

Refactoring To Patterns

version 1.13



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

Table of Contents

<i>Introduction</i>	5
<i>Chain Constructors*</i>	6
Motivation	7
Mechanics	7
Example	7
<i>Replace Multiple Constructors with Creation Methods*</i>	9
Motivation	9
Mechanics	10
Example	10
Parameterized Creation Methods	13
<i>Encapsulate Subclasses with Creation Methods*</i>	15
Motivation	15
Mechanics	16
Example	16
<i>Extract Creation Class*</i>	18
Motivation	18
Mechanics	19
Example	19
<i>Replace Conditional Calculations with Strategy*</i>	22
Motivation	23
Mechanics	23
Example	24
<i>Replace Implicit Tree with Composite*</i>	31
Motivation	31
Mechanics	32
Example	32
<i>Encapsulate Composite with Builder**</i>	35
Motivation	35
Mechanics	36
Example	36
Extended Example	38

<i>Extract Special-Case Logic into Decorators</i>	41
Motivation	42
Mechanics	43
Example	43
<i>Replace Hard-Coded Notifications with Observer*</i>	50
Motivation	51
Mechanics	52
Example	52
<i>Move Accumulation to Collecting Parameter*</i>	55
Motivation	55
Mechanics	56
Example	56
Example 2	59
<i>Replace One/Many Distinctions with Composite</i>	60
Motivation	60
Mechanics	61
Example	61
<i>Compose Method**</i>	62
Motivation	62
Mechanics	63
Example 1	64
Example 2	67
Example 3	73
<i>Separate Versions with Adapters *</i>	75
Motivation	76
Mechanics	76
Example	77
Adapting Legacy Systems	82
<i>Adapt Interface</i>	83
Motivation	83
Mechanics	84
Example	84
<i>References</i>	86
<i>Conclusion</i>	87

Acknowledgements..... 87

Introduction

I've yet to write a formal introduction to this work. 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

An additional goal I have is to fill a gap, first noted by my friend Martin Fowler in his classic work: *Refactoring: Improving The Design Of Existing Code*:

Many of the refactorings, such as *Replace Type Code with State/Strategy* (227) and *Form Template Method* (345) are about introducing patterns into a system. As the essential Gang of Four book says, "Design Patterns...provide targets for your refactorings." 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. I don't have refactorings for all known patterns in this book, not even for all the Gang of Four patterns. This is another aspect of the incompleteness of this catalog. I hope someday the gap will be closed.

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:

- Zero, one or two asterisks beside the name of every refactoring. Zero asterisks mean I don't use the refactoring very often, one asterisk means I use it sometimes, and two asterisks mean I use it quite often.
- 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/>

There is also 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** (9).

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 amount 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.

Replace Multiple Constructors with Creation Methods*

Many 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*

Loan
+Loan(notional, outstanding, customerRating, expiry) +Loan(notional, outstanding, customerRating, expiry, maturity) +Loan(capitalStrategy, outstanding, customerRating, expiry, maturity) +Loan(type, capitalStrategy, outstanding, customerRating, expiry) +Loan(type, capitalStrategy, outstanding, customerRating, expiry, maturity)



Loan
#Loan(type, capitalStrategy, notional, outstanding, customerRating, expiry, maturity) <u>+newTermLoan(notional, outstanding, customerRating, expiry) : Loan</u> <u>+newTermLoanWithStrategy(capitalStrategy, notional, outstanding, customerRating, expiry) : Loan</u> <u>+newRevolver(notional, outstanding, customerRating, expiry) : Loan</u> <u>+newRevolverWithStrategy(capitalStrategy, notional, outstanding, customerRating, expiry) : Loan</u> <u>+newRCTL(notional, outstanding, customerRating, expiry, maturity) : Loan</u> <u>+newRCTLWithStrategy(capitalStrategy, notional, outstanding, customerRating, expiry, maturity) : Loan</u>

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 only one constructor, this isn't a problem. But 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 so wrong with that? A lot. Multiple 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 make the call to 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 or Joshua Bloch use the term *Factory Method* in their excellent books, *Refactoring* [Fowler] and *Effective Java* [Bloch], respectively.

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

1. Identify the class that has copious constructors. These constructors will often have abundant parameters, which further confuses developers when they want to choose which constructor to call to obtain an instance.
2. Identify the *catch-all* constructor or create one using *Chain Constructors (6)*. If your catch-all constructor is public, make it private or protected.
3. For every type of object that can be created using one of the many constructors, create an intention-revealing Creation Method. Test that each Creation Method returns the correct type of object and confirm that each Creation Method is called by some client code (if there is no user of the Creation Method, remove it until there is a need for it).
4. Replace all calls to constructors with calls to the appropriate Creation Methods. This can take some work but will result in easier-to-read client code.

