Chapter 6. Composing Methods

A large part of my refactoring is composing methods to package code properly. Almost all the time the problems come from methods that are too long. Long methods are troublesome because they often contain lots of information, which gets buried by the complex logic that usually gets dragged in. The key refactoring is Extract Method, which takes a clump of code and turns it into its own method. Inline Method is essentially the opposite. You take a method call and replace it with the body of the code. I need Inline Method when I've done multiple extractions and realize some of the resulting methods are no longer pulling their weight or if I need to reorganize the way I've broken down methods.

The biggest problem with Extract Method is dealing with local variables, and temps are one of the main sources of this issue. When I'm working on a method, I like Replace Temp with Queryto get rid of any temporary variables that I can remove. If the temp is used for many things, I use Split Temporary Variable first to make the temp easier to replace.

Sometimes, however, the temporary variables are just too tangled to replace. I need Replace Method with Method Object. This allows me to break up even the most tangled method, at the cost of introducing a new class for the job.

Parameters are less of a problem than temps, provided you don't assign to them. If you do, you need Remove Assignments to Parameters.

Once the method is broken down, I can understand how it works much better. I may also find that the algorithm can be improved to make it clearer. I then use Substitute Algorithm to introduce the clearer algorithm.

Extract Method

You have a code fragment that can be grouped together.

Turn the fragment into a method whose name explains the purpose of the method.

```
void printOwing(double amount) {
    printBanner();

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + amount);
}
```

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails (double amount) {
```

```
        System.out.println ("name:" + _name);
        System.out.println ("amount" + amount);
    }
```

Motivation

Extract Method is one of the most common refactorings I do. I look at a method that is too long or look at code that needs a comment to understand its purpose. I then turn that fragment of code into its own method.

I prefer short, well-named methods for several reasons. First, it increases the chances that other methods can use a method when the method is finely grained. Second, it allows the higher-level methods to read more like a series of comments. Overriding also is easier when the methods are finely grained.

It does take a little getting used to if you are used to seeing larger methods. And small methods really work only when you have good names, so you need to pay attention to naming. People sometimes ask me what length I look for in a method. To me length is not the issue. The key is the semantic distance between the method name and the method body. If extracting improves clarity, do it, even if the name is longer than the code you have extracted.

Mechanics

- Create a new method, and name it after the intention of the method (name it by what it does, not by how it does it).

> *? If the code you want to extract is very simple, such as a single message or function call, you should extract it if the name of the new method will reveal the intention of the code in a better way. If you can't come up with a more meaningful name, don't extract the code.*

- Copy the extracted code from the source method into the new target method.
- Scan the extracted code for references to any variables that are local in scope to the source method. These are local variables and parameters to the method.
- See whether any temporary variables are used only within this extracted code. If so, declare them in the target method as temporary variables.
- Look to see whether any of these local-scope variables are modified by the extracted code. If one variable is modified, see whether you can treat the extracted code as a query and assign the result to the variable concerned. If this is awkward, or if there is more than one such variable, you can't extract the method as it stands. You may need to use Split Temporary Variable and try again. You can eliminate temporary variables with Replace Temp with Query (see the discussion in the examples).
- Pass into the target method as parameters local-scope variables that are read from the extracted code.
- Compile when you have dealt with all the locally-scoped variables.
- Replace the extracted code in the source method with a call to the target method.

> *? If you have moved any temporary variables over to the target method, look to see whether they were declared outside of the extracted code. If so, you can now remove the declaration.*

- Compile and test.

Example: No Local Variables

In the simplest case, Extract Method is trivially easy. Take the following method:

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

It is easy to extract the code that prints the banner. I just cut, paste, and put in a call:

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}

void printBanner() {
    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
```

Mechanics

- Inspect all uses of the candidate fields to ensure they are used in the same way.
- If the fields do not have the same name, rename the fields so that they have the name you want to use for the superclass field.
- Compile and test.
- Create a new field in the superclass.

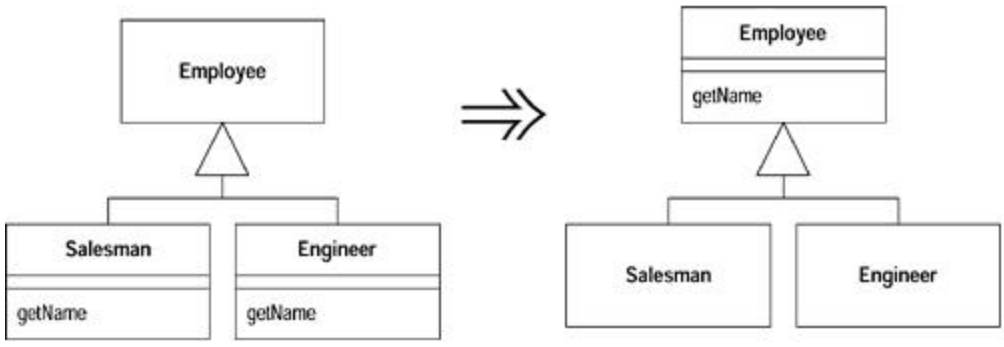
> ?rarr; *If the fields are private, you will need to protect the superclass field so that the subclasses can refer to it.*

- Delete the subclass fields.
- Compile and test.
- Consider using Self Encapsulate Field on the new field.

Pull Up Method

You have methods with identical results on subclasses.

Move them to the superclass.



Motivation

Eliminating duplicate behavior is important. Although two duplicate methods work fine as they are, they are nothing more than a breeding ground for bugs in the future. Whenever there is duplication, you face the risk that an alteration to one will not be made to the other. Usually it is difficult to find the duplicates.

The easiest case of using *Pull Up Method* occurs when the methods have the same body, implying there's been a copy and paste. Of course it's not always as obvious as that. You could just do the refactoring and see if the tests croak, but that puts a lot of reliance on your tests. I usually find it valuable to look for the differences; often they show up behavior that I forgot to test for.

Often *Pull Up Method* comes after other steps. You see two methods in different classes that can be parameterized in such a way that they end up as essentially the same method. In that case the smallest step is to parameterize each method separately and then generalize them. Do it in one go if you feel confident enough.

A special case of the need for *Pull Up Method* occurs when you have a subclass method that overrides a superclass method yet does the same thing.

The most awkward element of *Pull Up Method* is that the body of the methods may refer to features that are on the subclass but not on the superclass. If the feature is a method, you can either generalize the other method or create an abstract method in the superclass. You may need to change a method's signature or create a delegating method to get this to work.

If you have two methods that are similar but not the same, you may be able to use Form Template Method.

Mechanics

- Inspect the methods to ensure they are identical.

    ?rarr; *If the methods look like they do the same thing but are not identical, use algorithm substitution on one of them to make them identical.*

- If the methods have different signatures, change the signatures to the one you want to use in the superclass.
- Create a new method in the superclass, copy the body of one of the methods to it, adjust, and compile.

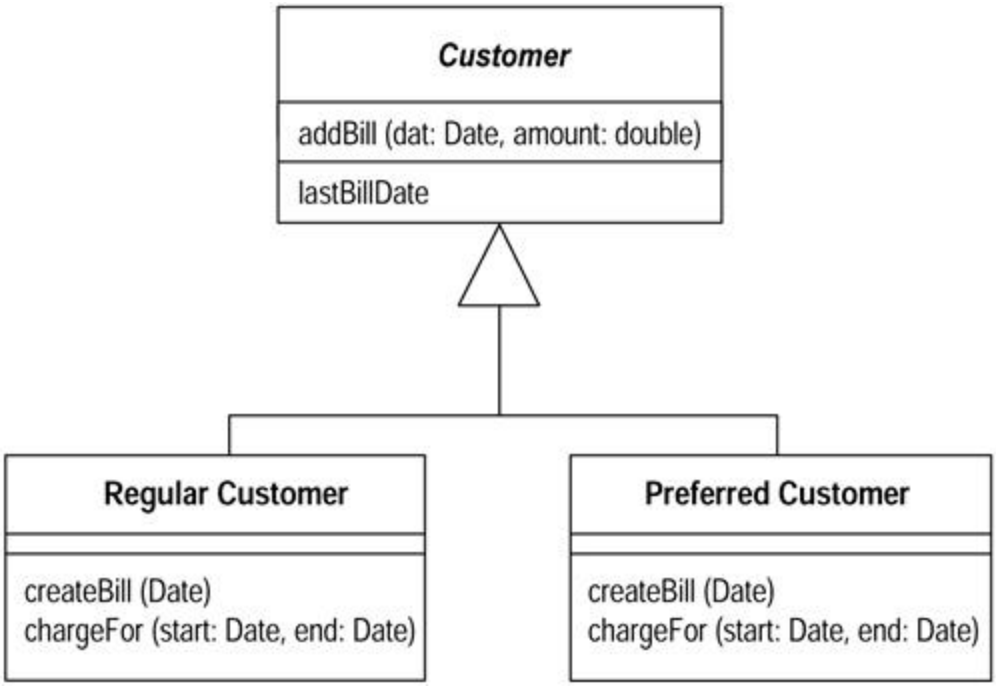
    ?rarr; *If you are in a strongly typed language and the method calls another method that is present on both subclasses but not the superclass, declare an abstract method on the superclass.*

    ?rarr; *If the method uses a subclass field, use* Pull Up Field *or* Self Encapsulate Field *and declare and use an abstract getting method.*

- Delete one subclass method.
- Compile and test.
- Keep deleting subclass methods and testing until only the superclass method remains.
- Take a look at the callers of this method to see whether you can change a required type to the superclass.

