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 (110)*, which takes a clump of code and turns it into its own method. *Inline Method (117)* is essentially the opposite. You take a method call and replace it with the body of the code. I need *Inline Method (117)* 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 (110)* 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 Query (120)* to get rid of any temporary variables that I can remove. If the temp is used for many things, I use *Split Temporary Variable (128)* 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 (135)*. 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 (131)*.

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 (139)* to introduce the clearer algorithm.

Extract Method

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

Extract Method

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 (128)* and try again. You can eliminate temporary variables with *Replace Temp with Query (120)* (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.

Extract Method

Example: No Local Variables

In the simplest case, *Extract Method (110)* 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 ******");
    System.out.println ("**************************");
}
```


Extract Method

Example: Using Local Variables

So what's the problem? The problem is local variables: parameters passed into the original method and temporaries declared within the original method. Local variables are only in scope in that method, so when I use *Extract Method (110)*, these variables cause me extra work. In some cases they even prevent me from doing the refactoring at all.

The easiest case with local variables is when the variables are read but not changed. In this case I can just pass them in as a parameter. So if I have the following method:

```
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);
}
```

I can extract the printing of details with a method with one parameter:

```
void printOwing() {

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

    printBanner();

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

    printDetails(outstanding);
}

void printDetails (double outstanding) {
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

You can use this with as many local variables as you need.

The same is true if the local variable is an object and you invoke a modifying method on the variable. Again you can just pass the object in as a parameter. You only have to do something different if you actually assign to the local variable.

Example: Reassigning a Local Variable

It's the assignment to local variables that becomes complicated. In this case we're only talking about temps. If you see an assignment to a parameter, you should immediately use *Remove Assignments to Parameters (131)*.

For temps that are assigned to, there are two cases. The simpler case is that in which the variable is a temporary variable used only within the extracted code. When that happens, you can move the temp into the extracted code. The other case is use of the variable outside the code. If the variable is not used after the code is extracted, you can make the change in just the extracted code. If it is used afterward, you need to make the extracted code return the changed value of the variable. I can illustrate this with the following method:

```
void printOwing() {

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

    printBanner();

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

    printDetails(outstanding);
}
```

Now I extract the calculation:

```
void printOwing() {
    printBanner();
    double outstanding = getOutstanding();
    printDetails(outstanding);
}

double getOutstanding() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    return outstanding;
}
```


The enumeration variable is used only in the extracted code, so I can move it entirely within the new method. The outstanding variable is used in both places, so I need to return it from the extracted method. Once I've compiled and tested for the extraction, I rename the returned value to follow my usual convention:

```
double getOutstanding() {
    Enumeration e = _orders.elements();
    double result = 0.0;
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        result += each.getAmount();
    }
    return result;
}
```

In this case the outstanding variable is initialized only to an obvious initial value, so I can initialize it only within the extracted method. If something more involved happens to the variable, I have to pass in the previous value as a parameter. The initial code for this variation might look like this:

```
void printOwing(double previousAmount) {

    Enumeration e = _orders.elements();
    double outstanding = previousAmount * 1.2;

    printBanner();

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

    printDetails(outstanding);
}
```

In this case the extraction would look like this:

```
void printOwing(double previousAmount) {
    double outstanding = previousAmount * 1.2;
    printBanner();
    outstanding = getOutstanding(outstanding);
    printDetails(outstanding);
}

double getOutstanding(double initialValue) {
    double result = initialValue;
    Enumeration e = _orders.elements();
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        result += each.getAmount();
    }
    return result;
}
```

After I compile and test this, I clear up the way the outstanding variable is initialized:

```
void printOwing(double previousAmount) {
    printBanner();
    double outstanding = getOutstanding(previousAmount * 1.2);
    printDetails(outstanding);
}
```

At this point you may be wondering, "What happens if more than one variable needs to be returned?"

Here you have several options. The best option usually is to pick different code to extract. I prefer a method to return one value, so I would try to arrange for multiple methods for the different values. (If your language allows output parameters, you can use those. I prefer to use single return values as much as possible.)

Temporary variables often are so plentiful that they make extraction very awkward. In these cases I try to reduce the temps by using *Replace Temp with Query (120)*. If whatever I do things are still awkward, I resort to *Replace Method with Method Object (135)*. This refactoring doesn't care how many temporaries you have or what you do with them.

Inline Method

A method's body is just as clear as its name.

Put the method's body into the body of its callers and remove the method.

```
int getRating() {
    return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
    return _numberOfLateDeliveries > 5;
}
```

```
int getRating() {
    return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Motivation

A theme of this book is to use short methods named to show their intention, because these methods lead to clearer and easier to read code. But sometimes you do come across a method in which the body is as clear as the name. Or you refactor the body of the code into something that is just as clear as the name. When this happens, you should then get rid of the method. Indirection can be helpful, but needless indirection is irritating.

Another time to use *Inline Method (117)* is when you have a group of methods that seem badly factored. You can inline them all into one big method and then reextract the methods. Kent Beck finds it is often good to do this before using *Replace Method with Method Object (135)*. You inline the various calls made by the method that have behavior you want to have in the method object. It's easier to move one method than to move the method and its called methods.

I commonly use *Inline Method (117)* when someone is using too much indirection and it seems that every method does simple delegation to another method, and I get lost in all the delegation. In these cases some of the indirection is worthwhile, but not all of it. By trying to inline I can flush out the useful ones and eliminate the rest.

Mechanics

- Check that the method is not polymorphic.

  ⇒ *Don't inline if subclasses override the method; they cannot override a method that isn't there.*

- Find all calls to the method.
- Replace each call with the method body.
- Compile and test.
- Remove the method definition.

Written this way, *Inline Method (117)* is simple. In general it isn't. I could write pages on how to handle recursion, multiple return points, inlining into another object when you don't have accessors, and the like. The reason I don't is that if you encounter these complexities, you shouldn't do this refactoring.

Inline Temp

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 (120)*, 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 (110)*, it's time to inline it.

Mechanics

- Check that the right hand side of the assignment is free of side-effects.
- 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 new method. 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 (110)*. 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 only assigned 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 (128)* or *Separate Query from Modifier (279)* 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 (128).
- 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 (279).
- Compile and test.
- *Inline Temp (119)* 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 *Inline Temp (119)*. 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 (110)* 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, however, when local variables make it difficult to use *Extract Method (110)*. That's when I use *Introduce Explaining Variable (124)*.

Mechanics

- Declare a final temporary variable, and set it to the result of part of the complex expression.
- Replace the result part of the expression with the value of the temp.
  - ⇛ *If the result part of the expression is repeated, you can replace the repeats one at a time.*
- Compile and test.
- Repeat for other parts of the expression.

Example

I start with a simple calculation:

```
double price() {
    // price is base price - quantity discount + shipping
    return _quantity * _itemPrice -
        Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
        Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

Simple it may be, but I can make it easier to follow. First I identify the base price as the quantity times the item price. I can turn that part of the calculation into a temp:

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;
    return basePrice -
        Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
        Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

Quantity times item price is also used later, so I can substitute with the temp there as well:

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;
    return basePrice -
        Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
        Math.min(basePrice * 0.1, 100.0);
}
```

Next I take the quantity discount:

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;
    final double quantityDiscount = Math.max(0, _quantity - 500) * _itemPrice * 0.05;
    return basePrice - quantityDiscount +
        Math.min(basePrice * 0.1, 100.0);
}
```

Finally, I finish with the shipping. As I do that, I can remove the comment, too, because now it doesn't say anything the code doesn't say:

```
double price() {
    final double basePrice = _quantity * _itemPrice;
    final double quantityDiscount = Math.max(0, _quantity - 500) * _itemPrice * 0.05;
    final double shipping = Math.min(basePrice * 0.1, 100.0);
    return basePrice - quantityDiscount + shipping;
}
```

Example with *Extract Method*

For this example I usually wouldn't have done the explaining temps; I would prefer to do that with *Extract Method (110)*. I start again with

```
double price() {
    // price is base price - quantity discount + shipping
    return _quantity * _itemPrice -
        Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
        Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

but this time I extract a method for the base price:

```
double price() {
    // price is base price - quantity discount + shipping
    return basePrice() -
        Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
        Math.min(basePrice() * 0.1, 100.0);
}

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


I continue one at a time. When I'm finished I get

```
double price() {
    return basePrice() - quantityDiscount() + shipping();
}

private double quantityDiscount() {
    return Math.max(0, _quantity - 500) * _itemPrice * 0.05;
}

private double shipping() {
    return Math.min(basePrice() * 0.1, 100.0);
}

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

I prefer to use *Extract Method (110)*, because now these methods are available to any other part of the object that needs them. Initially I make them private, but I can always relax that if another object needs them. I find it's usually no more effort to use *Extract Method (110)* than it is to use *Introduce Explaining Variable (124)*.

So when do I use *Introduce Explaining Variable (124)*? The answer is when *Extract Method (110) is* more effort. If I'm in an algorithm with a lot of local variables, I may not be able to easily use *Extract Method (110)*. In this case I use *Introduce Explaining Variable (124)* to help me understand what is going on. As the logic becomes less tangled, I can always use *Replace Temp with Query (120)* later. The temp also is valuable if I end up having to use *Replace Method with Method Object (135)*.

Split Temporary Variable

You have a temporary variable assigned to more than once, but is not a loop variable nor a collecting temporary variable.

Make a separate temporary variable for each assignment.

```
double temp = 2 * (_height + _width);
System.out.println (temp);
temp = _height * _width;
System.out.println (temp);
```

```
final double perimeter = 2 * (_height + _width);
System.out.println (perimeter);
final double area = _height * _width;
System.out.println (area);
```

Motivation

Temporary variables are made for various uses. Some of these uses naturally lead to the temp's being assigned to several times. Loop variables [Beck] change for each run around a loop (such as the `i` in `for (int i=0; i<10; i++)`). Collecting temporary variables [Beck] collect together some value that is built up during the method.

Many other temporaries are used to hold the result of a long-winded bit of code for easy reference later. These kinds of variables should be set only once. That they are set more than once is a sign that they have more than one responsibility within the method. Any variable with more than one responsibility should be replaced with a temp for each responsibility. Using a temp for two different things is very confusing for the reader.

Mechanics

- Change the name of a temp at its declaration and its first assignment.
  - ⇛ *If the later assignments are of the form i = i + some expression, that indicates that it is a collecting temporary variable, so don't split it. The operator for a collecting temporary variable usually is addition, string concatenation, writing to a stream, or adding to a collection.*

- Declare the new temp as final.
- Change all references of the temp up to its second assignment.
- Declare the temp at its second assignment.
- Compile and test.
- Repeat in stages, each stage renaming at the declaration, and changing references until the next assignment.

Example

For this example I compute the distance traveled by a haggis. From a standing start, a haggis experiences an initial force. After a delayed period a secondary force kicks in to further accelerate the haggis. Using the common laws of motion, I can compute the distance traveled as follows:

```
double getDistanceTravelled (int time) {
    double result;
    double acc = _primaryForce / _mass;
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * acc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;
    if (secondaryTime > 0) {
        double primaryVel = acc * _delay;
        acc = (_primaryForce + _secondaryForce) / _mass;
        result +=  primaryVel * secondaryTime + 0.5 * acc * secondaryTime * secondaryTime;
    }
    return result;
}
```

A nice awkward little function. The interesting thing for our example is the way the variable `acc` is set twice. It has two responsibilities: one to hold the initial acceleration caused by the first force and another later to hold the acceleration with both forces. This I want to split.

I start at the beginning by changing the name of the temp and declaring the new name as final. Then I change all references to the temp from that point up to the next assignment. At the next assignment I declare it:

```
double getDistanceTravelled (int time) {
    double result;
    final double primaryAcc = _primaryForce / _mass;
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * primaryAcc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;
```

```
    if (secondaryTime > 0) {
        double primaryVel = primaryAcc * _delay;
        double acc = (_primaryForce + _secondaryForce) / _mass;
        result +=  primaryVel * secondaryTime + 0.5 * acc * secondaryTime * secondaryTime;
    }
    return result;
}
```

I choose the new name to represent only the first use of the temp. I make it final to ensure it is only set once. I can then declare the original temp at its second assignment. Now I can compile and test, and all should work.

I continue on the second assignment of the temp. This removes the original temp name completely, replacing it with a new temp named for the second use.