Example

1. I'll use the example shown in the code sketch. We start with a single 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 {
    private static String TERM_LOAN = "TL";
    private static String REVOLVER = "RC";
    private static String RCTL = "RCTL";
    private String type;
    private CapitalStrategy strategy;
    private float notional;
    private float outstanding;
    private int customerRating;
    private Date maturity;
    private Date expiry;

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

```

    Date maturity) {
        this(RCTL, new RevolvingTermROC(), notional, outstanding, customerRating,
            expiry, maturity);
    }
    public Loan(CapitalStrategy strategy, float notional, float outstanding,
        int customerRating, Date expiry, Date maturity) {
        this(RCTL, strategy, notional, outstanding, customerRating,
            expiry, maturity);
    }
    public Loan(String type, CapitalStrategy strategy, float notional,
        float outstanding, int customerRating, Date expiry) {
        this(type, strategy, notional, outstanding, customerRating, expiry, null);
    }
    public Loan(String type, CapitalStrategy strategy, float notional,
        float outstanding, int customerRating, Date expiry, Date maturity) {
        this.type = type;
        this.strategy = strategy;
        this.notional = notional;
        this.outstanding = outstanding;
        this.customerRating = customerRating;
        this.expiry = expiry;
        if (RCTL.equals(type))
            this.maturity = maturity;
    }
}
}

```

There are 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 do other programmers who will use this code know this? If they don't, will the constructors communicate sufficiently when I'm not around? Well, they will probably be able to figure it out, with a little work. But they shouldn't have to work at all to obtain the type of Loan they need.

Before we continue with the refactoring, I want to know what other assumptions are made in the above constructors. There is a major one: if you call the first constructor you will get back a Term Loan, not a Revolver. If you want a Revolver instance, you have to call one of the last two constructors, which lets you pass in a Loan type. Hmmm, I wonder if all users of this code will know this? Or will they have to learn by encountering some ugly bugs?

2. Our 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(String type, CapitalStrategy strategy, float notional, float outstanding,
    int customerRating, Date expiry, Date maturity) {
    this.type = type;
    this.strategy = strategy;
    this notional = notional;
    this.outstanding = outstanding;
    this.customerRating = customerRating;
    this.expiry = expiry;
    if (RCTL.equals(type))
        this.maturity = maturity;
}

```

I make this constructor protected:

```

protected Loan(String type, CapitalStrategy strategy, float notional, float outstanding,
    int customerRating, Date expiry, Date maturity)

```

3. Next, I must identify the types of objects that may be created using one of the many Loan constructors. In this case I identify the following types:

- a Term Loan with default capital strategy
- a Term Loan with custom capital strategy
- a Revolver with default capital strategy

- a Revolver with custom capital strategy
- an RCTL with default capital strategy
- an RCTL with custom capital strategy

I begin by writing a test for a new Creation Method that will return a Term Loan with a default Term Loan capital strategy:

```
public void testTermLoan() {
    String custRating = 2;
    Date expiry = createDate(2001, Calendar.NOVEMBER, 20);
    Loan loan = Loan.newTermLoan(1000f, 250f, CUSTOMER_RATING, expiry);
    assertNotNull(loan);
    assertEquals(Loan.TERM_LOAN, loan.getType());
}
```

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

```
public class Loan...
    static Loan newTermLoan(float notional, float outstanding, int customerRating,
        Date expiry) {
        return new Loan(TERM_LOAN, new TermROC(), notional, outstanding,
            customerRating, expiry, null);
    }
}
```

Notice how this method delegates to the protected catch-all constructor identified in step 1.

I will create five similar tests and five additional intention-revealing Creation Methods for the remaining object types. As this work is done, the public Loan constructors are eliminated a little at a time. Here's what the refactored Loan class looks like:

```
public class Loan {
    private static String TERM_LOAN = "TL";
    private static String REVOLVER = "RC";
    private static String RCTL = "RCTL";
    private String type;
    private CapitalStrategy strategy;
    private float notional;
    private float outstanding;
    private int customerRating;
    private Date maturity;
    private Date expiry;

    protected Loan(String type, CapitalStrategy strategy, float notional,
        float outstanding, int customerRating, Date expiry, Date maturity) {
        this.type = type;
        this.strategy = strategy;
        this.notional = notional;
        this.outstanding = outstanding;
        this.customerRating = customerRating;
        this.expiry = expiry;
        if (RCTL.equals(type)
            this.maturity = maturity;
        }
    }
    static Loan newTermLoan(float notional, float outstanding, int customerRating,
        Date expiry) {
        return new Loan(TERM_LOAN, new TermROC(), notional, outstanding, customerRating,
            expiry, null);
    }
    static Loan newTermWithStrategy(CapitalStrategy strategy, float notional,
        float outstanding, int customerRating, Date expiry) {
        return new Loan(TERM_LOAN, strategy, new TermROC(), notional, outstanding,
            customerRating, expiry, null);
    }
    static Loan newRevolver(float notional, float outstanding, int customerRating,
        Date expiry) {
        return new Loan(REVOLVER, new RevolverROC(), notional, outstanding,
            customerRating, expiry, null);
    }
    static Loan newRevolverWithStrategy(CapitalStrategy strategy, float notional,
        float outstanding, int customerRating, Date expiry) {
```