Example

Consider a customer with two subclasses: regular customer and preferred customer.



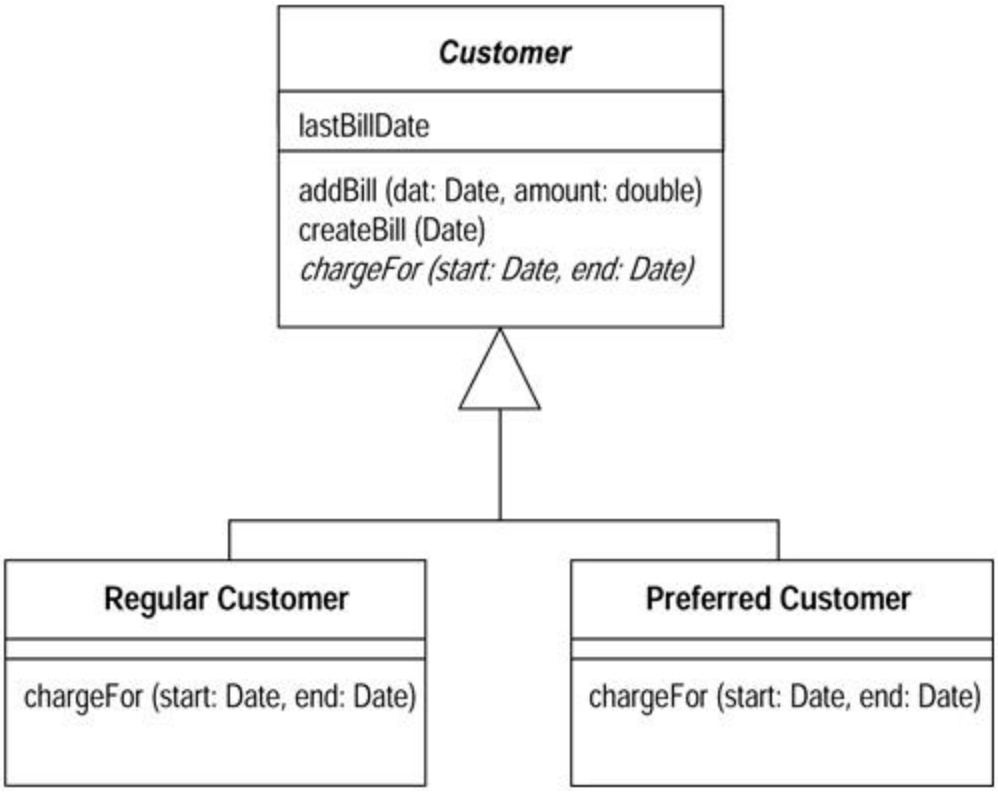
The `createBill` method is identical for each class:

```
void createBill (date Date) {
  double chargeAmount = charge (lastBillDate, date);
  addBill (date, charge);
}
```

I can't move the method up into the superclass, because `chargeFor` is different on each subclass. First I have to declare it on the superclass as abstract:

```
class Customer...
  abstract double chargeFor(date start, date end)
```

Then I can copy `createBill` from one of the subclasses. I compile with that in place and then remove the `createBill` method from one of the subclasses, compile, and test. I then remove it from the other, compile, and test:



Pull Up Constructor Body

You have constructors on subclasses with mostly identical bodies.

Create a superclass constructor; call this from the subclass methods.

```
class Manager extends Employee...
  public Manager (String name, String id, int grade) {
      _name = name;
      _id = id;
      _grade = grade;
  }
```

```
  public Manager (String name, String id, int grade) {
      super (name, id);
      _grade = grade;
  }
```

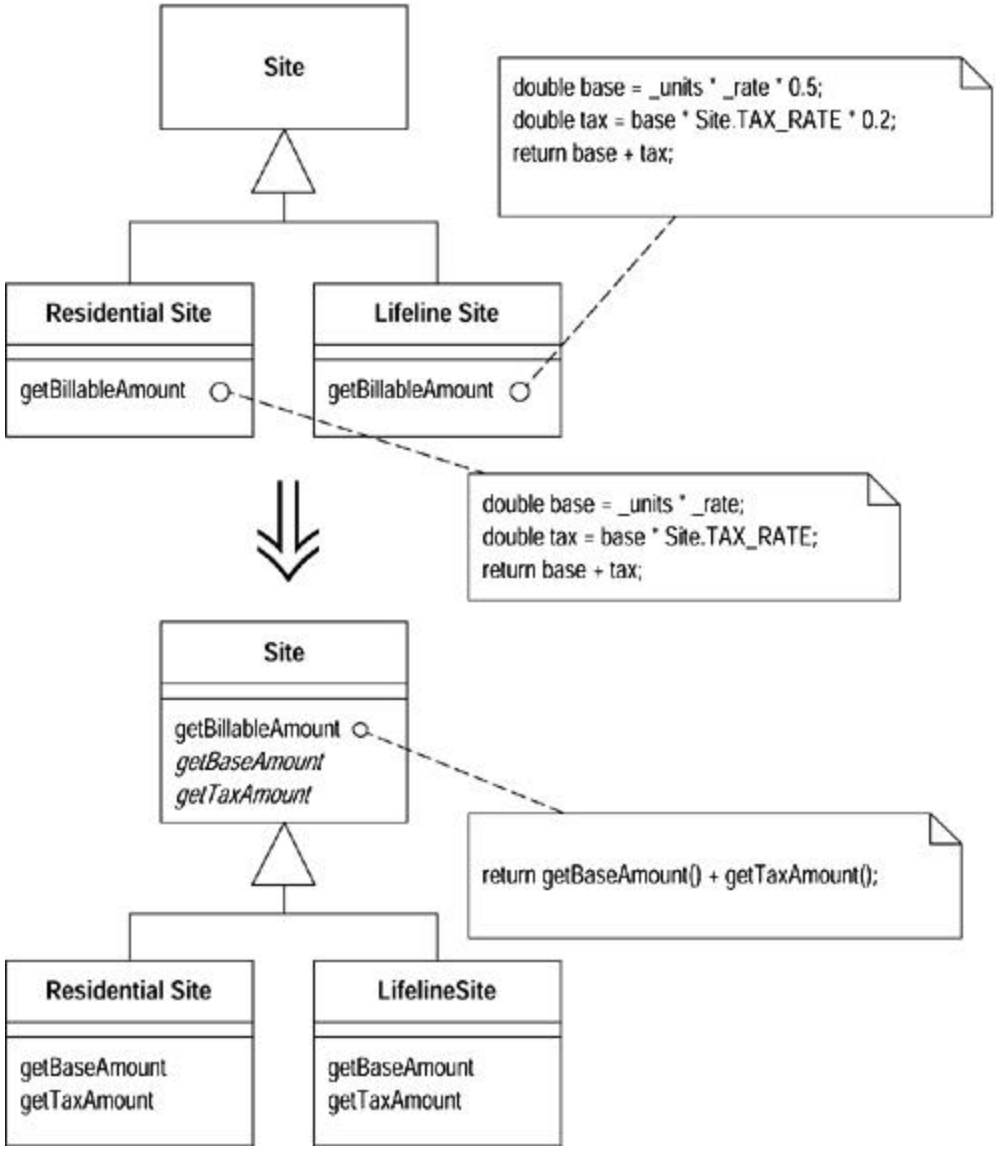
Motivation

- Use Pull Up Field and Pull Up Method or Push Down Method and Push Down Field to move all the behavior and data of the removed class to the class with which it is being merged.
- Compile and test with each move.
- Adjust references to the class that will be removed to use the merged class. This will affect variable declarations, parameter types, and constructors.
- Remove the empty class.
- Compile and test.

Form Template Method

You have two methods in subclasses that perform similar steps in the same order, yet the steps are different.

Get the steps into methods with the same signature, so that the original methods become the same. Then you can pull them up.



Motivation

Inheritance is a powerful tool for eliminating duplicate behavior. Whenever we see two similar methods in a subclass, we want to bring them together in a superclass. But what if they are not exactly the same? What do we do then? We still need to eliminate all the duplication we can but keep the essential differences.

A common case is two methods that seem to carry out broadly similar steps in the same sequence, but the steps are not the same. In this case we can move the sequence to the superclass and allow polymorphism to play its role in ensuring the different steps do their things differently. This kind of method is called a *template method* [Gang of Four].