```
double getDistanceTravelled (int time) {
    double result;
    final double primaryAcc = _primaryForce / _mass;
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * primaryAcc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;
    if (secondaryTime > 0) {
        double primaryVel = primaryAcc * _delay;
        final double secondaryAcc = (_primaryForce + _secondaryForce) / _mass;
        result +=  primaryVel * secondaryTime + 0.5 *
                   secondaryAcc * secondaryTime * secondaryTime;
    }
    return result;
}
```

I'm sure you can think of a lot more refactoring to be done here. Enjoy it. (I'm sure it's better than eating the haggis—do you know what they put in those things?)

Remove Assignments to Parameters

The code assigns to a parameter.

Use a temporary variable instead.

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
```

```
int discount (int inputVal, int quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
```

Motivation

First let me make sure we are clear on the phrase "assigns to a parameter." This means that if you pass in an object named *foo*, in the parameter, assigning to the parameter means to change *foo* to refer to a different object. I have no problems with doing something to the object that was passed in; I do that all the time. I just object to changing *foo* to refer to another object entirely:

```
void aMethod(Object foo) {
  foo.modifyInSomeWay();              // that's OK
  foo = anotherObject;                // trouble and despair will follow you
```

The reason I don't like this comes down to lack of clarity and to confusion between *pass by value* and *pass by reference*. Java uses pass by value exclusively (see later), and this discussion is based on that usage.

With pass by value, any change to the parameter is not reflected in the calling routine. Those who have used pass by reference will probably find this confusing.

The other area of confusion is within the body of the code itself. It is much clearer if you use only the parameter to represent what has been passed in, because that is a consistent usage.

In Java, don't assign to parameters, and if you see code that does, apply *Remove Assignments to Parameters*.

Of course this rule does not necessarily apply to other languages that use output parameters, although even with these languages I prefer to use output parameters as little as possible.

Mechanics

- Create a temporary variable for the parameter.
- Replace all references to the parameter, made after the assignment, to the temporary variable.
- Change the assignment to assign to the temporary variable.
- Compile and test.

⇒ *If the semantics are call by reference, look in the calling method to see whether the parameter is used again afterward. Also see how many call by reference parameters are assigned to and used afterward in this method. Try to pass a single value back as the return value. If there is more than one, see whether you can turn the data clump into an object, or create separate methods.*

Example

I start with the following simple routine:

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
    if (quantity > 100) inputVal -= 1;
    if (yearToDate > 10000) inputVal -= 4;
    return inputVal;
}
```

Replacing with a temp leads to

```
int discount (int inputVal, int quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
    if (quantity > 100) result -= 1;
    if (yearToDate > 10000) result -= 4;
    return result;
}
```

You can enforce this convention with the final keyword:

```
int discount (final int inputVal, final int quantity, final int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
    if (quantity > 100) result -= 1;
    if (yearToDate > 10000) result -= 4;
    return result;
}
```


I admit that I don't use final much, because I don't find it helps much with clarity for short methods. I use it with a long method to help me see whether anything is changing the parameter.

Pass By Value in Java

Use of pass by value often is a source of confusion in Java. Java strictly uses pass by value in all places, thus the following program:

```
class Param {
  public static void main(String[] args) {
      int x = 5;
      triple(x);
      System.out.println ("x after triple: " + x);
  }
  private static void triple(int arg) {
      arg = arg * 3;
      System.out.println ("arg in triple: " + arg);
  }
}
```

produces the following output:

```
arg in triple: 15
x after triple: 5
```

The confusion exists with objects. Say I use a date, then this program:

```
class Param {

  public static void main(String[] args) {
      Date d1 = new Date ("1 Apr 98");
      nextDateUpdate(d1);
      System.out.println ("d1 after nextDay: " + d1);

      Date d2 = new Date ("1 Apr 98");
      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.

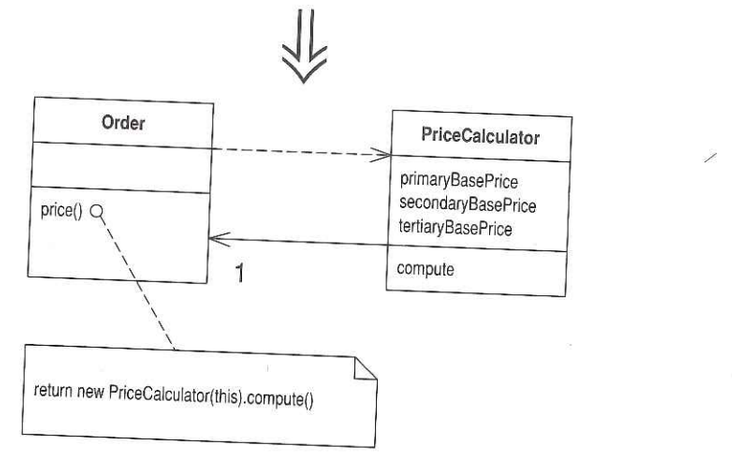
Java 1.1 and later versions allow you to mark a parameter as *final;* this prevents assignment to the variable. It still allows you to modify the object the variable refers to. I always treat my parameters as final, but I confess I rarely mark them so in the parameter list.

