelCapacity - getFuelConsumed()) * refuelPrice;
    }
    public void setRefuelOnReturn(float pricePerGallon) {
        refuelPrice = pricePerGallon;
        hasRefuelOnReturn = true;
    }
}

class Model {
    public float fuelCapacity;
    public float price;
    public String name;

    public Model(float fuelCapacity, float price, String name) {
        this.fuelCapacity = fuelCapacity;
        this.price = price;
        this.name = name;
    }
}
```

In CarRental's `calcPrice()` method you can see that the algorithm handles cases in which a rental car has insurance or the refuel on return option or both. Below, I show how three different CarRental instances may be created: one that uses none of the special options, one that uses insurance and one that uses both insurance and the refuel option:

```
Model m = new Model(10.0f, 50.0f, "Ford Taurus");
CarRental r1 = new CarRental(m, 5);
assert(r1.calcPrice() == 250.0f);

CarRental r2 = new CarRental(m, 5);
r2.setInsurance(12.5f);
assert(r2.calcPrice() == 312.5f);

CarRental r3 = new CarRental(m, 5);
r3.setInsurance(12.5f);
r3.setRefuelOnReturn(3.75f);
assert(r3.calcPrice() == 350.0f);
```

We will see how the above client code changes after we do the refactoring. Our task now is to choose which piece of special-case logic we want to extract from CarRental's `calcPrice()` method. I will choose the insurance option.

2. Now I must create a common interface to be implemented by the CarRental class and any new Decorators that we create. This interface must be composed of all of CarRental's public methods, since we want existing client code to communicate with CarRental instances (or decorated CarRental instances) using this new interface. After creating the interface, we make CarRental implement it:

```
interface Rental{
    public float calcPrice();
    public int getDaysRented();
    public Model getModel();
    public float getFuelConsumed();
    public void setFuelConsumed(float amount);
    public void setInsurance(float rate);
    public void setRefuelOnReturn(float pricePerGallon);
}

class CarRental implements Rental. . .
```

3. Next, I'll create a concrete Decorator called Insurance. The Insurance Decorator will be used to add an insurance option to CarRental instances. Insurance will also implement the Rental interface:

```
class Insurance implements Rental {
    public float calcPrice() {}
    public int getDaysRented() {}
    public Model getModel() {}
    public float getFuelConsumed() {}
    public void setFuelConsumed(float amount) {}
    public void setInsurance(float rate) {}
    public void setRefuelOnReturn(float pricePerGallon) {}
}
```

4. The next step is to give Insurance a Rental instance variable and let users set that instance from a constructor:

```
class Insurance implements Rental. . .
    private Rental rental;
    public Insurance(Rental rental) {
        this.rental = rental;
    }
```

5. Now, each of Insurance's methods will forward their method calls to the rental instance variable:

```
class Insurance implements Rental {
    private Rental rental;
    public Insurance(Rental rental) {
        this.rental = rental;
    }
    public float calcPrice() {
        return rental.calcPrice();
    }
    public int getDaysRented() {
        return rental.getDaysRented();
    }
    public Model getModel() {
        return rental.getModel();
    }
    public float getFuelConsumed() {
        return rental.getFuelConsumed();
    }
    public void setFuelConsumed(float amount) {
        rental.setFuelConsumed(amount);
    }
    public void setInsurance(float rate) {
        rental.setInsurance(rate);
    }
    public void setRefuelOnReturn(float pricePerGallon) {
        rental.setRefuelOnReturn(pricePerGallon);
    }
}
```

6. I'll now test that the Insurance Decorator works:

```
Model m = new Model(10.0f, 50.0f, "Ford Taurus");
Rental ford = new CarRental(m, 5);
ford.setInsurance(12.5f);
int fordPrice = ford.calcPrice();

Rental insuredFord = new Insurance(new CarRental(m, 5));
insuredFord.setInsurance(12.5f);
int insuredFordPrice = insuredFord.calcPrice();
assert(fordPrice == insuredFordPrice);
```

7. Next, I move the insurance logic from CarRental's `calcPrice()` method and place it in the Insurance Decorator. This involves moving insurance-related variables and methods from CarRental to Insurance. It also provides an opportunity for simplifying the Rental interface, since CarRental's `setInsurance(float rate)` method can be replaced by an insuranceRate parameter being passed to an Insurance constructor:

```
interface Rental{
    public float calcPrice();
    public int getDaysRented();
    public Model getModel();
    public float getFuelConsumed();
    public void setFuelConsumed(float amount);
    ~~public void setInsurance(float rate);~~
    public void setRefuelOnReturn(float pricePerGallon);
}

class CarRental implements Rental {
    ~~protected float insuranceRate;~~
    ~~protected boolean hasInsurance;~~

    public CarRental(Model m, int rentalDays) {
        model = m;
        days = rentalDays;
        ~~hasInsurance = false;~~
        hasRefuelOnReturn = false;
    }
    public float calcPrice() {
        float price = (model.price * days);
        ~~if (hasInsurance)~~
            ~~price += insuranceAmount();~~
```

```
        if (hasRefuelOnReturn)
            price += refuelPrice();
        return price;
    }
    private float insuranceAmount() {
        return insuranceRate * getDaysRented();
    }
    public void setInsurance(float rate) {
        insuranceRate = rate;
        hasInsurance = true;
    }
}
```

Moving insurance logic to the Insurance Decorator involves:

- replacing the `setInsurance(float rate)` method with a constructor argument
- creating an instance variable, called `rate`, to hold the insurance amount
- creating a copy of the old CarRental method, `insuranceAmount()`
- updating the `calcPrice()` method to add the computed insurance amount to the rate computed by the delegate variable, rental.

```
class Insurance implements Rental {
    private float rate;
    private Rental rental;

    public Insurance(Rental rental, float insuranceRate) {
        this.rental = rental;
        rate = insuranceRate;
    }

    private float insuranceAmount() {
        return rate * rental.getDaysRented();
    }
    public float calcPrice() {
        return rental.calcPrice() + insuranceAmount();
    }
    public void setInsurance(float rate) {
        rental.setInsurance(rate);
    }
}
```

8. I now test the Insurance Decorator:

```
Model m = new Model(10.0f, 50.0f, "Ford Taurus");
Rental insuredFord = new Insurance(new CarRental(m, 5), 12.5f);
float insuredFordPrice = insuredFord.calcPrice();
assert(insuredFordPrice == 312.5f);
```

9. I repeat the above steps to turn CarRental's refueling rental option into a Decorator. This further simplifies the CarRental class, which can now be decorated when necessary. In the code below, you can see the reduction of CarRental's responsibilities by looking at the reduction of its public methods and the size of its `calcPrice()` method. In addition, since we now have two Decorators, it makes sense to factor out common behavior into an abstract Decorator superclass.

```
interface Rental{
    public float calcPrice();
    public int getDaysRented();
    public float getFuelConsumed();
    public void setFuelConsumed(float amount);
    public Model getModel();
}

class CarRentalDecorator implements Rental {
    protected Rental rental;
    protected CarRentalDecorator(Rental r) {
        rental = r;
    }
    public float calcPrice() {
```

```
        return rental.calcPrice();
    }
    public int getDaysRented() {
        return rental.getDaysRented();
    }
    public float getFuelConsumed() {
        return rental.getFuelConsumed();
    }
    public void setFuelConsumed(float amount) {
        rental.setFuelConsumed(amount);
    }
    public Model getModel() {
        return rental.getModel();
    }
}

class Insurance extends CarRentalDecorator {
    protected float rate;

    public Insurance(Rental r, float rate) {
        super(r);
        this.rate = rate;
    }
    private float insuranceAmount() {
        return rate * rental.getDaysRented();
    }
    public float calcPrice() {
        return rental.calcPrice() + insuranceAmount();
    }
}

class RefuelOnReturn extends CarRentalDecorator {
    private float refuelPrice;
    public RefuelOnReturn(Rental r, float refuelPrice) {
        super(r);
        this.refuelPrice = refuelPrice;
    }
    private float refuelPrice() {
        return(rental.getModel().fuelCapacity - rental.getFuelConsumed()) * refuelPrice;
    }
    public float calcPrice() {
        return rental.calcPrice() + refuelPrice();
    }
}
```

We must now test that multiple CarRental Decorators work. Here's how:

```
Model m = new Model(10.0f, 50.0f, "Ford Taurus");
Rental insuredFord = new Insurance(new CarRental(m, 5), 12.5f);
Rental refuelInsuredFord = new RefuelOnReturn(insuredFord, 3.75f);
float price = refuelInsuredFord.calcPrice();
assert(price == 350.0f);

Rental refuelFord = new RefuelOnReturn(new CarRental(m, 5), 3.75f);
Rental insuredRefuelFord = new Insurance(refuelFord, 12.5f);
float price = insuredRefuelFord.calcPrice();
assert(insuredFordPrice == 350.0f);
```

10. We change client code that looked like this:

```
Model m = new Model(10.0f, 50.0f, "Ford Taurus");
CarRental r1 = new CarRental(m, 5);
r2.setInsurance(12.5f);
```

to code that looks like this:

```
Model m = new Model(10.0f, 50.0f, "Ford Taurus");
Rental r1 = new Insurance(new CarRental(m, 5), 12.5f);
```

The refactored version of CarRental came out to be 34 lines longer than the original code. That may or may not happen when you do this refactoring – it all depends on the kind of code you'll

be replacing with Decorator. If it is complex conditional code, chances are that adding Decorator may *decrease* the lines of code. But in any event, introducing Decorator into your system should make your code simpler and easier to understand. It may even help you reduce duplication if your code must handle numerous special-case combinations of behavior.

Let me finish by repeating what I said at the beginning of this refactoring: please don't overuse the Decorator pattern. If you'd like to see an excellent example of using Decorator in a design, study the Decorator code in the extensions package of the JUnit testing framework (http://www.junit.org).

Collections.synchronizedMap

[Todo: Write up the story of the move from the synchronized Vector and Hashtable classes to the unsynchronized collections classes that use Collections.synchronizedMap() to obtain a synchronization decorator].

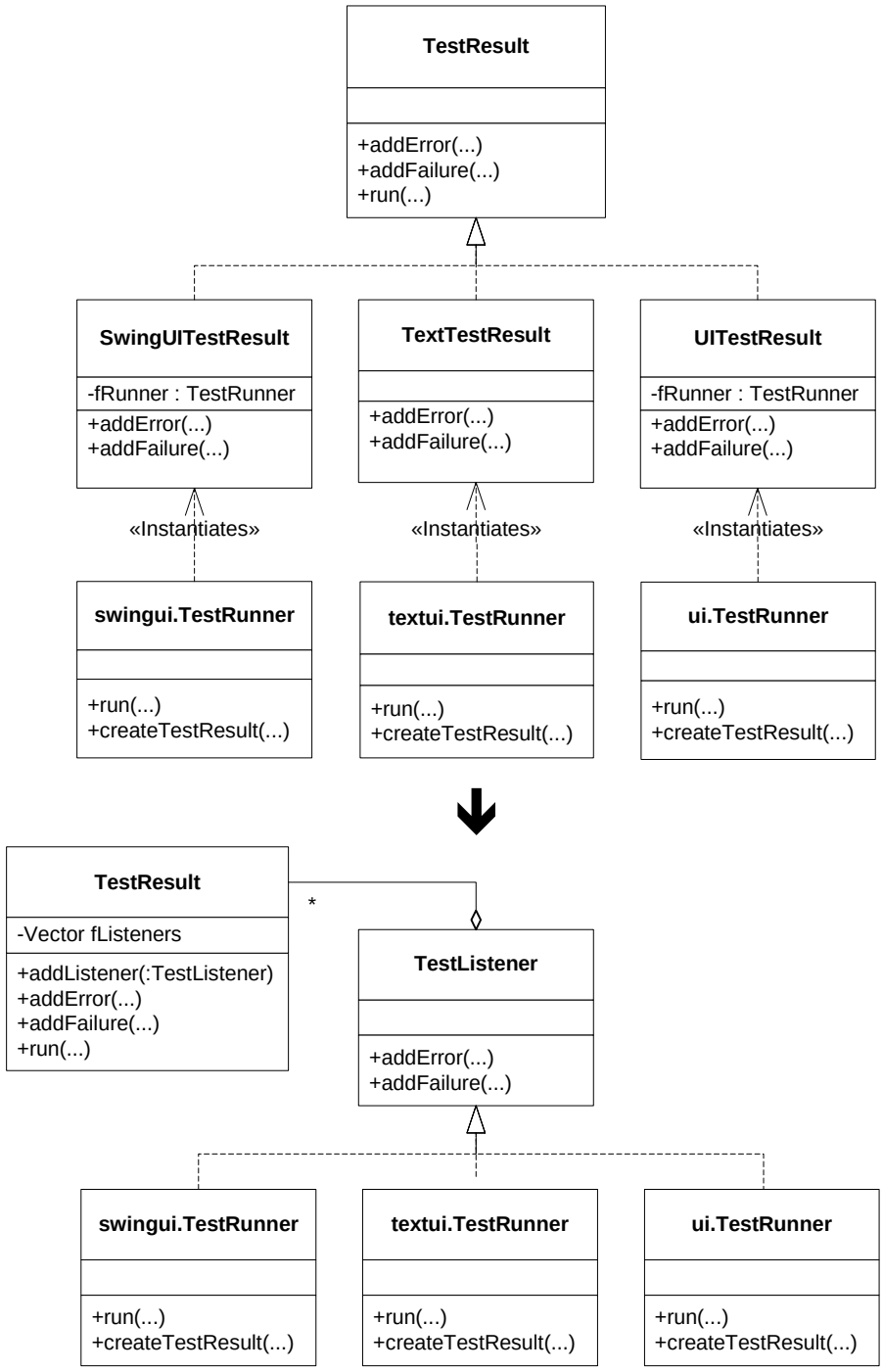
```
Vector
    public synchronized void addElement(Object obj) {
        modCount++;
        ensureCapacityHelper(elementCount + 1);
        elementData[elementCount++] = obj;
    }


    static class SynchronizedCollection implements Collection, Serializable {
    Collection c;   // Backing Collection
    Object      mutex;  // Object on which to synchronize

    SynchronizedCollection(Collection c) {
       this.c = c; mutex = this;
         }

    public boolean add(Object o) {
       synchronized(mutex) {return c.add(o);}
         }
    public boolean remove(Object o) {
       synchronized(mutex) {return c.remove(o);}
         }


Collections…
    public static List synchronizedList(List list) {
    return new SynchronizedList(list);
    }

static class SynchronizedList extends SynchronizedCollection
                            implements List {
    private List list;

    SynchronizedList(List list) {
       super(list);
       this.list = list;
    }
    SynchronizedList(List list, Object mutex) {
              super(list, mutex);
       this.list = list;
         }
```

```
public void add(int index, Object element) {
   synchronized(mutex) {list.add(index, element);}
      }
public Object remove(int index) {
   synchronized(mutex) {return list.remove(index);}
      }

}
```

Replace Hard-Coded Notifications with Observer

Your class or numerous subclasses perform
custom object notifications at designated times

Replace your custom notification code
with the Observer pattern



Motivation

The Observer pattern is popular. Many programmers know it well and use it often. But the trick is to learn when you actually *need* to use Observer and when you don't.

Consider under what circumstances the authors of Design Patterns suggest using Observer (see *Design Patterns,* page 294):

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.

- When a change to one object requires changing others, and you don't know how many objects need to be changed.

- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Now, what happens when you *do* know the object you want to update and it isn't necessarily to have *loose coupling* with that object? For example, class A needs to update objects of type B, based on some event. Since this is a notification responsibility, you may want to implement a solution using the Observer pattern (or Java's Listeners -- essentially the same idea). But do you really need to go that far? Could Observer be too heavyweight a solution given this example? What if you simply wrote code in class A that would notify B objects at appropriate times?

Certainly that could work just fine, until objects of type C also need to be notified about A's events. You could then experiment with your code. See if adding more hard-coded notification logic in class A overcomplicates the class. If it doesn't, you've solved your immediate need without writing much new code.

Eventually, class A's notification responsibilities may grow. As the responsibilities grow, you must observe your own interactions with the code. Ask yourself questions like:

- Am I finding duplicate notification code?
- Am I creating relatively dumb subclasses just to satisfy new notification needs?
- Is my notification logic becoming too complex?
- Is it awkward to pass in object references to class A just for the purpose of notification?

The answers to these questions may lead you to refactor to Observer. Doing so should lead to simpler, smaller and easier-to-read code. Just remember that once you do decide to refactor to Observer, try to do so in the simplest way possible. For example, if your observers will never need to stop getting notifications, do *not* write the removeObserver() code on your Subject class -- it would only be wasted code that no one uses.

Communication	Duplication	Simplicity
Hard-coded object notifications enable runtime collaborations, but the code doesn't communicate this very well: objects get passed into constructors, and notifications happen in random methods. Compare this to a class that implements the Observer pattern – both *who* can observe its events and *when* they get notified is clearly communicated in the class declaration.	If you are compelled to write special code for every class that must be notified at runtime, you can easily produce more code than you need, perhaps resulting in parallel or near-parallel class hierarchies. For only a few notifications, this is no big deal. But as you add more and more special notification code, duplication and code bloat take over.	A few runtime object notifications can be easily handled with simple custom code. But when the number of notifications increases, lots of special code will be written or more and more subclasses will be produced to obtain the necessary behavior. At that point, your code can be simplified by using the Observer pattern.

Mechanics

1. Identify a Subject: a class that accepts an object reference and contains hard-coded notification instructions that couple it directly to the object reference type.

2. Define an Observer: an interface that consists of the set of public methods called by the Subject on the referenced object.

3. Add to the Subject an Observers list and a way for clients to add to that list via a public addObserver(Observer o) method. Add a corresponding removeObserver(Observer o) method only if one is needed.

4. For code in the Subject that accepted an object reference and directly notified that reference, replace with code that iterates over Subject's Observer list, updating each Observer instance.

5. For any class that needs to get notified by Subject, make it implement the Observer interface.

6. Replace code that passed in an object reference to the Subject with code that registers that object reference as an Observer of the Subject. You'll use Subject's addObserver(Observer o) method for this purpose.

7. Compile and test.

Example

The code sketch above is from Kent Beck and Erich Gamma's JUnit Testing Framework. Prior to JUnit 3.x, the authors defined specific TestResult subclasses (like UITestResult, SwingTestResult and TextTestResult) that were responsible for gathering up test information and reporting it to TestRunners. Each TestResult subclass was coupled to a specific TestRunner, such as an AWT TestRunner, Swing TestRunner or Text-based TestRunner. At runtime, after creating a TestResult subclass, a TestRunner would pass itself in as a reference to that TestResult, and then wait to be notified by the TestResult. Each TestResult subclass was hard-coded this way to talk with a specific TestRunner, and that is where our refactoring begins.

In JUnit 3.1, Kent and Erich refactored the TestResult/TestRunner code to use the Observer pattern. This enabled them to eliminate *all* of the special TestResult subclasses (UITestResult, SwingTestResult and TextTestResult) and simplify each of the concrete TestRunners.

Our example will look at this real-world refactoring of the JUnit framework. I've deliberately simplified some of the JUnit code in order to concentrate on the refactoring, not the inner workings of JUnit. However, if you want to study the JUnit code (which I highly recommend), you can download it at http://www.junit.org.

1. Our first task is to find a Subject. In this case, the UITestResult class will be our Subject, but later our Subject will become the TestResult class. What is the reason for this? Well, as a subclass of TestResult, UITestResult doesn't add much new behavior: it exists only because it has the ability to talk directly to an AWT TestRunner class. Our refactoring will seek to eliminate UITestResult and move its behavior up to the TestResult class.

Let's look at the code for all three classes, minus some details you don't need to worry about. I highlight in bold the coupling between UITestResult and its AWT TestRunner:

```
package junit.framework;
public class TestResult extends Object {
    protected Vector fFailures;
```

```
    public TestResult() {
        fFailures= new Vector(10);
    }
    public synchronized void addFailure(Test test, Throwable t) {
        fFailures.addElement(new TestFailure(test, t));
    }
    public synchronized Enumeration failures() {
        return fFailures.elements();
    }
    protected void run(TestCase test) {
        startTest(test);
        try {
            test.runBare();
        }
        catch (AssertionFailedError e) {
            addFailure(test, e);
        }
        endTest(test);
    }
}

package junit.ui;
class UITestResult extends TestResult {
    private TestRunner fRunner;
    UITestResult(TestRunner runner) {
        fRunner= runner;
    }
    public synchronized void addFailure(Test test, Throwable t) {
        super.addFailure(test, t);
        fRunner.addFailure(this, test, t);
    }
    ...
}

package junit.ui;
public class TestRunner extends Frame {
    private TestResult fTestResult;
    ...
    protected TestResult createTestResult(TestRunner runner) {
        return new UITestResult(TestRunner.this);
    }
    synchronized public void runSuite() {
        ...
        fTestResult = createTestResult(TestRunner.this);
        testSuite.run(fTestResult);
    }
    public void addFailure(TestResult result, Test test, Throwable t) {
        fNumberOfFailures.setText(Integer.toString(result.testFailures()));
        appendFailure("Failure", test, t);
    }
}
```

2. Our next task is to define an Observer interface. Kent and Erich call this a TestListener:

```
package junit.framework;
public interface TestListener {
    public void addError(Test test, Throwable t);
    public void addFailure(Test test, Throwable t);
    public void endTest(Test test);
    public void startTest(Test test);
}
```

3. We must now add a list of Observers to our Subject and provide clients (that implement the Observer interface) a way to add themselves to this list. We do this work on the TestResult class rather than the UITestResult class, which we hope to eliminate:

```
public class TestResult extends Object {
    protected Vector fFailures;
    protected Vector fListeners;
    public TestResult() {
        fFailures= new Vector();
        fListeners= new Vector();
    }
```

```
    public synchronized void addListener(TestListener listener) {
        fListeners.addElement(listener);
    }
}
```

4. Now we need to make our Subject update its Observers when an event happens. This involves refactoring TestResult methods like addFailure(), addError() and so on. For simplicity, we will examine only how addFailure() is refactored. Here's what the original method looked like on UITestResult:

```
class UITestResult. . .
    public synchronized void addFailure(Test test, Throwable t) {
        super.addFailure(test, t);
        fRunner.addFailure(this, test, t);
    }
```

Rather than refactor UITestResult's addFailure() method, we focus on the same method in TestResult, the superclass. TestResult's addFailure method will continue to do what it used to do, but it will now iterate through its registered Observers, calling each one's addFailure() method. In this context, since Observers are usually TestRunners, this code will inform each registered TestRunner that a failure has been added. When that happens, the TestRunners have a chance to do things like update a GUI to reflect just how many test failures have occurred. Here's what TestResult's refactored addFailure() method looks like:

```
class TestResult. . .
    public synchronized void addFailure(Test test, AssertionFailedError t) {
        fFailures.addElement(new TestFailure(test, t));
        for (Enumeration e= cloneListeners().elements(); e.hasMoreElements(); ) {
            ((TestListener)e.nextElement()).addFailure(test, t);
        }
    }
```

5. Now, in order for the AWT TestRunner to register itself as an Observer of a TestResult, we must make the ui.TestRunner class implement the TestListener interface:

```
package junit.ui;
public class TestRunner extends Object implements TestListener . . .
```

6. The final step is to register the Observer with the Subject of choice. In this case, we'll look at the code that registers the ui.TestRunner with a TestResult instance:

```
package junit.ui;
public class TestRunner extends Object implements TestListener {
    private Vector fFailedTests;
    private TestResult fTestResult;

    protected TestResult createTestResult() {
        return new TestResult();
    }
    synchronized public void runSuite() {
        . . .
        fTestResult = createTestResult();
        fTestResult.addListener(TestRunner.this);
        . . .
    }
}
```

7. Finally, we can now compile and test that our refactored ui.TestRunner and TestResult work together the way we expect. In the real world, Kent and Erich refactored all of the TestResult subclasses and TestRunners to use the Observer pattern.

Move Accumulation to Collecting Parameter

You have a single bulky method
that accumulates information to a variable

Accumulate your result to a Collecting Parameter
that you pass to extracted methods.

```
class TagNode. . .
public String toString() {
    String result = new String();
    result += "<" + tagName + " " + attributes + ">";
    Iterator it = children.iterator();
    while (it.hasNext()) {
        TagNode node = (TagNode)it.next();
        result += node.toString();
    }
    if (!tagValue.equals(""))
        result += tagValue;
    result += "</" + tagName + ">";
    return result;
 }
```

```
class TagNode. . .
public String toString() {
    return toStringHelper(new StringBuffer(""));
}
private String toStringHelper(StringBuffer result) {
    writeOpenTagTo(result);
    writeChildrenTo(result);
    writeEndTagTo(result);
    return result.toString();
}
```

Motivation

Kent Beck defined the Collecting Parameter pattern in his classic book, *Smalltalk Best Practice Patterns.* A Collecting Parameter is an object that you pass to methods in order to collect information from those methods. A good reason to use this pattern is when you want to decompose a bulky method into smaller methods (using *Extract Method* [Fowler]), and you need to accumulate information from each of the extracted methods. Instead of making each of the extracted methods return a result, which you later combine into a final result, you can incrementally accumulate your result by passing a collecting parameter to each of the extract methods, which in turn, write their results to the collecting parameter.

Collecting Parameter works nicely with the Composite pattern, since you can use a Collecting Parameter to recursively accumulate information from a Composite structure. Kent Beck and Erich Gamma combined these two patterns in their JUnit testing framework to enable a single TestResult object to gather test result information from every test in a hierarchical structure of test case objects.

I recently combined Collecting Parameter with Composite when I refactored a class's `toString()` method (see the code sketch above). My initial goal was to replace a lot of slow String concatenation code with faster `StringBuffer` code, but when I realized that a simple replacement would generate lots of `StringBuffer` instances (because the code is recursive), I retreated from this approach. Then my programming partner at the time, Don Roberts, seized the keyboard, saying "I've got it, I've got it" and then quickly refactored the code to use a single

`StringBuffer` as a Collecting Parameter. The resulting code (partially shown in the code sketch) had a far simpler design, communicated better with the reader and, thanks to the `StringBuffer`, was far more efficient.

Communication	Duplication	Simplicity
Bulky methods don't communicate well. Communicate what you are accumulating by placing each step into intention-revealing methods that write results to a parameter.	You don't often reduce duplicate code using this refactoring. The only exception would be if you have different types of Collecting Parameters that can be passed into the same methods.	Extract Method is at the heart of this refactoring. You use it to reduce a bulky method into a simpler method that delegates to intention-revealing methods.

Mechanics

1. Identify a chunk of code that accumulates information into a variable (we'll call that variable "result"). Result will become your Collecting Parameter. If result's type won't let you iteratively gather data across methods, change result's type. For example, Java's String won't let us accumulate results across methods, so we use a StringBuffer.

2. Find an information accumulation step and extract it into a private method (using *Extract Method* [Fowler]). Make the method's return type be void and pass it result. Inside the method, write information to result.

3. Repeat steps 2 for every accumulation step, until the original code has been replaced with calls to extracted methods that accept and write to result.

4. Compile and test.

Example

In this example, we will see how to refactor Composite-based code to use a Collecting Parameter. We'll start with a composite that can model an XML tree (see *Replace Primitive Tree Construction with Composite* for a complete example of this XML composite code).

The composite is modeled with a single class, called `TagNode`, which has a toString() method. The toString() method recursively walks the nodes in the XML tree, and produces a final String representation of what it finds. It does a fair amount of work in 11 lines of code. We will refactor toString() to make it simpler and easier to understand.

1. The following toString() method recursively accumulates information from every tag in a composite structure and stores results in a variable called "result":

```
class TagNode. . .
public String toString() {
    String result = new String();
    result += "<" + tagName + " " + attributes + ">";
    Iterator it = children.iterator();
    while (it.hasNext()) {
        TagNode node = (TagNode)it.next();
        result += node.toString();
    }
    if (!tagValue.equals(""))
        result += tagValue;
    result += "</" + tagName + ">";
    return result;
 }
```

I change result's type to be a StringBuffer in order to support this refactoring:

```
    StringBuffer result = new StringBuffer("");
```

2. I identify the first information accumulation step: code that concatenates an xml open tag along with any attributes to the result variable. I Extract Method on this code as follows:

```
    result += "<" + tagName + " " + attributes + ">";
```

is extracted to:

```
    private void writeOpenTagTo(StringBuffer result) {
        result.append("<");
        result.append(name);
        result.append(attributes.toString());
        result.append(">");
    }
```

The original code now looks like this:

```
    StringBuffer result = new StringBuffer("");
    writeOpenTagTo(result);
    ...
```

3. Next, I want to continue to extract methods from toString(). I focus on the code that adds child XML nodes to the result. This code contains a recursive step (which I highlight below in bold):

```
class TagNode. . .
    public String toString(). . .
        Iterator it = children.iterator();
        while (it.hasNext()) {
            TagNode node = (TagNode)it.next();
            result += node.toString();
        }
        if (!tagValue.equals(""))
            result += tagValue;
        . . .
    }
```

Since this code makes a recursive call, it isn't so easy to extract into a method. The following code will show you why:

```
private void writeChildrenTo(StringBuffer result) {
    Iterator it = children.iterator();
    while (it.hasNext()) {
        TagNode node = (TagNode)it.next();
        node.toString(result); // can't do this because toString() doesn't take arguments.
    }
    . . .
}
```

Since toString() doesn't take a StringBuffer as an argument I can't simply extract the method. I have to find another solution and I decide to solve the problem using a helper method. This method will do the work that toString() used to do, but it will take a StringBuffer as a Collecting Parameter:

```
public String toString() {
    return toStringHelper(new StringBuffer(""));
}

private String toStringHelper(StringBuffer result) {
    writeOpenTagTo(result);
    . . .
    return result.toString();
}
```

With the new toStringHelper() method in place, I can go back to my original task: extracting the next accumulation step:

```
private String toStringHelper(StringBuffer result) {
    writeOpenTagTo(result);
    writeChildrenTo(result);
    …
    return result.toString();
}
private void writeChildrenTo(StringBuffer result) {
    Iterator it = children.iterator();
    while (it.hasNext()) {
        TagNode node = (TagNode)it.next();
        node.toStringHelper(result);  // now recursive call will work
    }
    if (!value.equals(""))
        result.append(value);
}
```

As I stare at the `writeChildrenTo()` method, I realize that it is handling two steps: adding children recursively and adding a value to a tag, when one exists. To make these two separate steps stand out, I extract the code for handling a value into its own method:

```
private void writeValueTo(StringBuffer result) {
    if (!value.equals(""))
        result.append(value);
}
```

To finish the refactoring, I extract one more method that writes an XML close tag. Here's what the final code looks like:

```
public class TagNode . . .
    public String toString() {
        return toStringHelper(new StringBuffer(""));
    }
    private String toStringHelper(StringBuffer result) {
        writeOpenTagTo(result);
        writeChildrenTo(result);
        writeValueTo(result);
        writeEndTagTo(result);
        return result.toString();
    }
    private void writeOpenTagTo(StringBuffer result) {
        result.append("<");
        result.append(name);
        result.append(attributes.toString());
        result.append(">");
    }
    private void writeChildrenTo(StringBuffer result) {
        Iterator it = children.iterator();
        while (it.hasNext()) {
            TagNode node = (TagNode)it.next();
            node.toStringHelper(result);
        }
    }
    private void writeValueTo(StringBuffer result) {
        if (!value.equals(""))
            result.append(value);
    }
    private void writeEndTagTo(StringBuffer result) {
        result.append("</");
        result.append(name);
        result.append(">");
    }
}
```

Or so I thought that was the final code. An astute reader of the above code pointed out that when the `writeChildrenTo()` method recursively calls `toStringHelper()`, it is returned a String, which it promptly ignores. In other words, the only time that the return result of

`toStringHelper()` is used is when it is called from the `toString()` method. This means that the code can be made more efficient as follows:

```
public String toString() {
    StringBuffer result = new StringBuffer("");
    toStringHelper(result);
    return result.toString();
}
public void toStringHelper(StringBuffer result) {
    writeOpenTagTo(result);
    writeChildrenTo(result);
    writeValueTo(result);
    writeEndTagTo(result);
}
```

4. I compile, run my tests and everything is good.

JUnit's Collecting Parameter

To get a better understanding of the Collecting Parameter pattern, let's have a look at another example, which comes from the unit testing framework, JUnit. In JUnit, every test is an object. Test objects get put into suites, which may be put into more suites, which results in a composite of tests. To report on how each test performs (did it pass, fail or generate errors?), some object needs to accumulate and report results as each test in the Composite is executed. TestResult is that object and it serves the role of Collecting Parameter.

[add uml and more description]

Replace One/Many Distinctions with Composite

You have separate code for handling
single elements and collections of those elements

*Combine the code to handle single
or multiple elements using Composite*

```
public class Product...
  protected Vector singleParts = new Vector();
  protected Vector collectedParts = new Vector();

  public void add(Part part) {
    singleParts.addElement(part);
  }
  public void add(PartSet set) {
    collectedParts.addElement(set);
  }
  public float getPrice() {
    float price = 0.0f;
    Enumeration e;
    for (e=singleParts.elements(); e.hasMoreElements();) {
      Part p = (Part)e.nextElement();
      price += p.getPrice();
    }
    for (e=collectedParts.elements(); e.hasMoreElements();) {
      PartSet set = (PartSet)e.nextElement();
      price += set.getPrice();
    }
    return price;
  }
```

```
public class Product...
  protected Vector parts = new Vector();

  public void add(Part p) {
    parts.addElement(p);
  }
  public float getPrice() {
    float price = 0.0f;
    for (Enumeration e=parts.elements(); e.hasMoreElements();) {
      Part p = (Part)e.nextElement();
      price += p.getPrice();
    }
    return price;
  }
```

```
public class DomainRepository...
  List repository;

  public List isSatisfiedBy(SearchCriteria criteria) {
     loop on repository
        collect all objects that meet search criteria
     return list
  }
  public List isSatisfiedBy(List searchCriteriaList) {
     for each criteria in list
        loop on repository
           collect all objects that meet search criteria
     return list
  }
```