Mechanics

- Decompose the methods so that all the extracted methods are either identical or completely different.
- Use Pull Up Method to pull the identical methods into the superclass.
- For the different methods use Rename Method so the signatures for all the methods at each step are the same.

> ?rarr; *This makes the original methods the same in that they all issue the same set of method calls, but the subclasses handle the calls differently.*

- Compile and test after each signature change.
- Use Pull Up Method on one of the original methods. Define the signatures of the different methods as abstract methods on the superclass.
- Compile and test.
- Remove the other methods, compile, and test after each removal.

Example

I finish where I left off in Chapter 1. I had a customer class with two methods for printing statements. The `statement` method prints statements in ASCII:

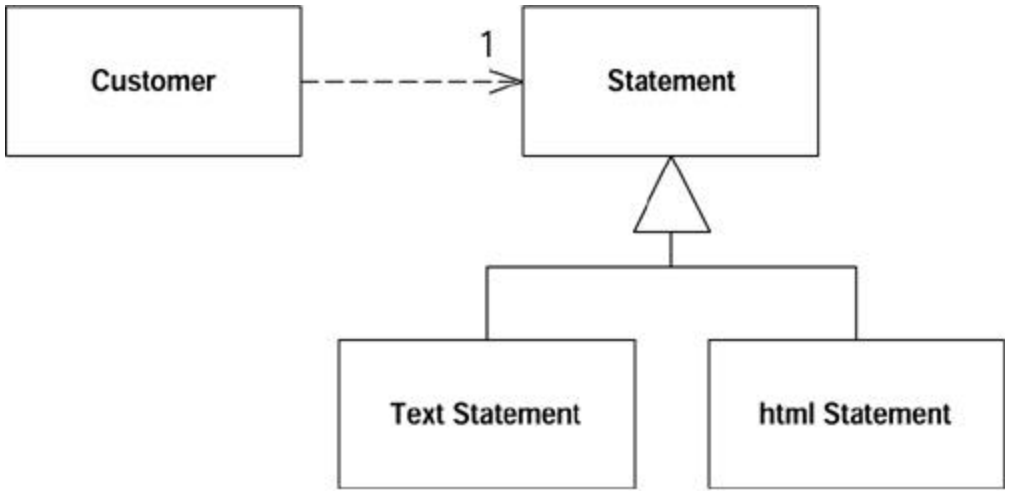
```
    public String statement() {
       Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                String.valueOf(each.getCharge()) + "\n";
        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(getTotalCharge())
+ "\n";
        result += "You earned " +
String.valueOf(getTotalFrequentRenterPoints()) +
             " frequent renter points";
        return result;
    }
```

while the `htmlStatement` does them in HTML:

```
    public String htmlStatement() {
        Enumeration rentals = _rentals.elements();
        String result = "<H1>Rentals for <EM>" + getName() +
"</EM></H1><P>\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            //show figures for each rental
            result += each.getMovie().getTitle()+ ": " +
                String.valueOf(each.getCharge()) + "<BR>\n";
        }
        //add footer lines
        result +=  "<P>You owe <EM>" + String.valueOf(getTotalCharge())
+ "</EM><P>\n";
        result += "On this rental you earned <EM>" +
            String.valueOf(getTotalFrequentRenterPoints()) +
            "</EM> frequent renter points<P>";
        return result;
    }
```

Before I can use Form Template Method I need to arrange things so that the two methods are subclasses of some common superclass. I do this by using a method object [Beck] to create a separate strategy hierarchy for printing the statements (Figure 11.1).

Figure 11.1. Using a strategy for statements




```
class Statement {}
class TextStatement extends Statement {}
class HtmlStatement extends Statement {}
```

Now I use Move Method to move the two statement methods over to the subclasses:

```
  class Customer...
public String statement() {
    return new TextStatement().value(this);
}
public String htmlStatement() {
    return new HtmlStatement().value(this);
}
```

```
 class TextStatement {
   public String value(Customer aCustomer) {
         Enumeration rentals = aCustomer.getRentals();
         String result = "Rental Record for " + aCustomer.getName() +
"\n";
         while (rentals.hasMoreElements()) {
             Rental each = (Rental) rentals.nextElement();

             //show figures for this rental
             result += "\t" + each.getMovie().getTitle()+ "\t" +
                 String.valueOf(each.getCharge()) + "\n";
         }

         //add footer lines
         result +=  "Amount owed is " +
String.valueOf(aCustomer.getTotalCharge()) + "\n";
         result += "You earned " +
String.valueOf(aCustomer.getTotalFrequentRenterPoints()) +
           " frequent renter points";
         return result;
    }
 class HtmlStatement {
   public String value(Customer aCustomer) {
         Enumeration rentals = aCustomer.getRentals();
         String result = "<H1>Rentals for <EM>" + aCustomer.getName() +
"</EM></H1><P>\n";
         while (rentals.hasMoreElements()) {
             Rental each = (Rental) rentals.nextElement();
             //show figures for each rental
             result += each.getMovie().getTitle()+ ": " +
                         String.valueOf(each.getCharge()) + "<BR>\n";
         }
         //add footer lines
         result +=  "<P>You owe <EM>" +
String.valueOf(aCustomer.getTotalCharge()) +
              "</EM><P>\n";
         result += "On this rental you earned <EM>"

String.valueOf(aCustomer.getTotalFrequentRenterPoints()) +
                 "</EM> frequent renter points<P>";
         return result;
 }
```

As I moved them I renamed the statement methods to better fit the strategy. I gave them the same name because the difference between the two now lies in the class rather than the method. (For those trying this from the example, I also had to add a `getRentals` method to customer and relax the visibility of `getTotalCharge` and `getTotalFrequentRenterPoints.`

With two similar methods on subclasses, I can start to use Form Template Method. The key to this refactoring is to separate the varying code from the similar code by using Extract Method to extract the pieces that are different between the two methods. Each time I extract I create methods with different bodies but the same signature.

The first example is the printing of the header. Both methods use the customer to obtain information, but the resulting string is formatted differently. I can extract the formatting of this string into separate methods with the same signature:

```
 class TextStatement...
   String headerString(Customer aCustomer) {
     return "Rental Record for " + aCustomer.getName() + "\n";
   }
   public String value(Customer aCustomer) {
         Enumeration rentals = aCustomer.getRentals();
         String result =headerString(aCustomer);
         while (rentals.hasMoreElements()) {
             Rental each = (Rental) rentals.nextElement();

             //show figures for this rental
             result += "\t" + each.getMovie().getTitle()+ "\t" +
                 String.valueOf(each.getCharge()) + "\n";
         }

         //add footer lines
         result +=  "Amount owed is " +
String.valueOf(aCustomer.getTotalCharge()) + "\n";
         result += "You earned " +
String.valueOf(aCustomer.getTotalFrequentRenterPoints()) +
              " frequent renter points";
         return result;
    }

 class HtmlStatement...
   String headerString(Customer aCustomer) {
         return "<H1>Rentals for <EM>" + aCustomer.getName() +
"</EM></H1><P>\n";
 }
 public String value(Customer aCustomer) {
       Enumeration rentals = aCustomer.getRentals();
       String result = headerString(aCustomer);
       while (rentals.hasMoreElements()) {
           Rental each = (Rental) rentals.nextElement();
           //show figures for each rental
           result += each.getMovie().getTitle()+ ": " +
                       String.valueOf(each.getCharge()) + "<BR>\n";
       }
```

```
       //add footer lines
     result +=  "<P>You owe <EM>" +
String.valueOf(aCustomer.getTotalCharge()) + "</ EM><P>\n";
      result += "On this rental you earned <EM>" +
          String.valueOf(aCustomer.getTotalFrequentRenterPoints()) +
          "</EM> frequent renter points<P>";
      return result;
  }
```

I compile and test and then continue with the other elements. I did the steps one at a time. Here is the result:

```
 class TextStatement …
   public String value(Customer aCustomer) {
       Enumeration rentals = aCustomer.getRentals();
       String result = headerString(aCustomer);
       while (rentals.hasMoreElements()) {
           Rental each = (Rental) rentals.nextElement();
           result += eachRentalString(each);
       }
       result +=  footerString(aCustomer);
       return result;
    }
 String eachRentalString (Rental aRental) {
       return "\t" + aRental.getMovie().getTitle()+ "\t" +
           String.valueOf(aRental.getCharge()) + "\n";
     }
 String footerString (Customer aCustomer) {
         return "Amount owed is " +
String.valueOf(aCustomer.getTotalCharge()) + "\n" +
            "You earned " +
String.valueOf(aCustomer.getTotalFrequentRenterPoints()) +
            " frequent renter points";
     }
 class HtmlStatement…
   public String value(Customer aCustomer) {
         Enumeration rentals = aCustomer.getRentals();
         String result = headerString(aCustomer);
         while (rentals.hasMoreElements()) {
             Rental each = (Rental) rentals.nextElement();
             result += eachRentalString(each);
         }
         result +=  footerString(aCustomer);
         return result;
   }
   String eachRentalString (Rental aRental) {
       return aRental.getMovie().getTitle()+ ": " +
           String.valueOf(aRental.getCharge()) + "<BR>\n";
   }
   String footerString (Customer aCustomer) {
         return "<P>You owe <EM>" +
String.valueOf(aCustomer.getTotalCharge()) +
         "</EM><P>
```