Replace Method with Method Object

You have a long method that uses local variables in such a way that you cannot apply *Extract Method (110).*

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 (120)* 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 (135)* turns all these local variables into fields on the method object. You can then use *Extract Method (110)* 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 (110)* 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.*

Chapter 7

Moving Features Between Objects

One of the most fundamental, if not the fundamental, decision in object design is deciding where to put responsibilities. I've been working with objects for more than a decade, but I still never get it right the first time. That used to bother me, but now I realize that I can use refactoring to change my mind in these cases.

Often I can resolve these problems simply by using *Move Method (142)* and *Move Field (146)* to move the behavior around. If I need to use both, I prefer to use *Move Field (146)* first and then *Move Method (142)*.

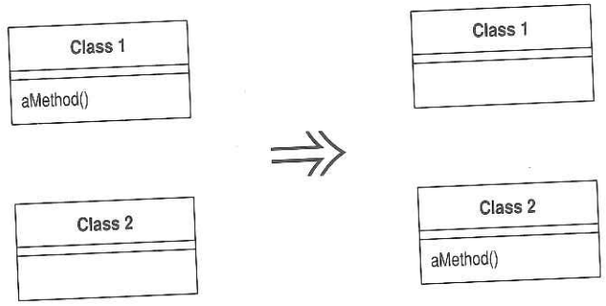
Often classes become bloated with too many responsibilities. In this case I use *Extract Class (149)* to separate some of these responsibilities. If a class becomes too irresponsible, I use *Inline Class (154)* to merge it into another class. If another class is being used, it often is helpful to hide this fact with *Hide Delegate (157)*. Sometimes hiding the delegate class results in constantly changing the owner's interface, in which case you need to use *Remove Middle Man (160)*

The last two refactorings in this chapter, *Introduce Foreign Method* and *Introduce Local Extension (164)* are special cases. I use these only when I'm not able to access the source code of a class, yet I want to move responsibilities to this unchangeable class. If it is only one or two methods, I use *Introduce Foreign Method (162)*; for more than one or two methods, I use *Introduce Local Extension (164)*.

Move Method

A method is, or will be, using or used by more features of another class than the class on which it is defined.

Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.



Motivation

Moving methods is the bread and butter of refactoring. I move methods when classes have too much behavior or when classes are collaborating too much and are too highly coupled. By moving methods around, I can make the classes simpler and they end up being a more crisp implementation of a set of responsibilities.

I usually look through the methods on a class to find a method that seems to reference another object more than the object it lives on. A good time to do this is after I have moved some fields. Once I see a likely method to move, I take a look at the methods that call it, the methods it calls, and any redefining methods in the hierarchy. I assess whether to go ahead on the basis of the object with which the method seems to have more interaction.

It's not always an easy decision to make. If I am not sure whether to move a method, I go on to look at other methods. Moving other methods often makes the decision easier. Sometimes the decision still is hard to make. Actually it is then no big deal. If it is difficult to make the decision, it probably does not matter that much. Then I choose according to instinct; after all, I can always change it again later.

Mechanics

- Examine all features used by the source method that are defined on the source class. Consider whether they also should be moved.

  ⇛ *If a feature is used only by the method you are about to move, you might as well move it, too. If the feature is used by other methods, consider moving them as well. Sometimes it is easier to move a clutch of methods than to move them one at a time.*

- Check the sub- and superclasses of the source class for other declarations of the method.

  ⇛ *If there are any other declarations, you may not be able to make the move, unless the polymorphism can also be expressed on the target.*

- Declare the method in the target class.

  ⇛ *You may choose to use a different name, one that makes more sense in the target class.*

- Copy the code from the source method to the target. Adjust the method to make it work in its new home.

  ⇛ *If the method uses its source, you need to determine how to reference the source object from the target method. If there is no mechanism in the target class, pass the source object reference to the new method as a parameter.*

  ⇛ *If the method includes exception handlers, decide which class should logically handle the exception. If the source class should be responsible, leave the handlers behind.*

- Compile the target class.

- Determine how to reference the correct target object from the source.

  ⇛ *There may be an existing field or method that will give you the target. If not, see whether you can easily create a method that will do so. Failing that, you need to create a new field in the source that can store the target. This may be a permanent change, but you can also make it temporarily until you have refactored enough to remove it.*

- Turn the source method into a delegating method.

- Compile and test.

- Decide whether to remove the source method or retain it as a delegating method.

  ⇛ *Leaving the source as a delegating method is easier if you have many references.*

- If you remove the source method, replace all the references with references to the target method.

  ⇛ *You can compile and test after changing each reference, although it is usually easier to change all references with one search and replace.*

- Compile and test.

Example

An account class illustrates this refactoring:

```
class Account...
  double overdraftCharge() {
      if (_type.isPremium()) {
          double result = 10;
          if (_daysOverdrawn > 7) result += (_daysOverdrawn - 7) * 0.85;
          return result;
      }
      else return _daysOverdrawn * 1.75;
  }

  double bankCharge() {
      double result = 4.5;
      if (_daysOverdrawn > 0) result += overdraftCharge();
      return result;
  }
  private AccountType _type;
  private int _daysOverdrawn;
```

Let's imagine that there are going to be several new account types, each of which has its own rule for calculating the overdraft charge. So I want to move the overdraft charge method over to the account type.

The first step is to look at the features that the overdraftCharge method uses and consider whether it is worth moving a batch of methods together. In this case I need the _daysOverdrawn field to remain on the account class, because that will vary with individual accounts.

Next I copy the method body over to the account type and get it to fit.

```
class AccountType...
  double overdraftCharge(int daysOverdrawn) {
      if (isPremium()) {
          double result = 10;
          if (daysOverdrawn > 7) result += (daysOverdrawn - 7) * 0.85;
          return result;
      }
      else return daysOverdrawn * 1.75;
  }
```

In this case fitting means removing the _type from uses of features of the account type, and doing something about the features of account that I still need. When I need to use a feature of the source class I can do one of four things: (1) move this feature to the target class as well, (2) create or use a reference from the target class to the source, (3) pass the source object as a parameter to the method, (4) if the feature is a variable, pass it in as a parameter.

