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.

Chapter 7: Simplifying Conditional Expressions

Conditional logic has a way of getting tricky, so here are a number of refactorings you can use to simplify it. The core refactoring here is *Decompose Conditional (136)* which talks about how to decompose the conditional into pieces. In many ways it is an obvious refactoring, but it's one that often makes a huge difference to the clarity of code.

The other refactorings here look into other important cases. Use *Consolidate Conditional Expression (137)* when you have several tests all with the same effect, use *Consolidate duplicate conditional fragments (140)* to remove any duplication within the conditional code.

If you are working with code developed in a one-exit point mentality you often find control flags which are there to allow the conditions to work with this rule. I don't follow the rule about one exit point from a method. Hence I'll use *Replace Nested Conditional with Guard Clauses (145)* to clarify special case conditionals and *Remove Control Flag (141)* to get rid of the awkward control flags.

Object-oriented programs often have less conditional behavior than procedural programs because much of the conditional behavior is handled by polymorphism. Polymorphism is better because the caller does not need to know about the conditional behavior and it is thus easier to extend the conditions. As a result object-oriented programs rarely have switch (case) statements. So any that show up are prime candidates for *Replace Switch with Polymorphism (147)*.

One of the most useful, but less obvious, uses of polymorphism is to use *Introduce Null Object (151)* to remove checks for a null value.

Decompose Conditional

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

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

Motivation

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

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

Mechanics

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

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

Example

Suppose you are calculating the charge for something that has a separate rate for winter and for summer

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

Extract the conditional and each leg into

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

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

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

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

Here I've shown the result of the complete refactoring for clarity. In practice, however, I would do each extraction separately, compiling and testing after each one.

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

You can find a larger example of this on page 409.

Consolidate Conditional Expression

You have a sequence of conditional tests with the same result

Combine them into a single conditional expression, and extract it

Motivation

Sometimes you see a series of conditional checks, where each check is different yet the resulting action is the same. When you see this you should consolidate them into a single conditional check, using ands and ors, with the single result.

Consolidating the conditional code is important for two reasons. Firstly it makes the check clearer by showing that you are really making a single check that's or'ing the other checks together. The sequence has the same effect but it communicates carrying out a sequence of separate checks that just happen to be done together. The second reason for this refactoring is the most compelling: it sets you up to do *Extract Method (114)*. Extracting a condition is one of the most useful things you can do to clarify your code, for it replaces a statement of what you are doing with why you are doing it.

Those reasons in favor of consolidating conditionals also point to when you shouldn't do it. If you think the checks are really independent, and shouldn't be thought of as a single check; then don't do the refactoring. Your code already communicates your intention.

Mechanics

- Replace the string of conditionals with a single conditional statement using logical operators
- Compile and test
- Extract the condition into its own method

Example

The state of the code for this is along the lines of the following.

```
Money pay {
  If (dead()) return Money.dollars(0);
  If (sick()) return Money.dollars(0);
  If (retired()) return Money.dollars(0);
  Return doPay();
}
```

Here we see a sequence of conditional checks that all result in the same thing. With sequential code like this they are the equivalent of an or statement.

```
If (dead() || sick() || retired()) return Money.dollars(0)
```

```
return normalPayAmount()
```

Now I can look at the condition and use *Extract Method (114)* to communicate what the condition is looking for.

```
Money pay() {
  If (nonPayable()) return Money.dollars(0)
  else return normalPayAmount()
}

private boolean nonPayable() {
  return (dead() || sick() || retired());
}
```

That example showed ors, but you can do the same with ands. Here the set up is something like

```
if (onVacation())
    if (lengthOfService() > 10)
        return 1;
return 0.5;
```

This would be changed to

```
if (onVacation() && lengthOfService() > 10) return 1;
else return 0.5;
```

You may well find you get a combination of these that yield an expression with ands ors and nots.

If the routine you are looking at only tests the condition and returns a value; then you can turn the routine into a single return statement using the tertiary operator. So

```
if (onVacation() && lengthOfService() > 10) return 1;
else return 0.5;
```

becomes

```
return (onVacation() && lengthOfService() > 10) ? 1 : 0.5;
```

In this case the condition speaks for itself as clearly as any method name would, so I won't extract it.

Consolidate duplicate conditional fragments

The same fragment of code is in all branches of a conditional expression

Move it outside of the expression

Motivation

Sometimes you will find the same code executed in all legs of a conditional. In that case you should move the code to outside the conditional. This makes it clearer as it what varies, and what stays the same.

Mechanics

- Identify code that gets executed the same regardless of the condition
- If the common code is at the beginning, then move it to before the conditional
- If it is at the end, move it to after
- If it is in the middle look to see if the code before or after it changes anything. If so you can move the common code forwards or backwards to the ends you can then move it as above
- If there is more than a single statement then you should extract that code into a method.

Example

The kind of situation you find this is with code like

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
}
else {
    total = price * 0.98;
    send();
}
```

Since the send method is executed in either case you should move it out of the conditional.

```
if (isSpecialDeal())
```

```
        total = price * 0.95;
    else
        total = price * 0.98;
    send();
```

The same situation can apply to exceptions. If code is repeated after an exception causing statement in the try block and all the catch blocks, then you can move it to the finally block.

Remove Control Flag

You have a control flag that is acting as a control flag for a series of boolean expression.

Use a break or return instead

Motivation

When you have a series of conditional expressions, you often see a control flag used to determine when to stop looking.

```
set done to false
while not done
  if (condition)
      do something
      set done to true
  next step of loop
```

Such control flags are more trouble than they are worth. They come from the rules of structured programming that call for routines with one entry and one exit point. I agree with (and modern languages enforce) one entry point, but the one exit point rule leads you to very convoluted conditionals with these awkward flags in the code. This is why modern languages often have the break and continue statements to get out of a complex conditional. It is often surprising what you can do when you get rid of a control flag, the real purpose of the conditional becomes so much more clear.

Mechanics

The obvious way to deal with this is using the break or continue statements present in Java.

- Find the value of the control flag that get's you out of the logic statement
- Replace assignments of the break out value with a break or continue statement
  - ☞ Break is used to end processing in that code fragment, continue is used to take another trip round a loop.
- Compile and test after each replacement

Another possibility, usable in languages without break and continue

- Extract the logic into a method
- Find the value of the control flag that get's you out of the logic statement
- Replace assignments of the break out value with a return
- Compile and test after each replacement

Indeed even in languages with a break or continue, I quite like the use of a extraction and the use of return. Often if you have that kind of code, you need to extract that piece anyway.

Keep an eye on whether the control flag also indicates some result information. If so you still need it if you use the break, or you can return the value if you have extracted a method.

Example

The following function checks to see if a list of people contains a couple of hard coded suspicious characters.

```
void checkSecurity(String[] people) {
    boolean found = false;
    for (int i = 0; i < people.length; i++) {
        if (! found) {
           if (people[i].equals ("Don")){
              sendAlert();
              found = true;
           }
           if (people[i].equals ("John")){
              sendAlert();
              found = true;
           }
        }
```

```
    }
}
```

In a case like this it is easy to see the control flag, it's the piece that sets the found variable to true. I can introduce the breaks one at a time

```
void checkSecurity(String[] people) {
    boolean found = false;
    for (int i = 0; i < people.length; i++) {
        if (! found) {
           if (people[i].equals ("Don")){
             sendAlert();
             break;
           }
           if (people[i].equals ("John")){
             sendAlert();
             found = true;
           }
        }
    }
}
```

Until I have them all.

```
void checkSecurity(String[] people) {
    boolean found = false;
    for (int i = 0; i < people.length; i++) {
        if (! found) {
           if (people[i].equals ("Don")){
             sendAlert();
             break;
           }
           if (people[i].equals ("John")){
             sendAlert();
             break;
           }
        }
    }
}
```

Then I can remove all references to the control flag

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

```
    }
}
```

Sometimes logic like this also uses a result.

```
void checkSecurity(String[] people) {
    String found = "";
    for (int i = 0; i < people.length; i++) {
        if (found.equals("")) {
           if (people[i].equals ("Don")){
             sendAlert();
             found = "Don";
           }
           if (people[i].equals ("John")){
             sendAlert();
             found = "John";
           }
        }
    }
    someLaterCode(found);
}
```

Here found is doing two things, it is both indicating a result and acting as a control flag. When I see this I like to extract the code that is determining found into its own method.

```
void checkSecurity(String[] people) {
    String found = foundMiscreant(people);
    someLaterCode(found);
}

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

Then I can replace the control flag with a return.

```
String foundMiscreant(String[] people){
    String found = "";
```

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

Until I have removed the control flag.

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

Of course this has the problem of a function with side-effects. So I want to *Separate Query from Modifier (236)*. You'll find this example continued there....

Replace Nested Conditional with Guard Clauses

A method has conditional behavior that does not make clear what the normal path of execution is

Use Guard Clauses for all the special cases

Motivation

I often find that conditional expressions come in two forms. The first form is a check whether either course is part of the normal behavior, the second case is where one answer from the conditional indicates normal behavior and the other indicates an usual error condition.

These kinds of conditionals have a different intention, and this intention should come through in the code. If both are part of normal behavior, then use a condition with an if and an else leg. However if the condition is a rare error condition then check the condition and return if the condition is true. This kind of check is often called a *Guard Clause* [Beck].

The key point about this refactoring is one of emphasis. If you are using an if-then-else construct you are giving equal weight to the if leg and the else leg. This communicates to the reader that they are equally likely and important. Instead the guard clause says “this is rare, and if it happens just get out”.

I often find I use this refactoring when I’m working with a programmer who has been taught to have only one entry and one exit point from a method. One entry point is enforced by modern languages, and one exit point is really not a useful rule. Clarity is the key principle: if it is clearer with one exit point then use one exit point, otherwise don’t.

Mechanics

- For each check put the guard clause in.
  - The guard clause will either return, or throw an exception.
- Compile and test after each check is replaced with a guard clause.
  - If all the guard clauses yield the same result then *Consolidate the Conditional Expressions*.

Example

Imagine a run of a payroll system where the rule is that you cannot pay employees that are dead, sick, or retired. Furthermore earlier processing should have removed all such employees from those you are considering, but you are aware that an occasional incorrect employee slips in. So you still have to go through the checks.

If you write the code like this

```
Money payAmount() {
  Money result;
  If (dead()) result = deadAmount()
  else {
      If (sick()) result = sickAmount()
      else {
          If (retired()) result = retiredAmount()
          else result = normalPayAmount()
      };
  }
return result;
};
```

Then the checking is masking the normal course of action behind the checking. So instead it is clearer to use guard clauses [Beck].

```
Money payAmount() {
  If (dead()) return deadAmount();
  If (sick()) return sickAmount();
  If (retired()) return retiredAmount();
  Return doPay();
}
```

The important point about this example is the phrase "earlier processing should have removed all such employees". If this was the routine that was primarily responsible for the checking then I would be less inclined to do this refactoring.

Replace Switch with Polymorphism

You have a case statement that switches on a type code

Move each leg of the case statement to a subclass

Motivation

One of the most obvious symptoms of object-oriented code is its lack of case statements, especially those that depend on some kind of enumerated type code. Such statements look like this

```
switch (enum)
  case value1:
      doSomething1;
  case value2:
```

```
        doSomething2;
    case value3:
        doSomething2;
```

or this

```
if (var == value1) doSomething1;
else if (var == value2) doSomething2;
else if (var == value3) doSomething2;
```

or this

```
if (isSomething) doSomething1
else doSomething2
```

You can replace these statements by subclasses and polymorphism. Such a replacement gives you many advantages. The biggest gain occurs when this same case statement appears in many places in the program. If you want to add a new case you have to find and update all the case statements. But with subclasses you just create a new subclass and provide the appropriate methods. Clients of the class don't need to know about the subclasses, which reduces the dependencies in your system, making it easier to update.

Mechanics

The first step here is to create the necessary inheritance structure. If you have several case statements switching on the same type code you only need to create on inheritance structure for that type code. To do this you have two options: straight subclassing or using the state/strategy pattern. The vanilla subclassing is the simplest and so you should use it if you can. If you update the type code after the object is created, however, you cannot use subclassing and have to use the state/strategy pattern instead. You also need to use the state/strategy pattern if you are already subclassing this class for some other reason.

I've described these refactorings under *Replace Type Code with Subclasses (224)* and *Replace Type Code with State/Strategy (227)*.

With that done you have the appropriate inheritance hierarchy set up. You can now attack the code. The code you target may be a switch (case) statement or an if statement.

- If the conditional statement is one part of a larger method then take the conditional statement part and use *Extract Method (114)*

- Pick one of the subclasses. Create a subclass method that overrides the conditional statement method. Copy the body of that leg of the conditional statement into the subclass method, and adjust it to fit.
  - ☞ You may need to make some private members of the superclass protected in order to do this.
- Compile and test
- Remove that leg of the conditional statement
- Compile and test.
- Repeat with each leg of the conditional statement until all legs are turned into subclass methods.
- Make the superclass method abstract

Example

I'll use the tedious and simplistic employee pay example. I'm using the classes after the use of the state pattern.

```
class Employee…
  int payAmount() {
      switch (getType()) {
          case EmployeeType.ENGINEER:
             return _monthlySalary;
          case EmployeeType.SALESMAN:
             return _monthlySalary + _commission;
          case EmployeeType.MANAGER:
             return _monthlySalary + _bonus;
          default:
             throw new RuntimeException("Incorrect Employee");
      }
  }

  int getType() {
      return _type.getTypeCode();
  }
  private EmployeeType _type;

abstract class EmployeeType…
  abstract int getTypeCode();

class Engineer…
  int getTypeCode() {
      return Employee.ENGINEER;
  }

… and other subclasses
```

The case statement is already nicely extracted, so there is nothing to do there. I do need to move it into the employee type, as that is the class that is being subclassed.

```
int payAmount(Employee emp) {
    switch (getTypeCode()) {
        case ENGINEER:
           return emp.getMonthlySalary();
        case SALESMAN:
           return emp.getMonthlySalary() + emp.getCommission();
        case MANAGER:
           return emp.getMonthlySalary() + emp.getBonus();
        default:
           throw new RuntimeException("Incorrect Employee");
    }
}
```

Since I need data from the employee, I need to pass in the employee as an argument. Some of this data might be moved to the employee type object, but that is an issue for another refactoring.

When this compiles I change the payAmount method in employee to delegate to the new class.

```
int payAmount() {
    return _type.payAmount(this);
}
```

Now I can go to work on the case statement. It's rather like the way small boys kill insects – I remove one leg at a time. First I copy the engineer leg of the case statement onto the engineer class.

```
class Engineer…
  int payAmount(Employee emp) {
      return emp.getMonthlySalary();
  }
```

This new method will override the whole case statement for engineers. If you want to be sure of that you can put in a trap in the case statement.

```
int payAmount(Employee emp) {
    switch (getTypeCode()) {
        case ENGINEER:
           throw new RuntimeException ("Should be being overridden");
        case SALESMAN:
           return emp.getMonthlySalary() + emp.getCommission();
        case MANAGER:
           return emp.getMonthlySalary() + emp.getBonus();
        default:
```

```
            throw new RuntimeException("Incorrect Employee");
        }
    }
```

Carry on until all the legs are removed.

```
class Salesman…
    int payAmount(Employee emp) {
        return emp.getMonthlySalary() + emp.getCommission();
    }

class Manager…
    int payAmount(Employee emp) {
        return emp.getMonthlySalary() + emp.getBonus();
    }
```

And then declare the superclass method as abstract.

```
    abstract int payAmount(Employee emp);
```

Introduce Null Object

You have repeated checks for a null value

Replace the null value with a null object [Woolf]

Motivation

The essence of polymorphism is that instead of asking an object what type it is and then invoking some behavior based on the answer; you just invoke the behavior. The object, depending on its type, does the right thing. One of the less intuitive places to do this is when you have a null value in a field. I'll let Ron Jeffries tell the story.

We first started using the Null Object pattern when Rich Garzaniti found that there was lots of code in the system that would check objects for presence before sending a message to it. We might ask an object for its person, then ask the result whether it was null, then if it was there, ask it for its rate. We were doing this in several places and resulting duplicate code was getting annoying.

So we implemented a missing person object that answered a zero rate (we call our null objects missing objects). Soon missing person knew a lot of methods like rate.

Now we have over eighty null object classes.

Our most common use of these objects is in the display of information. When we display, say, a person, the object may or may not have any of perhaps twenty instance variables. If these were allowed to be null, the printing of a person would be very complex. Instead, we plug in various null objects, all of which know how to display themselves in an orderly way. This got rid of huge amounts of procedural code.

Our most clever use of null object was the missing Gemstone session. We use the Gemstone database for production, but we prefer to develop without it and push the new code to Gemstone every week or so. There are various points in the code where we have to log into a Gemstone session. When we are running without Gemstone we simply plug in a missing Gemstone session. It looks the same as the real thing, but allows us to develop and test without realizing the database isn't there.

