Chapter 5: Towards a Catalog of Refactorings

Chapters 6 to 12 form an initial catalog of refactorings. They've grown over the notes I've made in carrying out refactoring over the last few years. They are by no means comprehensive or watertight, but they should provide a solid starting point for your own refactoring work.

Format of the Refactorings

As I describe the refactorings in this, and other chapters, I've adopted a standard format to make it easier to work my around them as I read them. Each refactoring has the following parts

- I begin with a **name**. A name is important so that we can build a vocabulary of refactorings. This is the name I use elsewhere within this book.
- I follow the name with a short summary of the situation where you need the refactoring and a summary of what the refactoring does. This is there to help you find a refactoring more quickly.
- The **motivation** describes why the refactoring should be done, and also describes circumstances when it shouldn't be done.
- The **mechanics** are a concise, step by step description of how to carry out the refactoring. They come from my own notes to remember how to do the refactoring when I haven't done it for a while. I written the mechanics so I can carry out the refactoring with each step being as small as possible. In this I've stressed the safe way of doing the refactoring, which is to take very small steps, testing after every one. In reality I usually take larger steps than some of the baby steps here, but if I run into a bug I back out that step and take the smaller steps. The mechanics are somewhat terse, I give more long-winded explanations in the example.

- The **examples** are of the laughably simple textbook kind. My aim with the example is to help explain the basic refactoring with minimal distractions, so I hope you'll forgive the simplicity. (They are certainly not examples of good business object design.) I'm sure you'll be able to apply them to your rather more complex situations. There are one or two very simple refactorings that don't have an example as I didn't think the example would add much.

Finding References

Many of the refactorings call for you to find all references to a method, or a field, or a class. When you do this, enlist the computer to help you. By using the computer you reduce your chances of missing a reference, and can usually do it much more quickly than you would if you just eyeball the code.

Most languages treat computer programs as text files. Your best help here is a suitable text search. Many programming environments allow you to text search a single file or a group of files. The access control of the feature you are looking for will tell you what range of files you need to look for, (in an untyped language err on the cautious side).

Don't just search and replace blindly, inspect each reference to ensure it really refers to the thing you are replacing. You can get clever with your search pattern, but I always check mentally to ensure I am making the right replacement. When you can use the same method name on different classes, or on methods of a different signature on the same class there are too many chances you will get it wrong.

In a strongly typed language you can let the compiler help you do the hunting. You can often remove the old feature and let the compiler find the dangling references. The good thing about this is that the compiler will catch every dangling reference. However there are problems with this technique. Firstly the compiler will get confused when a feature is declared more than once in an inheritance hierarchy. This is particularly true when you are looking at a method that is overridden several times. If you are working in a hierarchy use the text search to see if any other class declares the method you are manipulating. The second problem is that the compiler may well be too slow to make this effective. If so use a text search first, at least the compiler double-checks your work. This only works when you intend to remove the

feature. Often you want to look at all the uses to decide what to do next. In these cases you have to use the text search alternative.

Some Java environments, notably IBM's VisualAge, are following the example of the Smalltalk Browser. With these you use menu options to find references instead of using text searches.

This is because Smalltalk and similar environments do not use text files to hold the code, instead they uses an in-memory database. You can take any method and ask for a list of its senders (which methods invoke the method) and implementers (which classes declare and implement the method). Get used to using those and you will find them often superior to the unavailable text search. A similar menu item will get you all references to an instance variable. (In Smalltalk you do have to be careful about methods declared on more than one class, as it is an untyped language.)

How Mature are These Refactorings?

Any technical author has the problem of deciding when to publish. The earlier you publish, the quicker people can take advantage of the ideas. However people are always learning and if you publish half-baked ideas too early, they can be incomplete and even lead to problems for those who try to use them.

The basic technique of refactoring, small steps and test often, has been well tested over many years, especially in the Smalltalk community. So I'm confident that the basic idea of refactoring is very stable.