```
public class DomainRepository...
  List repository;

  public List isSatisfiedBy(SearchCriteria criteria) {
     loop on repository
        collect all objects that meet search criteria
     return list
  }
```

Motivation

A good reason to refactor to Composite is to get rid of code that distinguishes between single objects and collections of those objects. You may find code that makes these distinctions when you have a hierarchy of objects, some of which are leaves and some of which are collections of leaves (or collections of collections). Treating both the leaf objects and the collections identically is an important goal of Composite.

[I have much more to write in this section]
.

Communication	Duplication	Simplicity
If you're performing the same behavior on single objects or collections of those objects, it is useful to communicate this in your code. But code that handles the different class types in separate bits of code doesn't communicate this message well. Make it clear by using the Composite pattern to treat your single and multiple objects uniformly.	One of the primary reasons to refactor to Composite is to remove duplication. Identify separate chunks of code that execute similar behavior on different class types and consolidate this code by treating the objects uniformly via an interface or common base class.	Having separate code to process single objects and collections of objects isn't simple – it is a symptom of code that is needlessly complex. Simplify this code by treating the class types uniformly via an interface or common base class.

Mechanics

[to do]

10 rules in validation
plus 5
only use 8 plus 2 more originals

Example

Compose Method

It isn't easy to understand your method's logic

Transform the logic into a small number of intention-revealing steps at the same level of detail

```
public boolean contains(Component c) {
   Point p = c.getLocation();
   int locX = new Double(p.getX()).intValue();
   int locY = new Double(p.getY()).intValue();
   boolean completelyWithin =
               (locX >= coords[0] &&
               locY >= coords[1] &&
               (locX+CardComponent.WIDTH) <= coords[2]) &&
               (locY+CardComponent.HEIGHT) <= coords[3];
   if (completelyWithin) return true;

   locX = locX+CardComponent.WIDTH;
   locY = locY+CardComponent.HEIGHT;
   boolean partiallyWithin =
               (locX > coords[0] &&
               locY > coords[1] &&
               (locX < coords[2]) &&
               (locY < coords[3]));

   return partiallyWithin;
}
```

```
public boolean contains(Component c) {
   return completelyWithin(c) || partiallyWithin(c) ;
}

private boolean completelyWithin(Component c) {
   Point p = c.getLocation();
   return (p.x >= coords[0] &&
            p.y >= coords[1] &&
           (p.x + CardComponent.WIDTH) <= coords[2] &&
           (p.y + CardComponent.HEIGHT) <= coords[3]);
}

private boolean partiallyWithin(Component c) {
   Point p = c.getLocation();
   return ((p.x + CardComponent.WIDTH) > coords[0] &&
             (p.y + CardComponent.HEIGHT) > coords[1] &&
             (p.x + CardComponent.WIDTH) < coords[2] &&
             (p.y + CardComponent.HEIGHT) < coords[3]);
}
```

Motivation

Kent Beck once said that some of his best patterns are those that he thought someone would laugh at him for writing. *Composed Method* [Beck] may be such a pattern. A Composed Method is a small, simple method that is easy to understand. Do you write a lot of Composed Methods? I like to think I do, but I often find that I don't, at first. So I have to go back and refactor to this pattern. When my code has many Composed Methods, it tends to be a easy to use, read and extend.

I find myself aggressively refactoring to this pattern quite often. For example, just the other day I was debugging a method in some code I've been writing with a friend. The method, called

`contains()`, wasn't very complex, but it was complex enough that I had to think about how it was doing its job. I knew this method would be easier to debug if I refactored it first. But my ego wasn't ready for that, just then: I just wanted to get rid of the bug. So, after writing an automated test to demonstrate the bug, I wrote new code in the `contains()` method to fix the bug. That code didn't fix the bug and after two more failed attempts, I was ready to refactor. It wasn't difficult to transform `contains()` into a Composed Method. But after doing so, it was so much easier to follow the logic. And moments after the refactoring, I found and fixed my bug.

Communication	Duplication	Simplicity
It may be clear *what* a method does but not *how* the method does what it does. Make the "how" easy to understand by clearly communicating every logical step. You'll often implement part of this refactoring using Extract Method [Fowler].	Duplicate code, whether blatant or subtle, clutters a method's logic. Remove the duplication to make the code smaller and simpler. Doing so often reveals further refactoring opportunities.	Composed Methods often read like English. If your method has too many lines of code, such that you can't easily explain how it does its job, simplify it by extracting logic till it is a Composed Method.

Mechanics

This is one of the most important refactorings I know of. Conceptually, it is also one of the simplest. So you'd think that this refactoring would lead to a simple set of mechanics. In fact, just the opposite is the case. While the steps themselves aren't complex, there is no simple, repeatable set of these steps. But there are guidelines for refactoring to Composed Method, some of which include:

- *Think Small* – Composed Methods are rarely more than 10 lines of code, and are usually more like 5.

- *Remove Duplication* – Reduce the amount of code in the method by getting rid of blatant and/or subtle code duplication.

- *Communicate Intention* – do so with the names of your variables and methods, and by making your code simple.

- *Simplify* – there are many ways to skin a cat. Refactor to the way that is most simple and that best communicates your intention. Simple methods may not be the most highly optimized methods. Don't worry about that. Make your code simple and optimize it later.

- *Similar Levels* – when you break up one method into chunks of behavior, make the chunks operate at similar levels. For example, if you have a piece of detailed conditional logic mixed in with some high-level method calls, you have code at different levels. Push the detail into a new or existing high-level chunk.

- *Group Related Code* – Some code is simply hard to extract into its own method. You can easily see a way to extract part of the code, but the rest remains in the original method. You now have *code at different levels*. In addition, because you have an unnatural split between related fragments of code, your code is harder to follow. In general, look for ways to *group* related code fragments, even if they aren't obvious at first.

Let's now look at three examples of refactoring to Composed Method:

Example 1

I'll start with the game example from the code sketch above. We begin with a single bulky method, called `contains()`, which figures out whether a Component is fully or partially contained within a rectangular area:

```
public boolean contains(Component c) {
   Point p = c.getLocation();
   int locX = new Double(p.getX()).intValue();
   int locY = new Double(p.getY()).intValue();
   boolean completelyWithin =
     (locX >= coords[0] &&
     locY >= coords[1] &&
     (locX+CardComponent.WIDTH) <= coords[2]) &&
     (locY+CardComponent.HEIGHT) <= coords[3];
   if (completelyWithin) return true;

   locX = locX+CardComponent.WIDTH;
   locY = locY+CardComponent.HEIGHT;
   boolean partiallyWithin =
     (locX > coords[0] &&
     locY > coords[1] &&
     (locX < coords[2]) &&
     (locY < coords[3]));

   return partiallyWithin;
}
```

Before we get into the refactoring, let's look at one of six test methods for the `contains()` method. The following method tests to see if a card is initially contained within the first player's play area, then moves the card out of the first player's play area and follows that with another test:

```
public void testCardOutOfPlayAreaOne() {
    Hand hand = (Hand)explanations.getCurrentPlayer().getHand();
    Card card = (Card)hand.elements().nextElement();
    CardComponent c = new CardComponent(card,explanations);
    PlayerArea area = explanations.getPlayerArea(0);
    explanations.moveCard(c, area.upperLeft());
    assertEquals("area contains card", true, area.contains(c));

    explanations.moveCard(c, CardComponent.WIDTH + 10, CardComponent.HEIGHT + 10);
    assertEquals("area does not contain card", false, area.contains(c));
}
```

The above test, and the other five tests, all pass (or "run green') before I begin refactoring. I run these tests after each of the small steps I am about to do below.

To begin, my first impulse is to make the `contains()` method *smaller*. That leads me to look at the conditional represented by the variable, `completelyWithin`:

```
   boolean completelyWithin =
     (locX >= coords[0] &&
     locY >= coords[1] &&
     (locX+CardComponent.WIDTH) <= coords[2]) &&
     (locY+CardComponent.HEIGHT) <= coords[3];
```

While that variable helps make it clear what the conditional logic does, the `contains()` method would be smaller and easier to read if this fragment were in it's own method. So I start with an Extract Method:

```
public boolean contains(Component c) {
    Point p = c.getLocation();
    int locX = new Double(p.getX()).intValue();
    int locY = new Double(p.getY()).intValue();
    if (completelyWithin(locX, locY)) return true;

    locX = locX+CardComponent.WIDTH;
```

```
    locY = locY+CardComponent.HEIGHT;
    boolean partiallyWithin =
            (locX > coords[0] &&
            locY > coords[1] &&
            (locX < coords[2]) &&
            (locY < coords[3]));
    return partiallyWithin;
}

private boolean completelyWithin(int locX, int locY) {
    return  (locX >= coords[0] &&
             locY >= coords[1] &&
            (locX+CardComponent.WIDTH) <= coords[2]) &&
            (locY+CardComponent.HEIGHT) <= coords[3];
}
```

Next, after seeing a similar temporary variable, called `partiallyWithin`, I do another Extract Method:

```
public boolean contains(Component c) {
    Point p = c.getLocation();
    int locX = new Double(p.getX()).intValue();
    int locY = new Double(p.getY()).intValue();
    if (completelyWithin(locX, locY)) return true;
    locX = locX+CardComponent.WIDTH;
    locY = locY+CardComponent.HEIGHT;
    return partiallyWithin(locX, locY);
}

private boolean partiallyWithin(int locX, int locY) {
    return (locX > coords[0] &&
             locY > coords[1] &&
            (locX < coords[2]) &&
            (locY < coords[3]));
}
```

The `contains()` method is now smaller and simpler, but it still seems cluttered with variable assignments. I notice that the assignments to `locX` and `locY` are performed simply for use by the new methods, `completelyWithin()` and `partiallyWithin()`. I decide to let those methods deal with the `locX` and `locY` assignments. The easiest way to do this is to just pass the `Point` variable, `p`, to each of the methods:

```
public boolean contains(Component c) {
    Point p = c.getLocation();
    if (completelyWithin(p)) return true;
    return partiallyWithin(p);
}
```

Now, the `contains()` method is really looking smaller and simpler. I feel like I'm done. But then I look at that first line of code:

```
Point p = c.getLocation();
```

The level of that code seems wrong – it is a detail, while the rest of the code in the method represents core pieces of logic. The two methods I'm calling each need the `Point` variable. But each of those methods could easily obtain the `Point` variable if I just sent them `Component c`. I consider doing that, but then I worry about violating the rule of doing things *once and only once*. For if I pass variable `c`, the `Component`, to each method, each method will have to contain code to obtain a `Point` from `c`, instead of just getting one passed in directly.

Hmmmm. What is my real goal here? Is it more important to get the levels of the code right or to say things once and only once? After some reflection, I realize that my goal is to produce a method that can be read and understood in seconds. But as it stands, that first line of code takes away from the readability and simplicity of the method. So I push down the code to obtain a `Point` into the two called methods and end up with the following:

```
public boolean contains(Component c) {
    return completelyWithin(c) || partiallyWithin(c);
}

private boolean completelyWithin(Component c) {
   Point p = c.getLocation();
   int locX = new Double(p.x).intValue();
   int locY = new Double(p.y).intValue();
   return (locX >= coords[0] &&
           locY >= coords[1] &&
          (locX + CardComponent.WIDTH) <= coords[2]) &&
          (locY + CardComponent.HEIGHT) <= coords[3];
}

private boolean partiallyWithin(Component c) {
   Point p = c.getLocation();
   int locX = new Double(p.x).intValue() + CardComponent.WIDTH;
   int locY = new Double(p.y).intValue() + CardComponent.HEIGHT;
   return (locX > coords[0] &&
           locY > coords[1] &&
          (locX < coords[2]) &&
          (locY < coords[3]));
}
```

Now I think I'm really done. But whenever you think you're really done, you're not. A reviewer of this refactoring, named Andrew Swan, observed that I was converting `p.x` and `p.y` to `ints`, when they are already `ints`! So this lead to a further simplification:

```
public boolean contains(Component c) {
   return completelyWithin(c) || partiallyWithin(c) ;
}

private boolean completelyWithin(Component c) {
   Point p = c.getLocation();
   return (p.x >= coords[0] &&
           p.y >= coords[1] &&
          (p.x + CardComponent.WIDTH) <= coords[2] &&
          (p.y + CardComponent.HEIGHT) <= coords[3]);
}

private boolean partiallyWithin(Component c) {
   Point p = c.getLocation();
   return ((p.x + CardComponent.WIDTH) > coords[0] &&
           (p.y + CardComponent.HEIGHT) > coords[1] &&
           (p.x + CardComponent.WIDTH) < coords[2] &&
           (p.y + CardComponent.HEIGHT) < coords[3]);
}
```

Example 2

```
public static Vector wrap(String s) {
   Vector wrapVector = new Vector();
   String words;
   String word;
   int lastPos;
   do {
      if (s.length() > 16) {
         words="";
         word="";
         lastPos=0;
         for (int i=0;i<16;i++) {
            if (s.charAt(i)==' ' || s.charAt(i)=='-') {
               words+=word+s.charAt(i);
               lastPos = i+1;
               word="";
            } else word+=s.charAt(i);
         }
         if (lastPos==0) {
            // Rare case that there was no space or dash, insert one and break
            words+=word+"-";
            lastPos=16;
         }
         wrapVector.addElement(words);
         s = s.substring(lastPos, s.length());
      }
   } while (s.length() > 16);
   if (s.length()>0) wrapVector.addElement(s);
   return wrapVector;
}
```

```
public static Vector wrap(StringBuffer cardText) {
   Vector wrapLines = new Vector();
   while (cardText.length() > 0)
      wrapLines.addElement(extractPhraseFrom(cardText));
   return wrapLines;
}

private static String extractPhraseFrom(StringBuffer cardText) {
   StringBuffer phrase = new StringBuffer("");
   StringBuffer word = new StringBuffer("");
   final int MAXCHARS = Math.min(MAX_LINE_WIDTH, cardText.length());
   for (int i=0; i<MAXCHARS; i++) {
      addCharacterTo(word, cardText.charAt(i));
      if (isCompleteWord(word, cardText))
         addCompleteWordTo(phrase, word);
   }
   addRemainingWordTo(phrase, word);
   removePhraseFrom(cardText, phrase);
   return phrase.toString();
}

private static boolean  addCharacterTo(StringBuffer word, char character) ...
private static boolean isCompleteWord(StringBuffer word, StringBuffer cardText) ...
private static void addCompleteWordTo(StringBuffer phrase, StringBuffer word) ...
private static void addRemainingWordTo(StringBuffer phrase, StringBuffer word) ...
private static void removePhraseFrom(StringBuffer cardText, StringBuffer phrase)...
```

In a game I've been writing with a friend, text needs to be displayed on graphical cards. The text is typically too long to fit on one line of each card, so it must be displayed on multiple lines of each card. To enable this behavior, we test-first programmed a `wrap()` method. Here are a few of the tests:

```
public void accumulateResult(String testString) {
    int i = 0;
    for (Enumeration e = CardComponent.wrap(testString).elements();e.hasMoreElements();)
        result[i++] = (String)e.nextElement();
}

public void testWrap() {
    accumulateResult("Developers Misunderstand Requirements");
    assertEquals("First line","Developers ",result[0]);
    assertEquals("Second line","Misunderstand ",result[1]);
    assertEquals("Third line","Requirements",result[2]);
}
public void testWrap2() {
    accumulateResult("Stories Are Too Complex");
    assertEquals("First line","Stories Are Too ",result[0]);
    assertEquals("Second line","Complex",result[1]);
}
public void testWrap3() {
    accumulateResult("Intention-Revealing Code");
    assertEquals("First line","Intention-",result[0]);
    assertEquals("Second line","Revealing Code",result[1]);
}
```

With these tests in place, I can work on refactoring the following bloated method:

```
public static Vector wrap(String s) {
    Vector wrapVector = new Vector();
    String words;
    String word;
    int lastPos;
    do {
        if (s.length() > 16) {
            words="";
            word="";
            lastPos=0;
            for (int i=0;i<16;i++) {
                if (s.charAt(i)==' ' || s.charAt(i)=='-') {
                    words+=word+s.charAt(i);
                    lastPos = i+1;
                    word="";
                } else word+=s.charAt(i);
            }
            if (lastPos==0) {
                // Rare case that there was no space or dash, insert one and break
                words+=word+"-";
                lastPos=16;
            }
           wrapVector.addElement(words);
           s = s.substring(lastPos, s.length());
        }
    } while (s.length() > 16);
    if (s.length()>0) wrapVector.addElement(s);
    return wrapVector;
}
```

The first thing I notice is that we have some blatant duplicate logic: the line, `s.length() > 16`, appears in a conditional statement at line 6 and at the end of the while statement. No good. I experiment with removing this duplication by using a `while` loop instead of a `do..while` loop. The tests confirm that the experiment works:

```
public static Vector wrap(String s) {
    Vector wrapVector = new Vector();
    String words;
    String word;
    int lastPos;
    while (s.length() > 16) {
        words="";
        word="";
        lastPos=0;
        for (int i=0;i<16;i++)
            if (s.charAt(i)==' ' || s.charAt(i)=='-') {
                words+=word+s.charAt(i);
```

```
                lastPos = i+1;
                word="";
            } else word+=s.charAt(i);
        if (lastPos==0) {
            // Rare case that there was no space or dash, insert one and break
            words+=word+"-";
            lastPos=16;
        }
        wrapVector.addElement(words);
        s = s.substring(lastPos, s.length());
    }
    if (s.length()>0) wrapVector.addElement(s);
    return wrapVector;
}
```

Next I notice more duplication. At two places in the middle of the method, the code says:

`word+=s.charAt(i)`.

By consolidating this logic, I see a way to simplify a conditional statement:

```
for (int i=0;i<16;i++) {
    word+=s.charAt(i);  // now we say this only once
    if (s.charAt(i)==' ' || s.charAt(i)=='-') {
        words+=word;
        lastPos = i+1;
        word="";
    } // else statement is no longer needed
}
```

Additional duplicate logic doesn't jump out at me just yet, so I continue to look (I know it is there!). I wonder about the variable, `lastPos`. What does it store? Can I figure out what the value of `lastPos` would be, without having to declare and set a variable for it? After a little bit of study, I try some experiments. Gradually it dawns on me that `words.length()` contains the exact value as that held by `lastPos`. This allows me to get rid of another variable, and all of the assignments to it:

```
public static Vector wrap(String s) {
    Vector wrapVector = new Vector();
    String words;
    String word;
    while (s.length() > 16) {
        words="";
        word="";
        for (int i=0;i<16;i++) {
            word+=s.charAt(i);
            if (s.charAt(i)==' ' || s.charAt(i)=='-') {
                words+=word;
                word="";
            }
        }
        if (words.length() == 0)  // if no space or dash, insert one
            words+=word+"-";
        wrapVector.addElement(words);
        s = s.substring(words.length(), s.length());
    }
    if (s.length()>0) wrapVector.addElement(s);
    return wrapVector;
}
```

The code is definitely getting smaller and more manageable. But the body of the while method still seems big and bulky. I decide to *Extract Method* [Fowler]:

```
public static Vector wrap(String s) {
    Vector wrapVector = new Vector();
    String words;
    while (s.length() > 16) {
        words = extractPhraseFrom(s);
```

```
        wrapVector.addElement(words);
        s = s.substring(words.length(), s.length());
    }
    if (s.length()>0) wrapVector.addElement(s);
    return wrapVector;
}

private static String extractPhraseFrom(String cardText) {
    String phrase = "";
    String word="";
    for (int i=0;i<16;i++) {
        word += cardText.charAt(i);
        if (cardText.charAt(i)==' ' || cardText.charAt(i)=='-') {
            phrase += word;
            word="";
        }
    }
    if (phrase.length() == 0) // no found space or dash, insert dash
        phrase+=word+"-";
    return phrase;
}
```

We're making progress. But I'm still not happy with the `wrap()` method: I don't like the fact that the code is adding elements to the `wrapVector` both inside and outside the while loop and I also don't like the mysterious line that changes the value of the String "s" (which is a bad name for a variable that holds on to a card's text):

```
s = s.substring(words.length(), s.length());
```

So I ask myself how I can make this logic clearer? Given some card text, I would like my code to show how the text is broken up into pieces, added to a collection and returned. I decide that the best way to achieve this objective is to push all code that is responsible for creating a "phrase" into the `extractPhraseFrom()` method. I hope to end up with a while loop that has one line of code.

My first step is to rename and change the type of the `String` variable, `s`. I call it `cardText` and change it to be `StringBuffer`, since it will be altered by the `extractPhraseFrom()` method. This change requires that I make all callers of `wrap()` pass in a StringBuffer instead of a String. As I go about doing this work, I see that I can also get rid of the temporary variable, `word`, leaving the following:

```
public static Vector wrap(StringBuffer cardText) {
    Vector wrapVector = new Vector();
    while (cardText.length() > 16) {
        wrapVector.addElement(extractPhraseFrom(cardText));
        cardText.delete(0, words.length());
    }
    if (cardText.length()>0) wrapVector.addElement(cardText.toString());
    return wrapVector;
}
```

Now I must figure out how to push the fragmented pieces of phrase-construction logic down into the `extractPhraseFrom()` method. My tests give me a lot of confidence as I go about this work. First, I go for the low-hanging fruit: the code that deletes a substring from `cardText` can easily be moved to `extractPhraseFrom()`, which yields the following:

```
public static Vector wrap(StringBuffer cardText) {
    Vector wrapVector = new Vector();
    while (cardText.length() > 16)
        wrapVector.addElement(extractPhraseFrom(cardText));
    if (cardText.length()>0) wrapVector.addElement(cardText.toString());
    return wrapVector;
}
```

Now, I've just got the line of code after the while loop to worry about:

```
if (cardText.length()>0) wrapVector.addElement(cardText.toString());
```

How can I get that code to live in the `extractPhraseFrom()` method? I study the while loop and see that I'm looping on a magic number, 16. First, I decide to make a constant for that number, called MAX_LINE_WIDTH. Then, as I continue to study the loop, I wonder why the `wrap()` method has two conditionals fragments that check `cardText.length()`, (one in the while loop and one after the while loop). I want to remove that duplication. I decide to change the `while` loop to do its thing while `cardText.length() > 0`.

This last change requires a few changes to the `extractPhraseFrom` method to make it capable of handling the case when a line of text isn't greater than 16 characters (now called `MAX_LINE_WIDTH`). Once the tests confirm that everything is working, wrap() now feels like a Composed Method, while `extractPhraseFrom()` is getting there. Here's what we have now:

```
public static Vector wrap(StringBuffer cardText) {
    Vector wrapLines = new Vector();
    while (cardText.length() > 0)
        wrapLines.addElement(extractPhraseFrom(cardText));
     return wrapLines;
}

private static String extractPhraseFrom(StringBuffer cardText) {
    String phrase = "";
    String word="";
    final int MAX_CHARS = Math.min(MAX_LINE_WIDTH, cardText.length());
    for (int i=0; i<MAX_CHARS; i++) {
        word += cardText.charAt(i);
        if (cardText.charAt(i)==' ' || cardText.charAt(i)=='-' ||
             cardText.toString().endsWith(word)) {
             phrase += word;
             word="";
        }
    }
    if (phrase.length() == 0)
        phrase=word+"-";
    cardText.delete(0, phrase.length());
    return phrase;
}
```

This code is simpler than the original, so we could stop here. But I'm not altogether happy with the `extractPhraseFrom()` method. It's not a Composed Method, so I'm drawn to continue refactoring it. What's wrong with it? Well, there's a lot of conditional logic in it, and that conditional logic doesn't communicate very well. For example, what does this mean:

```
if (cardText.charAt(i)==' ' || cardText.charAt(i)=='-' ||
    cardText.toString().endsWith(word)) {
    phrase += word;
    word="";
}
```

Since my pair and I wrote that code, I know that it means, "*if we've found a complete word, then add the word to the phrase, and blank out the word variable so we can find the next word.*" But the next reader will have to figure that out. So I'll make the intention clear, by using Extract Method (which also requires changing some variables from Strings to StringBuffers):

```
private static String extractPhraseFrom(StringBuffer cardText) {
   StringBuffer phrase = new StringBuffer("");
   StringBuffer word = new StringBuffer("");
   final int MAXCHARS = Math.min(MAX_LINE_WIDTH, cardText.length());
   for (int i=0; i<MAXCHARS; i++) {
        word.append(cardText.charAt(i));
        if (isCompleteWord(word, cardText))   // note how more intention-revealing this is
```

```
            addCompleteWordTo(phrase, word);  // same for this line
        }
        if (phrase.length() == 0)
            phrase.append(word + "-");
        cardText.delete(0, phrase.length());
        return phrase.toString();
    }

    private static boolean isCompleteWord(StringBuffer word, StringBuffer cardText) {
        return (word.charAt(word.length()-1) ==' ' ||  word.charAt(word.length()-1) =='-' ||
                cardText.toString().endsWith(word.toString()));
    }

    private static void addCompleteWordTo(StringBuffer phrase, StringBuffer word) {
        phrase.append(word);
        word.delete(0, word.length());
    }
```

We're getting closer. But I still don't like the cryptic conditional statement that comes after the `for` loop. So I apply *Extract Method* to it:

```
private static String extractPhraseFrom(StringBuffer cardText) {
    StringBuffer phrase = new StringBuffer("");
    StringBuffer word = new StringBuffer("");
    final int MAXCHARS = Math.min(MAX_LINE_WIDTH, cardText.length());
    for (int i=0; i<MAXCHARS; i++) {
        word.append(cardText.charAt(i));
        if (isCompleteWord(word, cardText))
            addCompleteWordTo(phrase, word);
    }
    addRemainingWordTo(phrase, word);  // now this code communicates intention
    cardText.delete(0, phrase.length());
    return phrase.toString();
}

private static void addRemainingWordTo(StringBuffer phrase, StringBuffer word) {
    if (phrase.length() == 0)
        phrase.append(word + "-");
}
```

The `extractPhraseFrom()` method is now 10 lines of code and reads a lot more like English. But it is still uneven! Consider these two lines of code:

```
word.append(cardText.charAt(i));

cardText.delete(0, phrase.length());
```

Both of these lines aren't complicated, but compared with the other code, which reads like English, these bits of code stick out, demanding that the reader concentrates to understand them. So I push myself to extract these 2 lines of code into 2 intention-revealing methods: `addCharacterTo()` and `removePhraseFrom()`. This yields a Composed Method:

```
private static String extractPhraseFrom(StringBuffer cardText) {
    StringBuffer phrase = new StringBuffer("");
    StringBuffer word = new StringBuffer("");
    final int MAXCHARS = Math.min(MAX_LINE_WIDTH, cardText.length());
    for (int i=0; i<MAXCHARS; i++) {
        addCharacterTo(word, cardText.charAt(i));
        if (isCompleteWord(word, cardText))
            addCompleteWordTo(phrase, word);
    }
    addRemainingWordTo(phrase, word);
    removePhraseFrom(cardText, phrase);
    return phrase.toString();
}
```

My tests run green and I'm satisfied.

Example 3

```
private void paintCard(Graphics g) {
   Image image = null;
   if (card.getType().equals("Problem")) {
      image = explanations.getGameUI().problem;
   } else if (card.getType().equals("Solution")) {
      image = explanations.getGameUI().solution;
   } else if (card.getType().equals("Value")) {
      image = explanations.getGameUI().value;
   }
   g.drawImage(image,0,0,explanations.getGameUI());

   if (highlight)
      paintCardHighlight(g);
   paintCardText(g);
}
```

```
private void paintCard(Graphics g) {
   paintCardImage(g);
   paintCardHighlight(g);
   paintCardText(g);
}
```

The above, original `paintCard()` method isn't long, nor is it complicated. It paints a card image, checks a flag to see if it must paint a card highlight, and then paints text onto the card. Painting the card highlight and card text are performed by the methods, `paintCardHighlight()` and `paintCardText()`. But the code that paints the card image lives not in a separate method but in the `paintCard()` method itself. So? Well, consider the refactored version of `paintCard()`. I can look at the refactored version and know what it does in 2 seconds, while I have to spend a few brain cycles to figure out what the previous version does. Trivial difference? *No*, not when you consider how much simpler an entire system is when it consists of *many* composed methods, like `paintCard()`.

So what was the *smell* that led to this refactoring? *Code at different levels*: raw code mixed with higher-level code. When the method contains code at the same levels, it is easier to read and understand. As the guidelines in the mechanics section say, above, Composed Methods tend to have code at the *same level.*

Implementing this refactoring was incredibly easy. I did Extract Method [Fowler] as follows:

```
private void paintCard(Graphics g) {
    paintCardImage(g);
    if (highlight)
        paintCardHighlight(g);
    paintCardText(g);
}

private void paintCardImage(Graphics g) {
   Image image = null;
   if (card.getType().equals("Problem")) {
      image = explanations.getGameUI().problem;
   } else if (card.getType().equals("Solution")) {
      image = explanations.getGameUI().solution;
   } else if (card.getType().equals("Value")) {
      image = explanations.getGameUI().value;
   }
   g.drawImage(image,0,0,explanations.getGameUI());
}
```

To finish this refactoring, I took the sole conditional statement in the method (`if (highlight)…`) and pushed it down into the `paintCardHightlight()` method. Why? I

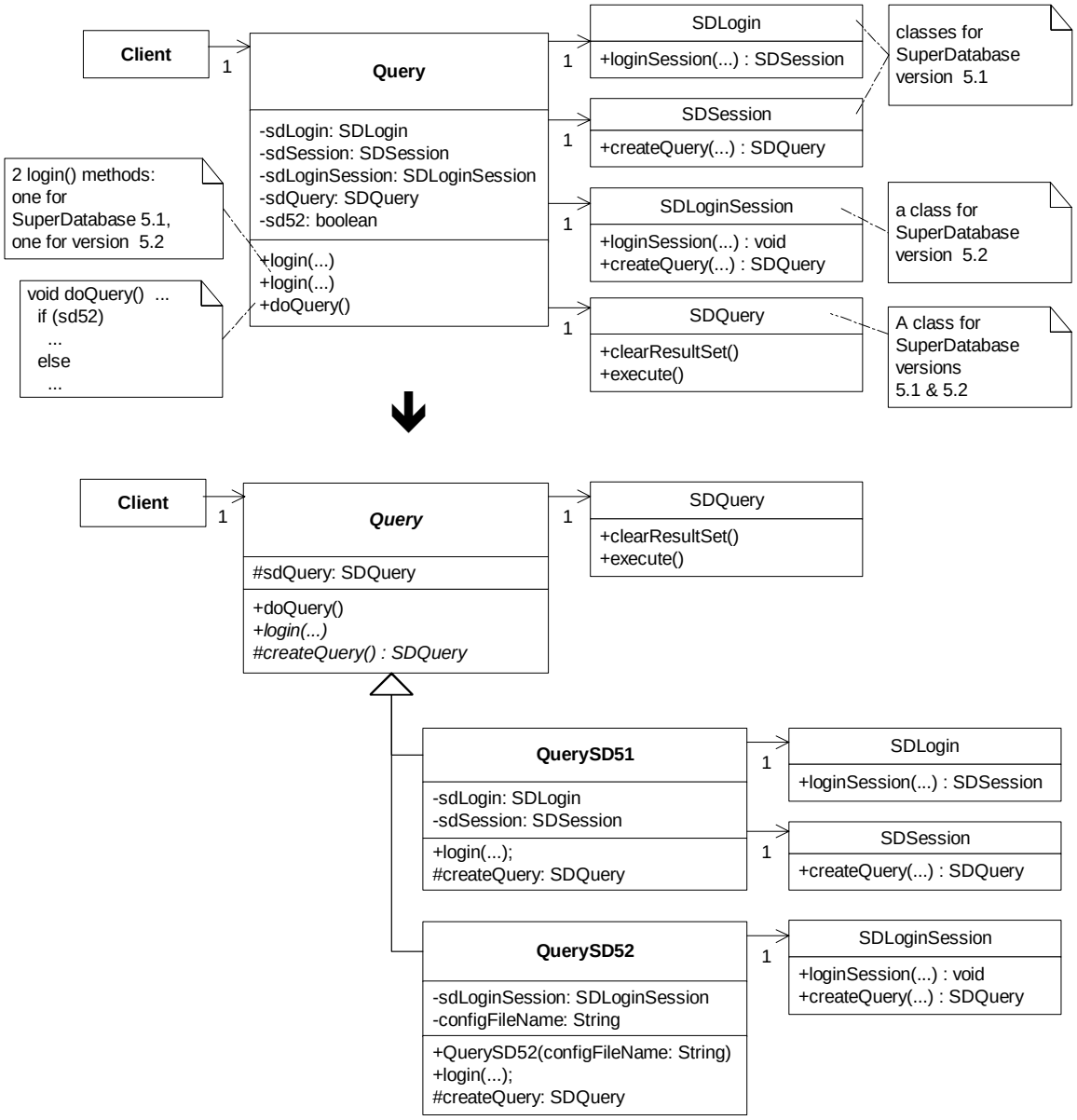
wanted the reader to simply see three steps: paint image, highlight image and paint card text. The detail of whether or not we do highlight the card isn't important to me – the reader can find that out if they look. But if that confuses other programmers, I'd be happy to see the method renamed to `paintCardHighlighIfNecessary(g)` or something similar.