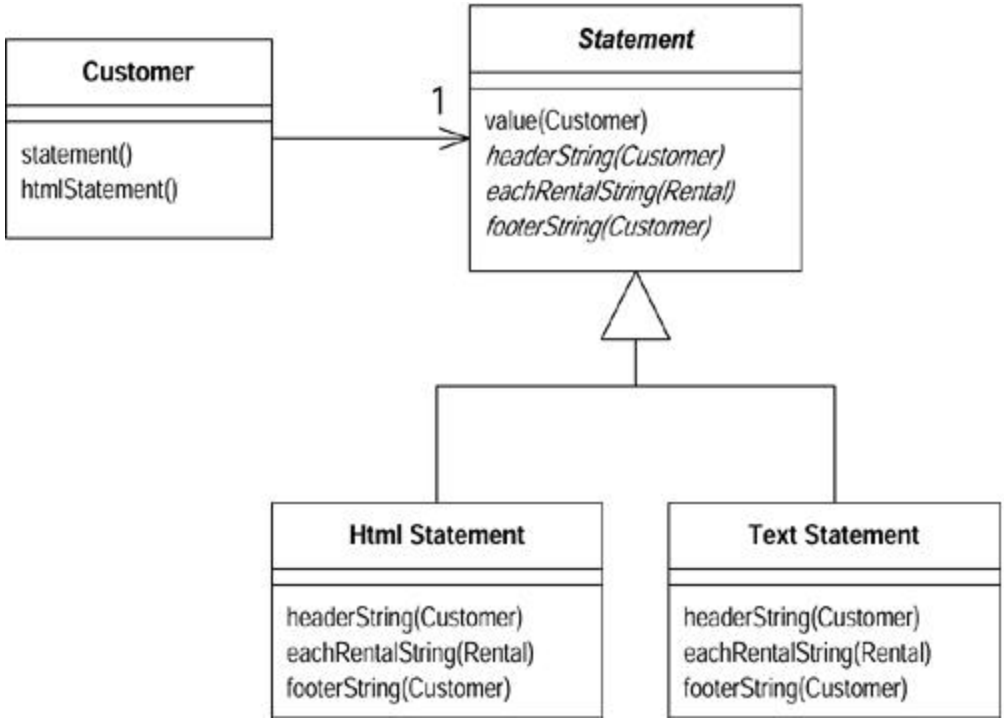
```
" + "On this rental you earned <EM>" +
         String.valueOf(aCustomer.getTotalFrequentRenterPoints()) +
           "</EM> frequent renter points<P>";
   }
```

Once these changes have been made, the two value methods look remarkably similar. So I use Pull Up Method on one of them, picking the text version at random. When I pull up, I need to declare the subclass methods as abstract:

```
class Statement...
  public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
        String result = headerString(aCustomer);
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += eachRentalString(each);
        }
        result +=  footerString(aCustomer);
        return result;
    }
  abstract String headerString(Customer aCustomer);
  abstract String eachRentalString (Rental aRental);
  abstract String footerString (Customer aCustomer);
```

I remove the value method from text statement, compile, and test. When that works I remove the value method from the HTML statement, compile, and test again. The result is shown in Figure 11.2

Figure 11.2. Classes after forming the template method

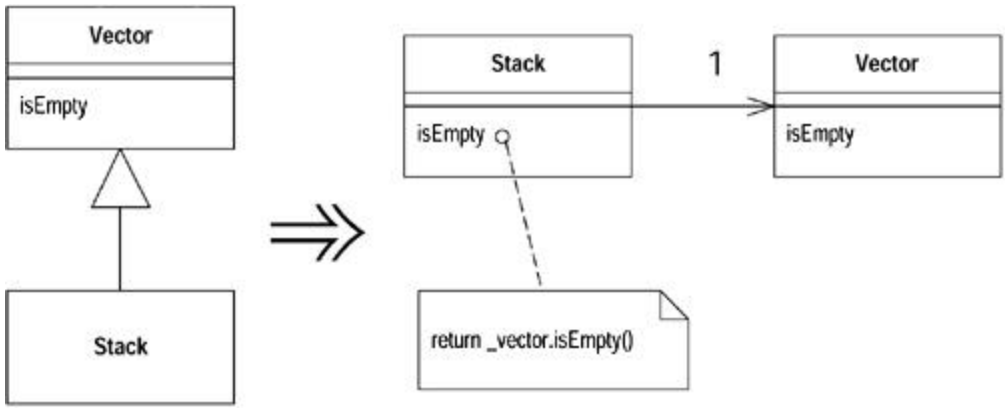


After this refactoring, it is easy to add new kinds of statements. All you have to do is create a subclass of statement that overrides the three abstract methods.

Replace Inheritance with Delegation

A subclass uses only part of a superclasses interface or does not want to inherit data.

Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.



Motivation

Inheritance is a wonderful thing, but sometimes it isn't what you want. Often you start inheriting from a class but then find that many of the superclass operations aren't really true of the subclass. In this case you have an interface that's not a true reflection of what the class does. Or

```
        nextDateReplace(d2);
        System.out.println ("d2 after nextDay: " + d2);
    }

    private static void nextDateUpdate (Date arg) {
        arg.setDate(arg.getDate() + 1);
        System.out.println ("arg in nextDay: " + arg);
    }

    private static void nextDateReplace (Date arg) {
        arg = new Date (arg.getYear(), arg.getMonth(), arg.getDate() +
1);
        System.out.println ("arg in nextDay: " + arg);
    }
  }
```

It produces this output

```
  arg in nextDay: Thu Apr 02 00:00:00 EST 1998
  d1 after nextDay: Thu Apr 02 00:00:00 EST 1998
  arg in nextDay: Thu Apr 02 00:00:00 EST 1998
  d2 after nextDay: Wed Apr 01 00:00:00 EST 1998
```

Essentially the object reference is passed by value. This allows me to modify the object but does not take into account the reassigning of the parameter.

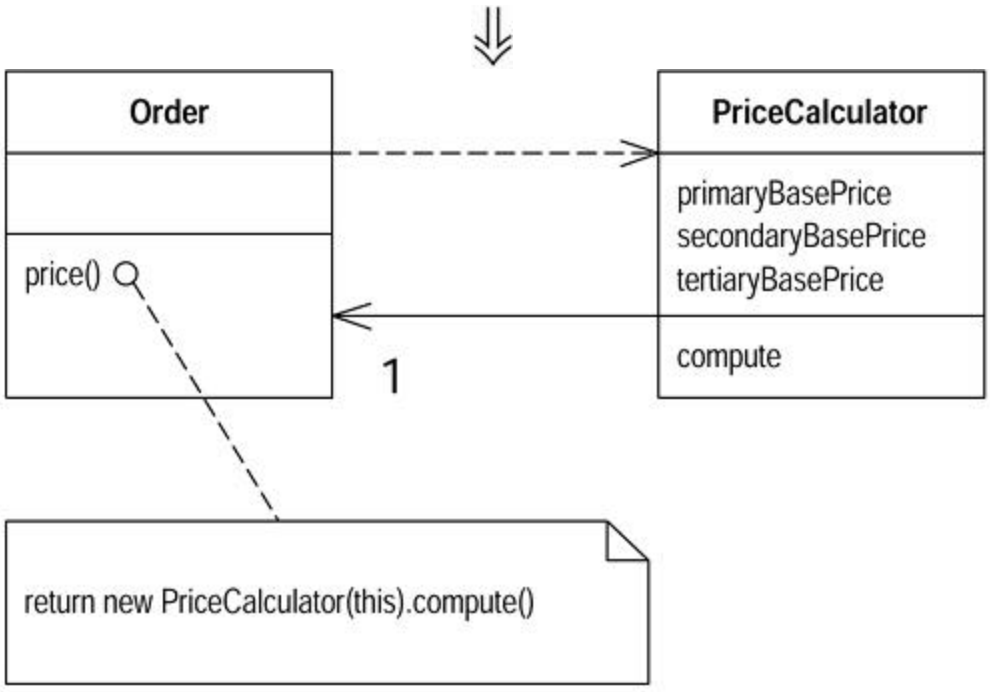
Java 1.1 and later versions allow you to mark a parameter as *final;* this prevents assignment to the variable. It still allows you to modify the object the variable refers to. I always treat my parameters as final, but I confess I rarely mark them so in the parameter list.

Replace Method with Method Object

You have a long method that uses local variables in such a way that you cannot apply Extract Method.