The refactorings in this book are my notes of the refactorings I use. I have used them all. However there is a difference between using the refactoring and boiling them down into the mechanical steps I give here. I cannot say that I have had a lot of people work from these steps to spot the problems that can crop up in odd cases.

So as you use the refactorings bear in mind that they are there as a starting point. You will doubtless find gaps in them. But I'm publishing them now because although they are not perfect I do believe they will be useful. I think they will give you a starting point that will improve your ability to refactor efficiently. That is what they do for me.

Chapter 6: Dealing with Long Methods

Long methods are usually a prime target in my refactoring, for they often contain lots of information which gets buried by the complex logic which usually gets dragged in there. The key refactoring is *Extract Method (114)*, which takes a clump of code and turns it into its own method. *Inline Method (120)* is essentially the opposite of this, you take a method call and replace it with the body of the code. I need *Inline Method (120)* 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 (114)* is dealing with local variables, and temps are one of the main sources of this issue. So when I'm working on a method I like to *Inline Temp (121)* to get rid of any temporary variables that I can remove. If the temp is used for many things, I use *Split Temporary Variable (124)* first to make it easier to replace.

Sometimes, however, the temporary variables are just too tangled to replace, then I need to *Replace Method with Method Object (130)*. This will allow 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 providing you don't assign to them. If you do, you need to *Remove Assignments to Parameters (126)*.

Once the method is broken down, I can understand how it works much better. I may also find that the algorithm could be improved to make it clearer. I then use *Substitute Algorithm (132)* to introduce the clearer algorithm.

Extract Method

You have a code fragment that can be grouped together

Create a new method. Make the fragment the body of the method. Replace the fragment with a call to the newly extracted method.

Motivation

This is one of the most common refactorings I do. I look at a method that is too long, or look at some 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. Firstly it increases the chances that other methods can use a method when the method is fine grained. Secondly it allows the higher level methods to read more like a series of comments. Overriding is also easier when the methods are fine grained.

It does take a little getting used to if you are used to seeing larger methods. And small methods only really work when you have good names, so you need to pay attention to naming. In the end the key to when to do this 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, name it after the intention of the method (name it by what it does, not how it does it)
  - ☞ If the code you are looking to extract is very simple, such as a single message or function call, then you should still extract it if the new method's name will reveal the intention of the code in a better way. If you can't come up with a more meaningful name, then 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 which are local in scope to the source method. These are local variables and

parameters to the method

- See if any temporary variables are only used within this extracted code. If so declare them in the target method as temporary variables.
- Look to see if any of these local scope variables are modified by the extracted code. If one of them is modified see if 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, then you can't extract the method as it stands. You may need to use *Split Temporary Variable (124)* and try again. You can eliminate temporary variables with *Inline Temp (121)*. (See the discussion in examples.)
- Those local scope variables which are read from in the extracted code must be passed into the target method as parameters.
- When you have dealt with all the locally scoped variables, compile.
  - ☞ In most C++ or Java environments you will need to recompile the class. In Smalltalk, and more sophisticated environments you can just accept to recompile the method.
- 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 if they were declared outside of the extracted code. If so you can now remove the declaration.
- Compile and test.

Examples

You'll find examples of this all over the book. The simple cut and paste cases are easy to do. All the complication in this refactoring comes from those pesky locally scoped variables: temps and parameters.

The easiest case is when you only need to read the temp

```
temp = expression;
…
// some extracted code
foo = temp.method();
// some more extracted code
```

Turns into

```
temp = expression;
newMethod(temp);

void newMethod (type temp) {
  // some extracted code
```

```
  foo = temp.method();
  // some more extracted code
}
```

Although if you can, it is better to replace the temp by a query.

```
newMethod();

void newMethod() {
  // some extracted code
  foo = newQuery().method();
  // some more extracted code
}

tempType newQuery() {
  return expression
}
```

If it is only code in the extracted method that uses the temp, then you should move the temp entirely within the extracted method.

```
newMethod();

void newMethod() {
  // some extracted code
  tempType temp = expression
  foo = temp.method();
  // some more extracted code
}
```

So that's pretty straightforward. So let me rephrase the problem: it's those pesky temps that you change.