Another very helpful use was the missing bin. A bin is a collection of payroll values that often need to be summed or looped over. If a particular bin doesn't exist, we answer a missing bin, which acts just like an empty bin. The missing bin knows it has zero balance and no values. By using this approach, we eliminated the creation of tens of empty bins for each of our thousands of employees.

An interesting characteristic of using null objects is that things almost never blow up. Since the null object responds to all the same messages as a real one, the system generally behaves normally. This can sometimes make it difficult to detect or find a problem, because nothing ever breaks. Of course, as soon as you being inspecting the objects, you'll find the null object somewhere where it shouldn't be.

Remember, null objects are always constant: nothing about them ever changes. Accordingly, we implement them using the Singleton pattern [Gang of Four]. Whenever you ask for, say, a missing person, you always get the single instance of that class.

- - Ron Jeffries

Mechanics

- Create a subclass of the source class to act is a null version of the class. Create an `isNull` operation on the source class and the null class. For the source class it should return false, for the null class it should return true.
  - ☞ You may find it useful to create an explicit Nullable interface
- Compile
- Find all places that can give out a null when asked for a source object. Replace them to give out a null object instead.

- Find all places that compare a variable of the source type to null, and replace them with a call is `isNull`.
  - You may be able to do this by replacing one source and its users at a time, and compiling and testing between working on sources.
  - A few assertions that check for null in places where you should no longer see it can be useful.
  - This step is trivial in Smalltalk if you always use `isNil` rather than `==nil` to test for nilness.
- Compile and test
- Look at the responses to the null test. Often you will see cases where if the object is not null the client invokes an operation on the object. If null the client does something else. If several clients react to nullness in the same way, override the non-null operation in the null object class and provide the common client behavior.
- Remove the condition check, compile and test.

Example

A utility company knows about Site's: the houses and apartments that use the utility's services. At any time a site will have a customer.

```
class Site...
  Customer getCustomer() {
      return _customer;
  }
  Customer _customer;
```

There are various features of a customer. I'll look at three of them.

```
class Customer...
  public String getName() {...}
  public BillingPlan getPlan() {...}
  public PaymentHistory getHistory() {...}
```

The payment history has its own features.

```
public class PaymentHistory...
  int getWeeksDelinquentInLastYear()
```

The getters I've shown allow clients to get at this data. However sometimes we don't have a customer for a site. Someone may have moved out and we don't know who has moved in yet. Because this can happen we have to ensure any code that uses the customer can handle nulls.

```
Customer customer = site.getCustomer();
```

```
BillingPlan plan;
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
String customerName;
if (customer == null) customerName = "occupant";
else customerName = customer.getName();
int weeksDelinquent;
if (customer == null) weeksDelinquent = 0;
else weeksDelinquent = customer.getHistory().getWeeksDelinquentInLastYear();
```

In these situations you may have many clients of site and customer, all of which have to check for nulls and all of which do the same thing when it finds one. Sounds like its time for a null object.

The first step is to create the null customer class and modify the customer class to support a query for a null test.

```
class NullCustomer extends Customer {
  public boolean isNull() {
      return true;
  }
}

class Customer...
  public boolean isNull() {
      return false;
  }

  protected Customer() {} //needed by the NullCustomer
```

If you like you can signal the use of null object by an interface

```
interface Nullable {
  boolean isNull();
}

class Customer implements Nullable
```

I like to add a factory method to create null customers. That way clients don't have to know about the null class.

```
class Customer...
  static Customer newNull() {
      return new NullCustomer();
  }
```

Now comes the difficult bit. Now we have to return this new null object whenever we expect a null, and replace the tests of the form `foo == null` with tests of the form `foo.isNull()`. I find it useful to look for all

the places where you ask for a customer, and modify them so that they return a null customer rather than null.

```
class Site...
  Customer getCustomer() {
      return (_customer == null) ?
          Customer.newNull():
          _customer;
  }
```

I also have to alter all uses of this value so that they test with `isNull()` rather than `== null`.

```
Customer customer = site.getCustomer();
BillingPlan plan;
if (customer.isNull()) plan = BillingPlan.basic();
else plan = customer.getPlan();
String customerName;
if (customer.isNull()) customerName = "occupant";
else customerName = customer.getName();
int weeksDelinquent;
if (customer.isNull()) weeksDelinquent = 0;
else weeksDelinquent = customer.getHistory().getWeeksDelinquentInLastYear();
```

There's no doubt that this is the trickiest part of this refactoring. For each source of a null you replace, you have to find all the times it is tested for nullness and replace them. If the object is widely passed around, these can be hard to track. You have to find every variable of type customer and find everywhere it is used. It is hard to break this into small steps. Sometimes you find one source that is only used in a few places, and you can replace that source only. But most of the time you have to make many widespread changes. The changes aren't too difficult to back out, since you can find calls of isNull without too much difficulty, but this is still a messy step.

Once this step is done, and I've compiled and tested, I can smile. Now the fun begins. As it stands I gain nothing from using `isNull` rather than `== null`. The gain comes as I move behavior to the null customer and remove conditionals. I can do these moves one at a time. I begin with the name. Currently I have client code that says

```
String customerName;
if (customer.isNull()) customerName = "occupant";
else customerName = customer.getName();
```

I add a suitable name method to the null customer.

```
class NullCustomer...
```

```
public String getName(){
    return "occupant";
}
```

Now I can make the conditional code go away.

```
String customerName = customer.getName();
```

I can do the same for any other method where there is a sensible general response to a query. I can also do appropriate action for modifiers. So client code such as

```
if (! customer.isNull())
    customer.setPlan(BillingPlan.special());
```

can be replaced by

```
customer.setPlan(BillingPlan.special());

class NullCustomer...
  public void setPlan (BillingPlan arg) {}
```

Remember that this movement of behavior makes sense only when most clients want the same response. Notice I said "most" not "all". Any clients who want a different response to the standard one can still test using `isNull`. You benefit when many clients want to do the same thing, they can just rely on the default null behavior.

The example contains a slightly different case. Client code that uses the result of a call to customer.

```
if (customer.isNull()) weeksDelinquent = 0;
else weeksDelinquent = customer.getHistory().getWeeksDelinquentInLastYear();
```

I can handle this by creating a null payment history.

```
class NullPaymentHistory extends PaymentHistory...
  int getWeeksDelinquentInLastYear() {
      return 0;
  }
```

I modify the null customer to return it when asked.

```
class NullCustomer...
  public PaymentHistory getHistory() {
      return PaymentHistory.newNull();
  }
```

And again I can remove the conditional code

```
int weeksDelinquent = customer.getHistory().getWeeksDelinquentInLastYear();
```

You often find that null objects return other null objects.

When carrying out this refactoring remember that you can have several kinds of null. Often there is a difference between there is no customer (new building and not yet moved in) and a unknown customer (we think there is someone there but we don't know who it is). If that is the case you can build separate classes for the different null cases. Sometimes null objects can actually carry data: such as usage records for the unknown customer so that we can bill them when we find out who they are.

Chapter 8: 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 over a decade, but I still never get it right first time. That used to bother me, but now I realize that I can use refactoring to change my mind with these cases.

Often I can resolve these problems by just using *Move Method (160)* and *Move Field (164)* to move the behavior around. If I need to use both I prefer to *Move Field (164)* first and then *Move Method (160)*.

Often classes get bloated with too many responsibilities. In this case I use *Extract Component (166)* to separate out some of these responsibilities. If a class becomes too irresponsible then use *Inline Component (170)* to merge it into another class.

If a component is being used, then it often is useful to hide this fact with *Hide Delegate (172)*. Sometimes hiding the component results in you constantly changing the owner's interface, in which case you need to use *Remove Middle Man (174)*

The last two refactorings in this chapter: *Create Foreign Method (176)* and *Create Extension (178)* are special cases. I only use these when I'm not able to access the source code of a class, yet I would like to move responsibilities to this immutable class. If it is just one or two methods I use *Create Foreign Method (176)*, if it is more than that I use *Create Extension (178)*.

Move Method

You need to move a method from one class to another

Create a new method in the target class, change all references to the existing method, and delete the existing method

Motivation

Moving methods is the bread and butter of refactoring. I do this because 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, looking for 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 based on which object the method seems to be interacting with more.

Its not always an easy decision to make. If I am not sure whether to move a method I go on and look at other methods. Often moving other methods makes the decision easier. Sometimes the decision still is hard to make. Actually that is not too much of a big deal. If it is difficult to decide then 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 the features that the source method uses that are defined on the source class. Consider whether they should be moved too.
  - In Smalltalk, use the messages menu item to show the methods used.
  - If a feature is only used by the method you are about to move, then 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.
- Declare the method in the target class
  - You may choose the use a different name 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, then you need to determine how to reference the source object from the target method. If there is no mechanism in the target class, then 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 really be responsible then leave the handlers behind.
- Compile the target class
- Determine how to references the correct target object from the source
  - There may be an existing field or method that will give you the target. If not see if you can easily create a method that will do so. Failing that you will 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.
- 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.
  - Turning the source into a delegating method is easier if you have many references.
- If you remove it, replace all the references with references to the target method.
  - In Smalltalk you should use the browser to find senders first, then change them, then remove the method.
  - You can compile and test after changing each reference, although it is usually easier to do them all with one search and replace.
- Compile and test.

Example

I'll use an account class to illustrate 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 their 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.

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 meant removing the _type from uses of features of the account type, and doing something about the features of account that I still need. When we need to use a feature of the source class we can do one of four things

- Move this feature to the target class as well
- Create or use a reference from the target class to the source
- Pass the source object as a parameter to the method
- If the feature is a variable, pass it in as a parameter.

Since this feature is a single field I can just pass it in as a variable. With methods we can't do that and we need to pass in the source object, like this.

```
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;
  }
```

You can also pass in the source object if you need several features of the class, although if you use to many it implies some further refactoring is needed (typically you need to decompose and move some pieces back).

One the method fits, and compiles in the target class, you can replace the source method body with a simple delegation.

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

At this point you can compile and test.

You can leave things like this, or you can remove the method in the source class. To remove the method you 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 you have replaced all of them you can remove the method declaration in account. You can compile and test after each removal, or do them in a batch. Remember to look for other classes that use this method if it isn't private. In a strongly typed language the compilation after removing the source declaration will find anything you missed.

Move Field

You need to move a field from one class to another

Create a new field in the target class, change all references to the existing field, and delete the existing field

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 was 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 is a decision based on interface. But if the methods seem sensible where they are I move the field.

Another cause for field moving is when I'm *extracting an object* from another. Then the fields go first, followed by the methods.

Mechanics

- ❍ If the field is public, encapsulate it.
  - ☞ If you are likely to be moving the methods that access it frequently, or there are a lot of methods that access the field, you may find it useful to *self-encapsulate* it
- ❍ Compile and test
- ❍ Create a field in the target class
- ❍ Add Getting and Setting Methods for the target field
- ❍ 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 if you can easily crate a method that will do so. Failing that you will 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
  - ☞ In Smalltalk you should use the browser to find the references first, then change them, then remove the instance variable.

- Replace all references to the source field with references to the appropriate method on the target.
  - For accesses to the variable, replace with a call to the target object's getting method, for assignments replace it with a call to the setting method.
  - Unless the field in 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 that reference, of which interestForAmount_days is one example. If there are a lot of methods that use the interest rate field I might start by self-encapsulating the field.

```
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;
  }
```

In either case 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. Also this way the compiler will help us spot any method I failed to redirect.

```
private double _interestRate; //tbd check this is struck through

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

If I had used the self-encapsulation, this redirection only needs to be done to the accessors.

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

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

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

Extract Component

You have a class that is doing the work of two classes

Create a new class and extract the relevant functionality from the old class into the new component.

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 gets too complicated.

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 split it. A good sign is if a subset of the data and a subset of the methods seems to go together. Or if a subset of the data usually changes together, or is 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 non-sense?

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 are subtyped 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.
- Move the data you wish to split from the old to the new class
  - If you aren't intending to move any fields, then you can make the new object a *fly-weight* [Gang of Four].
- Move the methods over from old to new, starting with lower level methods (that are called rather than calling) and building up 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 if it can be made one way.
- Decide whether to expose the component. If so decide whether to expose it as a reference object, or as an immutable value object.

Example

We'll 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 we can separate the telephone number behavior into its own class.

We start by defining a telephone number class

```
class TelephoneNumber {
}
```

That was easy!

We next make a link from the person to the telephone number.

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

Now we use *Move Field (164)* 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);
  }
```

We can then move the other field and use *Move Method (160)* 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 do I expose the component 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 it, I need to consider the dangers of aliasing. If I espouse the telephone number, and a client changes the area code in that component, how do I feel about it? Of course it may not be a direct client that does this change, it could be the client of a client of a client....

I have the following options

- 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 (191)*. In this case the Person would be the access point for the telephone number.
- I don't want anybody to change the value of the telephone number without going through the person. To do this I can either make the telephone number immutable, or I could provide an immutable interface for the telephone number.
- (Another possibility is to clone the telephone number before passing it out. But this can lead to confusion as people think they can change the value. It also may lead to aliasing problems between clients if the telephone number gets passed around a lot.)

Inline Component

Motivation

This is the reverse of *Extract Component (166)*. I use this if a class is no longer pulling it's weight and shouldn't be around any more. Often this is the result of refactoring, moving other responsibilities out of the class so there is little left. Then I just 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 component onto the absorbing class. Delegate all these methods to the component.

☞ If a separate interface makes sense for the component methods, use *Extract Inter-*

face (286) before inlining.

- Change all references to the component to the absorbing class.
  - Declare the class private to remove out of package references. Also change the name of the class so the compiler catches any dangling references to the component
- Compile and test
- Move the fields and methods from the component to the absorbing class until there is nothing left.

Example

Since I made a component out of telephone number, I shall 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 (160)* and *Move Field (164)* until the telephone class is no more.

Hide Delegate

A client is calling a component of an object

Create methods on the server to hide the delegate

Motivation

On of the key, 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, less 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 Java allows fields to be public. As you get 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 it means that the client needs to know about this delegate object. Should the delegate change, then the client also may have to change. This dependency can be removed by placing a simple delegating method on the server, which hides the delegate. Changes are now limited to server and don't propagate to the client.

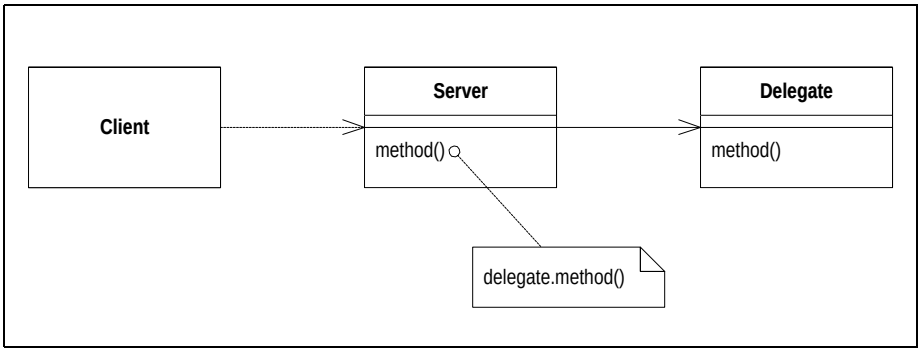


Figure 8.1: Simple Delegation

You may find it is worthwhile to *Extract Component (166)* 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.
- Adjust the client to call the server.
  - ☞ If the client is in a different package to 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'll 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 done for all methods of department, for all the client 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 *Extract Component (166)* I talked about the advantages of encapsulating the use of a delegated object. There is a price to 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 a while of doing this, 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 *Extract Component (166)* and *Remove Middle Man (174)* it does not matter so much. You can adjust your system as time goes on. As the system changes so the basis for how much you hide will change. A good encapsulation six months ago maybe 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'll use the 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;
  }
```

In order to find a person's manager, clients just ask.

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

This is simple to use, and encapsulates the department. However if there lots of methods that are doing this, we 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, and then use it.

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

I can then remove `getManager` from person. A compile will show if 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, yet show it to others. That will also leave some of the simple delegations in place.

Create 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

Motivation

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

If you only use the method once in the client class then it's no big deal, and probably wasn't really needed on the original class anyway. If you use it several times however, then you have to repeat this coding around. Since repetition is the root of all software evil, this repetitive code should be factored into a single method. When you do this 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 then you should *Create Extension (178)* instead.

Don't forget that foreign methods are a work-around. If you can, try to get the methods moved to their proper home. If it is code ownership that is the issue, send the foreign method to the server class's owner and ask them 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 some 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

We 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 will be 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);
}
```

Create Extension

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

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

Motivation

Sadly authors of classes 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 there are one or two methods you need, then you can *Create Foreign Method (176)*. Once you get beyond a couple of these, however, they get out of hand. So you need to group the methods together in a sensible place for them. An extension is a good way to do this.