```
private void paintCard(Graphics g) {
    paintCardImage(g);
    paintCardHighlight(g);
    paintCardText(g);
}
```

Separate Versions with Adapters

One class adapts multiple versions of a component, library, API or other entity

Write Adapters for each version



Motivation

While software must often support multiple versions of a component, library or API, code that handles these versions doesn't have to be a confusing mess. And yet, I routinely encounter code that attempts to handle multiple versions of something by overloading classes with version-specific state variables, constructors and methods. Accompanying such code are comments like "this is for version X – please delete this code when we move to version Y!" Sure, like that's ever gonna happen. Most programmers won't delete the version X code for fear that something they don't know about still relies on it. So the comments don't get deleted and many versions supported by the code remain in the code.

Now consider an alternative: for each version of something you need to support, create a separate class. The class name could even include the version number of what it supports, to be really explicit about what it does. We call such classes *Adapters* [GoF]. Adapters implement a common interface and are responsible for functioning correctly with one (and usually only one) version of some code. Adapters make it easy for client code to swap in support for one library or API version, or another. And programmers routinely rely on runtime information to configure their programs with the correct Adapter.

I refactor to Adapters fairly often. I like Adapters because they let me decide how I want to communicate with other people's code. In a fast-changing world, Adapters help me stay insulated from the highly useful but rapidly changing APIs, such as those springing eternally from the open-source world.

In several of the refactorings in this catalog, I assert the importance of not refactoring to a pattern too quickly in order to avoid overengineering. There must be a genuine need to refactor to a pattern, such as an overabundance of conditional logic, code bloat, duplication or unnecessary complexity. However, in the case of code that handles multiple versions of a component, library, API, etc., I often find compelling reasons to refactor to Adapters *early*, since not doing so can lead to a propagation of conditional or version-dependent logic throughout a system. So, while I'm not suggesting you adapt too early, be on guard for any complexity or propagating conditionality or maintanance issues accruing from code written to handle multiple versions of something. *Adapt* early and often so that it's easy to use or phase out various versions of code.

Communication	Duplication	Simplicity
A class that mixes together version-specific state variables, constructors and methods doesn't effectively communicate how each version is different or similar. Communicate version differences by isolating the differences in separate Adapter classes. Communicate how versions are similar by making each Adapter implement a common interface – either by subclassing an abstract class, implementing the same interface or a combination thereof.	When each version of a component, library, API, etc., isn't isolated in its own Adapter, but is instead accessed directly or through a single class, there tends to be the same repeating chunks of conditional logic that make version-specific calls to code. Such duplication bloats a class and makes the code harder to follow.	When a class is responsible for functioning correctly with several versions of some other code, it is rarely simple. Version-specfic code tends to bloat the single class and leads to conditional logic in the client code that uses it. Adapters provide a simple way to isolate versions and give clients a simple interface to every version.

Mechanics

There are different ways to go about this refactoring, depending on what your code looks like before you begin. For example, if you have a class that uses a lot of conditional logic to handle multiple versions of something, it's likely that your can create Adapters for each version by repeatedly applying *Replace Conditional with Polymorphism (255)* [Fowler]. If you have a case like that shown in the code sketch – in which a single class supports multiple versions of

something by containing version-specific variables and methods, you'll refactor to Adapter using a slightly different approach. I'll outline the mechanics for this latter scenario.

1. Identify the overburdened class (we'll call this class, "V").

2. Apply *Extract Subclass (330)* [Fowler] or *Extract Class (149)* [Fowler] for a single version of the multiple versions supported by V. Copy or move all instance variables and methods used exclusively for that version into the new class.

   To do this, you may need to make some private members of V public or protected. It may also be necessary to initialize some instance variables via a constructor in your new class, which will necessitate updates to callers of the new constructor.

3. Compile and test that your new class works as expected.

4. Repeat steps 2 –3 until there is no more version-specific code in V.

5. Remove any duplication found in the new classes, by applying refactorings like *Pull Up Method (322)* [Fowler] and *Form Template Method (345)* [Fowler].

6. Compile and test.

Example

The code we'll refactor in this example, which was depicted in the code sketch above, is based on real-world code that handles queries to a database using a third party library. To protect the innocent, I've renamed that library "SD," which stands for SuperDatabase.

1. We begin by identifying a class that is overburdened with support for multiple versions of SuperDatabase. This class, called `Query`, provides support for SuperDatabase versions 5.1 and 5.2 , which means it is already an *Adapter* to the SuperDatabase code. It just happens to be an Adapter that is *adapting* too much.

In the code listing below, notice the version-specific instance variables, duplicate `login()` methods and conditional code in `doQuery()`:

```
public class Query . . .
    private SDLogin sdLogin;                 // needed for SD version 5.1
    private SDSession sdSession;             // needed for SD version 5.1
    private SDLoginSession sdLoginSession;   // needed for SD version 5.2
    private boolean sd52;                    // tells if we're running under SD 5.2
    private SDQuery sdQuery;                 // this is needed for SD versions 5.1 & 5.2

    // this is a login for SD 5.1
    // NOTE: remove this when we convert all aplications to 5.2
    public void login(String server, String user, String password) throws QueryException {
        sd52 = false;
        try {
            sdSession = sdLogin.loginSession(server, user, password);
        } catch (SDLoginFailedException lfe) {
            throw new QueryException(QueryException.LOGIN_FAILED,
                                     "Login failure\n" + lfe, lfe);
        } catch (SDSocketInitFailedException ife) {
            throw new QueryException(QueryException.LOGIN_FAILED,
                                     "Socket fail\n" + ife, ife);
        }
    }

    // 5.2 login
    public void login(String server, String user, String password, String
    sdConfigFileName) throws QueryException {
        sd52 = true;
```

```
        sdLoginSession = new SDLoginSession(sdConfigFileName, false);
        try {
            sdLoginSession.loginSession(server, user, password);
        } catch (SDLoginFailedException lfe) {
            throw new QueryException(QueryException.LOGIN_FAILED,
                                     "Login failure\n" + lfe, lfe);
        } catch (SDSocketInitFailedException ife) {
            throw new QueryException(QueryException.LOGIN_FAILED,
                                     "Socket fail\n" + ife, ife);
        } catch (SDNotFoundException nfe) {
            throw new QueryException(QueryException.LOGIN_FAILED,
                                     "Not found exception\n" + nfe, nfe);
        }
    }

    public void doQuery() throws QueryException {
        if (sdQuery != null)
            sdQuery.clearResultSet();
        if (sd52)
            sdQuery = sdLoginSession.createQuery(SDQuery.OPEN_FOR_QUERY);
        else
            sdQuery = sdSession.createQuery(SDQuery.OPEN_FOR_QUERY);
        executeQuery();
    }
```

2. Because `Query` doesn't already have subclasses, I decide to apply *Extract Subclass (330)* [Fowler] to isolate code that handles SuperDatabase 5.1 queries. My first step is to define the subclass and create a constructor for it:

```
class QuerySD51 extends Query {
    public QuerySD51() {
        super();
    }
}
```

Next, I find all calls to the constructor of `Query` and, where appropriate, change the code to call the `QuerySD51` constructor. For example, I find the following:

```
public void loginToDatabase(String db, String user, String password)…
    query = new Query();
    try {
        if (usingSDVersion52()) {
            query.login(db, user, password, getSD52ConfigFileName());  // Login to SD 5.2
        } else {
            query.login(db, user, password); // Login to SD 5.1
        }
        …
    } catch(QueryException qe)…
```

And change this to:

```
public void loginToDatabase(String db, String user, String password)…
    try {
        if (usingSDVersion52()) {
            query = new Query();
            query.login(db, user, password, getSD52ConfigFileName()); // Login to SD 5.2
        } else {
            query = new QuerySD51();
            query.login(db, user, password);  // Login to SD 5.1
        }
        …
    } catch(QueryException qe) {
```

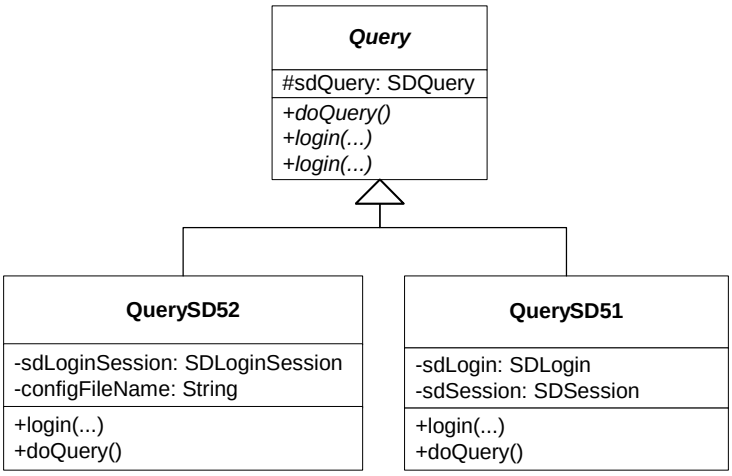
Next, I apply *Push Down Method (328)* [Fowler] and *Push Down Field (329)* [Fowler] to outfit `QuerySD51` with the methods and instance variables it needs. During this step, I have to be careful to consider the clients that are make calls to public Query methods, for if I move a public method like `login()` from `Query` to a `QuerySD51`, the caller will not be able to call the public method unless its type is changed to `QuerySD51`. Since I don't want to make such changes to

client code, I proceed cautiously, sometimes copying and modifying public methods instead of completely removing them from `Query`. While I do this, I generate duplicate code, but that doesn't bother me now - I'll get rid of the duplication in step 5.

```
class Query…
    private SDLogin sdLogin;
    private SDSession sdSession;
    protected SDQuery sdQuery;

    // this is a login for SD 5.1
    public void login(String server, String user, String password) throws QueryException {
        // I make this a do-nothing method
    }

    public void doQuery() throws QueryException {
        if (sdQuery != null)
            sdQuery.clearResultSet();
        if (sd52)
        sdQuery = sdLoginSession.createQuery(SDQuery.OPEN_FOR_QUERY);
        else
            sdQuery = sdSession.createQuery(SDQuery.OPEN_FOR_QUERY);
        executeQuery();
    }

class QuerySD51 {
    private SDLogin sdLogin;
    private SDSession sdSession;

    public void login(String server, String user, String password) throws QueryException {
        sd52 = false;
        try {
            sdSession = sdLogin.loginSession(server, user, password);
        } catch (SDLoginFailedException lfe) {
            throw new QueryException(QueryException.LOGIN_FAILED,
                                     "Login failure\n" + lfe, lfe);
        } catch (SDSocketInitFailedException ife) {
            throw new QueryException(QueryException.LOGIN_FAILED,
                                     "Socket fail\n" + ife, ife);
        }
    }

    public void doQuery() throws QueryException {
        if (sdQuery != null)
            sdQuery.clearResultSet();
        if (sd52)
            sdQuery = sdLoginSession.createQuery(SDQuery.OPEN_FOR_QUERY);
        else
        sdQuery = sdSession.createQuery(SDQuery.OPEN_FOR_QUERY);
        executeQuery();
    }
}
```

3. I compile and test that `QuerySD51` works. No problems.

4. Next, I perform steps 2 and 3 to create `QuerySD52`. Along the way, I can make the `Query` class abstract, along with the `doQuery()` method. Here's what I have now:



`Query` is now free of version-specific code, but it is not free of duplicate code.

5. I now go on a mission to remove duplication. I quickly find some in the two implementations of `doQuery()`:

```
abstract class Query…
    public abstract void doQuery() throws QueryException;

class QuerySD51…
    public void doQuery() throws QueryException {
        if (sdQuery != null)
            sdQuery.clearResultSet();

        sdQuery = sdSession.createQuery(SDQuery.OPEN_FOR_QUERY);
        executeQuery();
    }

class QuerySD52…
    public void doQuery() throws QueryException {
        if (sdQuery != null)
            sdQuery.clearResultSet();

        sdQuery = sdLoginSession.createQuery(SDQuery.OPEN_FOR_QUERY);
        executeQuery();
    }
```

Each of the above methods simply initializes the `sdQuery` instance in a different way. This means that I can apply *Introduce Polymorphic Creation with Factory Method* (36) and *Form Template Method (345)* [Fowler] to create a single superclass version of `doQuery()`:

```
public abstract class Query …
    protected abstract SDQuery createQuery();        // a Factory Method [GoF]

    public void doQuery() throws QueryException {    // a Template Method [GoF]
        if (sdQuery != null)
            sdQuery.clearResultSet();
        sdQuery = createQuery();                     // call to the Factory Method
        executeQuery();
    }

class QuerySD51…
    protected SDQuery createQuery() {
        return sdSession.createQuery(SDQuery.OPEN_FOR_QUERY);
    }

class QuerySD52…
    protected SDQuery createQuery() {
        return sdLoginSession.createQuery(SDQuery.OPEN_FOR_QUERY);
```

```
    }
```

After compiling and testing the changes, I now face a more obvious duplication problem: `Query` still declares public method for the SD 5.1 & 5.2 `login()` methods, even though they don't do anything anymore (i.e. the real login work is handled by the subclasses). The signatures for these two `login()` method are identical, except for 1 parameter:

```
// SD 5.1 login
public void login(String server, String user, String password) throws QueryException …

// SD 5.2 login
public void login(String server, String user,
                  String password, String sdConfigFileName) throws QueryException …
```

I decide to make the `login()` signatures the same, by simply supplying `QuerySD52` with the `sdConfigFileName` information via its constructor:

```
class QuerySD52 …
    private String sdConfigFileName;
    public QuerySD52(String sdConfigFileName) {
        super();
        this.sdConfigFileName = sdConfigFileName;
    }
```

Now `Query` has only one abstract `login()` method:

```
abstract class Query …
    public abstract void login(String server, String user,
                               String password) throws QueryException …
```

And client code is updated as follows:

```
public void loginToDatabase(String db, String user, String password)…
    if (usingSDVersion52())
        query = new QuerySD52(getSD52ConfigFileName());
    else
        query = new QuerySD51();

    try {
        query.login(db, user, password);
        …
    } catch(QueryException qe)…
```

I'm nearly done. Since `Query` is an abstract class, I decide to rename it `AbstractQuery`, which communicates more about its nature. But making that name change necessitates changing client code to declare variables of type `AbstractQuery` instead of `Query`. Since I don't want to do that, I apply *Extract Interface (341)* [Fowler] on `AbstractQuery` to obtain a `Query` interface that `AbstractQuery` can implement:

```
interface Query {
    public void login(String server, String user, String password) throws QueryException;
    public void doQuery() throws QueryException;
}

abstract class AbstractQuery implements Query …
    public abstract void login(String server, String user,
                               String password) throws QueryException …
```

Now, subclasses of `AbstractQuery` implement `login()`, while `AbstractQuery` doesn't even need to declare the `login()` method, since it is an abstract class.

6. I compile and test and everything works as planned. Each version of SuperDatabase is now fully *adapted*. The code is smaller and treats each version in a more uniform way, all of which makes it easier to

- see similarities and differences between the versions
- remove support for older, unused versions
- add support for newer versions

Adapting with Annonymous Inner Classes

JDK 1.0 included an interface called `Enumeration`, which was used to iterate over collections like `Vectors` or `Hashtables`. Over time, better collections classes were added to the JDK, along with a new interface, called Iterator. To make it possible to interoperate with code written using the Enumeration interface, the JDK provided the following Creation Method, which uses Java's anonymous inner class capability to adapt an Iterator with an Enumeration:

```
public class Collections…
    public static Enumeration enumeration(final Collection c) {
        return new Enumeration() {
            Iterator i = c.iterator();

            public boolean hasMoreElements() {
                return i.hasNext();
            }
            public Object nextElement() {
                return i.next();
            }
        };
    }
```

Adapting Legacy Systems

An organization has an extremely sophisticated system which brings in most of their income, but which happens to be written in about 2 million lines of COBOL, little of which was ever refactored over a decade of development. Sound familiar? Systems like this are usually hard to extend because they were never refactored. And as a result, organizations that maintain such systems can't easily add new features to them, which makes them less competitiveness, which can ultimately put them out of business.

What to do? One popular approach is to use Adapters to model new views of the legacy system. Client code talks to the Adapters, which in turn talk to the legacy code. Over time, teams rewrite entire sytstems by simply writing new implementations for each Adapter. The process goes like this:

- Identify a subsystem of your legacy system
- Write Adapters for that subsystem
- Write new client programs that rely on calls to the Adapters
- Create versions of each Adapter using newer technologies
- Test that the newer and older Adapters function identically
- Update client code to use the new Adapters
- Repeat for the next subsystem

This is an example of applying *Separate Versions with Adapter (99)*, only it is performed across an entire system or subsystem, so the mechanics are a bit different.

Adapt Interface

Your class implements an interface but only provides code for some of the interface's methods.

Move the implemented methods to an Adapter of the interface and make the Adapter accessible from a Creation Method.

```
public class CardComponent extends Container implements MouseMotionListener ...
  public CardComponent(Card card,Explanations explanations) {
     ...
     addMouseMotionListener(this);
  }
  public void mouseDragged(MouseEvent e) {
    e.consume();
    dragPos.x = e.getX();
    dragPos.y = e.getY();
    setLocation(getLocation().x+e.getX()-currPos.x,
                getLocation().y+e.getY()-currPos.y);
    repaint();
  }
  public void mouseMoved(MouseEvent e) {
  }
```

```
public class CardComponent extends Container ...
  public CardComponent(Card card,Explanations explanations) {
     ...
     addMouseMotionListener(createMouseMotionAdapter());
  }
  private MouseMotionAdapter createMouseMotionAdapter() {
    return new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        e.consume();
        dragPos.x = e.getX();
        dragPos.y = e.getY();
        setLocation(getLocation().x+e.getX()-currPos.x,
                    getLocation().y+e.getY()-currPos.y);
        repaint();
      }
    };
  }
```

Motivation

Empty methods in concrete classes bother me. I often find that they're there because a class needs to satisfy a contract by implementing an interface, but only really needs code for *some* of the interface's methods. The rest of the methods get declared, but remain empty: they were added to satisfy a compiler rule. I find that these empty methods add to the heftiness of a class's interface (i.e. it's public methods), falsely advertise behavior (I'm a class that can, among other things, do X(), Y() and Z() – only I really only provide code for X()), and forces me to do work (like declaring empty methods) that I'd rather not do.

The Adapter pattern provides a nice way to refactor this kind of code. By implementing empty methods for every method defined by an interface, the Adapter lets me subclass it to supply just the code I need. In Java, I don't even have to formally declare an Adapter subclass: I can just create an anonymous inner Adapter class and supply a reference to it from a Creation Method.

Communication	Duplication	Simplicity
Empty methods on a class don't communicate very much at all. Either someone forgot to delete the empty method, or it is just there because an interface forces you to have it there. It is far better to communicate only what you actually implement, and an Adapter can make this feasible.	If more than one of your classes partially implements an interface, you'll have numerous empty methods in your classes. You can remove this duplication by letting each of the classes work with an Adapter which handles the empty method declarations.	It is always simpler to supply less code than more. This refactoring gives you a way to cut down on the number of methods your classes declare. In addition, when used to adapt multiple interfaces, it can provide a nice way to partition methods in each of their respective adapters.

Mechanics

1. If you don't already have an adapter for the interface (which we'll call A), create a class that implements the interface and provides do-nothing behavior. Then write a Creation Method that will return a reference to an instance of your Adapter (which we'll call AdapterInstance).

2. Delete every empty method in your class that's solely there because your class implements A.

3. For those methods specified by A for which you have code, move each to your AdapterInstance.

4. Remove code declaring that your class implements A.

5. Supply the AdapterInstance to clients who need it.

Example

We'll use the example from the code sketch above. In this case we have a class called `CardComponent` that extends the JDK `Component` class and implements the JDK's `MouseMotionListener` interface. However, it only implements one of the two methods declared by the `MouseMotionListener` interface. So our task here is to replace a partially implemented interface with an Adapter.

1. The first step involves writing a Creation Method for our AdapterInstance. If we don't have an AdapterInstance, we need to create one using the refactoring, *Adapt Interface*. In this case, the JDK already supplies us with an adapter for the `MouseMotionListener` interface. It's called `MouseMotionAdapter`. So we create the following new method on the `CardComponent` class, using Java's handy anonymous inner class capability:

```
private MouseMotionAdapter createMouseMotionAdapter() {
    return new MouseMotionAdapter() {
    };
}
```

2. Next, we delete the empty method(s) that `CardComponent` declared because it implemented `MouseMotionListener`. In this case, it implemented `mouseDragged()`, but did not implement `mouseMoved()`.