In this case I passed the variable as a parameter.

Once the method fits and compiles in the target class, I can replace the source method body with a simple delegation:

```
class Account...
  double overdraftCharge() {
      return _type.overdraftCharge(_daysOverdrawn);
  }
```

At this point I can compile and test.

I can leave things like this, or I can remove the method in the source class. To remove the method I need to find all callers of the method and redirect them to call the method in account type:

```
class Account...
  double bankCharge() {
      double result = 4.5;
      if (_daysOverdrawn > 0) result += _type.overdraftCharge(_daysOverdrawn);
      return result;
  }
```

Once I've replaced all the callers, I can remove the method declaration in account. I can compile and test after each removal, or do them in a batch. If the method isn't private, I need to look for other classes that use this method. In a strongly typed language, the compilation after removal of the source declaration finds anything I missed.

In this case the method referred only to a single field, so I could just pass this field in as a variable. If the method called another method on the account, I wouldn't have been able to do that. In those cases I need to pass in the source object:

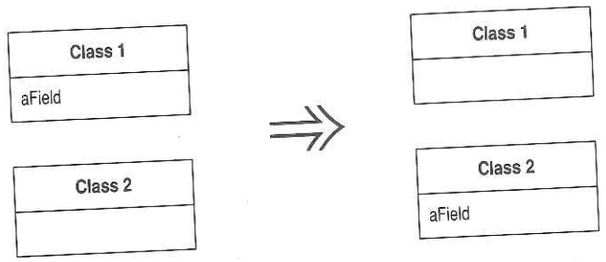
```
class AccountType...
  double overdraftCharge(Account account) {
      if (isPremium()) {
          double result = 10;
          if (account.getDaysOverdrawn() > 7)
             result += (account.getDaysOverdrawn() - 7) * 0.85;
          return result;
      }
      else return account.getDaysOverdrawn() * 1.75;
  }
```

I also pass in the source object if I need several features of the class, although if there are too many, further refactoring is needed. Typically I need to decompose and move some pieces back.

Move Field

A field is, or will be, used by another class more than the class on which it is defined.

Create a new field in the target class, and change all its users.



Motivation

Moving state and behavior between classes is the very essence of refactoring. As the system develops, you find the need for new classes and the need to shuffle responsibilities around. A design decision that is reasonable and correct one week can become incorrect in another. That is not a problem; the only problem is not to do something about it.

I consider moving a field if I see more methods on another class using the field than the class itself. This usage may be indirect, through getting and setting methods. I may choose to move the methods; this decision based on interface. But if the methods seem sensible where they are, I move the field.

Another reason for field moving is when doing *Extract Class (149)*. In that case the fields go first and then the methods.

Mechanics

- If the field is public, use *Encapsulate Field (206)*.
  - ⇛ *If you are likely to be moving the methods that access it frequently or if a lot of methods access the field, you may find it useful to use* Self Encapsulate Field (171)
- Compile and test.
- Create a field in the target class with getting and setting methods.
- Compile the target class.

- Determine how to reference the target object from the source.
  - ⇛ *An existing field or method may give you the target. If not, see whether you can easily create a method that will do so. Failing that, you need to create a new field in the source that can store the target. This may be a permanent change, but you can also do it temporarily until you have refactored enough to remove it.*
- Remove the field on the source class.
- Replace all references to the source field with references to the appropriate method on the target.
  - ⇛ *For accesses to the variable, replace the reference with a call to the target object's getting method; for assignments, replace the reference with a call to the setting method.*
  - ⇛ *If the field is not private, look in all the subclasses of the source for references.*
- Compile and test.

Example

Here is part of an account class:

```
class Account...
  private AccountType _type;
  private double _interestRate;

  double interestForAmount_days (double amount, int days) {
      return _interestRate * amount * days / 365;
  }
```

I want to move the interest rate field to the account type. There are several methods with that reference, of which `interestForAmount_days` is one example.

I next create the field and accessors in the account type:

```
class AccountType...
  private double _interestRate;


  void setInterestRate (double arg) {
      _interestRate = arg;
  }

  double getInterestRate () {
      return _interestRate;
  }
```

I can compile the new class at this point.

Now I redirect the methods from the account class to use the account type and remove the interest rate field in the account. I must remove the field to be sure that the redirection is actually happening. This way the compiler helps spot any method I failed to redirect.

```
private double _interestRate;

double interestForAmount_days (double amount, int days) {
    return _type.getInterestRate() * amount * days / 365;
}
```

Example: Using Self-Encapsulation

If a lot of methods use the interest rate field, I might start by using *Self Encapsulate Field (171)*:

```
class Account...
  private AccountType _type;
  private double _interestRate;

  double interestForAmount_days (double amount, int days) {
      return getInterestRate() * amount * days / 365;
  }

  private void setInterestRate (double arg) {
      _interestRate = arg;
  }

  private double getInterestRate () {
      return _interestRate;
  }
```

That way I only need to do the redirection for the accessors:

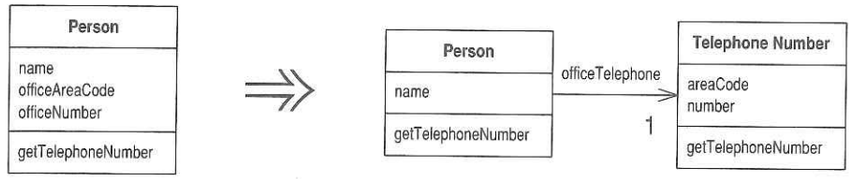
```
double interestForAmountAndDays (double amount, int days) {
    return getInterestRate() * amount * days / 365;
}

private void setInterestRate (double arg) {
    _type.setInterestRate(arg);
}

private double getInterestRate () {
    return _type.getInterestRate();
}
```

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 (142)* 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.
  - ⇛ *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.
  - ⇛ *You may need a two-way link. But don't make the back link until you find you need it.*
- Use *Move Field (146)* on each field you wish to move.
- Compile and test after each move.
- Use *Move Method (142)* 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.
  - ⇛ *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 (146)* 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 (142)* 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 (179)*. 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.