An extension is a separate class, but is a subtype of the class that 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 create an extension instead and use it.

By doing this 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.

Mechanics

The most obvious way to do this is to make the extension a subclass of the original.

```
Class mfDate extends Date {
  public nextDay() …
  public dayOfYear()…
```

You can also do this by making the extension a wrapper of the original.

```
class mfDate {
  private Date _original;
```

With the wrapper approach you need to write simple delegating methods for all the methods on the original. These are easy to do, but tedious. The subclass doesn't need these. However if you already have the object created, the subclass forces you to create a separate object, while the wrapper just uses the existing object.

In either case you need methods to convert between the original and the extension, so that clients can easily use one or the other. With subclassing you need to provide a constructor for each of the original's constructors. You can usually just delegate to the superclass. With the wrapper you need to create the original and place it in an instance variable.

- Create an extension class either as a subclass or wrapper of the original
- Add converting constructors to the extension
- 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.

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

```
public boolean equals (Date arg)
```

Since you can't alter the original you will only be able to do equals in one direction.

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

The same problem is not an issue with subclassing, providing you don't override the operation. If you do override you'll get yourself completely confused with the method lookup. I don't find I do override methods with extensions, however, I usually just add methods.

Example

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. So I'll use it as an example here.

The first case to show is using subclassing. The first step is to create the new date as a subclass of the original.

```
class MfDateSub extends Date
```

Next is dealing with changing between dates and the extension. The original constructors of the original need to be repeated with simple delegation.

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

You should provide a constructor that takes an original as an argument.

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

You can now add new features to the extension and move any foreign methods over to the extension.

With the wrapping approach, you need to set up these conversion methods differently. The original constructors are implemented with simple delegation

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

The constructor with the original as argument now just sets the instance variable

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

And then there's the tedious task of delegating all the methods of the original class: I'll only show a couple.

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

public boolean equals (MfDateWrap arg) {
```

```
    return (toDate().equals(arg.toDate()));
}
```

Chapter 9: Organizing Data

In this chapter I'll discuss several refactorings that make working with data easier. For many people *Self Encapsulate Field (184)* will seem unnecessary. It's long been a matter of good-natured debate about whether an object should access its own data directly or through accessors. Sometimes you do need the accessors and then you can get them with *Self Encapsulate Field (184)*. I generally use direct access as I find it simple to do this refactoring when I need it.

One of the useful things about object languages is that they allow you to define new types that go beyond what can be done with the simple data types of traditional languages. It takes a while to get used to how to do this, however, and often you start with a simple data value and then realize that an object would be more useful. *Replace Data Value with Object (187)* allows you to turn dumb data into articulate objects. When you realize that these objects are instances that will be needed in many parts of the program then you can use *Change Value to Reference (191)* to make them into reference objects. If you see an array acting as a data structure, you can make the data structure clearer with *Replace Array with Object (194)*. In all these cases the object is but the first step - the real advantage comes as you use *Move Method (160)* to add behavior to the new objects.

Magic numbers, numbers with special meaning, have long been a problem. I remember being told not to use them in my earliest programming days. They do keep appearing, however, and I use *Replace Magic Number with Symbolic Constant (210)* to get rid of them whenever I figure out what they are doing.

Links between objects can be one-way or two-way. One-way links are easier, but sometimes you need to *Change Unidirectional Association to Bidirectional (204)* to support new function. *Change Bidirectional Associa-*

tion to Unidirectional (207) removes unnecessary complexity should you find you don't need this any more.

I've often run into cases where gui classes are doing business logic that they shouldn't. To move the behavior into proper domain classes you need to have the data in the domain class as well as supporting the gui by using *Duplicate Data From Presentation to Domain (197).* Normally I don't like duplicating data, but this is an exception that is usually impossible to avoid.

One of the key tenets of object-oriented programming is encapsulation. So if there's any public data streaking around, you can use *Encapsulate Field (211)* to decorously cover it up. If that data is a collection then use *Encapsulate Collection (212)* instead as that has special protocol. If you have a whole naked record then use *Replace Record with Data Class (218)*

One form of data that requires particular treatment is the type code: some special value that indicates something particular about what type of instance we are dealing with. These often show up as enumerations, often implemented as static final integers. If the codes are for information and do not alter the behavior of the class, then you can use *Replace Type Code with Class (219),* which gives you better type checking and a platform for moving behavior later. If the behavior of a class is affected by a type code then use *Replace Type Code with Subclasses (224)* if possible, or if you can't do that use the more complicated (but more flexible) *Replace Type Code with State/Strategy (227).*

Self Encapsulate Field

You are using accessing a field directly, but the coupling to the field is becoming awkward.

Create Getting and Setting Methods for the field and use only those to access the field

Motivation

When it comes to accessing fields there are two schools of thought. One is that within the class where the variable is defined, you should access it the variable freely (Direct Variable Access). The other says that even within the class you should always use accessors (Indirect Variable Access). Debates between the two can get heated. I'm always in two minds with it, so I'm usually happy to do what the rest of the team wants to do. Left to myself though I like to use Direct Variable Access as a first resort, until it gets in the way. Once things start becoming awkward I switch to Indirect Variable Access. Refactoring gives you the freedom to change your mind.

The most important time to do this is when you are accessing a field in a superclass, but you want to override this variable access with a computed value in the subclass. Self-encapsulating the field is the first step, after that you can override the Getting and Setting methods as you need to.

Mechanics

- Create a Getting and Setting Method for the field
- Find all references to the field and replace them with a getting or setting method
  - For accesses to the field, replace with a call to the getting method, for assignments replace it with a call to the setting method.
  - You can't entirely rely on the compiler in a strongly typed language here, as it is not an error to refer to the field in its own class.
- Make the field private.
  - Smalltalk cannot do this (all subclasses can see a superclass variable). Making a field private will allow the compiler to catch any subclass using the field, but the compiler will still not catch references with the field's class
- Double check you have caught all references
- Compile and Test

Refactory does this with the menu item "abstract" to either a class or an instance variable.

Example

This seems almost too simple for an example, but hey at least it'll be quick to write.

```
class IntRange {
```

```
    private int _low, _high;

    boolean includes (int arg) {
        return arg >= _low && arg <= _high;
    }

    void grow(int factor) {
        _high = _high * 2;
    }

    IntRange (int low, int high) {
        _low = low;
        _high = high;
    }
```

To self encapsulate I define getting and setting methods (if they don't already exist) and use those.

```
class IntRange {

    boolean includes (int arg) {
        return arg >= getLow() && arg <= getHigh();
    }

    void grow(int factor) {
        setHigh (getHigh() * 2);
    }

    private int _low, _high;

    int getLow() {
        return _low;
    }

    int getHigh() {
        return _high;
    }

    void setLow(int arg) {
        _low = arg;
    }

    void setHigh(int arg) {
        _high = arg;
    }
```

In cases like this you have to be more careful about using the setting method in the constructor. Often it is assumed that you use the setting

method for changes after the object is created, so the setting method has different semantics. So in cases like this I prefer a separate initialization method.

```
IntRange (int low, int high) {
    initialize (low, high);
}

private void initialize (int low, int high) {
    _low = low;
    _high = high;
}
```

The value in doing all this comes when you have a subclass.

```
class CappedRange extends IntRange {

  CappedRange (int low, int high, int cap) {
      super (low, high);
      _cap = cap;
  }

  private int _cap;

  int getCap() {
      return _cap;
  }

  int getHigh() {
      return Math.min(super.getHigh(), getCap());
  }
}
```

I can override all of `IntRange`'s behavior to take into account the cap, without changing any of that behavior.

Replace Data Value with Object

You have a data item that needs additional data or behavior

Turn the data item into an object

Motivation

Often in early stages of development you make decisions about representing simple facts as simple data items. As development proceeds you realize that those simple items aren't so simple any more. A telephone number may be represented as a string for a while, but later on you realize that the telephone needs special behavior for formatting, extracting the area code and the like. For one or two items you may just put the methods in the owning object, but quickly the code smells of duplication and feature envy.

When the smell begins turn the data value into an object.

Mechanics

- Create the class for the value. Give it a field of the same type as the value in the source class. Add an getter and a constructor that takes the field as an argument.
- Compile
- Change the type of the field in the source class to the new class
- Change the getter in the source class to call the getter in the new class
- If the field is mentioned in the source class constructor, assign the field using the new class's constructor
- Change the getting method to create a new instance of the new class.
- Compile and test.
- You may now need to use *Change Value to Reference (191)* on the new object.

Example

I'll start with an order class that's stored the customer of the order as a string and would like to turn the customer into an object. This way we have something to store data such as an address, credit rating and the like.

```
class Order...
  public Order (String customer) {
      _customer = customer;
  }
  public String getCustomer() {
      return _customer;
  }
```

```
public void setCustomer(String arg) {
    _customer = arg;
}
private String _customer;
```

Some client code that uses this looks like

```
 int numberOfOrdersFor(Vector listOfOrders, String customer) {
    int result = 0;
    Enumeration e = listOfOrders.elements();
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        if (each.getCustomer().equals(customer)) result++;
    }
    return result;
}
```

First I create the new customer class. I give it field for a string attribute, as that is that the order currently uses. I call it name, since that seems to be what the string is used for. I also add a getting method and provide a constructor that uses the attribute. Notice that I don't provide a setter.

```
class Customer {
  public Customer (String name) {
      _name = name;
  }
  public String getName() {
      return _name;
  }
  private String _name;
}
```

Now I change the type of the customer field and change methods that reference it to use the appropriate references on the customer class. The getter and constructor are obvious. For the setter I create a new customer.

```
class Order...
  public Order (String customer) {
      _customer = new Customer(customer);
  }
  public String getCustomer() {
      return _customer.getName();
  }
  private Customer _customer;

  public void setCustomer(String arg) {
```

```
        _customer = new Customer(customer);
    }
```

The setter creates a new customer because the old string attribute was a value objects, and thus the customer currently is also a value object. This means that each order has its own customer objects. As a rule value objects should be immutable — this avoids some nasty aliasing bugs. Later on we will want customer to be a reference object, but that's another refactoring.

At this point we can compile and test.

Now we should look at the methods on order that manipulate customer and make some changes to make the new state of affairs clearer.

With the getter we use *Rename Method (234)*

```
public String getCustomerName() {
    return _customer.getName();
}
```

On the constructor and setter, we don't need to change the signature, but the name of the arguments should change.

```
public Order (String customerName) {
    _customer = new Customer(customerName);
}
public void setCustomer(String customerName) {
    _customer = new Customer(customerName);
}
```

Further refactoring may well cause us to add a new constructor and setter that takes an existing customer.

This finishes this refactoring, but in this case, as in many others, there is another step. If we wish to add such things as credit ratings and addresses to our customer, we cannot do so now. This is because the customer is treated as a value object. Each customer has it's own customer object. To give a customer these attributes we need to apply *Change Value to Reference (191)* to the customer. You'll find this example continued there....

Change Value to Reference

You have a class with many equal objects that you want to replace with a single object

Turn the object into a reference object

Motivation

There is a useful classification you can make of objects in many systems: reference objects and value objects. A reference object is something like customer, or account. Each object stands for one object in the real world. You don't usually copy reference objects, and you use the object identity to test if they are equal. Value objects are things like date, or money. They are entirely defined through their data values. You don't mind copying them, you may have hundreds of "1/1/2000" objects around your system. You do need to tell if two of them are equal, so you need to override the equals method (and the hash method too.)

The decision between reference and value is not always clear cut. Sometimes you start with a value and want to give it some data that really makes it a reference object. You can do this by turning it into a reference object.

Mechanics

- Use *Replace Constructor with Factory Method (256).*
- Compile and test
- Decide what object is responsible for providing access to the objects
  - This may be a static dictionary or a registry object.
  - You may have more than one object that acts as an access point for the new object.
- Decide whether the objects are pre-created or created on the fly.
  - If they are pre-created and you are retrieving them from memory you need to ensure they are loaded before they are needed.
- Alter the factory method to return the reference object.
  - If the objects are pre-computed, you need to decide how to handle errors if someone asks for one that does not exist.
  - You may wish to *Rename Method (234)* on the factory to convey that it returns an existing object.

- Compile and test

Example

I'll start with where I left off in the example for *Replace Data Value with Object (187)*. I have the following customer class.

```
class Customer {
  public Customer (String name) {
      _name = name;
  }
  public String getName() {
      return _name;
  }
  private String _name;
}
```

It is used by an order class.

```
class Order...
  public Order (String customerName) {
      _customer = new Customer(customerName);
  }
  public void setCustomer(String customerName) {
      _customer = new Customer(customerName);
  }
  public String getCustomerName() {
      return _customer.getName();
  }
  private Customer _customer;
```

and some client code

```
 int numberOfOrdersFor(Vector listOfOrders, String customerName) {
    int result = 0;
    Enumeration e = listOfOrders.elements();
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        if (each.getCustomer().equals(customerName)) result++;
    }
    return result;
}
```

At the moment it is a value. Each order has it's own customer object. I want to change this so that orders share the same customer object they represent the same customer. For this case this means that there should only be one customer object for each customer name.

I begin by using *Replace Constructor with Factory Method (256)*. I define the factory method on customer.

```
class Customer {
  public static Customer create (String name) {
      return new Customer(name);
  }
```

Then replace the calls to the constructor with calls to the factory.

```
  class Order {
  public Order (String customer) {
      _customer = Customer.create(customer);
  }
```

Then I make the constructor private.

```
class Customer {
  private Customer (String name) {
      _name = name;
  }
```

Now I have to decide how I access the customers. My preference is to use another object. Such a situation works well with something like the line items on an order. The order is responsible for providing access to the line items. However here there isn't such an obvious object. In this situation I usually create a registry object to be the access point. For simplicity in this example however, I will store them using a static field on customer, making the customer class the access point.

```
  private static Dictionary _instances = new Hashtable();
```

Then I decide whether I create customers on the fly when asked, or whether they are pre-created. I'll use the latter. In my application start-up code I'll load up the customers that are in use. These could come from a database or from a file, but again for simplicity I'll use explicit code. I can always use *Substitute Algorithm (132)* to change it later.

```
class Customer...
  static void loadCustomers() {
      new Customer ("Lemon Car Hire").store();
      new Customer ("Associated Coffee Machines").store();
      new Customer ("Bilston Gasworks").store();
  }
  private void store() {
      _instances.put(this.getName(), this);
  }
```

Now I'll alter the factory method to return the pre-created customer.

```
  public static Customer create (String name) {
      return (Customer) _instances.get(name);
  }
```

I have to decide what to do if a client asks for a customer that doesn't exist yet. The dictionary will just return null, in this case I'll raise an exception.

```
class Customer...
  public static Customer create (String name) {
      Customer result = (Customer) _instances.get(name);
      if (result == null)
          throw new IllegalArgumentException ("No customer with name: " + name);
      return result;
  }
```

Since the create method always returns an existing customer, I should make this clear by using *Rename Method (234)*.

```
class Customer...
  public static Customer getNamed (String name) {
      Customer result = (Customer) _instances.get(name);
      if (result == null)
          throw new IllegalArgumentException ("No customer with name: " + name);
      return result;
  }
```

Replace Array with Object

You have an array where certain elements mean different things

Replace the array with an object, with a field for each element

Motivation

Array's are a common structure for organizing data. However they should only be used to contain a collection of similar objects in some order. Sometimes, however you see them used to contain a number of different things. Conventions such as "the first element on the array is the person's name" are hard to remember. With an object you can use names of fields and methods to convey this information so you don't have remember it, or hope the comments are up to date. You can also

encapsulate the information, and use *Move Method (160)* to add behavior to it.

Mechanics

- Create a new class to represent the information in the array. Give it a public field for the array
- Change all users of the array to use the new class
- Compile and test
- One by one, add getters and setters for each element of the array. Name the accessors after the purpose of the array element. Change the clients to use the accessors. Compile and test after each change
- When all array accesses are replaced by methods, make the array private
- Compile
- For each element of the array, create a field in the class and change the accessors to use the field.
- Compile and test after each element is changed
- When all elements have been replaced with fields, delete the array.

Example

I'll start with an array that's used to hold the name, wins, and losses of a sports team. It would be declared as

```
String[] row = new String[3];
```

It would be used with code like

```
row [0] = "Liverpool";
row [1] = "15";

String name = row[0];
int wins = Integer.parseInt(row[1]);
```

To turn this into an object, we begin by creating a class.

```
class Performance {}
```

For our first step we give the new class a public data member. (I know this is evil and wicked, but I'll reform in due course.)