```
        return new Loan(REVOLVER, strategy, new RevolverROC(), notional, outstanding,
            customerRating, expiry, null);
    }
    static Loan newRCTL(float notional, float outstanding, int customerRating,
        Date expiry, Date maturity) {
        return new Loan(RCTL, new RCTLROC(), notional, outstanding,
            customerRating, expiry, maturity);
    }
    static Loan newRCTLWithStrategy(CapitalStrategy strategy, float notional,
        float outstanding, int customerRating, Date expiry, Date maturity) {
        return new Loan(RCTL, strategy, new RevolverROC(), notional, outstanding,
            customerRating, expiry, maturity);
    }
}
```

Now it is quite clear how to obtain the kind of Loan instance you need – you simply look at the available types and call the correct method. The new Creation Methods still take a fair number of parameters. *Introduce Parameter Object (295)* [Fowler] is a refactoring that can help you reduce the number of parameters you pass to methods.

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();
```

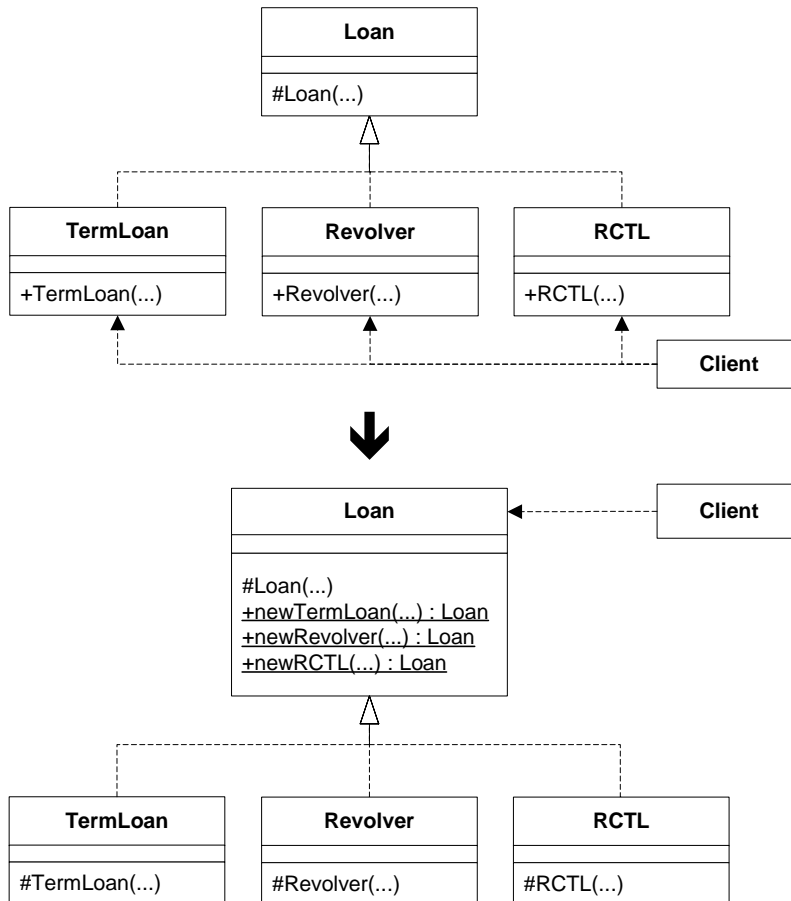
These Creation Methods would not altogether replace the public constructors, but would supplement them and perhaps reduce their number. However, because the above Creation Methods aren't parameterized, they could easily multiply over time, yielding a forrest of Creation Methods that would also make it 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 Subclasses with Creation Methods*

Subclasses implement a common interface
but are constructed in diverse ways

*Encapsulate the subclasses with intention-revealing
Creation Methods in the base class*



Motivation

The motivation for doing this refactoring is to hide implementation details from those who don't need to know about them.

Consider a design in which you have numerous subclasses that implement a common interface. You expect client code to communicate with these subclasses via the common interface. But if you don't protect the subclass constructors, you make it possible for clients to directly instantiate them. What's so wrong with that?