Turn the method into its own object so that all the local variables become fields on that object. You can then decompose the method into other methods on the same object.

```
  class Order...
    double price() {
    double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // long computation;
        ...
    }
```



Motivation

In this book I emphasize the beauty of small methods. By extracting pieces out of a large method, you make things much more comprehensible.

The difficulty in decomposing a method lies in local variables. If they are rampant, decomposition can be difficult. Using Replace Temp with Query helps to reduce this burden, but occasionally you may find you cannot break down a method that needs breaking. In this case you reach deep into the tool bag and get out your *method object* [Beck].

Applying Replace Method with Method Object turns all these local variables into fields on the method object. You can then use Extract Method on this new object to create additional methods that break down the original method.

Mechanics

Stolen shamelessly from Beck [Beck].

- Create a new class, name it after the method.
- Give the new class a final field for the object that hosted the original method (the source object) and a field for each temporary variable and each parameter in the method.
- Give the new class a constructor that takes the source object and each parameter.
- Give the new class a method named "compute."
- Copy the body of the original method into compute. Use the source object field for any invocations of methods on the original object.
- Compile.
- Replace the old method with one that creates the new object and calls compute.

Now comes the fun part. Because all the local variables are now fields, you can freely decompose the method without having to pass any parameters.

Example

A proper example of this requires a long chapter, so I'm showing this refactoring for a method that doesn't need it. (Don't ask what the logic of this method is, I made it up as I went along.)

```
  Class Account
    int gamma (int inputVal, int quantity, int yearToDate) {
        int importantValue1 = (inputVal * quantity) + delta();
        int importantValue2 = (inputVal * yearToDate) + 100;
        if ((yearToDate - importantValue1) > 100)
            importantValue2 -= 20;
        int importantValue3 = importantValue2 * 7;
        // and so on.
        return importantValue3 - 2 * importantValue1;
    }
```

To turn this into a method object, I begin by declaring a new class. I provide a final field for the original object and a field for each parameter and temporary variable in the method.

```
  class Gamma...
    private final Account _account;
    private int inputVal;
    private int quantity;
    private int yearToDate;
    private int importantValue1;
    private int importantValue2;
    private int importantValue3;
```

I usually use the underscore prefix convention for marking fields. But to keep small steps I'll leave the names as they are for the moment.

I add a constructor:

```
    Gamma (Account source, int inputValArg, int quantityArg, int
yearToDateArg) {
        _account = source;
        inputVal = inputValArg;
        quantity = quantityArg;
        yearToDate = yearToDateArg;
    }
```

Now I can move the original method over. I need to modify any calls of features of account to use the `_account` field

```
    int compute () {
        importantValue1 = (inputVal * quantity) + _account.delta();
        importantValue2 = (inputVal * yearToDate) + 100;
        if ((yearToDate - importantValue1) > 100)
            importantValue2 -= 20;
        int importantValue3 = importantValue2 * 7;
        // and so on.
        return importantValue3 - 2 * importantValue1;
    }
```

I then modify the old method to delegate to the method object:

```
    int gamma (int inputVal, int quantity, int yearToDate) {
        return new Gamma(this, inputVal, quantity,
yearToDate).compute();
    }
```

That's the essential refactoring. The benefit is that I can now easily use Extract Method on the compute method without ever worrying about the argument's passing:

```
   int compute () {
       importantValue1 = (inputVal * quantity) + _account.delta();
       importantValue2 = (inputVal * yearToDate) + 100;
       importantThing();
       int importantValue3 = importantValue2 * 7;
       // and so on.
       return importantValue3 - 2 * importantValue1;
   }

   void importantThing() {
       if ((yearToDate - importantValue1) > 100)
           importantValue2 -= 20;
   }
```

Substitute Algorithm

You want to replace an algorithm with one that is clearer.

Replace the body of the method with the new algorithm.

```
    String foundPerson(String[] people){
        for (int i = 0; i < people.length; i++) {
            if (people[i].equals ("Don")){
                return "Don";
            }
            if (people[i].equals ("John")){
                return "John";
```

```
            }
            if (people[i].equals ("Kent")){
                return "Kent";
            }
        }
        return "";
    }
```

```
    String foundPerson(String[] people){
        List candidates = Arrays.asList(new String[] {"Don", "John",
"Kent"});
        for (int i=0; i<people.length; i++)
            if (candidates.contains(people[i]))
                return people[i];
        return "";
    }
```

Motivation

I've never tried to skin a cat. I'm told there are several ways to do it. I'm sure some are easier than others. So it is with algorithms. If you find a clearer way to do something, you should replace the complicated way with the clearer way. Refactoring can break down something complex into simpler pieces, but sometimes you just reach the point at which you have to remove the whole algorithm and replace it with something simpler. This occurs as you learn more about the problem and realize that there's an easier way to do it. It also happens if you start using a library that supplies features that duplicate your code.

Sometimes when you want to change the algorithm to do something slightly different, it is easier to subtitute the algorithm first into something easier for the change you need to make.

When you have to take this step, make sure you have decomposed the method as much as you can. Substituting a large, complex algorithm is very difficult; only by making it simple can you make the substitution tractable.

Mechanics

- Prepare your alternative algorithm. Get it so that it compiles.
- Run the new algorithm against your tests. If the results are the same, you're finished.
- If the results aren't the same, use the old algorithm for comparison in testing and debugging.

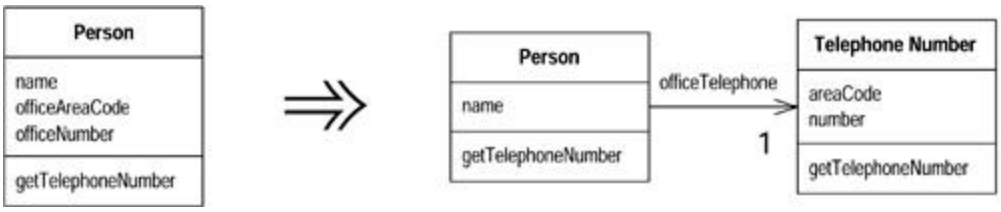
> *? Run each test case with old and new algorithms and watch both results. That will help you see which test cases are causing trouble, and how.*

I can redirect the clients of the accessors to use the new object later if I want. Using self-encapsulation allows me to take a smaller step. This is useful if I'm doing a lot of things with the class. In particular, it simplifies use Move Method to move methods to the target class. If they refer to the accessor, such references don't need to change.

Extract Class

You have one class doing work that should be done by two.

Create a new class and move the relevant fields and methods from the old class into the new class.



Motivation

You've probably heard that a class should be a crisp abstraction, handle a few clear responsibilities, or some similar guideline. In practice, classes grow. You add some operations here, a bit of data there. You add a responsibility to a class feeling that it's not worth a separate class, but as that responsibility grows and breeds, the class becomes too complicated. Soon your class is as crisp as a microwaved duck.

Such a class is one with many methods and quite a lot of data. A class that is too big to understand easily. You need to consider where it can be split, and you split it. A good sign is that a subset of the data and a subset of the methods seem to go together. Other good signs are subsets of data that usually change together or are particularly dependent on each other. A useful test is to ask yourself what would happen if you removed a piece of data or a method. What other fields and methods would become nonsense?

One sign that often crops up later in development is the way the class is subtyped. You may find that subtyping affects only a few features or that some features need to be subtyped one way and other features a different way.

Mechanics

- Decide how to split the responsibilities of the class.
- Create a new class to express the split-off responsibilities.

  ?rarr; *If the responsibilities of the old class no longer match its name, rename the old class.*

- Make a link from the old to the new class.

  ?rarr; *You may need a two-way link. But don't make the back link until you find you need it.*

- Use Move Field on each field you wish to move.
- Compile and test after each move.
- Use Move Method to move methods over from old to new. Start with lower-level methods (called rather than calling) and build to the higher level.
- Compile and test after each move.
- Review and reduce the interfaces of each class.

> ?rarr; *If you did have a two-way link, examine to see whether it can be made one way.*

- Decide whether to expose the new class. If you do expose the class, decide whether to expose it as a reference object or as an immutable value object.