```
    public String[] _data = new String[3];
```

Now I find the spots that create and access the array. Where the array is created I use

```
Performance row = new Performance();
```

Where it is used, I change to

```
row._data [0] = "Liverpool";
row._data  [1] = "15";

String name = row._data[0];
int wins = Integer.parseInt(row._data[1]);
```

Now, one by one, I add more meaningful getters and setters. I start with the name.

```
class Performance...
  public String getName() {
      return _data[0];
  }
  public void setName(String arg) {
      _data[0] = arg;
  }
```

I alter the users of that row to use the getters and setters instead,

```
row.setName("Liverpool");
row._data  [1] = "15";

String name = row.getName();
int wins = Integer.parseInt(row._data[1]);
```

I can do the same with the second element. To make matters easier I can encapsulate the data type conversion as well.

```
class Performance...
  public int getWins() {
      return Integer.parseInt(_data[1]);
  }
  public void setWins(String arg) {
      _data[1] = arg;
  }

....
client code...
  row.setName("Liverpool");
  row.setWins("15");

  String name = row.getName();
  int wins = row.getWins();
```

Once I've done this for each element, I can make the array private.

```
private String[] _data = new String[3];
```

The most important part of this refactoring, changing the interface, is now done. However it is also useful to replace the array internally. I

can do this by adding a field for each array element, and changing the accessors to use it.

```
class Peformance...
  public String getName() {
      return _name;
  }
  public void setName(String arg) {
      _name = arg;
  }
  private String _name;
```

I do this for each element in the array. When I've done them all I delete the array.

Duplicate Data From Presentation to Domain

You have domain data embedded in a gui control

Copy the data to a domain object. Set up a mechanism to synchronize the two pieces of data.

Motivation

A well layered system will separate the code that handles the user interface from code that handles the business logic. It does this for several reasons: you may want several interfaces for similar business logic, the user-interface gets too complicated if it does both, it is easier to maintain and evolve domain objects separate from the gui, or you may have different developers handling these different pieces.

Although the behavior can be separated easily, the data often cannot. Data needs to be embedded in gui controls which has the same meaning as data that lives in the domain model. UI frameworks, from MVC onwards, that are used to a multi-tiered system provide mechanisms to allow you to provide this data and keep everything in sync.

If you come across that has been developed within a two tier approach, where business logic is embedded into the UI, you will need to separate out this behavior. Much of this is about decomposing and moving methods. For the data however, you cannot just move the

data, you have to duplicate it and provide the synchronization mechanism.

Mechanics

- Make the presentation class an observer of the domain class
  - If there is no domain class yet, create one.
  - If there is no link from the presentation class to the domain class, put the domain class in a field of the presentation class.
- Use *Self Encapsulate Field (184)* on the domain data within the gui class.
- Compile and test
- Add a call to the setting method in the event handler, to update the component with its current value using direct access.
  - That is put a method in the event handler that updates the value of the component based on its current value. Of course this is completely unnecessary, you are just setting the value to its current value, but by using the setting method you allow any behavior there to execute.
  - When you do this change, don't use the getting method for the component, use direct access to the component. Later on the getting method will pull the value from the domain, which does not change until the setting method executes.
  - Make sure the event handling mechanism is triggered by the test code
- Compile and test
- Define the data and accessor methods in the domain class
  - Make sure the setting method on the domain triggers the notify mechanism in the observer pattern.
  - Use the same data type in the domain as was on the presentation (usually a string). Convert the data type in a later refactoring.
- Redirect the accessors to write to the domain field.
- Modify the observer's update method to copy the data from the domain field to the gui control.
- Compile and test

Example

I'll start with the window in Figure 9.1. The behavior is very simple. Whenever you change the value in one of the text fields, the other ones

update. If you change the start or end fields, it calculates the length, if you change the length field it calculates the end.

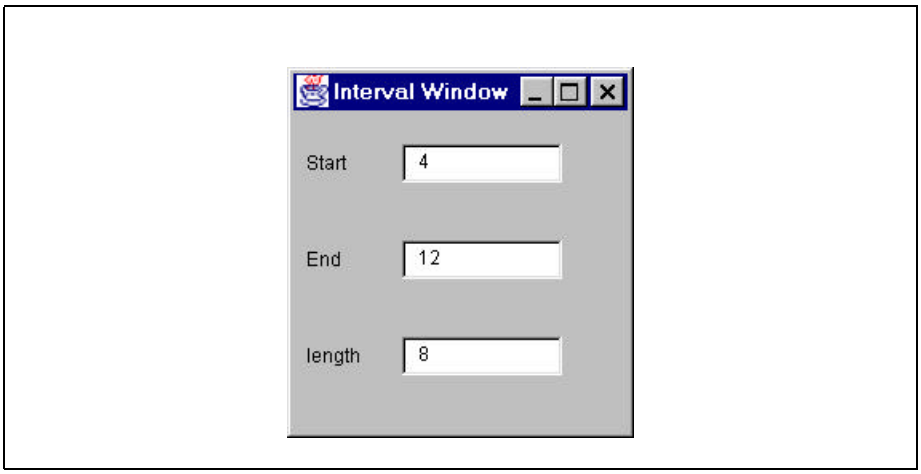


Figure 9.1: A simple gui window

All the methods are on a single `IntervalWindow` class. The fields are set to respond to the loss of focus from the field.

```
class SymFocus extends java.awt.event.FocusAdapter
{
    public void focusLost(java.awt.event.FocusEvent event)
    {
        Object object = event.getSource();
        if (object == _startField)
           StartField_FocusLost(event);
        else if (object == _endField)
           EndField_FocusLost(event);
        else if (object == _lengthField)
           LengthField_FocusLost(event);
    }
}
```

The listener reacts by calling `StartField_FocusLost` when focus is lost on the start field, and `EndField_FocusLost` and `LengthField_FocusLost` for the other fields. These event handling methods look like this:

```
void StartField_FocusLost(java.awt.event.FocusEvent event) {
    if (isNotInteger(_startField.getText()))
        _startField.setText("0");
    calculateLength();
}
```

```
  void EndField_FocusLost(java.awt.event.FocusEvent event) {
      if (isNotInteger(_endField.getText()))
          _endField.setText("0");
      calculateLength();
  }

  void LengthField_FocusLost(java.awt.event.FocusEvent event) {
      if (isNotInteger(_lengthField.getText()))
          _lengthField.setText("0");
      calculateEnd();
  }
```

All of them insert a zero if any non integer characters appear and call the relevant calculation routine.

```
void calculateLength(){
    try {
      int start = Integer.parseInt(_startField.getText());
      int end = Integer.parseInt(_endField.getText());
      int length = end - start;
      _lengthField.setText(String.valueOf(length));
    } catch (NumberFormatException e) {
      throw new RuntimeException ("Unexpected Number Format Error");
    }
}
void calculateEnd() {
    try {
      int start = Integer.parseInt(_startField.getText());
      int length = Integer.parseInt(_lengthField.getText());
      int end = start + length;
      _endField.setText(String.valueOf(end));
    } catch (NumberFormatException e) {
      throw new RuntimeException ("Unexpected Number Format Error");
    }
}
```

My task, should I choose to accept it, is to separate out the non-visual logic from the gui. Essentially this means moving `calculateLength` and `calculateEnd` to a separate domain class. However to do this they need to refer to the start, end, and length data *without* referring to the window class. The only way I can do this is to duplicate this data in the domain class and synchronize the data with the gui.

I don't currently have a domain class, so I create an (empty) one.

```
class Interval extends Observable {}
```

The interval window needs a link to this new domain class.

```
    private Interval _subject;
```

I then need to properly initialize this field, and make interval window an observer of the interval. I can do this by putting the following code in interval window's constructor.

```
_subject = new Interval();
_subject.addObserver(this);
update(_subject, null);
```

I like to put this code at the end of construction process. The call to update will ensure that as I duplicate the data in the domain class the gui is initialized from the domain class.

Of course to do this I need to declare that interval window implements observable.

```
public class IntervalWindow extends Frame implements Observer
```

In order to implement Observer I need to create an update method. For the moment this can be blank.

```
public void update(Observable observed, Object arg) {
}
```

I can compile and test at this point. I haven't made any real changes yet, but I can make mistakes in the simplest places.

Now I can turn my attention to moving fields. As usual I make the changes one field at a time. To demonstrate my command of the English language I'll start with the end field. The first task is to apply *Self Encapsulate Field (184)*. Text fields are updated with `getText` and `setText` methods. I create accessors that call these

```
String getEnd() {
    return _endField.getText();
}

void setEnd (String arg) {
    _endField.setText(arg);
}
```

I find every reference to `_endField` and replace them the appropriate accessors.

```
void calculateLength(){
  try {
    int start = Integer.parseInt(_startField.getText());
    int end = Integer.parseInt(getEnd());
    int length = end - start;
    _lengthField.setText(String.valueOf(length));
```

```
    } catch (NumberFormatException e) {
      throw new RuntimeException ("Unexpected Number Format Error");
    }
  }

void calculateEnd() {
    try {
      int start = Integer.parseInt(_startField.getText());
      int length = Integer.parseInt(_lengthField.getText());
      int end = start + length;
      setEnd(String.valueOf(end));
    } catch (NumberFormatException e) {
      throw new RuntimeException ("Unexpected Number Format Error");
    }
}

 void EndField_FocusLost(java.awt.event.FocusEvent event) {
     if (isNotInteger(getEnd()))
         setEnd("0");
     calculateLength();
 }
```

That's the normal process for *Self Encapsulate Field (184)*. However when you are working with a GUI, there is a complication. The user can change the field value directly without calling setEnd. So I need to put a call to setEnd into the event handler for the GUI. This call will change value of the end field to the current value of the end field. Of course this does nothing at the moment, but it does ensure the user input goes through the setting method.

```
void EndField_FocusLost(java.awt.event.FocusEvent event) {
    setEnd(_endField.getText());
    if (isNotInteger(getEnd()))
        setEnd("0");
    calculateLength();
}
```

You should notice that in this call I don't use getEnd, instead I access the field directly. I do this because later on in the refactoring getEnd will get a value from the domain object, not from the field. At that point using it would mean that every time the user changed the value of the field this code would just change it back again, so here I must use direct access.

At this point I can compile and test the encapsulated behavior.

Now I add the end field to the domain class.

```
class Interval...
    private String _end = "0";
```

I initialize it to the same value that it is initialized to in the gui. I now add getting and setting methods.

```
class Interval...

  String getEnd() {
      return _end;
  }
  void setEnd (String arg) {
      _end = arg;
      setChanged();
      notifyObservers();
  }
```

Since I'm using the observer pattern I have to add the notification code into the setting method.

I can now do one more compile and test before I perform the duplication. By getting all this preparatory work done I've minimized the risk in this tricky step.

The first change is updating the accessors on interval window to use interval.

```
class IntervalWindow...
  String getEnd() {
      return _subject.getEnd();
  }
  void setEnd (String arg) {
      _subject.setEnd(arg);
  }
```

I also need to update `update` to ensure the gui reacts to the notification.

```
class IntervalWindow...
    public void update(Observable observed, Object arg) {
      _endField.setText(_subject.getEnd());
    }
```

This is the other place where I have to use direct access. If I called the setting method I would get into an infinite recurse.

I can now compile and test and the data is properly duplicated.

I can repeat for the other two fields. Once this is done I can apply *Move Method (160)* to move `calculateEnd` and `calculateLength` over to the interval class.

Change Unidirectional Association to Bidirectional

You have classes that need to have a two way reference

Add back-pointers, change modifiers to update both sets

Motivation

You may find that you have initially set up two classes so that one class refers to the other. Over time you may find that a client of the referred class needs to get to the objects that refer to it. Effectively this means navigating backwards along the pointer. Pointers are one-way links, so you can't do this. Often you can get around this by find another route. This may cost in computation, but is still reasonable, and you can have a method on the referred class that uses this behavior.

However sometimes this is not easy, and you need to set up a two-way reference (sometimes referred to as back pointers). If you aren't used to these, it's easy to get tangled up doing this, but once you get used to the idiom it is not too complicated.

The idiom is awkward enough that you should have tests, at least until you are comfortable with the idiom. Since I usually don't bother testing accessors (the risk is not high enough) this is the rare case of a refactoring that adds a test.

Mechanics

- Add a field for the back--pointer
- Decide which class will control the association
- Create a helper method on the non-controlling side of the association. Name this to clearly indicate its restricted use.
- If the existing modifier is on the controlling side, modify it to update the back pointers
- If the existing modifier is on the controlled side, create a controlling method on the controlling side and call it from the existing modifier.

Example

I'll start with a simple program that has an order which refers to a customer

```
class Order...
  Customer getCustomer() {
      return _customer;
  }
  void setCustomer (Customer arg) {
      _customer = arg;
  }
  Customer _customer;
```

The customer class has no reference back to the order.

I start the refactoring by adding a field to the customer. As a customer can have several orders this field will be a collection. Since we don't want a customer to have the same order more than once in it's collection, the correct collection would be a set. However since I'm using vanilla Java 1.1 I'll have to make do with a vector.

```
class Customer  {
  private Vector _orders = new Vector();
```

Now I need to decide which class will take charge of the association. My decision process runs like this

- If both objects are reference objects, and it is a one to many association, then the object which has the one reference is the controller. (That is if one customer has many orders, then the order controls the association)
- If one object is a component of the other, then the composite should control the association.
- If both objects are reference objects, and it is a many to many association, then it doesn't matter.

As the order will take charge I need to add a helper method to the customer that the order's modifier will use. I use name frankforters to signal that this method is only to be used in special cases. I also minimize its visibility by making it package visibility if at all possible. I do have to make it public if the other class is in another package.

```
class Customer ...
  Vector friendOrders() {
```

```
    /** should only be used by Order */
        return _orders;
    }
```

Now I update the modifier to update the back pointers.

```
    void setCustomer (Customer arg) ...
        if (_customer != null) _customer.friendOrders().removeElement(this);
        _customer = arg;
        if (_customer != null) _customer.friendOrders().addElement(this);
    }
```

The exact code in the controlling modifier will be different depending on the multiplicity of the association. If the customer is not allowed to be null, I can forego the null checks (but I need to check for a null argument). But the basic pattern is always the same, first tell the target object to disconnect, second reconnect at your end, then tell the new target to reconnect.

If you want to modify the link through the customer, you let it call the controlling method.

```
class Customer ...
  void addOrder(Order arg) {
      arg.setCustomer(this);
  }
```

If an order could have many customers, then we would have a many to many case, and the methods would look like this.

```
class Order... //controlling methods
  void addCustomer (Customer arg) {
      arg.friendOrders().addElement(this);
      _customers.addElement(arg);
  }
  void removeCustomer (Customer arg) {
      arg.friendOrders().removeElement(this);
      _customers.removeElement(arg);
  }

class Customer...
  void addOrder(Order arg) {
      arg.addCustomer(this);
  }
  void removeOrder(Order arg) {
      arg.removeCustomer(this);
  }
```

Change Bidirectional Association to Unidirectional

You have a two way association that no longer needs to be two way

Drop the unneeded end of the association

Motivation

Bidirectional associations are useful but they carry a price. The price is the added complexity of maintaining the two way links and ensuring that objects are properly created and removed. Bidirectional associations are not natural for many programmers, and so they often are a source of errors.

Lots of two way links also make it too easy for mistake to lead to zombies: objects that should be dead but still hang around due to some reference that never got cleared.

Finally bidirectional associations force an interdependency between the two classes. Any change to one may cause a change to another. If the classes are in separate packages, this means you get a interdependency between the packages. Many interdependencies lead to a highly coupled system, where any little change leads to lots of unpredictable ramifications.

As such you should use bidirectional associations when you need to, but not when you don't. As soon as you see a bidirectional association is no longer pulling its weight, drop the unnecessary end.

Mechanics

- Examine all the readers of the field that needs to be eliminated to see if it is feasible to eliminate the field
  - Look at direct readers and further methods that call the methods
  - Consider the possibility of using *Substitute Algorithm (132)* on the getter for the field
  - Consider adding the object as an argument to all methods that use the field
- If the change seems feasible then decide whether you need to substitute the getting method.
- If you are substituting the getting method, use *Self Encapsulate Field*

(184), carry out *Substitute Algorithm (132)* on the getter, compile and test.

- If you aren't substituting the getter then change each user of the field so that it gets the object in the field another way. Compile and test after each change.
- When there is no reader left of the field, remove all updates to the field, and remove the field.
  - ☞ If there are many places that assign the field then use *Self Encapsulate Field (184)* so that they all use a single setter. Compile and test. Then change the setter to have an empty body. Compile and test. If that works then remove the field, the setter, and all calls to the setter.
- Compile and test.

Example

I'll start from where I ended up from the example in *Change Unidirectional Association to Bidirectional (204)*. I have a customer and order with a bi-directional link.