Here's what's wrong: once programmers write code that talks directly to a subclass type rather than through its common interface, the door is open for changing the subclass interface in response to the needs of client code. This happens quiet often on projects. Once it happens, the altered subclass is different from the other subclasses and its base class: it has special methods on it that are only available via *its* interface and not the common interface. That may seem like no big deal, but it leads to the creation of a lot of special-case logic.

I recently implemented this refactoring on a project where we had about 25 similar subclasses, each of which had its own small collection of public constructors. Client code on that system directly instantiated the various subclasses, and programmers would sometimes instantiate the wrong subclass or the right subclass with the wrong constructor. The refactoring encapsulated the details and produced a single place to get a well-named subclass instance.

Communication	Duplication	Simplicity
If it is important that client code talk with subclass instances via a common interface, it is important to communicate this in your code. Creating subclasses doesn't communicate this very well. Instead, create protected subclass constructors and Creation Methods that give clients access to the various subclass types.	Duplication isn't an issue with this refactoring.	Figuring out which subclass to instantiate and how to instantiate it isn't as simple as calling intention-revealing Creation Methods, particularly when you have several subclasses and numerous ways to instantiate them. It is simpler if clients don't have to know about subclasses, but can instead obtain instances from superclass Creation Methods.

Mechanics

1. Make the subclass constructor that you'd like to encapsulate via a superclass Creation Method protected.
2. Create an intention-revealing Creation Method on the superclass. Makes its body a call to the protected subclass constructor.
3. Replace all calls to the subclass constructor with calls to the superclass Creation Method.
4. Compile and test after each replacement.

Example

1. In the refactoring, *Replace Multiple Constructors with Creation Methods** (9), three loan types – Term Loan, Revolver and RevolverTerm – were represented in the single Loan class. In this example, each of the loan types will be a subclass of an abstract Loan superclass:

```

abstract class Loan {
    ...
    protected Loan(...) {
        ...
    }
}

public class TermLoan extends Loan {
    public TermLoan(...) {
        super(...);
    }
}

public class Revolver extends Loan {
    public Revolver(...) {
        super(...);
    }
}

public class RCTL extends Loan {
    public RCTL(...)
        super(...);
}

```


The abstract Loan superclass constructor is protected, and the constructors for the three subclasses are public. We'll focus on the TermLoan class. The first step is to protect its constructor:

```
public class TermLoan extends Loan {
    protected TermLoan(...) {
        super(...);
    }
}
```

2. The next step is to create intention-revealing Creation Methods on the Loan class for the protected TermLoan constructor:

```
abstract class Loan...
    public static Loan newTermLoan(...) {
        return new TermLoan(...);
    }
```

3. We must now replace client calls to the TermLoan constructor with a call to the superclass Creation Method:

```
client code...
    Loan termLoan = Loan.newTermLoan(1000f, 250f, getCustRating(), expiryDate);
```

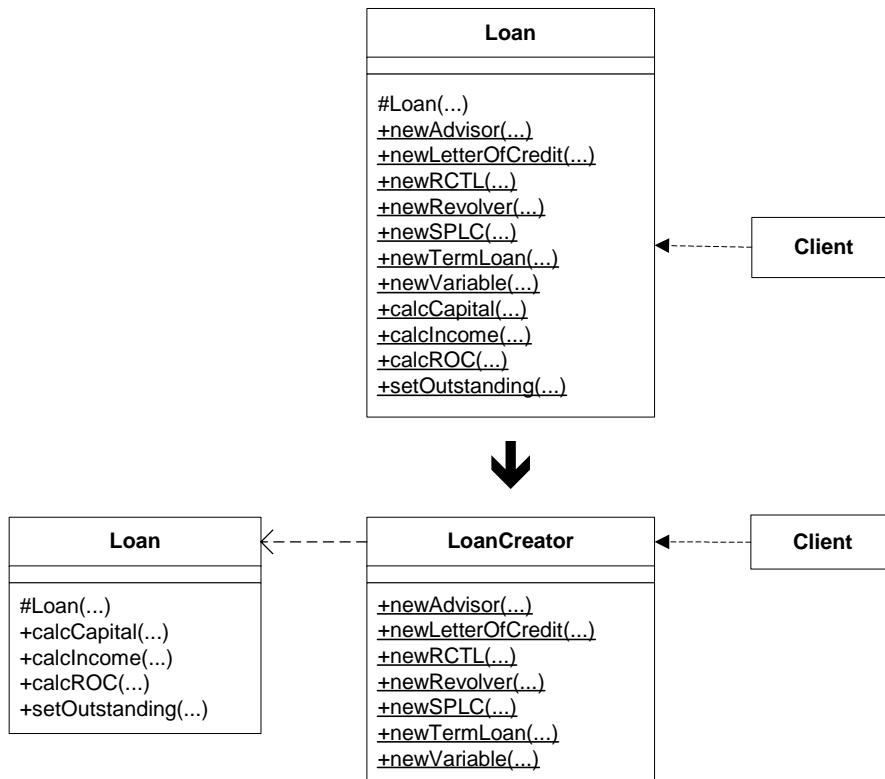
Consider what this code now does: it gives access to the subclass via the superclass, and ensures that clients obtain subclass instances via the Loan interface. This prevents clients from directly instantiating a TermLoan. More important, the design communicates to other programmers Loan subclasses are not meant to be public – the convention is to offer up access to subclasses via the superclass and a common interface.