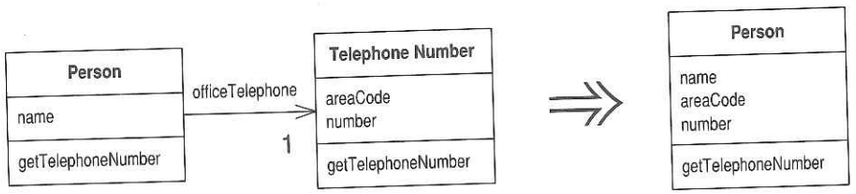
Extract Class (149) is a common technique for improving the liveness of a concurrent program because it allows you to have separate locks on the two resulting classes. If you don't need to lock both objects you don't have to. For more on this see section 3.3 in Lea [Lea].

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 in 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 (149)*. 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.

Mechanics

- Declare the public protocol of the source class onto the absorbing class. Delegate all these methods to the source class.

  ⇛ *If a separate interface makes sense for the source class methods, use* Extract Interface (341) *before inlining.*

- Change all references from the source class to the absorbing class.

  ⇛ *Declare the source class private to remove out-of-package references. Also change the name of the source class so the compiler catches any dangling references to the source class.*

- Compile and test.
- Use *Move Method* and *Move Field* to move features from the source class to the absorbing class until there is nothing left.
- Hold a short, simple funeral service.

Example

Because I made a class out of telephone number, I now inline it back into person. I start with separate classes:

```
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;
```

I begin by declaring all the visible methods on telephone number on person:

```
class Person...
  String getAreaCode() {
      return _officeTelephone.getAreaCode();
  }
  void setAreaCode(String arg) {
      _officeTelephone.setAreaCode(arg);
  }
  String getNumber() {
      return _officeTelephone.getNumber();
```

```
}
void setNumber(String arg) {
    _officeTelephone.setNumber(arg);
}
```

Now I find clients of telephone number and switch them to use the person's interface. So

```
Person martin = new Person();
martin.getOfficeTelephone().setAreaCode ("781");
```

becomes

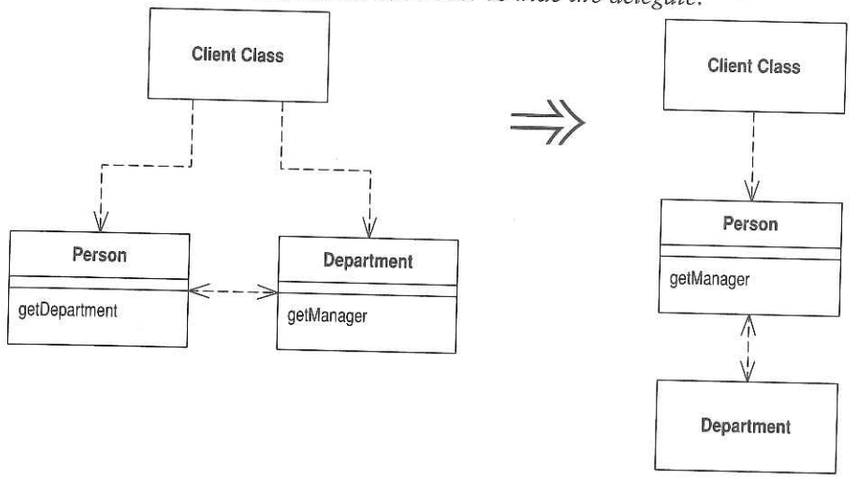
```
Person martin = new Person();
martin.setAreaCode ("781");
```

Now I can use *Move Method (142)* and *Move Field (146)* until the telephone class is no more.

Hide Delegate

A client is calling a delegate class of an object.

Create methods on the server to hide the delegate.



Motivation

One of the keys, if not *the* key, to objects is encapsulation. Encapsulation means that objects need to know less about other parts of the system. Then when things change, fewer objects need to be told about the change—which makes the change easier to make.

Anyone involved in objects knows that you should hide your fields, despite the fact that Java allows fields to be public. As you become more sophisticated, you realize there is more you can encapsulate.

If a client calls a method defined on one of the fields of the server object, the client needs to know about this delegate object. If the delegate changes, the client also may have to change. You can remove this dependency by placing a sim-

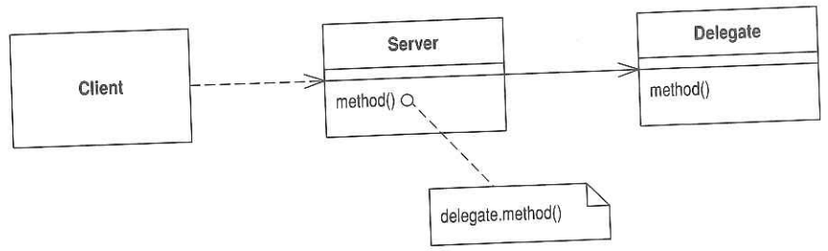


Figure 7.1 *Simple delegation*

ple delegating method on the server, which hides the delegate (Figure 7.1). Changes become limited to the server and don't propagate to the client.

You may find it is worthwhile to use *Extract Class (149)* for some clients of the server or all clients. If you hide from all clients, you can remove all mention of the delegate from the interface of the server.

Mechanics

- For each method on the delegate, create a simple delegating method on the server.
- Adjust the client to call the server.
  - ⇛ *If the client is not in the same package as the server, consider changing the delegate method's access to package visibility.*
- Compile and test after adjusting each method.
- If no client needs to access the delegate anymore, remove the server's accessor for the delegate.
- Compile and test.