```
class Order...
  Customer getCustomer() {
      return _customer;
  }
  void setCustomer (Customer arg) {
      if (_customer != null) _customer.friendOrders().removeElement(this);
      _customer = arg;
      if (_customer != null) _customer.friendOrders().addElement(this);
  }
  private Customer _customer;

class Customer...
  void addOrder(Order arg) {
      arg.setCustomer(this);
  }
  private Vector _orders = new Vector();
  Vector friendOrders() {
      /** should only be used by Order */
      return _orders;
  }
```

I've now found that in my application I don't have orders unless I already have a customer, so I want to break the link to customer.

The hardest part of this refactoring is checking that I can do it. Once I know it's safe to do, then it's easy. The issue is whether code is relying on the customer field being there. Then in order to remove it, I need to provide an alternative.

So my first move is to study all the readers of the field, and the methods that use those readers. Can I find another way to provide the customer object? Often this means passing in the customer as an argument for an operation. Here's a simplistic example of this.

```
class Order...
  double getDiscountedPrice() {
      return getGrossPrice() * (1 - _customer.getDiscount());
  }
```

changes to

```
class Order...
  double getDiscountedPrice(Customer customer) {
      return getGrossPrice() * (1 - customer.getDiscount());
  }
```

This works particularly well when the behavior is being called by the customer, since it's then easy to pass itself in as an argument. So

```
class Customer...
  double getPriceFor(Order order) {
      Assert.isTrue(_orders.contains(order));
      return order.getDiscountedPrice();
```

becomes

```
class Customer...
  double getPriceFor(Order order) {
      Assert.isTrue(_orders.contains(order));
      return order.getDiscountedPrice(this);
  }
```

Another alternative I might have is to give order an operation to get a customer, that does not use the field. Here my aim is to use *Substitute Algorithm (132)* on the body of `Order.getCustomer`. I might do something like this

```
  Customer getCustomer() {
      Enumeration e = Customer.getInstances();
      while (e.hasMoreElements()) {
          Customer each = (Customer)e.nextElement();
          if (each.containsOrder(this)) return each;
      }
      return null;
  }
```

Slow, but it works. In a database context it may not even be that slow if I use a database query. If the order class contains methods that use the

customer I can change them to use `getCustomer` by using *Self Encapsulate Field (184).*

If I retain the accessor, then the association is still bidirectional in interface, but unidirectional in implementation. I remove the back-pointer, but still retain the interdependencies between the two classes.

When I'm doing this I first study all the uses to see if it seems feasible. If I substitute the getting method, then I substitute that and leave the rest till later. Otherwise I change the callers one at a time to use the customer from another source. I compile and test after each change. In practice this usually is pretty rapid, because if it were complicated I would give up on this refactoring.

Once I've eliminated the readers of the field, I can work on the writers of the field. This is as simple as removing any assignments to the field, and then removing the field. Since nobody is reading it any more, that shouldn't matter.

Replace Magic Number with Symbolic Constant

You have a quoted number with a particular meaning

Create a constant, name it after the meaning, and replace the number with it.

Motivation

Magic numbers are one of oldest ills in computing. They are numbers with special values, that are usually not obvious. They are really nasty when you need to reference the same logical number in more than one place. If the numbers might ever change, then making the change is a nightmare. Even if you don't make a change, you have the difficulty of figuring out what to do.

Many languages allow you to declare a constant. There is no cost in performance and a great improvement in readability.

Mechanics

- Declare a constant and set it to the value of the magic number.
- Find all occurrences of the magic number
- See if it matches the usage of the constant, if so change it to use the constant
- Compile
- When all are changed compile and test, at this point all should work as nothing has been changed.
  - ☞ A good test for this to see if you can change the constant easily. This may mean altering some expected results to match the new value. This isn't always possible, but it is a good trick when it works.

Encapsulate Field

There is a public field

Make it private and provide accessors

Motivation

One of the principal tenets of object-orientation is encapsulation, or data hiding. This says that you should never make your data public. When you make data public it allows other objects to change and access data values without the owning object knowing about it. This separates data from behavior.

This is seen as a bad thing because it reduces the modularity of the program. When the data and behavior that uses it are clustered together, then it is easier to change, because the changed code is in one place rather than scattered all over the program.

This refactoring begins the process by hiding the data and adding accessors. But this is only the first step. A class with only accessors is a dumb class that doesn't really take advantage of the opportunities of objects, and an object is terrible thing to waste. Once I've done *Encap-*

sulate Field (211) I look for those methods that use the new methods to see if they fancy packing their bags and moving to the new object with a quick *Move Method (160).*

Mechanics

- Create getting and setting methods for the field
- Find all places outside the class that reference the field. If it uses the value then replace the reference with a call to the getting method. If it changes the value then replace with a call to the setting method.
  - ☞ If the field is an object and the client invokes a modifier on the object, then that is a use. Only use the setting method to replace assignment.
- Compile and test after each change
- Once all references are changed, declare the field as private.
- Compile and test

Encapsulate Collection

A method returns a collection

Make it return an iterator and provide add/remove methods

Motivation

Often a class will contain a collection of instances. This collection might be an array, vector, dictionary, or one of the more sophisticated collections available in Java 1.2. In such cases there is often the usual getter and setter for that collection.

However collections should use a slightly different protocol to other kinds of data. The getter should not return the collection object itself, for that allows clients to manipulate the contents of the collection without the owning class knowing what is going on. Also it reveals too much to clients about the object's internal data structures. So a getter for a multi-valued attribute should return an iterator, which in Java means an Enumeration.

In addition there should not be a setter for collection, rather there should be operations to add and remove elements. This gives the owning object control over adding and removing elements from the collection.

With this protocol the collection is properly encapsulated, which reduces the coupling of the owning class to its clients.

Mechanics

- Add an add and remove method for the collection
- Initialize the field to an empty collection
- Compile
- Find callers of the setting method. Either modify the setting method to use the add/remove operations or have the clients call those operations instead.
  - Two cases are when the setter only is used when the collection is empty, and when the setter is used to replace a full collection.
  - You may wish to rename the setting method to better communicate the intention using *Rename Method (234)*
- Compile and test
- Find all users of the getter that are modifying the collection. Change them to use the modifier. Compile and test after each change
- When all uses of the getter to modify have been changed, modify the getter to return a copy of the collection
- Compile and test
- Change the name of the current getter and add a getter to return an enumeration. Find users of the getter and change them to use one of the new methods.
  - If this is too big a jump use *Rename Method (234)* on the getter, create a new method that returns an enumeration, and change callers to use the new method.
- Compile and test
- Find the users of the getter. Look for code that should be on the host object. Use *Extract Method (114)* and *Move Method (160)* to get it there.
-

Example

As an example we will have a person taking courses. Our course is pretty simple

```
class Course ...
  public Course (String name, boolean isAdvanced) {...};
```

```
  public boolean isAdvanced() {...};
```

We will not bother with anything further on the course. The interesting class is the person.

```
class Person...
  public Vector getCourses() {
      return _courses;
  }
  public void setCourses(Vector arg) {
      _courses = arg;
  }
  private Vector _courses;
```

With this interface, clients adds courses with code like

```
Person kent = new Person();
Vector v = new Vector();
v.addElement(new Course ("Smalltalk Programming", false));
v.addElement(new Course ("Appreciating Single Malts", true));
kent.setCourses(v);
Assert.equals ("courses", 2, kent.getCourses().size());
Course refact = new Course ("Refactoring", true);
kent.getCourses().addElement(refact);
kent.getCourses().addElement(new Course ("Brutal Sarcasm", false));
Assert.equals ("2nd courses", 4, kent.getCourses().size());
kent.getCourses().removeElement(refact);
Assert.equals ("3rd courses", 3, kent.getCourses().size());
```

A client that wants to know about advanced courses might do it this way

```
Enumeration e = person.getCourses().elements();
int count = 0;
while (e.hasMoreElements()) {
    Course each = (Course) e.nextElement();
    if (each.isAdvanced()) count ++;
}
```

The first thing I want to do is to create the proper modifiers for the collection and compile

```
class Person
  public void addCourse(Course arg) {
      _courses.addElement(arg);
  }
  public void removeCourse(Course arg) {
      _courses.removeElement(arg);
  }
```

Life will be easier if I initialize the field as well

```
private Vector _courses = new Vector();
```

I then look at the users of the setter. If there are many clients, and the vector is used heavily then I will need to replace the body of the setter to use the add / remove operations. The complexity of this depends on how the setter is used. There are two cases. In the simplest case the client uses the setter to initialize the values, i.e. there are no courses before the setter is applied. In this case I replace the body of the setter to use the add method.

```
class Person...
  public void setCourses(Vector arg) {
      Assert.isTrue(_courses.isEmpty());
      Enumeration e = arg.elements();
      while (e.hasMoreElements()) {
          addCourse((Course) e.nextElement());
      }
  }
```

After changing the body like this it is wise to use *Rename Method (234)* to make the intention clearer

```
public void initializeCourses(Vector arg) {
    Assert.isTrue(_courses.isEmpty());
    Enumeration e = arg.elements();
    while (e.hasMoreElements()) {
        addCourse((Course) e.nextElement());
    }
}
```

In the more general case I have to use the remove method to remove every element first and then add the elements. But I find that occurs rarely (general cases often are).

However if the clients simply set up a vector and use it, I can get them to use the add and remove methods directly, and remove the setter completely. So code like

```
Person kent = new Person();
Vector v = new Vector();
v.addElement(new Course ("Smalltalk Programming", false));
v.addElement(new Course ("Appreciating Single Malts", true));
kent.setCourses(v);
```

becomes

```
Person kent = new Person();
kent.addCourse (new Course ("Smalltalk Programming", false));
kent.addCourse (new Course ("Appreciating Single Malts", true));
```

Now I start looking at users of the getter. My first concern are those cases where somebody uses the getter to modify the underlying collection, cases like

```
kent.getCourses().addElement(new Course ("Brutal Sarcasm", false));
```

I need to replace this with a call to the new modifier.

```
kent.addCourse(new Course ("Brutal Sarcasm", false));
```

Once I've done this for everyone I can check that nobody is modifying through the getter by changing the getter body to return a copy.

```
class Person...
  Vector getCourses() {
      return (Vector) _courses.clone();
  }
```

At this point I've encapsulated the collection. Nobody can change the elements of collection except through the person. However returning a collection is still poor style, and most of the time not that helpful to its clients. Often an enumeration is more helpful since they will usually be enumerating through the collection. So in this case change the name of the existing getter and create a new getter that returns an enumeration

```
public Vector getCoursesVector() {
    return (Vector) _courses.clone();
}
public Enumeration getCourses() {
    return _courses.elements();
}
```

Now I go back to all the users of the getter. I either change them to use the enumeration, or let them call the new method. In many cases callers will get the vector in order to get the enumeration

```
Enumeration e = person.getCourses().elements();
```

The new interface makes that simpler.

```
Enumeration e = person.getCourses();
```

When I've changed all these I can compile and test. If that is too big a change I can use *Rename Method (234)* to change `getCourses` to `getCoursesVector`, and then alter callers one by one to use the enumeration directly. Most of the time I find it easier to do the two together as I am working through the caller anyway.

Now I have the right interface. Once I've done that, however, I like to look at the users of the getter, looking for code that ought to be on Person. Code like

```
Enumeration e = person.getCourses();
int count = 0;
while (e.hasMoreElements()) {
    Course each = (Course) e.nextElement();
    if (each.isAdvanced()) count ++;
}
```

Is probably better moved to person. First I use *Extract Method (114)* on the code

```
int numberOfAdvancedCourses(Person person) {
    Enumeration e = person.getCourses();
    int count = 0;
    while (e.hasMoreElements()) {
        Course each = (Course) e.nextElement();
        if (each.isAdvanced()) count ++;
    }
    return count;
}
```

And then I use *Move Method (160)* to move it to person.

```
class Person...
  public int numberOfAdvancedCourses() {
      Enumeration e = getCourses();
      int count = 0;
      while (e.hasMoreElements()) {
          Course each = (Course) e.nextElement();
          if (each.isAdvanced()) count ++;
      }
      return count;
  }
```

A common case is

```
kent.getCoursesVector().size()
```

which can be changed to

```
kent.numberOfCourses()

class Person...
  public int numberOfCourses() {
      return _courses.size();
  }
```

A few years ago I was concerned that moving this kind of behavior over to person would lead to a bloated person class. In practice, I've found that usually isn't a problem.

If your collection is an array, it is a little harder. You can provide an enumeration with an inner class

```
private String[] _skills = new String[10];

public Enumeration getSkills() {
    return new Enumeration () {
        private int _position = 0;
        public boolean hasMoreElements() {
           return (_position < _skills.length);
        }
        public Object nextElement() throws NoSuchElementException {
           if (hasMoreElements())
             return  _skills[_position++];
           else throw new NoSuchElementException();
        }
    };
}
```

Similarly you can provide methods to update the array. Usually, however, I find it easier to just change the field to be a vector instead of an array. That way I get all of the behavior of vector and I don't have to worry about sizing the array. You can use the `copyInto` method of vector to provide an array for clients who need the array.

```
private Vector _skills = new Vector();
public String[] getSkillsArray() {
    Object[] result = new Object[_skills.size()];
    _skills.copyInto(result);
    return (String[]) result;
}
```

Replace Record with Data Class

You have a record structure in a traditional programming environment. How do you begin to make this object oriented?

Make a dumb data object for the record.

Motivation

Record structures a are common feature of programming environments. There are various reasons you may want to bring them into an object-oriented program. You could be copying a legacy program as in *Replace Program With Class (305)*, you could be communicating a structured record with a traditional programming API, or a data base record. In these cases it is useful to create an interfacing class to deal with this external element. It is simplest to make the class look just like the external record. You then move other fields and methods into the class later.

A less obvious, but very compelling case of this is an array where the element in each index has some special meaning. In this case you use *Replace Array with Object (194)*.

Mechanics

- Create a class to represent the record
- Give the class a private field with a getting method and a setting method for each data item.

You now have a dumb data object, it has no behavior yet, but further refactoring will explore that issue.

Replace Type Code with Class

You have a numeric type code that does not affect behavior

Replace the number with a new class

Motivation

Numeric type codes, or enumerations, are a common feature of C-based languages. With symbolic names they can be quite readable. The problem is that the symbolic name is only an alias, the compiler still sees the underlying number. Thus the compiler type checks using the number not the symbolic name. Any method that takes the type code as an argument will expect a number, and there is nothing to force a

symbolic name to be used. This can reduce readability and be a source of bugs.

If you replace the number with a class, the compiler can now type check on the class. By providing factory methods for the class you can statically check that only valid instances are created, and that those instances are passed on to the correct objects.

Before you do this, however, you need to consider the other type code replacements. Only replace the type code with a class if the type code is pure data, that is it does not cause different behavior inside a switch statement somewhere. For a start Java can only switch on an integer, not an arbitrary class, so the replacement will fail. But more importantly than that, any switch needs to be removed by *Replace Switch with Polymorphism (147)*. In order to do that the type code has to be dealt with by *Replace Type Code with Subclasses (224)* or *Replace Type Code with State/Strategy (227)*.

Mechanics

- ❍ Create a new class for the type code
  - ☞ The class needs a code field which matches the type code, and a getting method for this value. It should have static variables for the allowable instances of the class, and a static method which returns the appropriate instance from an argument based on the original code.
- ❍ Modify the implementation of the source class to use the new class
  - ☞ Maintain the old code-based interface, but change the static fields to use new class to generate the codes, and alter the other code based methods to get the code numbers from the new class
- ❍ Compile and Test
  - ☞ At this point the new class can do run time checking of the codes.
- ❍ For each method on the source class that uses the code, create a new method that uses the new class instead.
  - ☞ Methods that use the code as an argument need new methods using an instance of the new class as an argument. Methods that return a code need a new method returning the code. It is often wise to use *Rename Method (234)* on an old accessor before creating a new one, to make the program clearer when it is using an old code
- ❍ One by one, change the clients of the source class to use the new interface
- ❍ Compile and test after each client is updated
  - ☞ You may need to alter several methods before you have enough consistency to compile and test.
- ❍ Remove the old interface that uses the codes, and the static declara-

tions of the codes
- Compile and test.

Example

I'll start with a person who has a blood group modeled with a type code

```
class Person {

  public static final int O = 0;
  public static final int A = 1;
  public static final int B = 2;
  public static final int AB = 3;

  private int _bloodGroup;

  public Person (int bloodGroup) {
      _bloodGroup = bloodGroup;
  }

  public void setBloodGroup(int arg) {
      _bloodGroup = arg;
  }

  public int getBloodGroup() {
      return _bloodGroup;
  }
}
```

I start by creating a new blood group class with instances that contain the type code number.