4. In this final step we compile and test. If you do test-first design, instead of waiting till step 4 to test, you will begin this refactoring by writing a test to obtain the Loan instance you want from the intention-revealing superclass Creation Method.

Extract Creation Class*

Too many Creation Methods on a class obscure
it's primary responsibility (or responsibilities)

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



Motivation

This refactoring is essentially *Extract Class* [Fowler], only it is done on an existing 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 – its main purpose in life – can feel like it is obscured, outweighed by too much creational code. At such times, it's best to restore the class's identity by extracting all of its Creation Methods into their own home. We call such a home a Creation Class. Creation Classes are a little like Abstract Factories [GoF] in 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 that create object instances.

Communication	Duplication	Simplicity
<p>When we create classes, we give them responsibilities. One such responsibility, which is often not a class's primary responsibility, is object creation. When object creation begins to dominate the public interface of the class, the class no longer strongly communicates its main purpose. When that happens, it's best to communicate the act of object creation by creating a special class just to create instances of the object – i.e. a Creation Class.</p>	<p>Duplication is not an issue with respect to this refactoring.</p>	<p>One principle states that the fewer responsibilities a class has, the simpler it is. That principle can lead to an explosion of class, which can make a system not simple to work with at all. So one must use judgement. 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.</p>

Mechanics

1. Identify a class (which we'll call "A") that is overbalanced with Creation Methods.
2. Create a class that will be 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 privileges are accounted for.
4. Change all callers to obtain object references from your new Creation Class, rather than from A.

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` strategies) 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

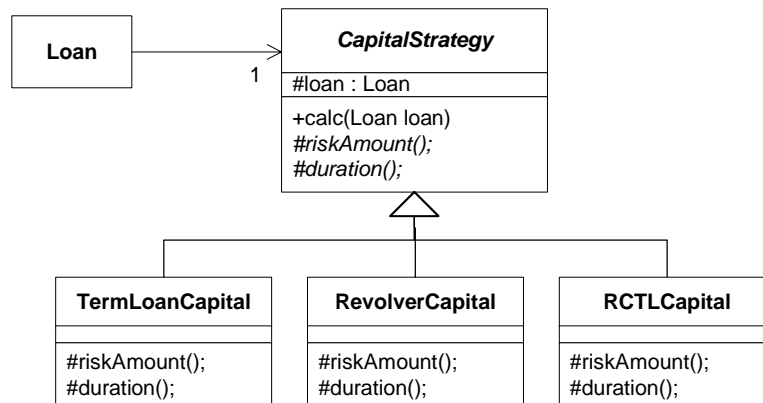
```
Loan termLoan = LoanCreator.newTermLoan(...)
```

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 amount 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 `calcUnusedRiskAmount()` 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 chose 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");
    assertEquals("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 <i>what</i> , <i>where</i> and <i>how</i> to add items to a tree. With a Builder, a client needs to know only <i>what</i> and <i>where</i> 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");
    assertEquals("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");
    assertEquals("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");
    assertEquals("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*).

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:

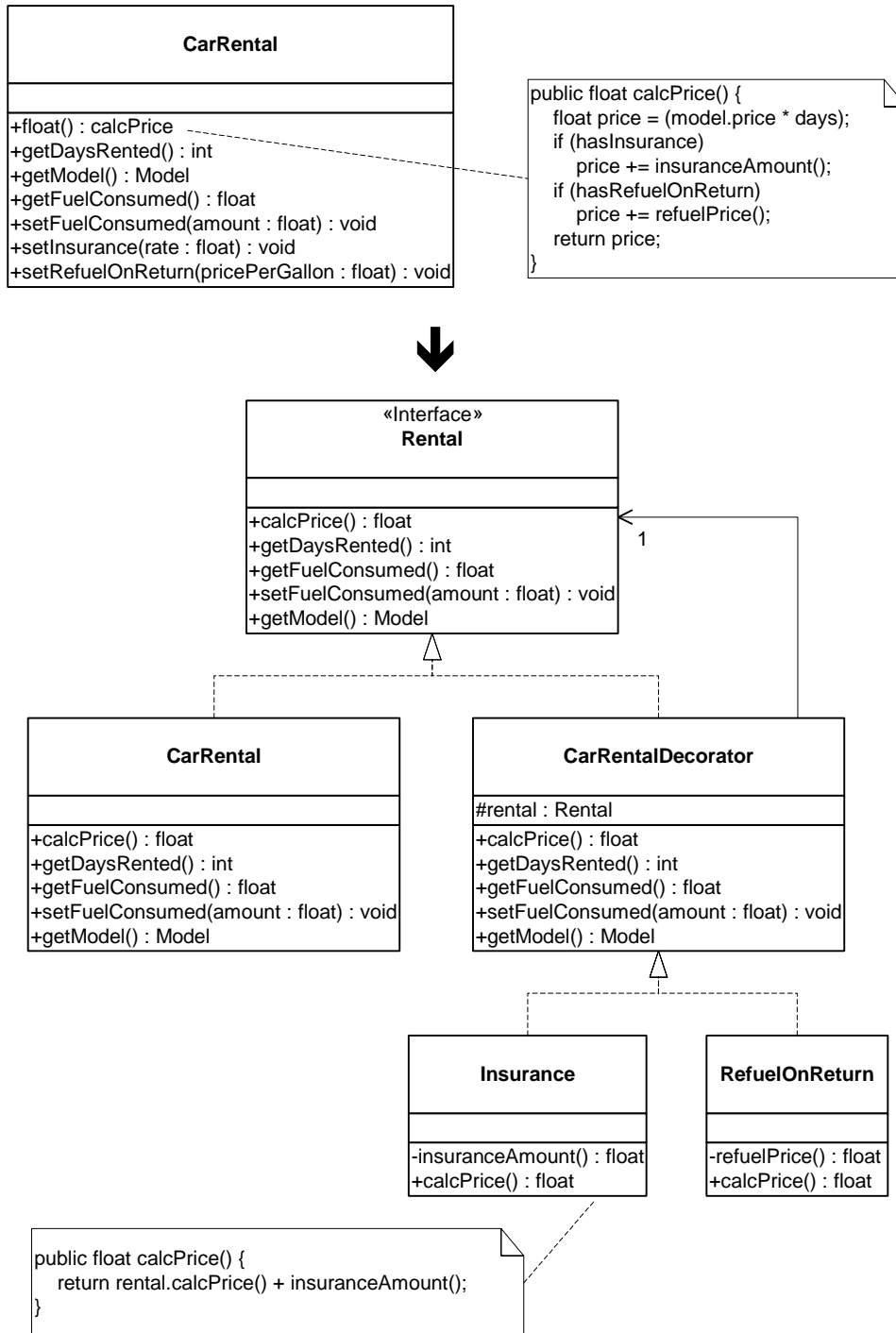
```
public String toString() {
    return root.toStringHelper(new StringBuffer(outputBufferSize));
}
```

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

Extract Special-Case Logic into Decorators

Your classes or methods have optional or special-case processing logic

Retain core logic but extract optional or special-case logic 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 <i>does</i> 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 3 or more rental options, which could be combined in different ways. But if the example contained all of that code, it might span 5 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 3 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