Example

I start with a simple person class:

```
class Person...
  public String getName() {
      return _name;
  }
  public String getTelephoneNumber() {
      return ("(" + _officeAreaCode + ") " + _officeNumber);
  }
  String getOfficeAreaCode() {
      return _officeAreaCode;
  }
  void setOfficeAreaCode(String arg) {
      _officeAreaCode = arg;
  }
  String getOfficeNumber() {
      return _officeNumber;
  }
  void setOfficeNumber(String arg) {
      _officeNumber = arg;
  }

  private String _name;
  private String _officeAreaCode;
  private String _officeNumber;
```

In this case I can separate the telephone number behavior into its own class. I start by defining a telephone number class:

```
class TelephoneNumber {
}
```

That was easy! I next make a link from the person to the telephone number:

```
class Person
  private TelephoneNumber _officeTelephone = new TelephoneNumber();
```

Now I use Move Field on one of the fields:

```
class TelephoneNumber {
  String getAreaCode() {
      return _areaCode;
  }
  void setAreaCode(String arg) {
      _areaCode = arg;
  }
  private String _areaCode;
}
class Person...
  public String getTelephoneNumber() {
      return ("(" + getOfficeAreaCode() + ") " + _officeNumber);
  }
  String getOfficeAreaCode() {
      return _officeTelephone.getAreaCode();
  }
  void setOfficeAreaCode(String arg) {
      _officeTelephone.setAreaCode(arg);
  }
```

I can then move the other field and use Move Method on the telephone number:

```
class Person...
  public String getName() {
      return _name;
  }
  public String getTelephoneNumber(){
      return _officeTelephone.getTelephoneNumber();
  }
  TelephoneNumber getOfficeTelephone() {
      return _officeTelephone;
  }

  private String _name;
  private TelephoneNumber _officeTelephone = new TelephoneNumber();
class TelephoneNumber...
  public String getTelephoneNumber() {
      return ("(" + _areaCode + ") " + _number);
  }
  String getAreaCode() {
      return _areaCode;
  }
  void setAreaCode(String arg) {
      _areaCode = arg;
  }
  String getNumber() {
      return _number;
  }
  void setNumber(String arg) {
      _number = arg;
  }
  private String _number;
```

```
    private String _areaCode;
```

The decision then is how much to expose the new class to my clients. I can completely hide it by providing delegating methods for its interface, or I can expose it. I may choose to expose it to some clients (such as those in my package) but not to others.

If I choose to expose the class, I need to consider the dangers of aliasing. If I expose the telephone number and a client changes the area code in that object, how do I feel about it? It may not be a direct client that makes this change. It might be the client of a client of a client.

I have the following options:

1. I accept that any object may change any part of the telephone number. This makes the telephone number a reference object, and I should consider Change Value to Reference. In this case the person would be the access point for the telephone number.
2. I don't want anybody to change the value of the telephone number without going through the person. I can either make the telephone number immutable, or I can provide an immutable interface for the telephone number.
3. Another possibility is to clone the telephone number before passing it out. But this can lead to confusion because people think they can change the value. It also may lead to aliasing problems between clients if the telephone number is passed around a lot.

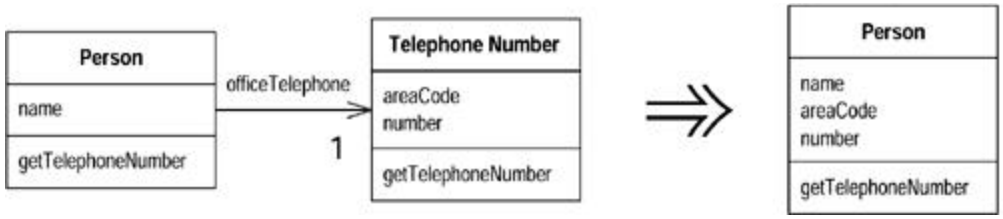
Extract Class is a common technique for improving the liveness of a concurrent program because it allows you to have separate locks on the two resulting classes. If you don't need to lock both objects you don't have to. For more on this see section 3.3 in Lea [Lea].

However, there is a danger there. If you need to ensure that both objects are locked together, you get into the area of transactions and other kinds of shared locks. As discussed in Lea by section 8.1 [Lea], this is complex territory and requires heavier machinery than it is typically worth. Transactions are very useful when you use them, but writing transaction managers is more than most programmers should attempt.

Inline Class

A class isn't doing very much.

Move all its features into another class and delete it.



Motivation

Inline Class is the reverse of Extract Class. I use *Inline Class* if a class is no longer pulling its weight and shouldn't be around any more. Often this is the result of refactoring that moves other responsibilities out of the class so there is little left. Then I want to fold this class into another class, picking one that seems to use the runt class the most.

You have a temp that is assigned to once with a simple expression, and the temp is getting in the way of other refactorings.

Replace all references to that temp with the expression.

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000)
```

```
return (anOrder.basePrice() > 1000)
```

Motivation

Most of the time *Inline Temp* is used as part of Replace Temp with Query, so the real motivation is there. The only time *Inline Temp* is used on its own is when you find a temp that is assigned the value of a method call. Often this temp isn't doing any harm and you can safely leave it there. If the temp is getting in the way of other refactorings, such as Extract Method, it's time to inline it.

Mechanics

- Declare the temp as final if it isn't already, and compile.

  ? *This checks that the temp is really only assigned to once.*

- Find all references to the temp and replace them with the right-hand side of the assignment.
- Compile and test after each change.
- Remove the declaration and the assignment of the temp.
- Compile and test.

Replace Temp with Query

You are using a temporary variable to hold the result of an expression.

Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods.

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```

```
    if (basePrice() > 1000)
        return basePrice() * 0.95;
    else
        return basePrice() * 0.98;
...
   double basePrice() {
       return _quantity * _itemPrice;
   }
```

Motivation

The problem with temps is that they are temporary and local. Because they can be seen only in the context of the method in which they are used, temps tend to encourage longer methods, because that's the only way you can reach the temp. By replacing the temp with a query method, any method in the class can get at the information. That helps a lot in coming up with cleaner code for the class.

Replace Temp with Query often is a vital step before Extract Method. Local variables make it difficult to extract, so replace as many variables as you can with queries.

The straightforward cases of this refactoring are those in which temps are assigned only to once and those in which the expression that generates the assignment is free of side effects. Other cases are trickier but possible. You may need to use Split Temporary Variable or Separate Query from Modifier first to make things easier. If the temp is used to collect a result (such as summing over a loop), you need to copy some logic into the query method.

Mechanics

Here is the simple case:

- Look for a temporary variable that is assigned to once.

  ? *If a temp is set more than once consider* Split Temporary Variable.

- Declare the temp as final.
- Compile.

  ? *This will ensure that the temp is only assigned to once.*

- Extract the right-hand side of the assignment into a method.

  ? *Initially mark the method as private. You may find more use for it later, but you can easily relax the protection later.*

  ? *Ensure the extracted method is free of side effects, that is, it does not modify any object. If it is not free of side effects, use* Separate Query from Modifier.

- Compile and test.
- Use Replace Temp with Query on the temp.

Temps often are used to store summary information in loops. The entire loop can be extracted into a method; this removes several lines of noisy code. Sometimes a loop may be used to sum up multiple values, as in the example on page 26. In this case, duplicate the loop for each temp so that you can replace each temp with a query. The loop should be very simple, so there is little danger in duplicating the code.

You may be concerned about performance in this case. As with other performance issues, let it slide for the moment. Nine times out of ten, it won't matter. When it does matter, you will fix the problem during optimization. With your code better factored, you will often find more powerful optimizations, which you would have missed without refactoring. If worse comes to worse, it's very easy to put the temp back.

Example

I start with a simple method:

```
double getPrice() {
    int basePrice = _quantity * _itemPrice;
    double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

I'm inclined to replace both temps, one at a time.

Although it's pretty clear in this case, I can test that they are assigned only to once by declaring them as final:

```
double getPrice() {
    final int basePrice = _quantity * _itemPrice;
    final double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

Compiling will then alert me to any problems. I do this first, because if there is a problem, I shouldn't be doing this refactoring. I replace the temps one at a time. First I extract the right-hand side of the assignment:

```
double getPrice() {
    final int basePrice = basePrice();
    final double discountFactor;
```

```
        if (basePrice > 1000) discountFactor = 0.95;
        else discountFactor = 0.98;
        return basePrice * discountFactor;
    }

    private int basePrice() {
    return _quantity * _itemPrice;
    }
```

I compile and test, then I begin with Replace Temp with Query. First I replace the first reference to the temp:

```
    double getPrice() {
        final int basePrice = basePrice();
        final double discountFactor;
        if (basePrice() > 1000) discountFactor = 0.95;
        else discountFactor = 0.98;
        return basePrice * discountFactor;
    }
```

Compile and test and do the next (sounds like a caller at a line dance). Because it's the last, I also remove the temp declaration:

```
    double getPrice() {
        final double discountFactor;
        if (basePrice() > 1000) discountFactor = 0.95;
        else discountFactor = 0.98;
        return basePrice() * discountFactor;
    }
```

With that gone I can extract discountFactor in a similar way:

```
    double getPrice() {
        final double discountFactor = discountFactor();
        return basePrice() * discountFactor;
    }

    private double discountFactor() {
        if (basePrice() > 1000) return 0.95;
        else return 0.98;
    }
```

See how it would have been difficult to extract `discountFactor` if I had not replaced `basePrice` with a query.

The `getPrice` method ends up as follows:

```
    double getPrice() {
        return basePrice() * discountFactor();
    }
```

Introduce Explaining Variable

You have a complicated expression.

Put the result of the expression, or parts of the expression, in a temporary variable with a name

that explains the purpose.