```
class BloodGroup {
  public static BloodGroup O = new BloodGroup(0);
  public static BloodGroup A = new BloodGroup(1);
  public static BloodGroup B = new BloodGroup(2);
  public static BloodGroup AB = new BloodGroup(3);
  private static BloodGroup[] _values = {O, A, B, AB};

  private int _code;

  private BloodGroup (int code ) {
      _code = code;
  }

  public int getCode() {
      return _code;
  }
```

```
  public static BloodGroup code(int arg) {
      return _values[arg];
  }

}
```

I then replace the code in Person with code that uses the new class.

```
class Person {

  public static final int O = BloodGroup.O.getCode();
  public static final int A = BloodGroup.A.getCode();
  public static final int B = BloodGroup.B.getCode();
  public static final int AB = BloodGroup.AB.getCode();

  private BloodGroup _bloodGroup;

  public Person (int bloodGroup) {
      _bloodGroup = BloodGroup.code(bloodGroup);
  }

  public int getBloodGroup() {
      return _bloodGroup.getCode();
  }

  public void setBloodGroup(int arg) {
      _bloodGroup = BloodGroup.code (arg);
  }
}
```

At this point I now have run-time checking within the blood group class. But to really gain from the change I have to alter the users of the person class to use blood group instead of integers.

To begin this I use *Rename Method (234)* on the accessor for the person's blood group to clarify the new state of affairs and add a new getting method that uses the new class

```
class Person...
  public int getBloodGroupCode() {
      return _bloodGroup.getCode();
  }
  public int getBloodGroup() {
      return _bloodGroup;
  }
```

I also create a new constructor and setting method that uses the class.

```
public Person (BloodGroup bloodGroup ) {
  _bloodGroup = bloodGroup;
}

public void setBloodGroup(BloodGroup arg) {
  _bloodGroup = arg;
}
```

Now I go to work on the clients of Person. The art is to do one client at a time so that you can take smaller steps. Each client may need various changes, however, that makes it more tricky. Any reference to the static variables needs to be changed. So

```
Person thePerson = new Person(Person.A)
```

becomes

```
Person thePerson = new Person(BloodGroup.A);
```

References to the getting method need to use the new one, so

```
thePerson.getBloodGroupCode()
```

becomes

```
thePerson.getBloodGroup()
```

The same is true for setting methods, so

```
thePerson.setBloodGroup(Person.AB)
```

becomes

```
thePerson.setBloodGroup(BloodGroup.AB)
```

Once this is done for all clients of person, I can remove the getting method, constructor, and setting methods that use the integer.

```
class Person ...
  public Person (int bloodGroup) {
      _bloodGroup = BloodGroup.code(bloodGroup);
  }
  public int getBloodGroup() {
      return _bloodGroup.getCode();
  }
  public void setBloodGroup(int arg) {
      _bloodGroup = BloodGroup.code (arg);
  }
```

I can also privatize the methods on blood group that use the code.

```
class BloodGroup...
  private int getCode() {
```

```
    return _code;
}

private static BloodGroup code(int arg) {
    return _values[arg];
}
```

Replace Type Code with Subclasses

You have a type code which affects the behavior of a class

Replace the type code with subclasses

Motivation

This refactoring is a necessary first step in *replacing a case statement with inheritance*. So see the motivation discussion there.

Mechanics

- *Self encapsulate* the type code.
  - ☞ If the type code is passed into the constructor, you will need to *replace the constructor with a factory method.*
- For each value of the type code create a subclass. Override the getting method of the type code in the subclass to return the relevant value.
  - ☞ This value will be hard coded into the return (e.g. “return 1). This looks messy but it is a temporary measure until all case statements have been replaced.
- Compile and test after replacing each type code value with a subclass.
- Remove the type code field from the superclass. Declare the accessors for the type code as abstract.
- Compile and test

Example

I will use the boring and unrealistic employee payment example.

```
class Employee {
```

```
private int _type;
static final int ENGINEER = 0;
static final int SALESMAN = 1;
static final int MANAGER = 2;

Employee (int type) {
    _type = type;
}

int payAmount() {
    switch (_type) {
        case ENGINEER:
           return _monthlySalary;
        case SALESMAN:
           return _monthlySalary + _commission;
        case MANAGER:
           return _monthlySalary + _bonus;
        default:
           throw new RuntimeException("Incorrect Employee");
    }
}
```

The first step is to self encapsulate the type code.

```
int getType() {
    return _type;
}

int payAmount() {
    switch (getType()) {
        case ENGINEER:
           return _monthlySalary;
        case SALESMAN:
           return _monthlySalary + _commission;
        case MANAGER:
           return _monthlySalary + _bonus;
        default:
           throw new RuntimeException("Incorrect Employee");
    }
}
```

Since the employee's constructor uses a type code as a parameter, I need to replace it with a factory method.

```
Employee create(int type) {
    return new Employee(type);
}

private Employee (int type) {
    _type = type;
```

```
}
```

I can now start with the engineer as a subclass. First I create the subclass and the overriding method for the type code.

```
class Engineer extends Employee {

  int getType() {
      return Employee.ENGINEER;
  }

}
```

I also need to alter the factory method to create the appropriate object.

```
class Employee
  static Employee create(int type) {
      if (type == ENGINEER) return new Engineer();
      else return new Employee(type);
  }
```

I continue, one by one, until all the codes are replaced by subclasses. At this point I can get rid of the type code field on employee and make `getType` an abstract method. At this point the factory method looks like this.

```
abstract int getType();

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

Of course this is the kind of switch statement I would prefer to avoid. But there is only one of them, and it is only used at creation. There are ways to deal with this too, see the discussion in *Replace Constructor with Factory Method (256)*.

Naturally once you have created the subclasses you should push down appropriate methods and fields.

Replace Type Code with State/Strategy

You have a type code which affects the behavior of a class. The class is already subclassed or the type code is mutable.

Replace it with a state object.

Motivation

This is similar to *replacing a tyw9pe code with subclassing*, but can be used if the type code changes during the life of the object, or if some other reason prevents subclassing. It uses either the state or strategy pattern [Gang of Four].

State and strategy are very similar, so the refactoring is the same whichever you use, and it doesn't really matter. Choose the pattern based on what fits the specific circumstances better. If you are moving for a single algorithm, then strategy is the better term. If you are going to move state specific data and you think of the object as changing state use the state pattern.

Mechanics

- *Self encapsulate* the type code.
- Create a new class, name it after the purpose of the type code. This is the state object.
- Add subclasses of the state object, one for each type code.
  - ☞ It is easier to do them all at once, rather than one at a time.
- Create an abstract query in the state object to return the type code. Create overriding queries of each state object subclass to return the correct type code.
- Compile
- Create a field in the old class for the new state object.
- Adjust the type code query on the original class to delegate to the state object.
- Adjust the type code setting methods on the original class to assign an instance of the appropriate state object subclass.
- Compile and test.

Example

I'll use the tiresome and brainless employee example.

```
class Employee {

  private int _type;
  static final int ENGINEER = 0;
  static final int SALESMAN = 1;
  static final int MANAGER = 2;

  Employee (int type) {
      _type = type;
  }

  int payAmount() {
      switch (_type) {
          case ENGINEER:
             return _monthlySalary;
          case SALESMAN:
             return _monthlySalary + _commission;
          case MANAGER:
             return _monthlySalary + _bonus;
          default:
             throw new RuntimeException("Incorrect Employee");
      }
  }
```

I assume this is an exciting and go-ahead company that allows promotion of managers to engineers. Thus the type code is mutable and I can't use subclassing. My first step, as ever, is to self-encapsulate the type code.

```
  Employee (int type) {
      setType (type);
  }

  int getType() {
      return _type;
  }

  void setType(int arg) {
      _type = arg;
  }

  int payAmount() {
      switch (getType()) {
          case ENGINEER:
             return _monthlySalary;
          case SALESMAN:
```

```
            return _monthlySalary + _commission;
        case MANAGER:
            return _monthlySalary + _bonus;
        default:
            throw new RuntimeException("Incorrect Employee");
    }
  }
```

Now I declare the state class. I declare this as an abstract class and provide an abstract method for returning the type code.

```
abstract class EmployeeType {
  abstract int getTypeCode();
}
```

I'll now create the subclasses.

```
class Engineer extends EmployeeType {

  int getTypeCode () {
      return Employee.ENGINEER;
  }
}

class Manager extends EmployeeType {

  int getTypeCode () {
      return Employee.MANAGER;
  }
}

class Salesman extends EmployeeType {

  int getTypeCode () {
      return Employee.SALESMAN;
  }
}
```

I compile so far, and it is all so trivial that, even for me, it compiles easily. Now I actually hook the subclasses into the employee by modifying the accessors for the type code.

```
class Employee…
  private EmployeeType _type;

  int getType() {
      return _type.getTypeCode();
  }

  void setType(int arg) {
      switch (arg) {
```

```
            case ENGINEER:
               _type = new Engineer();
               break;
            case SALESMAN:
               _type = new Salesman();
               break;
            case MANAGER:
               _type = new Manager();
               break;
            default:
               throw new IllegalArgumentException("Incorrect Employee Code");
        }
    }
```

Of course this means I now have a switch statement here. But once I'm done refactoring it will be the only one anywhere in the code, and it will only be executed when the type is changed. All the other case statements can now be eliminated by *Replace Switch with Polymorphism (147)*.

Still I like to finish the job by moving all knowledge of the type codes and subclasses over to the new class. First I copy the type code definitions into the employee type, create a factory method for employee types, and adjust the setting method on employee

```
class Employee…
  void setType(int arg) {
      _type = EmployeeType.newType(arg);
  }

class EmployeeType…
  static EmployeeType newType(int code) {
      switch (code) {
          case ENGINEER:
             return new Engineer();
          case SALESMAN:
             return new Salesman();
          case MANAGER:
             return new Manager();
          default:
             throw new IllegalArgumentException("Incorrect Employee Code");
      }
  }
  static final int ENGINEER = 0;
  static final int SALESMAN = 1;
  static final int MANAGER = 2;
```

Then I remove the type code definitions from the employee and replace them with references to the employee type.

```
class Employee…
  int payAmount() {
      switch (getType()) {
          case EmployeeType.ENGINEER:
             return _monthlySalary;
          case EmployeeType.SALESMAN:
             return _monthlySalary + _commission;
          case EmployeeType.MANAGER:
             return _monthlySalary + _bonus;
          default:
             throw new RuntimeException("Incorrect Employee");
      }
  }
```

Chapter 10: Making Method Calls Simpler

Objects are all about interfaces, and coming up with interfaces that are easy to understand and use is a key skill in developing good object-oriented software. This chapter explores refactorings that make interfaces more straightforward.

Often the simplest, and most important, thing you can do is to change a method's name. Naming is a key tool in communication, and as you understand what a program is doing you should not be afraid to use *Rename Method (234)* to pass on that knowledge. You can (and should) also rename variables and classes. On the whole these renamings are fairly simple text replacements, so I haven't added extra refactorings for them.

Parameters themselves have quite a role to play with interfaces. Often those new to objects use long parameter lists, which are typical of other development environments. Objects allow you to keep parameter lists shorter. If you are passing several values from an object then use *Preserve Whole Object (241)* to reduce all the values to a single object. If this object does not exist you can create it with *Introduce Parameter Object (247)*. If you can get the data from an object the method already has access to then you can eliminate parameters with *Replace Parameter with Method (245)*. On occasion, however, you can combine several similar methods by adding a parameter with *Parameterize Method (240)*.

One of the most valuable conventions that I've used over the years is to clearly separate methods that change state (modifiers) from those that query state (queries). I don't know how many times I've got myself into trouble, or seen others get into trouble, by mixing these up. So whenever I see them combined I use *Separate Query from Modifier (236)* to get rid of them.

Good interfaces show only what they have to, and no more. So you can make an interface better by hiding things. Of course all data should be hidden (I hope I don't need to tell you to do that), but also any methods that can be should be hidden too. Often when refactoring you need to make things visible for a while and then cover them all up with *Hide Method (255)* and *Remove Setting Method (252)*.

Constructors are a particularly awkward feature of Java and C++, because they force you to know what class an object is that you need to create. Often you don't need to know this. The need to know can be removed by using *Replace Constructor with Factory Method (256)*.

Casting is another bane of the Java programmer's life. As much as possible try to avoid making the user of a class do downcasting if you can contain it elsewhere by using *Encapsulate Downcast (258)*.

Java, like many modern languages, has an exception handling mechanism to make error handling easier. Programmers who are not used to this often use error codes to signal trouble, you can use *Replace Error Code With Exception (259)* to use the new exceptional features. But sometimes exceptions aren't the right answer, you should test first instead by using *Replace Exception With Test (264)*.

Rename Method

A method could be renamed to make its intention clearer

Change the name of the method

Motivation

An important part of the code style I am advocating here is small methods to factor complex processes. This can lead you on a merry dance to find out what all the little methods do. The key to avoiding this merry dance is the naming of the methods. Methods should be named in a way that communicates their intention. A good way to do this is to think what the comment for the method would be, and to turn that comment into the method's name.

Life being what it is, you won't get your names right first time. In this situation you may well be tempted to leave it – after all its only a name. That is the work of the evil demon "Obfusticatis", don't listen to him. If you see a badly named method it is imperative that you change it. Remember your code is for a human first, and a computer second. Humans need good names. Take note when you spend ages trying to do something that would have been easier if a couple of methods were better named. Good naming is a skill that requires practice, improving this skill is the key to being a truly skillful programmer.

The same applies to other aspects of the signature. If reordering the parameters makes matters clearer, then do it.

Mechanics

- Check to see if this method signature is implemented by a super-class or subclass. If so carry out these steps for each implementation.
- Declare a new method with the new name. Copy the old body of code over the new name and make any alterations to fit.
- Compile
- Change the body of the old method so that it calls the new one.
  - ☞ If you only have a few references you can reasonably skip this step.
- Compile and test.
- Find all references to the old method name and change them to refer to the new one. Compile and test after each change
  - ☞ In a strongly typed language you can do this by compiling. Otherwise use a text search to the old references
- Remove the old method
  - ☞ If the old method is part of the interface and you cannot remove it, you may leave it in place and mark it as deprecated.
- Compile and test.

Example

Say we have a method to get a person's telephone number

```
public String getTelephoneNumber() {
    return ("(" + _officeAreaCode + ") " + _officeNumber);
}
```

We want to rename the method to `getOfficeTelephoneNumber`. We begin by creating the new method, and copying the body over to the new method. The old method now changes to call the new one.

```
class Person...
```

```
public String getTelephoneNumber(){
    return getOfficeTelephoneNumber();
}
public String getOfficeTelephoneNumber() {
    return ("(" + _officeAreaCode + ") " + _officeNumber);
}
```

Now we find the callers of the old method, and switch them to call the new one. When we have switched them all, we can remove the old method.

The procedure is the same if we need to add or remove a parameter.

If there aren't many callers I just change the callers to call the new method, without using old method as a delegating method. If my tests throw a wobbly, I just back out and do it the slow way.

Separate Query from Modifier

You have a method that returns a value, but also changes the state of an object

Create two methods, one for the query and one for the modification

Motivation

When you have a function that gives you a value, and has no observable side-effects, you have a very valuable thing. You can call this function as often as you like, you can move to other places in the method. In short you have a lot less to worry about.

It is a good idea to clearly signal the difference between methods with side-effects and those without. A good rule to follow is to say that any method that returns a values should not have observable side effects. Some people (such as [Meyer]) treat this as an absolute rule. I'm not 100% pure on this (as on anything), but I try to follow it most of the time, and it has served me well.

So if you come across a method that returns a value but does also have side effects, then you should try to separate the query from the modifier.

You'll note I use the phrase *observable* side effects. A common optimization is to cache the value of a query in a field so that repeated calls go quicker. Although this changes the state of the object with the cache, the change is not observable, in that any sequence of queries will always return the same results for each query. (Again see [Meyer] for more details).

Mechanics

- Create a query that returns the same value as the original method
  - Look in the original method to see what is returned. If the returned value is a temporary look at all uses of the temp to see what value is returned.
- Modify the original method so it returns the result of a call to the query.
  - That is every return in the original method should say "`return newQuery()`", instead of returning anything else.
  - If the method used a temp to with a single assignment to capture the return value, you should be able to remove it.
- Compile and test
- For each call, replace the single call to the original method with a call to the query. Add a call to the original method before the line that calls the query. Compile and test and after each change to a calling method
- Make the original method have a void return type and remove the return expressions

Example

Here is a function that tells you the name of a miscreant for a security system and sends an alert. The rule is that only one alert is sent even if there is more than one miscreant.

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

```
        }
    }
    return "";
}
```

It is called by

```
void checkSecurity(String[] people) {
    String found = foundMiscreant(people);
    someLaterCode(found);
}
```

To separate the query from the modifier I first need to create a suitable query that returns the same value as the modifier does, but without doing the side effects.

```
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";
        }
    }
    return "";
}
```

Then replace every return in the original function, one at a time, with calls to the new query. Test after each replacement. When you are done the original method looks like

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

        }
        if (people[i].equals ("John")){
           sendAlert();
           return foundPerson(people);

        }
    }
    return foundPerson(people);
}
```

Now I alter all the calling methods to do two calls: first to the modifier, then to the query.