```
    public void mouseMoved(MouseEvent e) {}
```

3. We're now ready to move the `mouseDragged()` method from `CardComponent` to our instance of the `MouseMotionAdapter`:

```
private MouseMotionAdapter createMouseMotionAdapter() {
    return new MouseMotionAdapter() {
        public void mouseDragged(MouseEvent e) {
            e.consume();
            dragPos.x = e.getX();
            dragPos.y = e.getY();
            setLocation(getLocation().x+e.getX()-currPos.x,
                        getLocation().y+e.getY()-currPos.y);
            repaint();
        }
    };
}
```

4. Now we can remove the `implements MouseMotionListener` from `CardComponent`.

```
public class CardComponent extends Container implements MouseMotionListener {
```

5. Finally, we must supply the new adapter instance to clients that need it. In this case, we must look at the constructor. It has code that looks like this:

```
public CardComponent() {
   …
   addMouseMotionListener(this);
}
```

This needs to be changed to call our new, private, Creation Method:

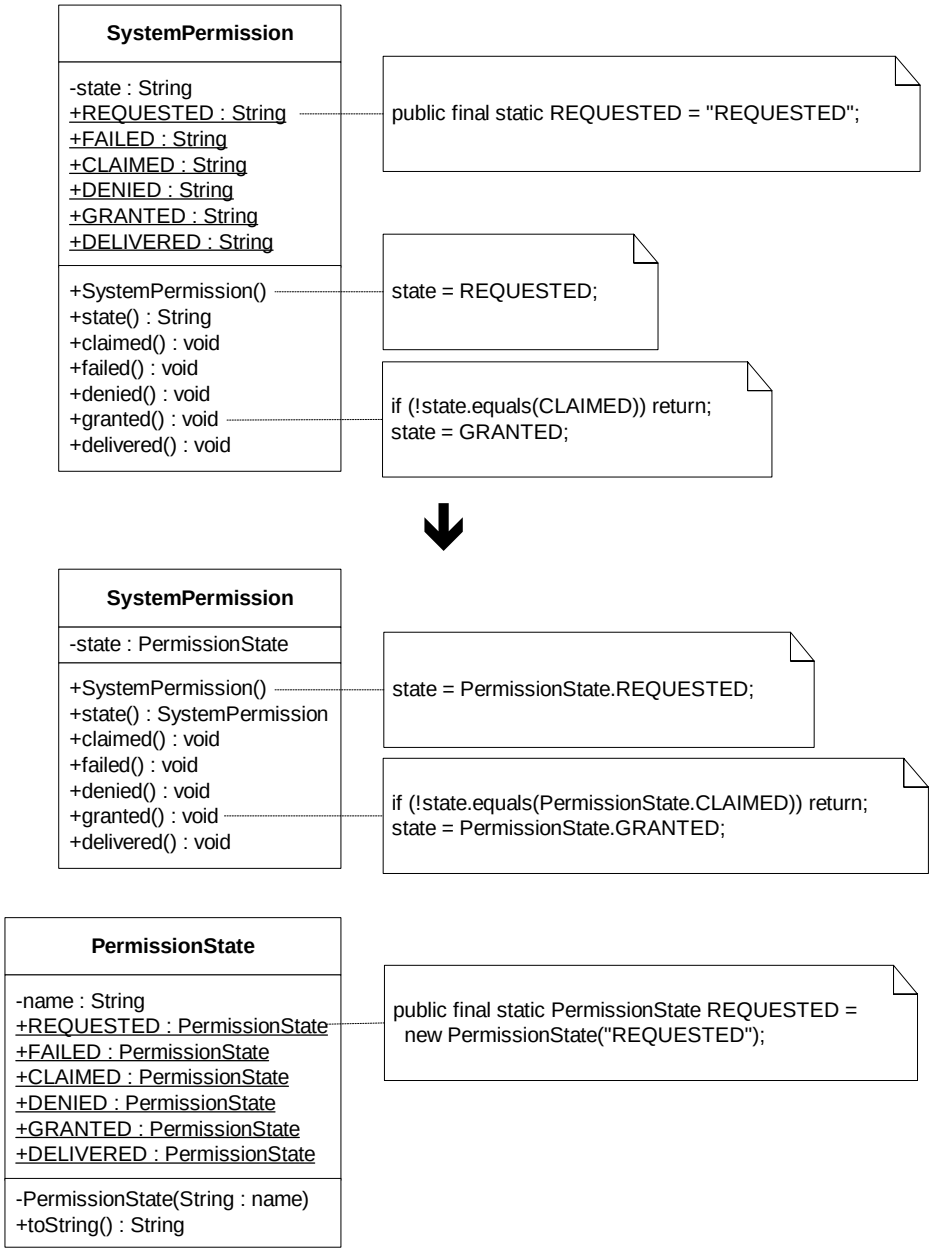
```
public CardComponent() {
   …
   addMouseMotionListener(createMouseMotionAdapter());
}
```

Now we test. Unfortunately, since this is mouse related code, I don't have automated unit tests. So I resort to some simple manual testing and confirm that everything is ok.

Replace Type with Type-Safe Enum

A field's language-defined type (e.g. String, int, etc.) fails to protect it from unsafe assignments and invalid equality comparisons

Constrain the assignments and equality comparisons by making the field type-safe



Motivation

A Type-Safe Enum bundles together a user-defined type with a set of constant instances of that type. A primary motivation for refactoring to Type-Safe Enum is to constrain the possible values that may be assigned to or equated with a variable.

To understand the value of this pattern, it helps to study code that isn't type-safe. Consider the following test case:

```
public void testPermissionRequest() {
    SystemPermission permission = new SystemPermission();
    assertEquals("permission state", permission.REQUESTED, permission.state());
    assertEquals("permission state", "REQUESTED", permission.state());
}
```

The first line of code creates a `SystemPermission` object. The constructor for this object sets its `state` instance variable equal to the `SystemPermission.REQUESTED` state:

```
public SystemPermission() {
   state = REQUESTED;
}
```

Other methods within `SystemPermission` assign `state` to system permission states such as `GRANTED` and `DENIED`. Now, given that each of these state types was defined using `String` constants (like `public final static String REQUESTED = "REQUESTED"`), and `state` was defined as type `String`, then the two tests above would both evaluate to true since `state` - accessible via `permission.state()` - would be considered equal to `SystemPermission.REQUESTED` and the `String`, "`REQUESTED`."

What's the problem with that? Glad you asked. The `String`, "`REQUESTED`" represents one object reference while the constant `String`, `SystemPermission.REQUESTED`, represents a different object reference, and yet the instance variable, `state`, is considered equal to both of them? That's not good, for just after a `SystemPermission` is instantiated, we want its `state` to be equal to the object reference, `SystemPermission.REQUESTED`, and no other object reference. A Type-Safe Enum can easily accomplish this.

Another motivation for refactoring to a Type-Safe Enum occurs when callers can change the value of an instance variable to an invalid value. For example, consider this code:

```
public class SystemPermission...
    public void setState(String newState){
        state = newState;
    }

permission.setState("thinking"); // "thinking" is not a valid SystemPermission state
```

If one didn't use a Type-Safe Enum to prevent such spurious assignments, you'd have to fill your classes with lots of unnecessary validation logic.

Communication	Duplication	Simplicity
It is useful to communicate the availability of a type and constant values of that type. A Type-Safe Enum does this well because it is a class that exists solely to define the type and constants.	Duplication isn't an issue with respect to this refactoring.	A family of constants defined using a language-based type is slightly simpler to declare than a family of Type-Safe Enums, but because Type-Safe Enums prevent spurious assignments from occurring, they often help us simplify code.

Mechanics

1. Identify a type-unsafe instance variable – i.e. a variable declared as a language-defined type, which is assigned to or compared against a family of constant values. Identify any getting/setting methods associated with this variable.

2. Rename the variable and any associated getting/setting methods, taking care to update all callers to the getting/setting methods.

   The type-unsafe variable often already has the name you want, so a quick rename now will later allow you to define your type-safe variable with the name you want.

3. Compile and test

4. Declare a new class to store the family of constant values, naming the class after the kinds of types it will store. This will be your Type-Safe Enum.

5. Choose one constant value that the type-unsafe instance variable is assigned to and/or compared against and define a new version of this constant in your Type-Safe Enum class by creating a public final static constant that is an instance of the Type-Safe Enum class.

6. In the class that declared the type-unsafe instance variable, create a type-safe version of it by declaring an instance variable whose type is the Type-Safe Enum class. Create any necessary getting/setting methods for this instance variable, mirroring the getting/setting methods declared for the type-unsafe instance variable.

7. Wherever the type-unsafe instance variable is assigned to the constant value choosen for step 5, *add* code to assign the type-safe instance variable equal to the type-safe enum constant created during step 5.

8. Wherever the type-unsafe instance variable is compared against the constant value choosen in step 5, *change* the code to compare it against the type-safe enum constant, created in step 5.

9. Compile and test.

10. Repeat steps 5, 7, 8 and 9 for every constant in the family of constant values.

11. Delete the type-unsafe instance variable, any getting/setting methods associated with it, any direct assignments to it and all of the type-unsafe constants.

12. Compile and test.

Example

This example, which was shown in the code sketch and mentioned in the Motivation section, deals with handling permission requests to access software systems. We'll begin by looking at relevant parts of the class, `SystemPermission`:

```
public class SystemPermission {
    private String state;
    private boolean granted;
    private boolean failed;

    public final static String REQUESTED = "REQUESTED";
```

```
    public final static String FAILED = "FAILED";
    public final static String CLAIMED = "CLAIMED";
    public final static String DENIED = "DENIED";
    public final static String GRANTED = "GRANTED";
    public final static String DELIVERED = "DELIVERED";

    public SystemPermission() {
        state = REQUESTED;
        failed = false;
        granted = false;
    }

    public boolean isGranted() {
        return granted;
    }

    public boolean hasFailed() {
        return failed;
    }

    public String state() {
        return state;
    }

    public void claimed() {
        if (state.equals(REQUESTED))
            state = CLAIMED;
    }

    public void failed() {
        if (!state.equals(REQUESTED)) return;
        state = FAILED;
        failed = true;
    }

    public void denied() {
        if (state.equals(CLAIMED))
            state = DENIED;
    }

    public void granted() {
        if (!state.equals(CLAIMED)) return;
        state = GRANTED;
        granted = true;
    }

    public void delivered() {
        if (state.equals(GRANTED) || state.equals(DENIED) )
            state = DELIVERED;
    }
}
```

1. The instance variable we're interested in here is called `state`, since it can be assigned to or compared against a family of `String` constants also defined inside `SystemPermission`. Our goal is to make `state` type-safe.

2. The first step is to rename `state` and its associated getting/setting methods. I'll rename it to `old_state`, and, since `state` only has a getting method and no setting method, I'll create a method called `old_state()` and update client code to use it:

```
public class SystemPermission...
  private String old_state;

  public SystemPermission() {
    old_state = REQUESTED;
    …
  }

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

```
  // etc.
```

And here is some client code that I update:

```
public class SystemPermissionTest extends TestCase...
  public void testPermissionRequest() {
    assertEquals("request", SystemPermission.REQUESTED, permission.old_state());
  }
```

Note: It is best to use an automated refactoring tool to handle the renaming of the variable and method(s).

3. I compile and test to make sure the name changes didn't break anything.

4. Now I create a class called `PermissionState`, which will be my Type-Safe Enum class:

```
public final class PermissionState {
}
```

I make it final because it will not need to be subclassed.

5. I now choose one constant value that the type-unsafe instance variable is assigned to or compared against, and I create a version of this constant in `PermissionState`, making it a public constant `PermissionState` member variable and instance of `PermissionState`:

```
public final class PermissionState {
  public final static PermissionState REQUESTED = new PermissionState();
}
```

This new type-safe constant will be easier to work with if I can query its `toString()` method to see which `PermissionState` type it is. So I make the following change:

```
public final class PermissionState {
  private String name;

  private PermissionState(String name) {
    this.name = name;
  }

  public final static PermissionState REQUESTED = new PermissionState("REQUESTED");

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

6. I create a new type-safe instance variable inside `SystemPermission`, using the type, `PermissionState`. Since `old_state` only had a getting method and not a setting method, I only need to create a getting method for `state`:

```
public class SystemPermission…
  private PermissionState state;

  public PermissionState state() {
    return state;
  }
```

7. Wherever I find code that assigns `old_state` to `SystemPermission.REQUESTED`, I must *add* code to assign `state` to `PermissionState.REQUESTED`:

```
public class SystemPermission...
  public SystemPermission() {
    old_state = REQUESTED;
```

```
    state = PermissionState.REQUESTED;
    failed = false;
    granted = false;
  }
```

Note: I'll delete the `old_state` assignment code later, when doing so won't cause logic problems with code that expects it to have a certain value.

8. Wherever `old_state` is compared against `SystemPermission.REQUESTED`, I must *replace* this code to compare `state` against `PermissionState.REQUESTED`:

Here is some test code that needs updating:

```
public class SystemPermissionTest extends TestCase...
  private SystemPermission permission;

  public void setUp() {
    permission = new SystemPermission();
  }
  public void testPermissionRequest() {
    assertEquals("request", SystemPermission.REQUESTED, permission.old_state());
  }
```

The `testPermissionRequest` method becomes:

```
  public void testPermissionRequest() {
    assertEquals("request", PermissionState.REQUESTED, permission.state());
  }
```

The following code in `SystemPermission` also needs updating:

```
public class SystemPermission...
  public void claimed() {
    if (old_state.equals(REQUESTED))
      old_state = CLAIMED;
  }

  public void failed() {
    if (!old_state.equals(REQUESTED)) return;
    old_state = FAILED;
    failed = true;
  }
```

I change this to:

```
public class SystemPermission...
  public void claimed() {
    if (state.equals(PermissionState.REQUESTED))
      old_state = CLAIMED;
  }

  public void failed() {
    if (!state.equals(PermissionState.REQUESTED)) return;
    old_state = FAILED;
    failed = true;
  }
```

9. Now I compile and test to see that everything is still working smoothly.

10. Next, I repeat steps 5, 7, 8 and 9 for every constant in the family of constant values. I'll spare you the details.

11. Finally, I have the pleasure of deleting `old_state`, the getting method, `old_state()`, all assignments made to `old_state`, and the entire family of `SystemPermission` type-unsafe constants. Here are a few of the deletions:

```
public class SystemPermission...
  private String old_state;

  public final static String REQUESTED = "REQUESTED";
  public final static String FAILED = "FAILED";
  public final static String CLAIMED = "CLAIMED";
  public final static String DENIED = "DENIED";
  public final static String GRANTED = "GRANTED";
  public final static String DELIVERED = "DELIVERED";

  public SystemPermission() {
    old_state = REQUESTED;
    state = PermissionState.REQUESTED;
    ...
  }

  public String old_state() {
    return old_state;
  }

  public void claimed() {
    if (state.equals(PermissionState.REQUESTED)) {
      old_state = CLAIMED;
      state = PermissionState.CLAIMED;
    }
  }