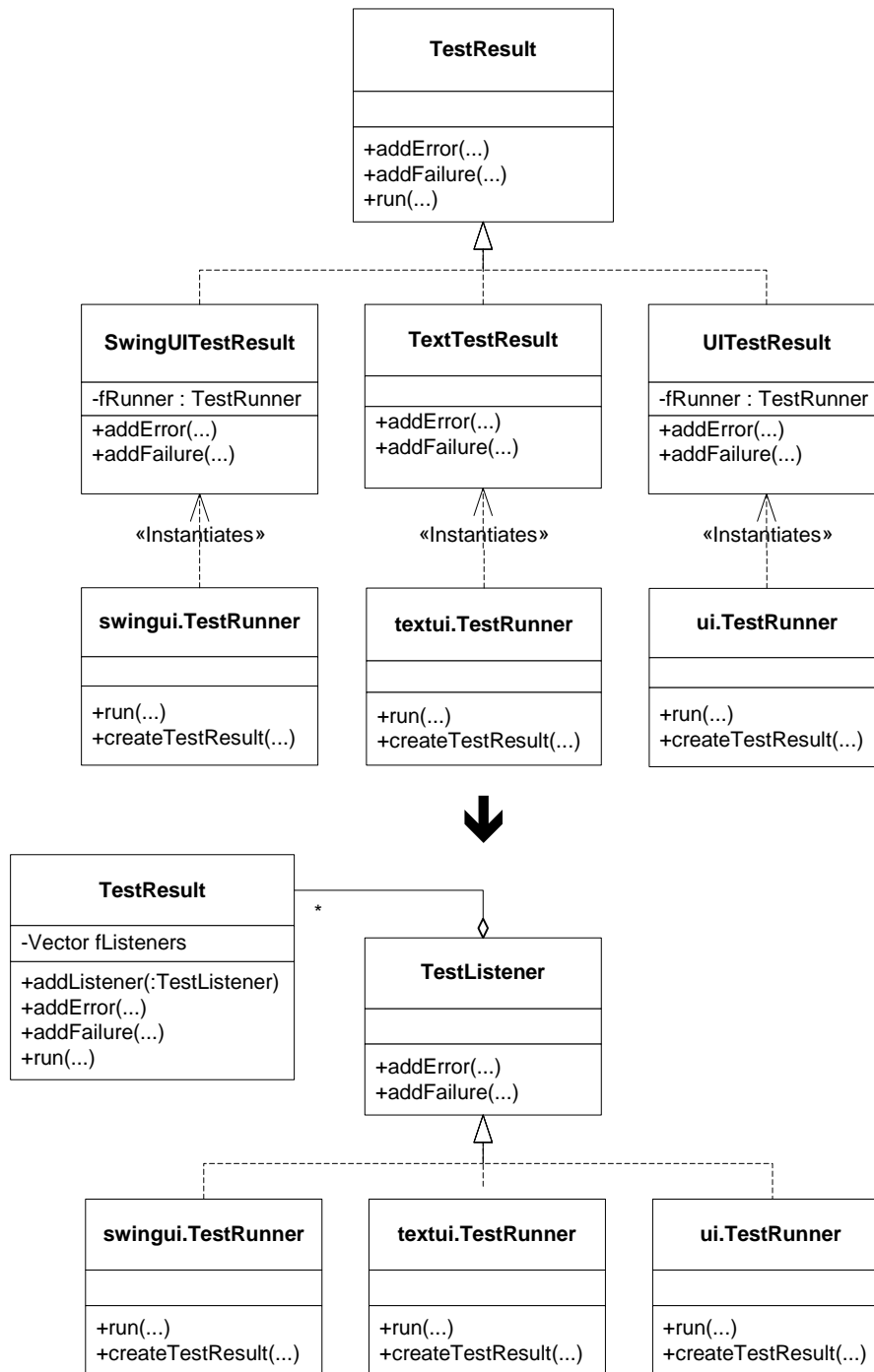
be replacing with Decorator. If it is complex conditional code, chances are that adding Decorator may *decrease* the lines of code. But in any event, introducing Decorator into your system should make your code simpler and easier to understand. It may even help you reduce duplication if your code must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

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 <i>who</i> can observe its events and <i>when</i> 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, AssertionError 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.

Example 2

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.