```
  if ( (platform.toUpperCase().indexOf("MAC") > -1) &&
       (browser.toUpperCase().indexOf("IE") > -1) &&
        wasInitialized() && resize > 0 )
{
       // do something
}
```

```
    final boolean isMacOs     = platform.toUpperCase().indexOf("MAC") >
-1;
    final boolean isIEBrowser = browser.toUpperCase().indexOf("IE")  >
-1;
    final boolean wasResized  = resize > 0;

    if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {
        // do something
    }
```

Motivation

Expressions can become very complex and hard to read. In such situations temporary variables can be helpful to break down the expression into something more manageable.

Introduce Explaining Variable is particularly valuable with conditional logic in which it is useful to take each clause of a condition and explain what the condition means with a well-named temp. Another case is a long algorithm, in which each step in the computation can be explained with a temp.

Introduce Explaining Variable is a very common refactoring, but I confess I don't use it that much. I almost always prefer to use Extract Method if I can. A temp is useful only within the context of one method. A method is useable throughout the object and to other objects. There are times,

```
public double getPrice() {
    return discountedPrice ();
}

private double discountedPrice () {
    if (getDiscountLevel() == 2) return getBasePrice() * 0.1;
    else return getBasePrice() * 0.05;
}

private double getBasePrice() {
    return _quantity * _itemPrice;
}
```

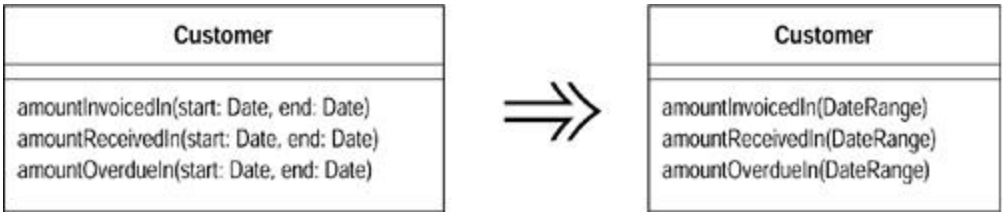
so I might as well use Inline Method on `discountedPrice:`

```
private double getPrice () {
    if (getDiscountLevel() == 2) return getBasePrice() * 0.1;
    else return getBasePrice() * 0.05;
}
```

Introduce Parameter Object

You have a group of parameters that naturally go together.

Replace them with an object.



Motivation

Often you see a particular group of parameters that tend to be passed together. Several methods may use this group, either on one class or in several classes. Such a group of classes is a data clump and can be replaced with an object that carries all of this data. It is worthwhile to turn these parameters into objects just to group the data together. This refactoring is useful because it reduces the size of the parameter lists, and long parameter lists are hard to understand. The defined accessors on the new object also make the code more consistent, which again makes it easier to understand and modify.

You get a deeper benefit, however, because once you have clumped together the parameters, you soon see behavior that you can also move into the new class. Often the bodies of the methods have common manipulations of the parameter values. By moving this behavior into the new object, you can remove a lot of duplicated code.

Mechanics

- Create a new class to represent the group of parameters you are replacing. Make the class immutable.
- Compile.
- Use Add Parameter for the new data clump. Use a null for this parameter in all the callers.

> ?rarr; *If you have many callers, you can retain the old signature and let it call the new method. Apply the refactoring on the old method first. You can then move the callers over one by one and remove the old method when you're done.*

- For each parameter in the data clump, remove the parameter from the signature. Modify the callers and method body to use the parameter object for that value.
- Compile and test after you remove each parameter.
- When you have removed the parameters, look for behavior that you can move into the parameter object with Move Method.

> ?rarr; *This may be a whole method or part of a method. If it is part of a method, use Extract Method first and then move the new method over.*

Example

I begin with an account and entries. The entries are simple data holders.

```
class Entry...
  Entry (double value, Date chargeDate) {
      _value = value;
      _chargeDate = chargeDate;
  }
  Date getDate(){
      return _chargeDate;
  }
  double getValue(){
      return _value;
  }
  private Date _chargeDate;
  private double _value;
```

My focus is on the account, which holds a collection of entries and has a method for determining the flow of the account between two dates:

```
class Account...
  double getFlowBetween (Date start, Date end) {
      double result = 0;
      Enumeration e = _entries.elements();
      while (e.hasMoreElements()) {
          Entry each = (Entry) e.nextElement();
          if (each.getDate().equals(start) ||
              each.getDate().equals(end) ||
               (each.getDate().after(start) &&
each.getDate().before(end)))
          {
```

```
                result += each.getValue();
            }
        }
        return result;
    }
    private Vector _entries = new Vector();

  client code...
    double flow = anAccount.getFlowBetween(startDate, endDate);
```

I don't know how many times I come across pairs of values that show a range, such as start and end dates and upper and lower numbers. I can understand why this happens, after all I did it all the time myself. But since I saw the range pattern [Fowler, AP] I always try to use ranges instead. My first step is to declare a simple data holder for the range:

```
  class DateRange {
    DateRange (Date start, Date end) {
        _start = start;
        _end = end;
    }
    Date getStart() {
        return _start;
    }
    Date getEnd() {
        return _end;
    }
    private final Date _start;
    private final Date _end;
  }
```

I've made the date range class immutable; that is, all the values for the date range are final and set in the constructor, hence there are no methods for modifying the values. This is a wise move to avoid aliasing bugs. Because Java has pass-by-value parameters, making the class immutable mimics the way Java's parameters work, so this is the right assumption for this refactoring.

Next I add the date range into the parameter list for the `getFlowBetween` method:

```
  class Account...
    double getFlowBetween (Date start, Date end, DateRange range) {
        double result = 0;
        Enumeration e = _entries.elements();
        while (e.hasMoreElements()) {
            Entry each = (Entry) e.nextElement();
            if (each.getDate().equals(start) ||
                each.getDate().equals(end) ||
                (each.getDate().after(start) &&
each.getDate().before(end)))
            {
                result += each.getValue();
            }
        }
        return result;
    }
```

```
  client code...
    double flow = anAccount.getFlowBetween(startDate, endDate, null);
```

At this point I only need to compile, because I haven't altered any behavior yet.

The next step is to remove one of the parameters and use the new object instead. To do this I delete the start parameter and modify the method and its callers to use the new object instead:

```
  class Account...
    double getFlowBetween (Date end, DateRange range) {
        double result = 0;
        Enumeration e = _entries.elements();
        while (e.hasMoreElements()) {
            Entry each = (Entry) e.nextElement();
            if (each.getDate().equals(range.getStart()) ||
                each.getDate().equals(end) ||
                (each.getDate().after(range.getStart()) &&
each.getDate().before(end)))
            {
                result += each.getValue();
            }
        }
        return result;
    }

  client code...
    double flow = anAccount.getFlowBetween(endDate, new DateRange
(startDate, null));
```

I then remove the end date:

```
  class Account...
    double getFlowBetween (DateRange range) {
        double result = 0;
        Enumeration e = _entries.elements();
        while (e.hasMoreElements()) {
            Entry each = (Entry) e.nextElement();
            if (each.getDate().equals(range.getStart()) ||
                each.getDate().equals(range.getEnd()) ||
                (each.getDate().after(range.getStart()) &&
each.getDate().before(range.getEnd())))
            {
                result += each.getValue();
            }
        }
        return result;
    }

  client code...
        double flow = anAccount.getFlowBetween(new DateRange
(startDate, endDate));
```

I have introduced the parameter object; however, I can get more value from this refactoring by moving behavior from other methods to the new object. In this case I can take the code in the condition and use Extract Method and Move Method to get

```
class Account...
  double getFlowBetween (DateRange range) {
      double result = 0;
      Enumeration e = _entries.elements();
      while (e.hasMoreElements()) {
          Entry each = (Entry) e.nextElement();
          if (range.includes(each.getDate())) {
              result += each.getValue();
          }
      }
      return result;
  }

class DateRange...
  boolean includes (Date arg) {
      return (arg.equals(_start) ||
              arg.equals(_end) ||
               (arg.after(_start) && arg.before(_end)));
  }
```

I usually do simple extracts and moves such as this in one step. If I run into a bug, I can back out and take the two smaller steps.

Remove Setting Method

A field should be set at creation time and never altered.

Remove any setting method for that field.

Motivation

Providing a setting method indicates that a field may be changed. If you don't want that field to change once the object is created, then don't provide a setting method (and make the field final). That way your intention is clear and you often remove the very possibility that the field will change.

This situation often occurs when programmers blindly use indirect variable access [Beck]. Such programmers then use setters even in a constructor. I guess there is an argument for consistency but not compared with the confusion that the setting method will cause later on.

Mechanics

```
   static Employee create(int type) {
       switch (type) {
           case ENGINEER:
              return Employee.createEngineer();
           case SALESMAN:
              return new Salesman();
           case MANAGER:
              return new Manager();
           default:
              throw new IllegalArgumentException("Incorrect type code
value");
       }
   }
```

I compile and test after changing each leg, until I've replaced them all:

```
   static Employee create(int type) {
       switch (type) {
           case ENGINEER:
              return Employee.createEngineer();
           case SALESMAN:
              return Employee.createSalesman();
           case MANAGER:
              return Employee.createManager();
           default:
              throw new IllegalArgumentException("Incorrect type code
value");
       }
   }
```

Now I move on to the callers of the old create method. I change code such as

```
  Employee kent = Employee.create(ENGINEER)
```

to

```
  Employee kent = Employee.createEngineer()
```

Once I've done that for all the callers of `create`, I can remove the `create` method. I may also be able to get rid of the constants.

Preserve Whole Object

You are getting several values from an object and passing these values as parameters in a method call.

Send the whole object instead.