Then the issue turns around the nature of the change. If you change the temp by invoking a modifier on the temp then that's fine. The fact that the object is referenced outside of the method will just mean that it is changed there after the method. This implies that the method is a modifier and thus should not return a value. If it does, then you should *Separate Query from Modifier (236)* later on.

So to be more precise: it's those pesky value objects that you reassign. Cases like

```
tempType temp;
…
// some extracted code
temp = expression;
// some more extracted code
…
```

```
foo = temp.method();
```

Or

```
tempType temp;
…
temp = expression;
foo = temp.method()
…
// some extracted code
temp = newExpression;
// some more extracted code
```

Or

```
temp = expression;
…
// some extracted code
temp = temp + newExpression;
// some more extracted code
…
foo = temp.method()
```

The first issue is how many temps you reassign. If you change only one variable then you can use it as the method's return value, I'll go into those cases in a moment. Before that I'll answer the inevitable question "what if there are more than one temp being reassigned?" In that second case I would not do the extraction. Instead I'd look for another extraction, or do something to get the number of reassigned temps down to one.

You'll also have noticed that I'm no longer talking about locally scoped variables, I'm only talking about temps. What happened to parameters? Well I don't reassign to passed in parameters as a matter of style, for I find that gets too confusing. In the principle OO languages (Smalltalk, C++, and Java) assignments to parameters inside a method are local to that method. In the calling method, the parameter retains its original value.

If I see any parameters assigned to, I *Remove Assignments to Parameters (126)*. So then this issue is about temps. Of course if your programming language allows you to reassign parameters then you can use the parameters to deal with multiple assigned temps. (Although I've not missed the ability to do this.) C and C++ use pointer variables to mimic this approach. Since this is primarily a Java book, I'll skip discussing how to deal with that, for the very good reason that I don't

have enough experience doing this with C/C++ to provide sound advice.

So onto the cases I outlined above. The first case is where you initialize the temp in the extracted code.

```
tempType temp;
…
// some extracted code
temp = expression;
// some more extracted code
…
foo = temp.method();
```

In this case I can return the temp as the result of the extracted method.

```
tempType temp;
temp = newMethod();
foo = temp.method();

tempType newMethod() {
  tempType result;
  // some extracted code
  result = expression;
  // some more extracted code
}
```

This works best when you don't have any code after the assignment to the result, then you can just return it.

```
tempType newMethod() {
  // some extracted code
  return expression;
}
```

This approach is most satisfying when the extracted method's purpose is about calculating the value of the temp. If the method does more than that, it questions whether this is the right block of code to extract. Even if it makes sense for the moment, there is more refactoring to do later.

A variation on this case is where the temp is initialized but not used before the extracted method. In that case you can remove the first (unused) initialization.

Another case is where the temp is initialized and used before the extracted method reassigns the value.

```
tempType temp;
…
```

```
temp = expression;
foo = temp.method()
…
// some extracted code
temp = newExpression;
// some more extracted code
```

In this case you should use *Split Temporary Variable (124)*, once you've done that you'll find yourself with a simpler case.