```
void checkSecurity(String[] people) {
    foundMiscreant(people);
    String found = foundPerson(people);
    someLaterCode(found);
}
```

Once I have done this for all calls I can alter the modifier to give it a void return type. At this point it seems a good idea

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

Now it seems better to change the name of the original

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

Of course in this case I have a lot of code duplication as the modifier uses the query to do its work. I can now do an algorithm substitution on the modifier to take advantage of this.

```
void sendAlert(String[] people){
    if (! foundPerson(people).equals(""))
        sendAlert();
}
```

Parameterize Method

Several methods do similar things, but with different values contained in the method body

Create one method that uses a parameter for the different values

Motivation

You may see a couple of methods that do similar things, but vary depending on a few values. In this case you can simplify matters by replacing the separate methods with a single method which handles the variations by parameters. Such a change removes duplicate code, and increases flexibility since you can deal with other variations by adding parameters.

Mechanics

- Create a parameterized method that can be substituted for each repetitive method
- Compile
- Replace one old method with a call to the new method
- Compile and test
- Repeat for all the methods, testing after each one.

You may find that you cannot do this for the whole method, but you could for a fragment of a method. In this case first extract the fragment into a method, then parameterize that method.

Example

The simplest case of this is methods along the following lines.

```
Class Employee {
  Void tenPercentRaise () {
      salary *= 0.1;
}

  Void fivePercentRaise () {
      salary *= 0.05;
}
```

which can be replaced with

```
void raise (double raiseAmount) {
  salary *= raiseAmount;
}
```

Of course that is so simple that anyone would spot it.

A less obvious case is (from the example on page 435)

```
protected Dollars baseCharge() {
    double result = Math.min(lastUsage(),100) * 0.03;
    if (lastUsage() > 100) {
        result += (Math.min (lastUsage(),200) - 100) * 0.05;
    };
    if (lastUsage() > 200) {
        result += (lastUsage() - 200) * 0.07;
    };


    return new Dollars (result);
}
```

which can be replaced by

```
protected Dollars baseCharge() {
    double result = usageInRange(0, 100) * 0.03;
    result += usageInRange (100,200) * 0.05;
    result += usageInRange (200, Integer.MAX_VALUE) * 0.07;

    return new Dollars (result);
}

protected int usageInRange(int start, int end) {
    if (lastUsage() > start) return Math.min(lastUsage(),end) - start;
    else return 0;
}
```

The trick is to spot code that is repetitive based on a few values that can be passed in as parameters.

Preserve Whole Object

You are getting several values from an object and passing these values to a method call

Send the whole object instead

Motivation

I often see code that looks like this

```
temp1 = anObject.getValue1();
temp2 = anObject.getValue2();
temp3 = anObject.getValue3();
calledObject.aMethod(temp1, temp2, temp3);
```

You are busy pulling values out of one object to pass these values to another object. Usually it is better to do

```
calledObject.aMethod(anObject);
```

The code is easier to see and the called object can get whatever it needs. If later on it needs to get something else it can do that itself, instead of involving the middle man.

There is a downside. In the first case the called object does not need to know about `anObject`, only about the values. *Preserve Whole Object (241)* thus causes a dependency between the called object and `anObject`. If this is going to mess up your dependency structure then don't use this refactoring.

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

If `aMethod` uses all these values from `anObject`, that is a signal that `aMethod` should really be defined on `anObject`. So when you are considering this refactoring, consider *Move Method (160)* instead.

You may not already have the whole object defined: in this case you need to *Introduce Parameter Object (247)*.

An important use of this refactoring is when `anObject` is the calling object. So instead of

```
calledObject.aMethod(field1, field2, field3)
```

consider using

```
calledObject.aMethod(this)
```

providing you have the appropriate Getting Methods.

Mechanics

- Create a new parameter for the whole object
- Add the whole object to the parameter list
- Compile and test
- Determine which parameters should be obtained from the whole object
- Take one parameter and replace references to it within the method body by invoking an appropriate method on the whole object parameter
- Delete the parameter.
- Compile and Test
- Repeat for each parameter that can be got from the whole object
- Remove the code in the calling method that obtains the deleted parameters.
  - ☞ Unless, of course, the code is using object somewhere else.
- Compile and Test

Example

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

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

Rather than unpack the range information when I pass it, I can pass the whole range. In this simple case I could do this as one step, but when there are more involved you can do it in smaller steps. First you can just add the whole object to the parameter list.

```
class HeatingPlan …
```

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

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

Then use one the whole object instead of one of the parameters.

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

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

And continue until you've changed all you need.

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

class Room …
  boolean withinPlan(HeatingPlan plan) {
      return plan.withinRange(daysRange());
  }
```

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

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

Replace Parameter with Method

An object computes a value then passes it as a parameter for a method. The receiver could also carry out this calculation.

Define a method to get the parameter object and replace references to the parameter with the new method

Motivation

If a method can get a value that is passed in as parameter by another means, then it should. Long parameter lists are difficult to understand, and we should reduce them as much as possible.

One way of doing this is to look to see if the receiving method could make the same calculation. If you are calling a method on the same object, and the calculation for the parameter does not reference any of the parameters of the calling method, then you should be able to remove the parameter by turning the calculation into its own method. This is also true if you are calling a method on an object that has a reference to the calling object.

You can't do this if the calculation of the parameter relies on a parameter of the calling method, since that parameter may change with each call. (Unless, of course, that parameter can be replaced with a method.) You also can't do it if the receiver does not have a reference to the sender, and you don't want to give it one.

In some cases the parameter may be there for a future parameterization of the method. In this case I would still get rid of it. Deal with the parameterization when you need it, you may well find out that you don't have the right parameter anyway. I would only make an exception to this rule when the resulting interface change would have painful consequences around the whole program, such as a long build or changing a lot of embedded code. If this worries you look into how painful such a change would really be. You should also look to see if you can reduce the dependencies that cause the change to be so pain-

ful. Stable interfaces are good, but freezing a poor interface is a problem.

Mechanics

- Extract the calculation of the parameter into a method
- Replace references to the parameter in method bodies with references to the method.
- Compile and test after each replacement
- Remove the parameter from method calls and declarations
- Compile and test.

Example

We'll start with yet another unlikely variation on discounting orders

```
public double getPrice() {
    int basePrice = _quantity * _itemPrice;
    int discountLevel;
    if (_quantity > 100) discountLevel = 2;
    else discountLevel = 1;
    double finalPrice = discountedPrice (basePrice, discountLevel);
    return finalPrice;
}

private double discountedPrice (int basePrice, int discountLevel) {
    if (discountLevel == 2) return basePrice * 0.1;
    else return basePrice * 0.05;
}
```

I can begin by extracting the calculation of the discount level.

```
public double getPrice() {
    int basePrice = _quantity * _itemPrice;
    int discountLevel = getDiscountLevel();
    double finalPrice = discountedPrice (basePrice, discountLevel);
    return finalPrice;
}

private int getDiscountLevel() {
    if (_quantity > 100) return 2;
    else return 1;
}
```

Then replace references to the parameter in `discountedPrice`.

```
private double discountedPrice (int basePrice, int discountLevel) {
    if (getDiscountLevel() == 2) return basePrice * 0.1;
    else return basePrice * 0.05;
}
```

Then I can get rid of the parameter.

```
public double getPrice() {
    int basePrice = _quantity * _itemPrice;
    int discountLevel = getDiscountLevel();
    double finalPrice = discountedPrice (basePrice);
    return finalPrice;
}

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

Of course I can now get rid of the temp, not to mention the other parameter and it's temp. Then I am left with.

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

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

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

So I might as well inline `discountedPrice`

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

Introduce Parameter Object

You have a group of parameters that naturally go together

Replace them with an object

Motivation

Often you see a particular group of parameters that tend to be often passed together. Several methods may use this group, either on one class or in several classes. Such a group of classes is a data clump and can be replaced by an object that carries all of this data.

It is worthwhile to turn these parameters into objects just to group the data together. This is useful because it reduces the size of the parameter lists, and long parameter lists are hard to understand. The defined accessors on the new object also make the code more consistent – which again makes it easier to understand and modify.

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

Mechanics

- Create a new class to represent the data clump. Make the class immutable.
- Compile
- Add a parameter for the new data clump to the method signature. Add a null for this parameter in all the callers and compile
- For each parameter in the data clump, remove the parameter from the signature, and modify the caller and method body to use the parameter object for that value.
- Compile and test after you remove each parameter.
- When you have removed the parameters, look for behavior that you move into the parameter object using *Move Method (160)*.
  - ☞ This may be a whole method, or part of a method. If it is part of a method, use *Extract Method (114)* first and then move the new method over.

Example

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

```
class Entry...
  Entry (double value, Date chargeDate) {
      _value = value;
      _chargeDate = chargeDate;
```

```
  }
  Date getDate(){
      return _chargeDate;
  }
  double getValue(){
      return _value;
  }
  private Date _chargeDate;
  private double _value;
```

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

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

  private Vector _entries = new Vector();

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

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

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

```
        return _end;
    }
    private Date _start;
    private Date _end;
}
```

I've made the date range class immutable, that is all the values for the date range are set in the constructor and I haven't provided any methods for modifying the values. This is a wise move to avoid aliasing bugs. Since Java has pass-by-value parameters, this mimics the way Java's parameters work, so is the right assumption for this refactoring.

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

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

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

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

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

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

```
            {
                result += each.getValue();
            }
        }
        return result;
    }

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

I then remove the end date

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

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

That's introduced the parameter object, however I can get more value from this refactoring by moving behavior from other methods to the new object. In this case I can take the code in the condition and use *Extract Method (114)* and *Move Method (160)* to get.

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

class DateRange...
```

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

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

Remove Setting Method

A field should be set at creation time and never altered

Remove any Getting Method for that field

Motivation

Providing a setting method indicates that a field may be changed. If you don't want that field to change once the object is created then don't provide a setting method. That way not just is your intention clearer, you will also often remove the very possibility of it happening.

This situation often occurs with people who blindly use Indirect Variable Access [Beck]. Such people then use setters even in a constructor. I guess there is an argument for consistency, but not compared to the confusion that that the setting method will cause later on.

Mechanics

- Check that the setting method is only called in the constructor, or in a method called by the constructor.
- Modify the constructor to access the variables directly.
  - ☞ You cannot do this if you are a subclass setting a superclass's private fields. In this case you should try to provide a protected superclass method (ideally a constructor) to set these values. Whatever you do, don't name it so it can be confused with a setting method.

Example

A simple example of this is

```
class Account {
```

```
  private String _id;

  Account (String id) {
      setId(id);
  }

  void setId (String arg) {
      _id = arg;
  }
```

Which can be replaced with

```
class Account {

  private String _id;

  Account (String id) {
      _id = id;
  }
```

The problems come in some variations. First there is the case where you are doing some computation on the argument.

```
class Account {

  private String _id;

  Account (String id) {
      setId(id);
  }

  void setId (String arg) {
      _id = "ZZ" + arg;
  }
```

If the change is simple (as here) and there is only one constructor, then you could do it in the constructor. But if it is complex or you need to call it from separate methods, then you really do need to provide a method. In that case you need to name the method to make it's intention clear.

```
class Account {

  private String _id;

  Account (String id) {
      initializeId(id);
  }

  void initializeId (String arg) {
```

```
        _id = "ZZ" + arg;
    }
```

If you are worried about the method being called at a later time, you can provide some further control values to enforce the non-update rule.

```
class Account {

    private boolean _isCreated = false;

    Account (String id) {
        initializeId(id);
        _isCreated = true;
    }

    void initializeId (String arg) {
        if (_isCreated) throw new RuntimeException("Account ID cannot be changed");
        _id = "ZZ" + arg;
    }
```

Of course this does not stop a determined programmer first changing `_isCreated` to false before changing the id. With this I am guarding against human error. By making it difficult to make a mistake, you reduce the chances of making one.

The other awkward case lies with subclasses initializing private superclass variables.

```
class InterestAccount extends Account…

    private double _interestRate;

    InterestAccount (String id, double rate) {
        setId(id);
        _interestRate = rate;
    }
```

The problem is that you cannot access the id directly to set it. The best solution is to use a superclass constructor.

```
class InterestAccount …

    InterestAccount (String id, double rate) {
        super(id);
        _interestRate = rate;
    }
```

If that is not possible, then a well named method is the best thing to do.

```
class InterestAccount …

  InterestAccount (String id, double rate) {
      initializeId(id);
      _interestRate = rate;
  }
```

Another case to consider is when we are setting the value of a collection.

```
class Person {
  Vector getCourses() {
      return _courses;
  }
  void setCourses(Vector arg) {
      _courses = arg;
  }
  private Vector _courses;
```

Here I want to replace the setter with add and remove operations. I talk about this in *Encapsulate Collection (212)*.

Hide Method

A Method is not used by any other class

Make the Method private

Motivation

Refactoring often causes you to change decisions about the visibility of methods. It is easy to spot cases when you need to make a method more visible: another class needs it and you thus relax the visibility. It is somewhat more difficult to tell when a method is too visible. Ideally a tool should check all methods to see if they can be hidden. Failing that you should make this check at regular intervals.

A particularly common case of this is hiding Getting and Setting methods as you work up a higher level interface. This is most common

when you are starting with a class that is little more than an encapsulated data holder. As it gets more behavior built into it, you may well find that many of the Getting and Setting Methods are no longer needed publicly, in which case they can be hidden. If you make a Getting or Setting Method private, and you are using Direct Variable Access, then you can remove the method.

Mechanics

- Check regularly for opportunities to make a method more private
  - Use a lint style tool, do manual checks every so often, check when you remove a call to a method in another class
  - Particularly look for cases like this with setting methods.
- Make each method as private as you can
- Compile after each batch
  - The compiler checks this part a system, so you don't need to compile with each change (if one goes wrong it is easy to spot).

Replace Constructor with Factory Method

You want to do more than simple construction when you create an object

Replace the constructor with a factory method

Motivation

The most obvious motivation for this comes with replacing a type code with subclassing. You have an object that is often created with a type code, but now needs subclasses. The exact subclass is determined based on the type code. Unfortunately constructors in Java (and C++) can only return an instance of the object that is asked for. (In Smalltalk you can return anything.) So you need to replace the constructor with a factory method [Gang of Four].

You can use factory methods for other situations where constructors are too limited. Factory methods are essential for *Change Value to Reference (191)*. They can also be used to signal different creation behavior that goes beyond the number and types of parameters.

Mechanics

- Create a factory method. Make its body a call to the current constructor.
- Replace all calls to the constructor with calls to the factory method
- Compile and test after each replacement.
- Declare the constructor as private
- Compile

Example

I'll quickly use the wearisome and labored employee example. Currently we have the following employee

```
class Employee {

  private int _type;
  static final int ENGINEER = 0;
  static final int SALESMAN = 1;
  static final int MANAGER = 2;

  Employee (int type) {
      _type = type;
  }
```

I want to make subclasses of employee corresponding to the type codes. So I need to create a factory method.

```
  static Employee create(int type) {
      return new Employee(type);
  }
```

I then change all callers of the constructor to use this new method, and make the constructor private.

```
client code...
  Employee eng = Employee.create(Employee.ENGINEER);

class Employee...
  private Employee (int type) {
      _type = type;
  }
```

I can then play around with the factory method to our heart's delight. See *Replace Type Code with Subclasses (224)* for examples.

Encapsulate Downcast

A method returns an object that needs to be downcasted by its callers

Move to downcast to within the method

Motivation

Downcasting is one of the most annoying things you have to do with strongly typed OO languages. It is annoying because it feels unnecessary, you are telling the compiler something it ought to be able to figure out itself. But since the figuring out is often rather complicated, you often have to do it yourself. This is particularly prevalent in Java, where the lack of templates means that you have to downcast whenever you take an object out of a collection.

Downcasting may be a necessary evil, but you should do this as little as possible. So if ever you return a value from a method, and you know the type of what is returned is more specialized than what the method signature says, you are putting unnecessary work on your clients. Rather than forcing them to do the downcasting, you should always provide them with the most specific type you can.

Often you find this with methods that return an iterator or collection. Instead look to see what people are using the iterator for, and provide the method for that.

Mechanics

- Look for cases where you have to downcast the result from calling a method
  - ☞ These cases often appear with methods that return a collection or iterator.
- Move the downcast into the method
  - ☞ With methods that return collections, use *Encapsulate Collection (212)*.

Example

The simple case of this is obvious. You have a method called `lastReading` which returns the last reading from a vector.

```
Object lastReading() {
  return readings.lastElement();
}
```

You should replace this with

```
Reading lastReading() {
  return (Reading) readings.lastElement();
}
```

A good lead in to doing this is where you have collection classes. Say this collection of readings is on a Site class and you see code like this.