  // and so on...
```

12. I compile and test after all of the deletions. Now the instance variable, `state`, is type-safe:

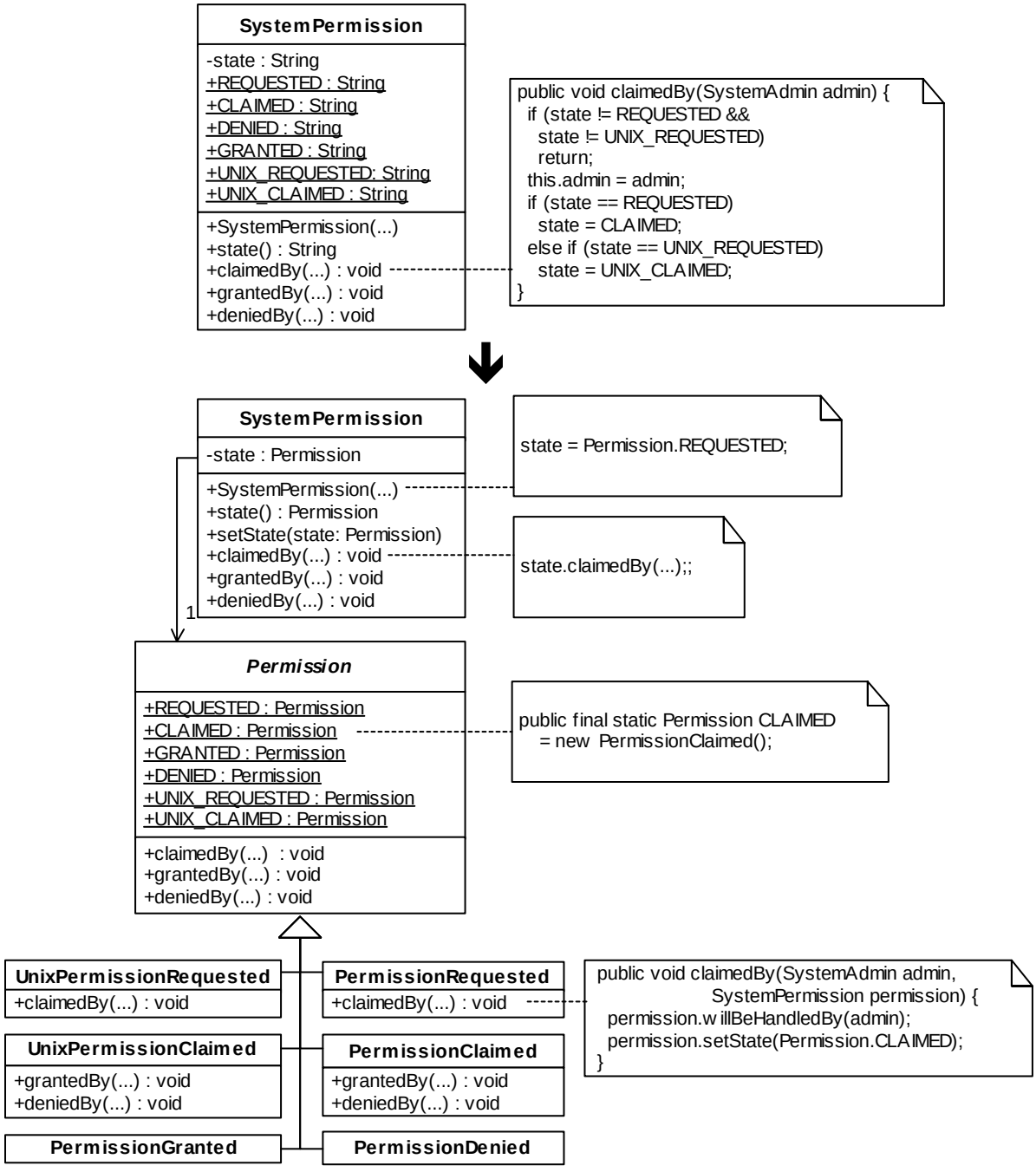
```
public class SystemPermission {
  private PermissionState state;
  private boolean granted;
  private boolean failed;

  public SystemPermission() {
    state = PermissionState.REQUESTED;
    failed = false;
    granted = false;
  }

  public boolean isGranted() {
    return granted;
  }

  public boolean hasFailed() {
    return failed;
  }

  public PermissionState state() {
    return state;
  }

  public void claimed() {
    if (state.equals(PermissionState.REQUESTED))
      state = PermissionState.CLAIMED;
  }

  public void failed() {
    if (!state.equals(PermissionState.REQUESTED)) return;
    state = PermissionState.FAILED;
    failed = true;
  }

  public void denied() {
    if (state.equals(PermissionState.CLAIMED))
      state = PermissionState.DENIED;
```

```
  }

  public void granted() {
    if (!state.equals(PermissionState.CLAIMED)) return;
    state = PermissionState.GRANTED;
    granted = true;
  }

  public void delivered() {
    if (state.equals(PermissionState.GRANTED) ||
      state.equals(PermissionState.DENIED))
      state = PermissionState.DELIVERED;
  }
}
```

Replace State-Altering Conditionals with State

Complex conditional expressions control an object's state transitions

Replace the conditionals with State classes that handle specific states and transitions between them



Motivation

The primary reason for refactoring to the State pattern is to tame overly-complex state-altering conditional logic. Such logic, which tends to spread itself thoughout a class, controls an object's state, including how states transition to other states. When you implement this pattern you create state classes that represent specific states of an object and the transitions between those states. The object that has its state changed is known as the *context*. A context delegates state-changing behavior to a state object. State objects make state transitions at runtime by making the context point to a different state object.

If you don't know the State pattern very well, you'll understand this refactoring better if you study the State pattern in *Design Patterns* [GoF]. If you do know this pattern, you might be using it when you don't need to be: i.e. when simple state-altering conditional logic would do. This refactoring is concerned with the edge – the place where state-altering conditional logic is no longer easy to follow or extend and when the State pattern can really make a difference.

Before I ever refactor to State, I always see if I can implement a simpler solution by applying low-level refactorings, like *Extract Method (110)* [Fowler]. If those refactorings still don't tame the conditional logic, I know I'm ready for State. The State pattern has a way of reducing or removing many lines of conditional logic, yielding clean, simple and extensible code.

If your state objects have no instance variables, context objects can share instances of them. Sharing state instances is often achieved via the Singleton or Flyweight patterns. If you need to easily write and configure mock objects for specific states, be careful that your Singleton or Flyweight code doesn't make working with mocks too difficult. If you don't need mock objects for state instances and your state instances are stateless, context objects can share the instances by getting access to them via Creation Methods on their superclass (see *Encapsulate Classes with Creation Methods (21)*).

This refactoring is different from *Replace Type Code with State/Strategy (227)* [Fowler] in a few areas. First, I don't have a single refactoring for the State and Strategy patterns because I view them as different patterns, I refactor to them for different reasons (see *Replace Conditional Calculations with Strategy (44)*) and the mechanics of the refactorings to each pattern differ. Second, Martin deliberately doesn't document a full refactoring to the State pattern, since the complete implementation depends on a further refactoring he wrote, called *Replace Conditional with Polymorphism (225)* [Fowler]. While I respect that decision, I thought it would be more helpful to readers to understand how the refactoring works from end to end, so my mechanics and example sections delineate all of the steps to get you from conditional state-changing logic to a State pattern implementation.

Communication	Duplication	Simplicity
Many lines of state-altering conditional logic don't communicate intent very well. Communicate this logic clearly by splitting out the state transition logic into classes that know how to handle their state transitions.	When you have a lot of state-altering conditional logic, you tend to see the same conditional phrases repeated throughout the methods of a class. Implementing the State pattern will allow you to remove much of this conditional logic.	One of the main reasons to perform this refactoring is to simplify complex state-changing logic. If you can't easily follow the state-changing logic in your class, it may be a good time to refactor to the State pattern.

Mechanics

1. A class (which we'll call the *context class*) contains a field (which we'll call the *original state field*) that gets assigned to or compared against a family of constants during state transitions. Rename this field and any associated getting/setting methods, taking care to update all callers to the getting/setting methods.

   Compile and test.

2. Declare a new abstract class and name it based on the name or general purpose of the *original state field*.

3. Declare subclasses of the abstract class, one for each of the states the context class may enter.

   - If you have 5 constant values that represent states, you'll create 5 subclasses.
   - Your subclasses won't have any methods in them to start – you'll add methods later.
   - If clients will interact with your state subclasses solely through the interface of their superclass, it's a good idea to make every subclass constructor non-public.

4. Create a non-public field (which we'll call the *state field*) in the context class, making its type that of the abstract class (from step 3). Create any necessary getting/setting methods for this field, mirroring the getting/setting methods on the field choosen in step 1.

5. Identify a state the context class can enter. For each context class method that transitions this state to one or more other states, declare a similar method on the abstract class (from step 3) and on the subclass that corresponds with this state.

   - It's best to start with the state the context class enters after being instantiated.

6. Implement the method(s) on the subclass, making whatever changes are necessary for each method to perform the state transition logic currently residing in the context class.

   - You may decide to pass a context class reference to the method(s) so the subclass code can call back on the context class.
   - At this point, you aren't replacing the state transition logic in the context class.

7. Wherever the *original state field* is assigned to the constant value for the state, add code to set the *state field* equal to an instance of the subclass you just worked with.

   - If this subclass is stateless, you can make a public static final version of it available via the abstract class or a creation class.

8. Wherever the *original state field* is compared against the constant value for the selected state, change the code to compare the *state field* against the subclass instance.

   Compile and test.

9. Repeat steps 5 through 8 for each context class state.

10. For every method in the context class that can change state, replace all of the code with a single delegation call to the state field.

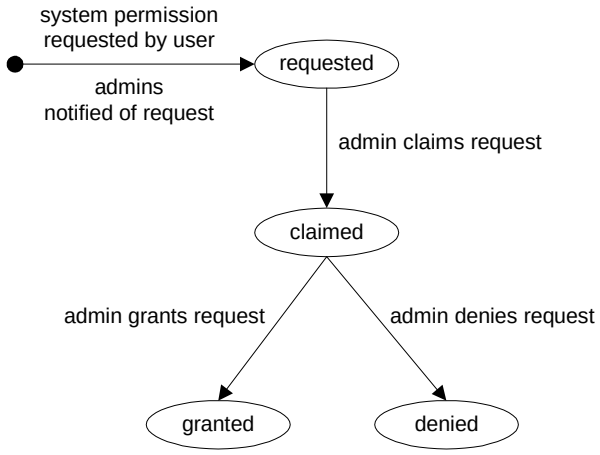
    Compile and test.

11. Delete the *original state field*, any getting/setting methods and constants associated with it and any direct assignments to it. This step involves deleting code in the state subclasses.

    Compile and test.

Example

To understand when it makes sense to refactor to the State pattern, it helps to study a class that manages its state without requiring the sophistication of the State pattern. `SystemPermission` is such a class. It uses simple conditional logic to keep track of the state of a permission request to access a software system. Over the lifetime of a `SystemPermission` object an instance variable named `state` transitions between the states *requested*, *claimed*, *denied* and *granted*. Here is a UML representation of the possible transitions:



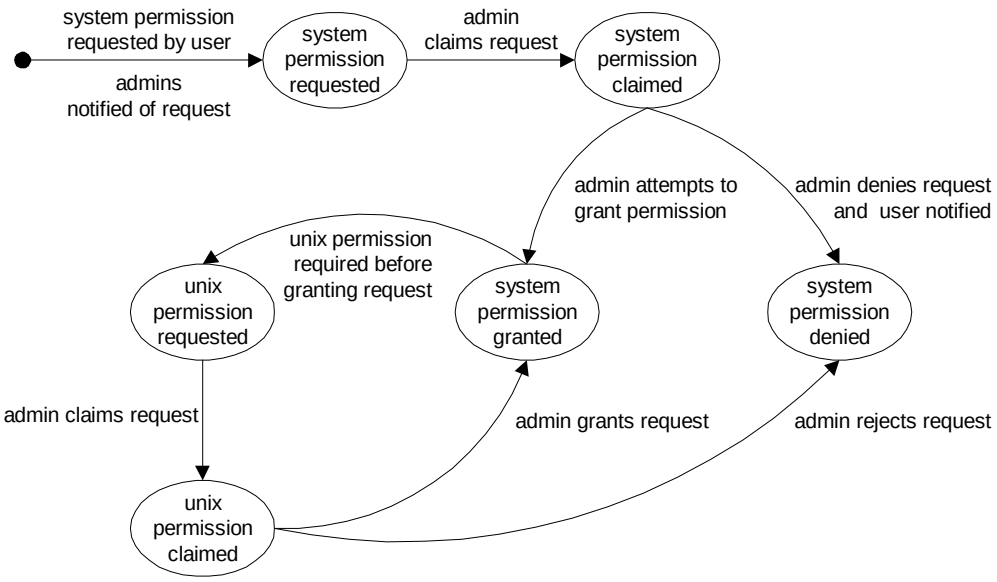
Below is the code for `SystemPermission` and a fragment of test code to show how the class gets used:

```
public class SystemPermission {
  private SystemProfile profile;
  private SystemUser requestor;
  private SystemAdmin admin;
  private boolean isGranted;
  private String state;

  public final static String REQUESTED = "REQUESTED";
  public final static String CLAIMED = "CLAIMED";
  public final static String GRANTED = "GRANTED";
  public final static String DENIED = "DENIED";

  public SystemPermission(SystemUser requestor, SystemProfile profile) {
    this.requestor = requestor;
    this.profile = profile;
    state = REQUESTED;
    isGranted = false;
    notifyAdminOfPermissionRequest();
  }

  public String state() {
    return state;
  }

  public void claimedBy(SystemAdmin admin) {
```

```
    if (state != REQUESTED)
      return;
    this.admin = admin;
    state = CLAIMED;
  }

  public void deniedBy(SystemAdmin admin) {
    if (state != CLAIMED)
      return;
    if (this.admin != admin) return;
    isGranted = false;
    state = DENIED;
    notifyUserOfPermissionRequestResult();
  }

  public void grantedBy(SystemAdmin admin) {
    if (state != CLAIMED)
      return;
    if (this.admin != admin) return;
    state = GRANTED;
    isGranted = true;
    notifyUserOfPermissionRequestResult();
  }

  public boolean isGranted() {
    return isGranted;
  }

  public void notifyAdminOfPermissionRequest() {
    // ...
  }
  public void notifyUserOfPermissionRequestResult() {
    // ...
  }
}

public class TestStates extends TestCase ...
  public void testGrantedBy() {
    permission.grantedBy(admin);
    assertEquals("requested", permission.REQUESTED, permission.state());
    assertEquals("not granted", false, permission.isGranted());
    permission.claimedBy(admin);
    permission.grantedBy(admin);
    assertEquals("granted", permission.GRANTED, permission.state());
    assertEquals("granted", true, permission.isGranted());
  }
```

Notice how the instance variable, `state`, gets assigned to different values as clients call specific `SystemPermission` methods. Now look at the overall conditional logic in `SystemPermission`. This logic is responsible for transitioning between states, but the logic isn't very complicated so the code doesn't require the sophistication of the State pattern.

This conditional state changing logic can quickly become hard to follow as more real-world behavior gets added to the `SystemPermission` class. For example, a client told us about their security system in which users needed to obtain unix and/or database permissions before the user could be granted general permission to access a given software system. The state transition logic for a system that requires unix permission before general permission may be granted looks like this:



Adding support for unix permission makes `SystemPermission`'s state-altering conditional logic a lot more complicated than it used to be. Consider the following:

```
public class SystemPermission...
  public void claimedBy(SystemAdmin admin) {
    if (state != REQUESTED &&
      state != UNIX_REQUESTED)
      return;
    this.admin = admin;
    if (state == REQUESTED)
      state = CLAIMED;
    else if (state == UNIX_REQUESTED)
      state = UNIX_CLAIMED;
  }

  public void deniedBy(SystemAdmin admin) {
    if (state != CLAIMED &&
      state != UNIX_CLAIMED) return;
    if (this.admin != admin) return;
    isGranted = false;
    isUnixPermissionGranted = false;
    state = DENIED;
    notifyUserOfPermissionRequestResult();
  }

  public void grantedBy(SystemAdmin admin) {
    if (state != CLAIMED &&
      state != UNIX_CLAIMED) return;
    if (this.admin != admin) return;

    if (profile.isUnixPermissionRequired() &&
      state == UNIX_CLAIMED)
      isUnixPermissionGranted = true;
    else if (profile.isUnixPermissionRequired() &&
      !isUnixPermissionGranted()) {
      state = UNIX_REQUESTED;
      notifyUnixAdminsOfPermissionRequest();
      return;
    }
    state = GRANTED;
    isGranted = true;
    notifyUserOfPermissionRequestResult();
  }
```

An attempt can be made to simplify the above code by applying *Extract Method (110)* [Fowler]. For example, one could refactor the `grantedBy()` method like so:

```
  public void grantedBy(SystemAdmin admin) {
    if (!isInClaimedState()) return;
    if (this.admin != admin) return;
    if (isUnixPermissionRequestedAndClaimed())
      isUnixPermissionGranted = true;
    else if (isUnixPermisionDesiredButNotRequested()) {
      state = PermissionState.UNIX_REQUESTED;
      notifyUnixAdminsOfPermissionRequest();
      return;
    }
    …
```

That's a little better but now the `SystemPermission` class has lots of state-specific boolean logic (i.e. methods liks is`UnixPermissionRequestedAndClaimed()`) and yet `grantedBy()` still isn't simple. It's time to simplify things by refactoring to the State pattern.

1. `SystemPermission` has a field called `state` and a corresponding accessor method called `state()`. I rename these to `old_state` because I want to use the name *state* for the State pattern implementation.

```
public class SystemPermission...
  private String old_state;

  public SystemPermission(SystemUser requestor, SystemProfile profile) {
    this.requestor = requestor;
    this.profile = profile;
    old_state = REQUESTED;
    isGranted = false;
    isUnixPermissionGranted = false;
    notifyAdminOfPermissionRequest();
  }

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

I make sure all client code is updated, compile and test that the name changes work.