The exception to splitting the temp is when you add to the temp

```
temp = expression;
…
// some extracted code
temp = temp + newExpression;
// some more extracted code
…
foo = temp.method()
```

Since here you are still using the temp for the same purpose, you should not split it. Instead you should pass the old value in and return the new.

```
temp = expression;
temp = newMethod(temp);
…
foo = temp.method();

tempType newMethod (tempType temp) {
  tempType result;
  // some extracted code
  result = temp + newExpression;
  // some more extracted code
  return result;
}
```

If you don't read the temp in the extracted method, you don't need to pass it in. Instead you can just return the changed value

```
temp = expression;
temp = temp + newMethod();
…
foo = temp.method();

tempType newMethod () {
  tempType result;
  // some extracted code
  result = newExpression;
  // some more extracted code
  return result;
```

```
}
```

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

Motivation

A theme of this book is to use short methods which are named to show their intention, as these methods lead to clearer and easier to read code. But sometimes you do come across a method where 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 just irritating.

Another time to do this is where you have a group of methods which seem badly factored. You can inline them all into one big method and then re-extract the methods. Kent finds it is often good to do this before using *Replace Method with Method Object (130)*. You do this by inlining the various calls that the method makes which have behavior you want to have on the method object. It's easier to move one method than the method and its called methods.

Mechanics

- ❍ Check the method is not polymorphic
  - ☞ Don't inline if subclasses override the method, for they cannot override a method that isn't there.
- ❍ Find all calls to the method
- ❍ Replace each one with the method body
- ❍ Compile and test
- ❍ Remove the method definition

Inline Temp

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 a reference to the new method.

Motivation

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

This is often a vital step before *Extract Method (114)*. Local variables make it difficult to extract, so replace as many as you can with queries.

The straightfoward case of this refactoring are those temps that are only assigned to once, and where the expression that generates the assignment is free of side effects. Other cases are more tricky but still possible. You may need to use *Split Temporary Variable (124)* or *Separate Query from Modifier (236)* first to make things easier. If the temp is used to collect a result (such as summing over a loop), you will need to copy some logic into the query method.

Mechanics

Here is the simple case

- Look for a temporary variable that is set once with an assignment.
  - If a temp is not set once consider *Split Temporary Variable (124)*.
- Declare the temp as `final`
- 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 relax the protection easily later.
  - Ensure the extracted method is free of side effects. If not you should *Separate*

Query from Modifier (236).

- Compile and test
- Find all references to the temp and replace them with a call to the new method.
- Compile and test after each change.
- Remove the declaration and the assignment of the temp.
- Compile and test.

Temps are often used to store summary information in loops. The whole loop can be extracted into a method, in Java or C++ this removes several lines of noisy code. Sometimes a loop may be used to sum up multiple values, as in the example on page 28. 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 with this case. Don't be concerned with performance when refactoring. If you find the loop to be a performance problem when you profile, then do something to fix it. You may end up with one loop summing multiple values. So be it, you trade design clarity for performance. That is a good trade-off if (and only if) it solves a genuine performance problem.

Example

You have 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'd be inclined to inline both temps. Although it's pretty clear in this case, I can test that they are only assigned 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 then inline them one at a time. First I extract the right hand side of the assignment.

```
double getPrice() {
    int basePrice = basePrice();
    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 replace the first reference to the temp.

```
double getPrice() {
    int basePrice = basePrice();
    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). As it's the last I'd also remove the temp declaration.

```
double getPrice() {
    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() {
    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 like.

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

Split Temporary Variable

You have a temporary variable that assigned to more than once, but is not a *Loop Variable* nor a *Collecting Temporary Variable*.

Make a separate temporary variable for each assignment.

Motivation

Temporary variables are made for various uses. Some of these uses naturally lead to the temp being assigned to several times. *Loop Variables* [Beck] change for each run around a loop. *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 some long-winded bit of code for easy reference later. These kinds of variables should be only set once. If they are set more than once, that is a sign that they have more than one responsibility within the method. Any variable with more than one responsibility should be replaced by a temp for each responsibility. Using a temp for two different things is very confusing for the reader.

Mechanics

- Change the name of temp at its declaration and its first assignment.
  - ☞ If the later assignments are of the form 'i = i + some expression' then that indicates that it is a Collecting Temporary Variable, so don't split it. The operator for a Collecting Temporary Variable is usually 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 by stages, each stage renaming at the declaration, and changing references until the next assignment.

Example

For this example I will 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. So using the common laws of motion I can compute the distance traveled as

```
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 due to the first force, and then later to hold the acceleration with both forces. This I would like 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;
}
```

Then I continue, section by section until the original temp is no more.