Example

I start with a person and a department:

```
class Person {
  Department _department;

  public Department getDepartment() {
      return _department;
  }
```



```
  public void setDepartment(Department arg) {
      _department = arg;
  }
}

class Department {
  private String _chargeCode;
  private Person _manager;

  public Department (Person manager) {
      _manager = manager;
  }

  public Person getManager() {
      return _manager;
  }
...
```

If a client wants to know a person's manager, it needs to get the department first:

```
manager = john.getDepartment().getManager();
```

This reveals to the client how the department class works and that the department is responsible to tracking the manager. I can reduce this coupling by hiding the department class from the client. I do this by creating a simple delegating method on person:

```
public Person getManager() {
    return _department.getManager();
}
```

I now need to change all clients of person to use this new method:

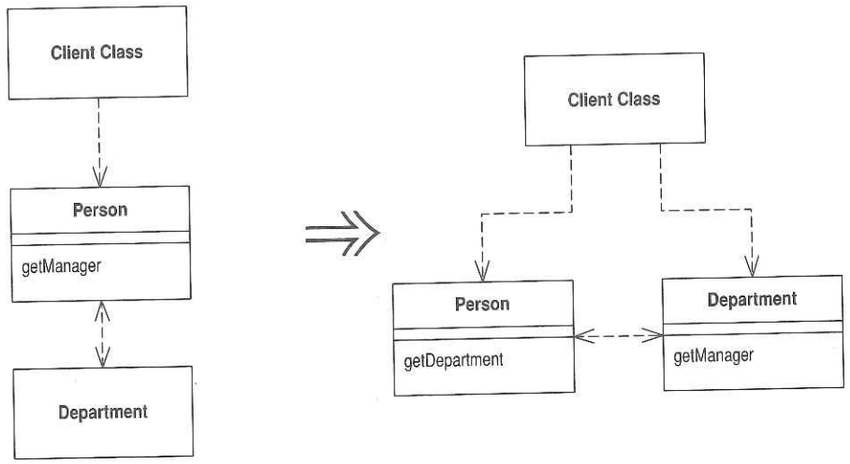
```
manager = john.getManager();
```

Once I've made the change for all methods of department and for all the clients of person, I can remove the `getDepartment` accessor on person.

Remove Middle Man

A class is doing too much simple delegation.

Get the client to call the delegate directly.



Motivation

In the motivation for *Hide Delegate*, I talked about the advantages of encapsulating the use of a delegated object. There is a price for this. The price is that every time the client wants to use a new feature of the delegate, you have to add a simple delegating method to the server. After adding features for a while, it becomes painful. The server class is just a middle man, and perhaps it's time for the client to call the delegate directly.

It's hard to figure out what the right amount of hiding is. Fortunately, with *Hide Delegate* and *Remove Middle Man (160)* it does not matter so much. You can adjust your system as time goes on. As the system changes, the basis for how much you hide also changes. A good encapsulation six months ago may be awkward now. Refactoring means you never have to say you're sorry—you just fix it.

Mechanics

- Create an accessor for the delegate.
- For each client use of a delegate method, remove the method from the server and replace the call in the client to call method on the delegate.
- Compile and test after each method.

Example

For an example I use person and department flipped the other way. I start with person hiding the department:

```
class Person...
  Department _department;
  public Person getManager() {
      return _department.getManager();

class Department...
  private Person _manager;
  public Department (Person manager) {
      _manager = manager;
  }
```

To find a person's manager, clients ask:

```
manager = john.getManager();
```

This is simple to use and encapsulates the department. However, if lots of methods are doing this, I end up with too many of these simple delegations on the person. That's when it is good to remove the middle man. First I make an accessor for the delegate:

```
class Person...
  public Department getDepartment() {
      return _department;
  }
```

Then I take each method at a time. I find clients that use the method on person and change it to first get the delegate. Then I use it:

```
manager = john.getDepartment().getManager();
```

I can then remove `getManager` from person. A compile shows whether I missed anything.

I may want to keep some of these delegations for convenience. I also may want to hide the delegate from some clients but show it to others. That also will leave some of the simple delegations in place.

Introduce Foreign Method

A server class you are using needs an additional method, but you can't modify the class.

Create a method in the client class with an instance of the server class as its first argument.

```
Date newStart = new Date (previousEnd.getYear(),
                  previousEnd.getMonth(), previousEnd.getDate() + 1);
```

```
Date newStart = nextDay(previousEnd);

private static Date nextDay(Date arg) {
    return new Date (arg.getYear(),arg.getMonth(), arg.getDate() + 1);
}
```

Motivation

It happens often enough. You are using this really nice class that gives you all these great services. Then there is one service it doesn't give you but should. You curse the class, saying, "Why don't you do that?" If you can change the source, you can add in the method. If you can't change the source, you have to code around the lack of the method in the client.

If you use the method only once in the client class then the extra coding is no big deal and probably wasn't needed on the original class anyway. If you use the method several times, however, you have to repeat this coding around. Because repetition is the root of all software evil, this repetitive code should be factored into a single method. When you do this refactoring, you can clearly signal that this method is really a method that should be on the original by making it a foreign method.

If you find yourself creating many foreign methods on a server class, or you find many of your classes need the same foreign method, you should use *Introduce Local Extension (164)* instead.

Don't forget that foreign methods are a work-around. If you can, try to get the methods moved to their proper homes. If code ownership is the issue, send the foreign method to the owner of the server class and ask the owner to implement the method for you.

Mechanics

- Create a method in the client class that does what you need.
  - ⇛ *The method should not access any of the features of the client class. If it needs a value, send it in as a parameter.*
- Make an instance of the server class the first parameter.
- Comment the method as "foreign method; should be in *server*."
  - ⇛ *This way you can use a text search to find foreign methods later if you get the chance to move the method.*

Example

I have some code that needs to roll over a billing period. The original code looks like this:

```
Date newStart = new Date (previousEnd.getYear(),
    previousEnd.getMonth(), previousEnd.getDate() + 1);
```

I can extract the code on the right-hand side of the assignment into a method. This method is a foreign method for date:

```
Date newStart = nextDay(previousEnd);

private static Date nextDay(Date arg) {
// foreign method, should be on date
    return new Date (arg.getYear(),arg.getMonth(), arg.getDate() + 1);
}
```

Introduce Local Extension

A server class you are using needs several additional methods, but you can't modify the class.