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;
}
```

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.

This refactoring has two asterisks beside its name because I find myself aggressively refactoring to it *often*. For example, just the other day I was debugging a method in some code I'm writing. 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 <i>what</i> a method does but not <i>how</i> 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 its 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-realing 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 `paintCardHighlight()` method. Why? I

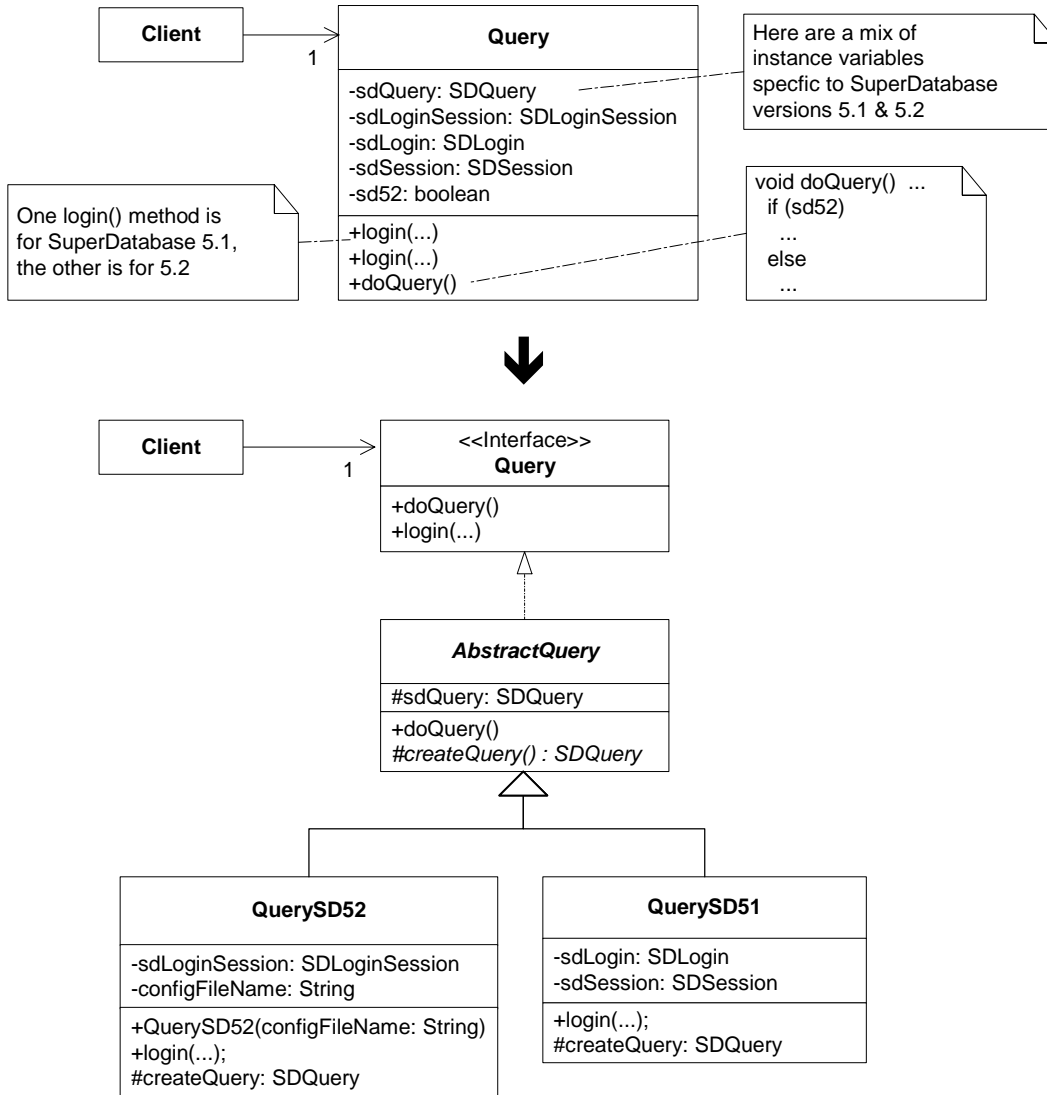
wanted the reader to simply see three steps: paint image, highlight image and paint card text. The detail of whether or not we do highlight the card isn't important to me – the reader can find that out if they look. But if that confuses other programmers, I'd be happy to see the method renamed to `paintCardHighlightIfNecessary(g)`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

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

Separate Versions with Adapters *

You're code must handle different versions of a component, library, API or other entity

Write Adapters for each version you need to support



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 maintenance 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
<p>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.</p>	<p>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.</p>	<p>When a class is responsible for functioning correctly with several versions of some other code, it is rarely simple. Version-specific 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.</p>

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 you 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 applications 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();
}

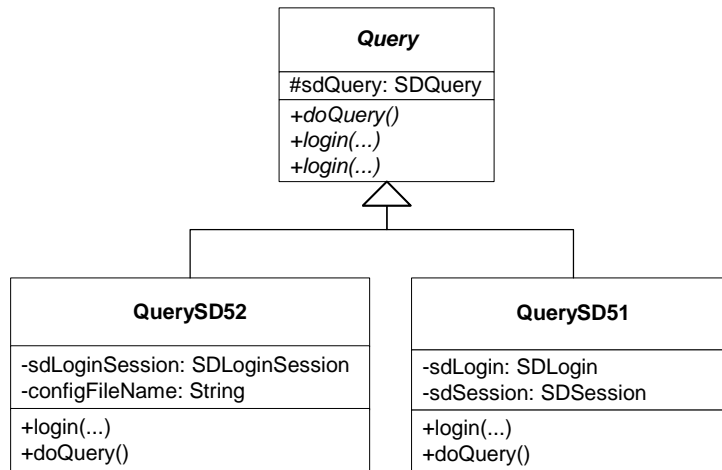
class QuerySD51 {
    private SDLogin sdLogin;
    private SDSession sdSession;

    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 *Pull Up Method* (322) [Fowler] to remove `doQuery()` from both subclasses, and apply *Form Template Method* (345) [Fowler] on `Query`'s `doQuery()`, such that it calls a *Factory Method* [GoF] to obtain the properly initialized `sdQuery` instance:

```

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 there is one abstract login() method on Query:

```
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 now 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 Legacy Systems

Organization X 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 systems 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*, 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 `MouseEventListener` interface. However, it only implements one of the two methods declared by the `MouseEventListener` 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 `MouseEventListener` interface. It's called `MouseEventAdapter`. So we create the following new method on the `CardComponent` class, using Java's handy anonymous inner class capability:

```
private MouseEventAdapter createMouseEventAdapter() {
    return new MouseEventAdapter() {
    };
}
```

2. Next, we delete the empty method(s) that `CardComponent` declared because it implemented `MouseEventListener`. 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 `MouseEventAdapter`:

```
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.

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>).

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 fantastic wife, Tracy, for her loving support and encouragement and the great food she brings by when I'm caught up in writing.

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 thank the following people:

- 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, Ted Young.