2. Now I create a new abstract class that will serve as a base class for all of the states that a `SystemPermission` can enter. "Permission" sounds like a good name for this class:

```
public abstract class Permission {
}
```

3. Next, it's time to create `Permission` subclasses for each of the states that a `SystemPermission` can enter.

```
public class PermissionRequested extends Permission {}
public class PermissionClaimed extends Permission {}
public class PermissionGranted extends Permission {}
public class PermissionDenied extends Permission {}
public class UnixPermissionClaimed extends Permission {}
public class UnixPermissionDenied extends Permission {}
```

Since the `SystemPermission` class will interact with each of these state subclasses via the interface of their superclass, I make each of their constructors protected.

4. Now I create a private `Permission` field in `SystemPermission` along with a getting method for it. I don't create a setting method for it (yet), because I'm simply mirroring what the field, `old_state`, had (i.e. a getting method and no setting method)

```
public class SystemPermission...
  private Permission state;
```

```
  public Permission state() {
    return state;
  }
```

5. Now comes the fun part. I identify the first state that a `SystemPermission` object can enter: the REQUESTED state. I study which `SystemPermission` method(s) can transition the REQUESTED state to some other state and find that `claimedBy(…)` is the only method that does so – it allows the transition from REQUESTED to CLAIMED. This leads me to declare a `claimedBy(…)` method on the `Permission` and `PermissionRequested` classes:

```
public abstract class Permission...
  public void claimedBy(SystemAdmin admin) {}

public class PermissionRequested extends Permission...
  public void claimedBy(SystemAdmin admin) {}
```

6. I can now implement the `PermissionRequested.claimedBy(…)` method. I start by studying the `SystemPermission.claimedBy(…)` method:

```
public class SystemPermission...
  public void claimedBy(SystemAdmin admin) {
    if (old_state != REQUESTED && old_state != UNIX_REQUESTED)
      return;
    this.admin = admin;
    if (old_state == REQUESTED)
      old_state = CLAIMED;
    else if (old_state == UNIX_REQUESTED)
      old_state = UNIX_CLAIMED;
  }
```

This method is weighted down with logic, much of which isn't important to my present task of writing code to handle the transition from the REQUESTED state to the CLAIMED state. The guard clause at the start of the method won't be necessary in my State-pattern implementation, and the conditional logic to check if `old_state` is equal to REQUESTED also isn't important, since I know I'll be in the PermissionRequested state when the claimedBy(…) method is called. Finally, I don't care at all about any logic relating to UNIX states. So, ignoring most of the logic in this method, I wrote the following code:

```
public abstract class Permission...
  public void claimedBy(SystemAdmin admin, SystemPermission permission)

public class PermissionRequested extends Permission...
  public void claimedBy(SystemAdmin admin, SystemPermission permission) {
    permission.willBeHandledBy(admin);
    permission.setOldState(permission.CLAIMED);
  }


public class SystemPermission...
  public void willBeHandledBy(SystemAdmin admin) {
    this.admin = admin;
  }
  public void setOldState(String state) {
    this.old_state = state;
  }
```

7. Now I find all places in `SystemPermission` where `old_state` is assigned to the REQUESTED constant and I *add* code to assign `state` equal to a `PermissionRequested` instance:

```
public class SystemPermission...
  public SystemPermission(SystemUser requestor, SystemProfile profile) {
    this.requestor = requestor;
    this.profile = profile;
    old_state = REQUESTED;
    state = Permission.REQUESTED;
    isGranted = false;
    isUnixPermissionGranted = false;
    notifyAdminOfPermissionRequest();
  }

public abstract class Permission...
  public final static Permission REQUESTED = new PermissionRequested();
```

8. Next, I find all places in `SystemPermission` where `old_state` is compared to the REQUESTED constant and I *change* the code to compare `state` with the `PermissionRequested` instance:

Here's some test code that needs updating:

```
public class TestStates extends TestCase...
  private SystemUser user = new SystemUser("Doe", "John");
  private SystemAdmin admin = new SystemAdmin("Joe", "Brontesaurus");
  private SystemProfile profile = new SystemProfile("Employee Benefits");
  private SystemPermission permission;

  public TestStates(String name) {
    super(name);
  }

  public void setUp() {
    permission = new SystemPermission(user, profile);
  }

  public void testRequestedBy() {
    assertEquals("requested", permission.REQUESTED, permission.old_state());
  }
```

The `testRequestedBy()` method becomes:

```
  public void testPermissionRequest() {
    assertEquals("request", Permission.REQUESTED, permission.state());
  }
```

The following `SystemPermission` code also requires updating:

```
public class SystemPermission...
  public void claimedBy(SystemAdmin admin) {
    if (old_state != REQUESTED && old_state != UNIX_REQUESTED)
      return;
    this.admin = admin;
    if (old_state == REQUESTED)
      old_state = CLAIMED;
    else if (old_state == UNIX_REQUESTED)
      old_state = UNIX_CLAIMED;
  }
```

I change this to:

```
public class SystemPermission...
  public void claimedBy(SystemAdmin admin) {
    if (!state.equals(Permission.REQUESTED) && old_state != UNIX_REQUESTED)
      return;
    this.admin = admin;
    if (state.equals(Permission.REQUESTED))
      old_state = CLAIMED;
    else if (old_state == UNIX_REQUESTED)
      old_state = UNIX_CLAIMED;
```

```
  }
```

I compile and test to confirm that these changes work.

9. Now I must repeat steps 5-8 for each of the additional states that a `SystemPermission` can enter. It would require too many pages to show you all of these changes, so I'll just show you what changes are necessary to implement the State pattern version of the CLAIMED state.

To implement step 5, I must identify which `SystemPermission` method(s) can transition the CLAIMED state to one or more other states. `SystemPermission.grantedBy(…)` and `deniedBy(…)` are those methods. So I write the following code:

```
public abstract class Permission...
  public final static Permission REQUESTED = new PermissionRequested();
  public void claimedBy(SystemAdmin admin, SystemPermission permission) {}
  public void deniedBy(SystemAdmin admin) {}
  public void grantedBy(SystemAdmin admin) {}

public class PermissionClaimed extends Permission...
  public void deniedBy(SystemAdmin admin) {}
  public void grantedBy(SystemAdmin admin) {}
```

To implement step 6, I must implement `PermissionClaimed.grantedBy(…)` and `deniedBy(…)`. Again, I look in the original methods to discover what actions are performed. I learn that the CLAIMED state may transition to either DENIED, UNIX_REQUESTED or GRANTED. So I write the following code:

```
public abstract class Permission...
  public void grantedBy(SystemAdmin admin, SystemPermission permission) {}
  public void deniedBy(SystemAdmin admin, SystemPermission permission) {}

public class PermissionClaimed extends Permission {
  public void deniedBy(SystemAdmin admin, SystemPermission permission) {
    permission.willBeHandledBy(admin);
    permission.setOldState(permission.DENIED);
    permission.setIsGranted(false);
    permission.notifyUserOfPermissionRequestResult();
  }
  public void grantedBy(SystemAdmin admin, SystemPermission permission) {
    permission.willBeHandledBy(admin);
    if (permission.profile().isUnixPermissionRequired() &&
      !permission.isUnixPermissionGranted()) {
      permission.setOldState(permission.UNIX_REQUESTED);
      permission.notifyUnixAdminsOfPermissionRequest();
      return;
    }
    permission.setOldState(permission.GRANTED);
    permission.setIsGranted(true);
    permission.notifyUserOfPermissionRequestResult();
  }
```

To implement step 7, I find all places where `old_state` gets assigned to the CLAIMED constant and I *add* code to assign `state` equal to a `PermissionClaimed` instance:

```
public abstract class Permission...
  public final static Permission CLAIMED = new PermissionClaimed();


public class SystemPermission...
  public void claimedBy(SystemAdmin admin) {
    ...
    if (state.equals(Permission.REQUESTED)) {
      old_state = CLAIMED;
      state = Permission.CLAIMED;
    }
    ...
  }
```

`PermissionRequested` also makes an assignment to the CLAIMED state, so I add code there as well:

```
public class PermissionRequested extends Permission {
  public void claimedBy(SystemAdmin admin, SystemPermission permission) {
    permission.willBeHandledBy(admin);
    permission.setOldState(permission.CLAIMED);
    permission.setState(Permission.CLAIMED);
  }
}
```

To implement step 8, I look for places where `old_state` is compared against the CLAIMED constant and I change the code to compare `state` against the `PermissionClaimed` instance. Here are the changes I make:

```
public class TestStates extends TestCase...
  public void testClaimedBy() {
    permission.claimedBy(admin);
    assertEquals("claimed", Permission.CLAIMED, permission.state());
  }

public class SystemPermission...
  public void deniedBy(SystemAdmin admin) {
    if (!state.equals(Permission.CLAIMED) && old_state != UNIX_CLAIMED) return;
    ...
  }
  public void grantedBy(SystemAdmin admin) {
    if (!state.equals(Permission.CLAIMED) && old_state != UNIX_CLAIMED) return;
    ...
  }
```

I compile and test to confirm that all of the changes work. Next, I continue to implement steps 5-8 for the remainder of `SystemPermission` states.

10. Now comes the fun part – making `SystemPermission` delegate to methods on the `state` field for all of its state transitions. This step allows me to delete many lines of code:

```
public class SystemPermission...
  public void claimedBy(SystemAdmin admin) {
    state.claimedBy(admin,this);

    if (!state.equals(Permission.REQUESTED) &&
      !state.equals(Permission.UNIX_REQUESTED))
      return;
    this.admin = admin;
    if (state.equals(Permission.REQUESTED)) {
      old_state = CLAIMED;
      state = Permission.CLAIMED;
    }
    else if (state.equals(Permission.UNIX_REQUESTED)) {
      old_state = UNIX_CLAIMED;
      state = Permission.UNIX_CLAIMED;
    }
  }

  public void deniedBy(SystemAdmin admin) {
    state.deniedBy(admin, this);

    if (!state.equals(Permission.CLAIMED) &&
      old_state != UNIX_CLAIMED) return;
    if (this.admin != admin) return;
    isGranted = false;
    isUnixPermissionGranted = false;
    old_state = DENIED;
    state = Permission.DENIED;
    notifyUserOfPermissionRequestResult();
  }
```

```
  public void grantedBy(SystemAdmin admin) {
    state.grantedBy(admin,this);

    if (!state.equals(Permission.CLAIMED) &&
      !state.equals(Permission.UNIX_CLAIMED)) return;
    if (this.admin != admin) return;

    if (profile.isUnixPermissionRequired() &&
      state.equals(Permission.UNIX_CLAIMED))
      isUnixPermissionGranted = true;
    else if (profile.isUnixPermissionRequired() &&
      !isUnixPermissionGranted()) {
      old_state = UNIX_REQUESTED;
      state = Permission.UNIX_REQUESTED;
      notifyUnixAdminsOfPermissionRequest();
      return;
    }
    old_state = GRANTED;
    state = Permission.GRANTED;
    isGranted = true;
    notifyUserOfPermissionRequestResult();
  }
```

I compile and test that unbelievably, everything works as expected.

11. Finally, I get a chance to remove more unnecessary code: i.e. everything associated with `old_state`, including the `old_state` assignments made from the state subclasses. Here are a few of the deletions I make:

```
public class PermissionRequested extends Permission {
  public void claimedBy(SystemAdmin admin, SystemPermission permission) {
    permission.willBeHandledBy(admin);
    permission.setOldState(permission.CLAIMED);
    permission.setState(Permission.CLAIMED);
  }
}

public class SystemPermission...
  private String old_state;

  public final static String REQUESTED = "REQUESTED";
  public final static String CLAIMED = "CLAIMED";
  public final static String GRANTED = "GRANTED";
  public final static String DENIED = "DENIED";
  public final static String UNIX_REQUESTED = "UNIX REQUESTED";
  public final static String UNIX_CLAIMED = "UNIX CLAIMED";

  public SystemPermission(SystemUser requestor, SystemProfile profile) {
    ...
    old_state = REQUESTED;
    ...
  }

  public String old_state() {
    return old_state;
  }

  public void setOldState(String state) {
    this.old_state = state;
  }
```

I compile and run my tests to confirm that everything is working. I've now fully implemented the State pattern. Was all that work worth it? I'll let you decide. Consider the code in a few of the state subclasses (listed below) and compare it against the state-altering conditional logic we started with:

```
public class PermissionRequested extends Permission {
  protected PermissionRequested() {
    super();
  }
  public void claimedBy(SystemAdmin admin, SystemPermission permission) {
    permission.willBeHandledBy(admin);
    permission.setState(Permission.CLAIMED);
  }
}

public class UnixPermissionRequested extends Permission {
  protected UnixPermissionRequested() {
    super();
  }
  public void claimedBy(SystemAdmin admin, SystemPermission permission) {
    permission.willBeHandledBy(admin);
    permission.setState(Permission.UNIX_CLAIMED);
  }
}

public class UnixPermissionClaimed extends Permission {
  protected UnixPermissionClaimed() {
    super();
  }
  public void deniedBy(SystemAdmin admin, SystemPermission permission) {
    permission.willBeHandledBy(admin);
    permission.setState(Permission.DENIED);
    permission.setIsGranted(false);
    permission.setIsUnixPermissionGranted(false);
    permission.notifyUserOfPermissionRequestResult();
  }
  public void grantedBy(SystemAdmin admin, SystemPermission permission) {
    permission.willBeHandledBy(admin);
    permission.setState(Permission.GRANTED);
    permission.setIsGranted(true);
    permission.setIsUnixPermissionGranted(true);
    permission.notifyUserOfPermissionRequestResult();
  }
}

public class PermissionGranted extends Permission {
  protected PermissionGranted() {
    super();
  }
}

public class PermissionDenied extends Permission {
  protected PermissionDenied() {
    super();
  }
}
```

Replace Singleton with Constant

Motivation

Mechanics

Example

```
public void denied(ApplicationPermission permission) {
   permission.setState(ApplicationPermissionDenied.getInstance());
}
```

becomes

```
public void denied(ApplicationPermission permission) {
   permission.setState(permission.DENIED);
}
```

Replace Retrieval with Listener

[Colloquium Example]
[SAX vs DOM]

Motivation

Mechanics

Example

References

[Beck]
Beck, Kent. *Smalltalk Best Practice Patterns.* Upper Saddle River, N.J.: Prentice Hall, 1997.

[Bloch]
Bloch, Joshua. *Effective Java*. Addison-Wesley, 2001.

[Fowler]
Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[GOF]
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Reading, Mass.: Addison-Wesley, 1995.

[JUnit]
Kent Beck and Erich Gamma. JUnit Testing Framework. Available on the Web (http://www.junit.org).

Appendix A – Naming Conventions

[describe forName, writeOn, claimedBy, etc].

Appendix B – Loan Terminology

A few of the code fragments used in the examples in this book are based on financial systems that calculate numbers for Loans. If you don't have experience writing systems like that, you may find that the example code is hard to understand. No problem. You don't need to know much to make sense of this code. The following describes the three major loan types used in the example code:

- Term Loan: often abbreviated as a TL, is the simplest of loans: I give you $100 and ask you to pay it back by some date, which is known as the maturity date of the loan.

- Revolver: a Revolver is an instrument that provides "revolving credit", like a credit card with a spending limit and expiry date. Financial companies often abbreviate Revolvers as "RC."

- RCTL – this is a combination of a Revolver and Term Loan. The loan starts its life as a Revolver, and on its expiry date, becomes a Term Loan. RCTLs have both expiry and maturity dates.

It is common to calculate numbers for loans, such as capital, risk-adjusted capital, return on capital, etc. When we do risk-adjusted calculations, we often need to use the numbers from some risk-table. In general, the kinds of calculations done on the various loan types shouldn't effect your understanding of the refactoring steps.

Conclusion

Stay tuned for more refactorings. This work is only the beginning of a larger work on this subject. I welcome your thoughts and feedback. If you are interested in seeing the latest copies of this work, please visit http://industriallogic.com/xp/refactoring/

Acknowledgements

I'd like to thank my wife, Tracy, for her loving support and continuous encouragement.

Eric Evans has contributed more than any one else to making this work what it is today. I want to thank him for his continued support, thoughtful conversations, great ideas and feedback.

I'd also like to the thank the following people:

- Russ Rufer and all of the many great members of the Silicon Valley Patterns Group (Tracy Bialik, Alan Harriman, Chris Lopez, Charlie Toland, Bob Evans, John Brewer, Jeff Miller, Patrick Manion, Debbie Utley, Carol Thistlethwaite, Summer Misherghi, Ted Young, Siqing Zhang). Your feedback has been invaluable.
- Robert Hirshfeld, for helping clarify the Decorator mechanics section.
- Martin Fowler for inspiration and encouragement, for giving me the advice that I once gave him (i.e. use code sketches at the beginning of each refactoring) and for numerous helpful suggestions and ideas.
- Kent Beck for his reviews and suggestions.
- John Vlissides for his reviews and suggestions.
- Ralph Johnson, Brian Foote, Brian Marick, Don Roberts, John Brant and others from the University of Illinois.
- Somik Raha – for many great pairing sessions, refactoring ideas and some poor code he once wrote when he was tired which provided great refactoring material.
- Many thanks to the following folks who provided excellent suggestions: Rob Mee, Jeff Grigg, Kaoru Hosokawa, Don Hinton, Andrew Swan, Erik Meade, Craig Demyanovich.