Create a new class that contains these extra methods. Make this extension class a subclass or a wrapper of the original.

Motivation

Authors of classes sadly are not omniscient, and they fail to provide useful methods for you. If you can modify the source, often the best thing is to add that method. However, you often cannot modify the source. If you need one or two methods, you can use *Introduce Foreign Method (162)*. Once you get beyond a couple of these methods, however, they get out of hand. So you need to group the methods together in a sensible place for them. The standard object-oriented techniques of subclassing and wrapping are an obvious way to do this. In these circumstances I call the subclass or wrapper a local extension.

A local extension is a separate class, but it is a subtype of the class it is extending. That means it supports all the things the original can do but also adds the extra features. Instead of using the original class, you instantiate the local extension and use it.

By using the local extension you keep to the principle that methods and data should be packaged into well-formed units. If you keep putting code in other classes that should lie in the extension, you end up complicating the other classes, and making it harder to reuse these methods.

In choosing between subclass and wrapper, I usually prefer the subclass because it is less work. The biggest roadblock to a subclass is that it needs to apply at object-creation time. If I can take over the creation process that's no problem. The problem occurs if you apply the local extension later. Subclassing forces me to create a new object of that subclass. If other objects refer to the old one, I have two objects with the original's data. If the original is immutable, there is no problem; I can safely take a copy. But if the original can change, there is a problem, because changes in one object won't change the other and I have to use a wrapper. That way changes made through the local extension affect the original object and vice versa.

Mechanics

- Create an extension class either as a subclass or a wrapper of the original.
- Add converting constructors to the extension.
  - ⇛ *A constructor takes the original as an argument. The subclass version calls an appropriate superclass constructor; the wrapper version sets the delegate field to the argument.*
- Add new features to the extension.
- Replace the original with the extension where needed.
- Move any foreign methods defined for this class onto the extension.

Examples

I had to do this kind of thing quite a bit with Java 1.0.1 and the date class. The calendar class in 1.1 gave me a lot of the behavior I wanted, but before it arrived, it gave me quite a few opportunities to use extension. I use it as an example here.

The first thing to decide is whether to use a subclass or a wrapper. Subclassing is the more obvious way:

```
class MfDateSub extends Date {
  public MfDateSub nextDay()...
  public int dayOfYear()...
```

A wrapper uses delegation:

```
class mfDateWrap {
  private Date _original;
```

Example: Using a Subclass

First I create the new date as a subclass of the original:

```
class MfDateSub extends Date
```

Next I deal with changing between dates and the extension. The constructors of the original need to be repeated with simple delegation:

```
public MfDateSub (String dateString) {
      super (dateString);
};
```

Now I add a converting constructor, one that takes an original as an argument:

```
public MfDateSub (Date arg) {
    super (arg.getTime());
}
```

I can now add new features to the extension and use *Move Method (142)* to move any foreign methods over to the extension:

```
client class...
  private static Date nextDay(Date arg) {
  // foreign method, should be on date
      return new Date (arg.getYear(),arg.getMonth(), arg.getDate() + 1);
  }
```

becomes

```
class MfDateSub...
  Date nextDay() {
      return new Date (getYear(),getMonth(), getDate() + 1);
}
```

Example: Using a Wrapper

I start by declaring the wrapping class:

```
class mfDateWrap {
  private Date _original;
}
```

With the wrapping approach, I need to set up the constructors differently. The original constructors are implemented with simple delegation:

```
public MfDateWrap (String dateString) {
   _original = new Date(dateString);
};
```


The converting constructor now just sets the instance variable:

```
public MfDateWrap (Date arg) {
    _original = arg;
}
```

Then there is the tedious task of delegating all the methods of the original class. I show only a couple.

```
public int getYear() {
    return _original.getYear();
}

public boolean equals(Object arg) {
    if (this == arg) return true;
    if (! (arg instanceof MfDateWrap)) return false;
    MfDateWrap other = ((MfDateWrap) arg);
    return  (_original.equals(other._original));
}
```

Once I've done this I can use *Move Method (142)* to put date-specific behavior onto the new class:

```
client class...
  private static Date nextDay(Date arg) {
  // foreign method, should be on date
      return new Date (arg.getYear(),arg.getMonth(), arg.getDate() + 1);
  }
```

becomes

```
class MfDateWrap...
  Date nextDay() {
      return new Date (getYear(),getMonth(), getDate() + 1);
}
```

A particular problem with using wrappers is how to deal with methods that take an original as an argument, such as

```
public boolean after (Date arg)
```

Because I can't alter the original, I can only do after in one direction:

```
aWrapper.after(aDate)                  // can be made to work
aWrapper.after(anotherWrapper)         // can be made to work
aDate.after(aWrapper)                  // will not work
```

The purpose of this kind of overriding is to hide the fact I'm using a wrapper from the user of the class. This is good policy because the user of wrapper really

shouldn't care about the wrapper and should be able to treat the two equally. However, I can't completely hide this information. The problem lies in certain system methods, such as `equals`. Ideally you would think that you could override `equals` on `MfDateWrap` like this

```
public boolean equals (Date arg)        // causes problems
```

This is dangerous because although I can make it work for my own purposes, other parts of the java system assume that equals is symmetric: that if `a.equals(b)` then `b.equals(a)`. If I violate this rule I'll run into a bevy of strange bugs. The only way to avoid that would be to modify Date, and if I could do that I wouldn't be using this refactoring. So in situations like this I just have to expose the fact that I'm wrapping. For equality tests this means a new method name.

```
public boolean equalsDate (Date arg)
```

I can avoid testing the type of unknown objects by providing versions of this method for both Date and MfDateWrap.

```
public boolean equalsDate (MfDateWrap arg)
```

The same problem is not an issue with subclassing, if I don't override the operation. If I do override, I become completely confused with the method lookup. I usually don't do override methods with extensions; I usually just add methods.