```
double getDistanceTravelled (int time) {
    double result;
    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'll be 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.

Motivation

First let me make sure we are clear on the phrase 'assigns to a parameter'. This means that if you pass in some object in the parameter named foo, 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 got passed in — I do that all the time. I just object with 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 also the confusion between pass by value and pass by reference. Java uses pass by value exclusively (see note below) and this discussion is based on that usage (In Smalltalk you cannot assign to a parameter).

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 only use the parameter to represent what has been passed in, as that is a consistent usage.

Of course this convention is a matter of style which does vary with language and environment. A number of approaches use output parameters to pass multiple values back to the caller. Personally I'm not too keen on this style, but that's more of a style issue. If you use input and output parameters then this refactoring does not apply to them. However I do recommend keeping output parameters to a minimum and using return values of functions whenever you can.

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 change the temporary variable.
- Compile and test
  - ☞ If the semantics are call by reference, look in the calling method to see if the parameter is used again afterwards, also see how many call by reference parameters are assigned to and used afterwards in this method. Try to pass a single value back as the return value. If there is more than one see if you can turn the data clump into an object, or create separate methods.

Example

We'll 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'll admit I don't use final much, as I don't find it helps much with clarity for short methods. I would use it with a long method, to help me see if anything was changing the parameter.

Pass by Value in Java

This issue is often a source of confusion in Java. Strictly Java uses pass by value in all places. Thus the program

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

Produces this output

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

The confusion exists with objects. Say we 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);
    }
}
```

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 it is the object reference that is passed by value, allowing you to modify the object, but not taking into account the reassigning of the parameter.

Java 1.1 allows you to mark a parameter as *final*, this prevents assignment to the variable. It does still allow you to modify the object that variable refers to.

Using Pass by Reference

Many languages allow pass by reference. In this case any assignment in the function body is reflected in the calling program. Often languages use special keywords, such as VAR in Pascal, to signal this. C uses pass by value only, but programmer frequently get pass by reference semantics by passing in pointers (which you cannot do with Java).

Objects help you to reduce parameter lists, they also help you to reduce what you need to return. Here's some of my advice on parameter passing when pass by reference is allowed

- Signal clearly those parameters that are carrying output back to the caller

- Try to use a single return value rather than parameters
- Use exceptions to report errors, rather than returning an error code
  - ☞ Return codes signaling errors are a common C idiom, but the exception handling in modern languages is a much better mechanism.
- Use *Introduce Parameter Object (247)* to reduce the number of values that need to be returned.
- Consider splitting the method into different methods for each value you are returning.

Replace Method with Method Object

Motivation

In this book I've stressed 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 too rampant it can be difficult to do the decomposition. Replacing temps with queries 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].

Mechanics

(Stolen shamelessly from [Beck])

- Create a new class, name it after the method
- Give the new class a field for 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. Since 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'll show 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) + 50;
      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 field for the original object, for each parameter, and for each temporary variable in the method.

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

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

That's the essential refactoring. The benefit is that I can now easily do decompositions of the compute method, without ever worrying about argument 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. Use the old body for comparative testing.

Motivation

I've never tried to skin at cat. I'm told there's several ways to do it. I'm sure some are easier than others. So it is with algorithms. If you need to modify how you do something, life is easier if the algorithm is

designed a certain way. It's a lot easier if you can understand what's going on.

Refactoring can break down something complex into simpler pieces, but sometimes you just reach a point where you have to remove the whole algorithm and replace it with something simpler.

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 done.
- If they 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.

Example

You can find an example for this at page 419.