```
      int low = daysTempRange().getLow();
```

```
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

```
withinPlan = plan.withinRange(daysTempRange());
```

Motivation

This type of situation arises when an object passes several data values from a single object as parameters in a method call. The problem with this is that if the called object needs new data values later, you have to find and change all the calls to this method. You can avoid this by passing in the whole object from which the data came. The called object then can ask for whatever it wants from the whole object.

In addition to making the parameter list more robust to changes, *Preserve Whole Object* often makes the code more readable. Long parameter lists can be hard to work with because both caller and callee have to remember which values were there. They also encourage duplicate code because the called object can't take advantage of any other methods on the whole object to calculate intermediate values.

There is a down side. When you pass in values, the called object has a dependency on the values, but there isn't any dependency to the object from which the values were extracted. Passing in the required object causes a dependency between the required object and the called object. If this is going to mess up your dependency structure, don't use *Preserve Whole Object.*

Another reason I have heard for not using *Preserve Whole Object* is that when a calling object need only one value from the required object, it is better to pass in the value than to pass in the whole object. I don't subscribe to that view. One value and one object amount to the same thing when you pass them in, at least for clarity's sake (there may be a performance cost with pass by value parameters). The driving force is the dependency issue.

That a called method uses lots of values from another object is a signal that the called method should really be defined on the object from which the values come. When you are considering *Preserve Whole Object,* consider Move Method as an alternative.

You may not already have the whole object defined. In this case you need Introduce Parameter Object.

A common case is that a calling object passes several of its *own* data values as parameters. In this case you can make the call and pass in `this` instead of these values, if you have the appropriate getting methods and you don't mind the dependency.

Mechanics

- Create a new parameter for the whole object from which the data comes.
- Compile and test.
- Determine which parameters should be obtained from the whole object.
- Take one parameter and replace references to it within the method body by invoking an appropriate method on the whole object parameter.
- Delete the parameter.
- Compile and test.
- Repeat for each parameter that can be got from the whole object.

- Remove the code in the calling method that obtains the deleted parameters.

  ?rarr; *Unless, of course, the code is using these parameters somewhere else.*

- Compile and test.

Example

Consider a room object that records high and low temperatures during the day. It needs to compare this range with a range in a predefined heating plan:

```
class Room...
  boolean withinPlan(HeatingPlan plan) {
      int low = daysTempRange().getLow();
      int high = daysTempRange().getHigh();
      return plan.withinRange(low, high);
  }
class HeatingPlan...
  boolean withinRange (int low, int high) {
      return (low >= _range.getLow() && high <= _range.getHigh());
  }
  private TempRange _range;
```

Rather than unpack the range information when I pass it, I can pass the whole range object. In this simple case I can do this in one step. When more parameters are involved, I can do it in smaller steps. First I add the whole object to the parameter list:

```
class HeatingPlan...
  boolean withinRange (TempRange roomRange, int low, int high) {
      return (low >= _range.getLow() && high <= _range.getHigh());
  }

class Room...
  boolean withinPlan(HeatingPlan plan) {
      int low = daysTempRange().getLow();
      int high = daysTempRange().getHigh();
      return plan.withinRange(daysTempRange(), low, high);
  }
```

Then I use a method on the whole object instead of one of the parameters:

```
class HeatingPlan...
  boolean withinRange (TempRange roomRange, int high) {
      return (roomRange.getLow() >= _range.getLow() && high <=
_range.getHigh());
  }

class Room...
  boolean withinPlan(HeatingPlan plan) {
      int low = daysTempRange().getLow();
```

```
        int high = daysTempRange().getHigh();
        return plan.withinRange(daysTempRange(), high);
    }
```

I continue until I've changed all I need:

```
  class HeatingPlan...
    boolean withinRange (TempRange roomRange) {
        return (roomRange.getLow() >= _range.getLow() &&
roomRange.getHigh() <= _range.getHigh());
    }
  class Room...
    boolean withinPlan(HeatingPlan plan) {
        int low = daysTempRange().getLow();
        int high = daysTempRange().getHigh();
        return plan.withinRange(daysTempRange());
    }
```

Now I don't need the temps anymore:

```
  class Room...
    boolean withinPlan(HeatingPlan plan) {
        int low = daysTempRange().getLow();
        int high = daysTempRange().getHigh();
        return plan.withinRange(daysTempRange());
    }
```

Using whole objects this way soon leads you to realize that you can usefully move behavior into the whole object to make it easier to work with.

```
  class HeatingPlan...
    boolean withinRange (TempRange roomRange) {
        return (_range.includes(roomRange));
    }
  class TempRange...
    boolean includes (TempRange arg) {
        return arg.getLow() >= this.getLow() && arg.getHigh() <=
this.getHigh();
    }
```

Replace Parameter with Method

An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method.

Remove the parameter and let the receiver invoke the method.

```
      int basePrice = _quantity * _itemPrice;
      discountLevel = getDiscountLevel();
      double finalPrice = discountedPrice (basePrice, discountLevel);
```

Chapter 9. Simplifying Conditional Expressions

Conditional logic has a way of getting tricky, so here are a number of refactorings you can use to simplify it. The core refactoring here is Decompose Conditional, which entails breaking a conditional into pieces. It is important because it separates the switching logic from the details of what happens.

The other refactorings in this chapter involve other important cases. Use Consolidate Conditional Expression when you have several tests and all have the same effect. Use Consolidate Duplicate Conditional Fragments to remove any duplication within the conditional code.

If you are working with code developed in a one exit point mentality, you often find control flags that allow the conditions to work with this rule. I don't follow the rule about one exit point from a method. Hence I use Replace Nested Conditional with Guard Clauses to clarify special case conditionals and Remove Control Flag to get rid of the awkward control flags.

Object-oriented programs often have less conditional behavior than procedural programs because much of the conditional behavior is handled by polymorphism. Polymorphism is better because the caller does not need to know about the conditional behavior, and it is thus easier to extend the conditions. As a result, object-oriented programs rarely have switch (case) statements. Any that show up are prime candidates for Replace Conditional with Polymorphism.

One of the most useful, but less obvious, uses of polymorphism is to use Introduce Null Object to remove checks for a null value.

Decompose Conditional

You have a complicated conditional (if-then-else) statement.

Extract methods from the condition, then part, and else parts.

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;
```

```
if (notSummer(date))
    charge = winterCharge(quantity);
else charge = summerCharge (quantity);
```

Motivation

One of the most common areas of complexity in a program lies in complex conditional logic. As you write code to test conditions and to do various things depending on various conditions, you quickly end up with a pretty long method. Length of a method is in itself a factor that makes it harder to read, but conditions increase the difficulty. The problem usually lies in the fact that the code, both in the condition checks and in the actions, tells you what happens but can easily obscure why it happens.

As with any large block of code, you can make your intention clearer by decomposing it and replacing chunks of code with a method call named after the intention of that block of code. With conditions you can receive further benefit by doing this for the conditional part and each of the alternatives. This way you highlight the condition and make it clearly what you are branching on. You also highlight the reason for the branching.

Mechanics

- Extract the condition into its own method.
- Extract the then part and the else part into their own methods.

If I find a nested conditional, I usually first look to see whether I should use Replace Nested Conditional with Guard Clauses. If that does not make sense, I decompose each of the conditionals.

Example

Suppose I'm calculating the charge for something that has separate rates for winter and summer:

```
    if (date.before (SUMMER_START) || date.after(SUMMER_END))
        charge = quantity * _winterRate + _winterServiceCharge;
    else charge = quantity * _summerRate;
```

I extract the conditional and each leg as follows:

```
    if (notSummer(date))
        charge = winterCharge(quantity);
    else charge = summerCharge (quantity);

private boolean notSummer(Date date) {
    return date.before (SUMMER_START) || date.after(SUMMER_END);
}

private double summerCharge(int quantity) {
    return quantity * _summerRate;
}

private double winterCharge(int quantity) {
    return quantity * _winterRate + _winterServiceCharge;
}
```

Here I show the result of the complete refactoring for clarity. In practice, however, I do each extraction separately and compile and test after each one.

Many programmers don't extract the condition parts in situations such as this. The conditions often are quite short, so it hardly seems worth it. Although the condition is often short, there often is a big gap between the intention of the code and its body. Even in this little case, reading `notSummer(date)` conveys a clearer message to me than does the original code. With the original I have to look at the code and figure out what it is doing. It's not difficult to do that here, but even so the extracted method reads more like a comment.