```
Reading lastReading = (Reading) theSite.readings().lastElement()
```

You can avoid the downcast, and hide which collection is being used, by

```
Reading lastReading = theSite.lastReading();

class Site {
  Reading lastReading() {
      return (Reading) readings().lastElement();
  }
```

Altering a method to return a subclass does alter the signature of the method, but will not break existing code since the compiler knows it can substitute a subclass for the superclass. Of course you should ensure that the subclass does not do anything that breaks the superclass's contract.

Replace Error Code With Exception

A method returns a special code to indicate an error

Throw an exception instead

Motivation

In computers, as in life, things go wrong occasionally. When things go wrong you need to do something about it. In the simplest case, you can just stop the program with an error code. This is the software equivalent of missing a flight and therefore committing suicide. (If I did that I wouldn't be alive even if I were a cat.) Despite my glib attempt at humor there is merit to the software suicide option. If the cost of a program crash is small, and the user is tolerant, then this is fine. However more important programs need more important measures.

The problem is that the part of the program that spots error, isn't always the part that can figure out what to do about it. Thus when such a routine finds an error it needs to let its caller know, and that caller may pass the error up the chain. In many languages a special output is used to indicate error. Unix and C based systems traditionally use a return code to signal success and failure of a routine.

Modern languages, such as Java, have a better way: exceptions. Exceptions are better because they clearly separate normal processing from error processing. This makes programs easier to understand, and as I hope you now believe, understandability is next to godliness.

Mechanics

- Decide if the exception should be checked or unchecked.
  - If the caller is responsible for testing the condition before calling, then make it unchecked.
  - If it is checked, either create a new exception or use an existing one.
- Find all the callers and adjust them to use the new method.
  - If this is an unchecked exception then adjust them to make the appropriate check before calling the method. Compile and test after each such change.
  - If this is a checked exception, adjust them to call the method in a try block.
- Change the signature of the method to reflect the new usage.

If you have many callers then this can be too big a change, you can make it more gradual with the following steps

- Decide if the exception should be checked or unchecked.
- Create a new method which uses the exception.
- Modify the body of the old method to call the new method.
- Compile and Test
- For each caller of the old method, adjust it to call the new method.

Compile and test after each change.

- Delete the old method.

Example

Isn't it strange that textbooks often assume you can't withdraw more than your balance from an account, although in real life you often can?

```
class Account...
  int withdraw(int amount) {
      if (amount > _balance)
          return -1;
      else {
          _balance -= amount;
          return 0;
      }
  }

  private int _balance;
```

To change this to use an exception I first need to decide whether to use a checked or unchecked exception. The decision hinges on whether it is the responsibility of the caller to test the balance before withdrawing, or whether it is the responsibility of the withdraw routine to make the check. If it is the caller's responsibility then it is a programming error to call withdraw with an amount greater than the balance. Since it is a programming error, that is a bug, I should use an unchecked exception. However if it is the withdraw routine's responsibility, then I must declare the exception in the interface. That way I signal to the caller that they should expect this to happen and to take appropriate measures.

Let's take the unchecked case first. Here I expect the caller to do the checking. First I look at the callers. In this case nobody should be using the return code since it is a programmer error to do so. So if I see code like

```
if (account.withdraw(amount) == -1)
    handleOverdrawn();
else doTheUsualThing();
```

I need to replace it with code like

```
if (amount > account.getBalance())
    handleOverdrawn();
else {
```

```
        account.withdraw(amount);
        doTheUsualThing();
    }
```

I can compile and test after each of these changes.

Now I need to remove the error code and throw an exception for the error case. Since the behavior is (by definition) exceptional, I should use a guard clause for the condition check.

```
void withdraw(int amount) {
    if (amount > _balance)
        throw new IllegalArgumentException ("Amount too large");
    _balance -= amount;
}
```

In fact since it is a programmer error, I should signal even more clearly by using an assertion.

```
class Account...
  void withdraw(int amount) {
      Assert.isTrue ("amount too large", amount > _balance);
      _balance -= amount;
  }

class Assert...
  static void isTrue (String comment, boolean test) {
      if (! test) {
          throw new RuntimeException ("Assertion failed: " + comment);
      }
  }
```

I handle the checked exception case slightly differently. First I create (or use) an appropriate new exception.

```
class BalanceException extends Exception {}
```

Then I adjust the callers to look like

```
    try {
        account.withdraw(amount);
        doTheUsualThing();
    } catch (BalanceException e) {
        handleOverdrawn();
    }
```

Now I change the withdraw method to use the exception.

```
void withdraw(int amount) throws BalanceException {
    if (amount > _balance) throw new BalanceException();
    _balance -= amount;
}
```

The awkward thing about this procedure is that I have to change all the callers and the called routine in one go. Otherwise the compiler spanks us. If there are a lot of callers, this can be too large a change without the compile test step.

For these cases you can use a temporary intermediate method. I begin with the same case as before

```
if (account.withdraw(amount) == -1)
    handleOverdrawn();
else doTheUsualThing();

class Account ...
  int withdraw(int amount) {
      if (amount > _balance)
          return -1;
      else {
          _balance -= amount;
          return 0;
      }
  }
```

Now our first step is to create a new withdraw method that uses the exception, just as above.

```
void newWithdraw(int amount) throws BalanceException {
    if (amount > _balance) throw new BalanceException();
    _balance -= amount;
}
```

Next I adjust the current withdraw method to use the new one.

```
int withdraw(int amount) {
    try {
        newWithdraw(amount);
        return 0;
    } catch (BalanceException e) {
        return -1;
    }
}
```

With that done, I can compile and test. Now I can replace each of the calls to the old method with a call to the new one.

```
    try {
        account.newWithdraw(amount);
        doTheUsualThing();
    } catch (BalanceException e) {
        handleOverdrawn();
    }
```

Now with both old and new methods in place, I can compile and test after each change. When I'm all done I can delete the old method and use *Rename Method (234)* to give the new method the old name.

Replace Exception With Test

You are throwing a checked exception on a condition the caller could have checked first

Change the caller to make the test first

Motivation

Exceptions are a significant advance in programming languages. They allow us to avoid those complex codes by *Replace Error Code With Exception (259)*. But like so many pleasures, they can be used to excess, and then cease to become pleasurable. (Yes even I can get tired of Aventinus.) Exceptions should be used for exceptional behavior, that is behavior that is an unexpected error. They should not act as a substitute for conditional tests. Therefore if you can reasonably expect the caller to check the condition before calling the operation, you should provide a test and the caller should use it.

Mechanics

- Put a test up-front and copy the code from the catch block into the appropriate leg of the if statement
- Add an assertion to the catch block to notify you if the catch block is executed
- Compile and test
- Remove the catch block and the try block if there are no other catch blocks
- Compile and test

Example

For this example we'll use an object that manages resources that are expensive to create but can be reused. Database connections are a good

example of this. Such a manager has two pools of resources, one that is available for use, and one that is allocated. When a client wants a resource the pool hands it out, and transfers it from the available pool to the allocated pool. When a client releases a resource the manager passes it back. If a client requests a resource and there are non available, then the manager creates a new one.

The method for giving out resources might look like this.

```
class ResourcePool
  Resource getResource() {
      Resource result;
      try {
          result = (Resource) _available.pop();
          _allocated.push(result);
          return result;
      } catch (EmptyStackException e) {
          result = new Resource();
          _allocated.push(result);
          return result;
      }
  }
  Stack _available;
  Stack _allocated;
```

In this case running out of resources is not an unexpected occurrence, so I should not use an exception.

To remove the exception I first add an appropriate up-front test, and do the empty behavior there.

```
  Resource getResource() {
      Resource result;
      if (_available.isEmpty()) {
          result = new Resource();
          _allocated.push(result);
          return result;
      }
      else {
          try {
             result = (Resource) _available.pop();
             _allocated.push(result);
             return result;
          } catch (EmptyStackException e) {
             result = new Resource();
             _allocated.push(result);
             return result;
          }
      }
```

```
}
```

With this the exception should never occur. I can add an assertion to check this

```
Resource getResource() {
    Resource result;
    if (_available.isEmpty()) {
        result = new Resource();
        _allocated.push(result);
        return result;
    }
    else {
        try {
           result = (Resource) _available.pop();
           _allocated.push(result);
           return result;
        } catch (EmptyStackException e) {
           Assert.shouldNeverReachHere("avaiable was empty on pop");
           result = new Resource();
           _allocated.push(result);
           return result;
        }
    }
}

class Assert...
  static void shouldNeverReachHere(String message) {
      throw new RuntimeException (message);
  }
```

Now I can compile and test. If all goes well I can then remove the try block completely

```
Resource getResource() {
    Resource result;
    if (_available.isEmpty()) {
        result = new Resource();
        _allocated.push(result);
        return result;
    }
    else {
        result = (Resource) _available.pop();
        _allocated.push(result);
        return result;
    }
}
```

After this you usually find you can clean up the conditional code in some way. Here I can *Consolidate duplicate conditional fragments (140)*

```
Resource getResource() {
    Resource result;
    if (_available.isEmpty())
        result = new Resource();
    else
        result = (Resource) _available.pop();
    _allocated.push(result);
    return result;
}
```

Chapter 11: Dealing with Generalization

Generalization produces its own batch of refactorings, mostly dealing with moving methods around a hierarchy of inheritance. *Pull Up Field (270)* and *Pull Up Method (271)* both look at promoting function up a hierarchy while *Push Down Method (276)* and *Push Down Field (276)* push function downwards. Constructors are a little bit more awkward to pull up so *Pull up Constructor Body (273)* talks about those issues. Rather than pushing down a constructor it's often useful to *Replace Constructor with Factory Method (256)* instead.

If you have methods that have a similar outline body but vary in details you can use *Form Template Method (289)* to separate the differences from the similarities.

As well as moving function around a hierarchy you can also change the hierarchy by creating new classes. *Extract Subclass (277)*, *Extract Superclass (282)*, and *Extract Interface (286)* all do this by forming new elements out of various points. *Extract Interface (286)* is particularly important where you want to mark just a small amount of function for the type system. If you find yourself with unnecessary classes in your hierarchy you can use *Collapse Hierarchy (288)* to remove them.

Sometimes you find that inheritance is not the best way of handling a situation and you need delegation instead, hence *Replace Inheritance with Delegation (295)* discusses how to make this change. Sometime's life is the other way round and you have to *Replace Delegation with Inheritance (297)*.

Pull Up Field

Two subclasses have the same field

Move the field to the superclass

Motivation

If subclasses are developed independently, or combined through refactoring, you often find that they duplicate features. In particular certain fields can be duplicates. Such fields sometimes have similar names, but not always. The only way to determine what is going on is to look at the fields and see how they are used by other methods. If they are being used in a similar way, then you can generalize them.

Doing this reduces duplication in two ways. Not just does it remove the duplicate data declaration, it also allows you to move behavior that uses the field from the subclasses to the superclass.

Mechanics

- ❍ Inspect all uses of the candidate fields to ensure they are used in the same way
- ❍ If the fields do not have the same name, rename the fields so that they have the name you want to use for the superclass field
- ❍ Compile and Test
- ❍ Create a new field in the superclass
  - ☞ If the fields are private, you will need to make the superclass field protected so that the subclasses can refer to it
- ❍ Delete the subclass fields
- ❍ Compile and Test
- ❍ Consider using *Self Encapsulate Field (184)* on the new field

Pull Up Method

You have methods with identical results on a subclass

Move them to the superclass

Motivation

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

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

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

A special case of this is where you have a subclass method that overrides a superclass method, yet does the same thing.

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

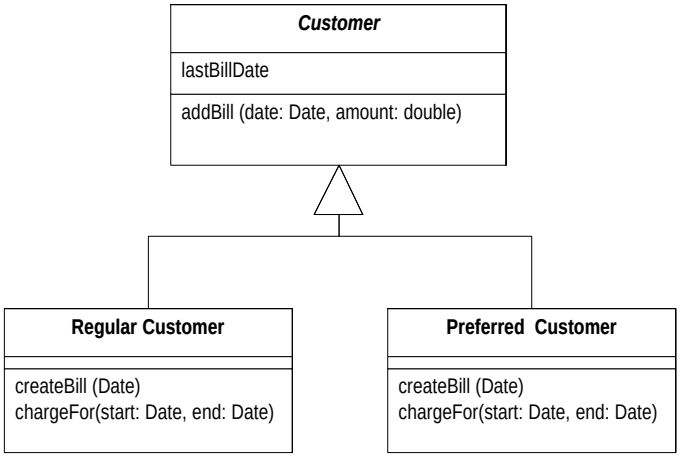
If you have two methods that are similar but not the same, see you may be able to *Form Template Method (289).*

Mechanics

- Inspect the methods to ensure they are identical
  - ☞ If they look like they do the same thing, but are not identical, use algorithm substitution on one of them to make them identical.
- If the methods have different signatures, then change the signatures to the one you want to use in the superclass
- Create a new method in the superclass, copy one of the methods' body to it, adjust and compile
  - ☞ If you are in a strongly typed language and the method calls another method that is present on both subclasses but not the superclass, then declare an abstract method on the superclass.
- Delete one subclass method
- Compile and test.
- Keep deleting subclass methods and testing until only the superclass method remains.

Example

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



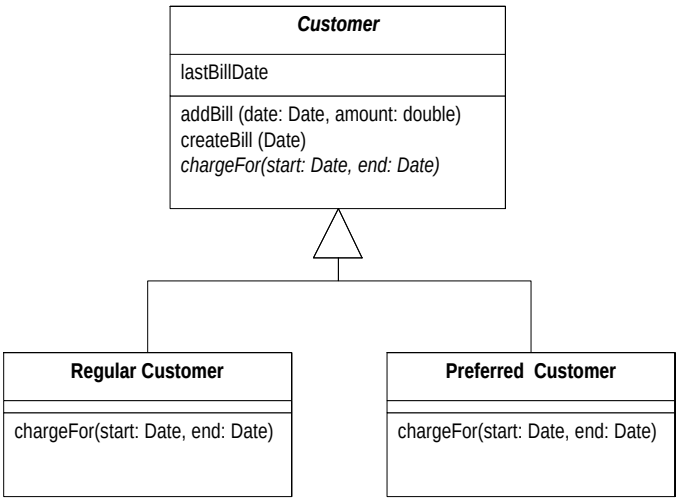
The `createBill` method is identical for each class

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

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

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

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



Pull up Constructor Body

You have constructors on subclasses with mostly identical bodies

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

Motivation

Constructors are tricky things, they aren't quite normal methods, so you are more restricted with what you can do with them than normal methods are.

If you see subclass methods with common behavior, your first thought is to extract that common behavior into a method, and pull it up into the superclass. With a constructor, however, that common behavior is often the construction. In this case you need a superclass constructor that is called by subclasses. In many cases this is the whole body of the constructor. You can't do *Pull Up Method (271)* here, because you can't inherit constructors (don't you just hate that?)

If this gets complex, you might want to consider *Replace Constructor with Factory Method (256).*

Mechanics

- Define a superclass constructor
- Move the common code at the beginning from the subclass to the superclass constructor
  - This may well be all the code
  - Try to move common code to the beginning of the constructor if you can.
- Call the superclass constructor as the first step in the subclass constructor.
  - If all the code was common, this will be the only line the subclass constructor
- Compile and test
  - If there is any common code later on use *Extract Method (114)* to factor out any more common code and *Pull Up Method (271)* to pull it up.

Example

Here is a manager and an employee.

```
class Employee...
  protected String _name;
  protected String _id;

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

```
  private int _grade;
```

The fields from employee should be set in a constructor for employee. So we define one, and make it protected to signal that subclasses should call it.

```
class Employee
  protected Employee (String name, String id) {
      _name = name;
      _id = id;
  }
```

Then we call it from the subclass

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

A variation on this occurs when there is also some common code later on. Say we have the following code.

```
class Employee...
  boolean isPriviliged() {..}
  void assignCar() {..}
class Manager...
  public Manager (String name, String id, int grade) {
      super (name, id);
      _grade = grade;
      if (isPriviliged()) assignCar(); //every subclass does this
  }
  boolean isPriviliged() {
      return _grade > 4;
  }
```

We can't move the assignCar behavior into the superclass constructor because it must be executed after grade has been assigned to the field. So we need to *Extract Method (114)* and *Pull Up Method (271).*

```
class Employee...
  void initialize() {
      if (isPriviliged()) assignCar();
  }
class Manager...
  public Manager (String name, String id, int grade) {
      super (name, id);
      _grade = grade;
      initialize();
  }
```

Push Down Method

Behavior on a superclass is only relevant for some of it's subclasses

Move it to the subclasses

Motivation

This is the opposite of *Pull Up Method (271)* I use it when I need to move behavior from a superclass to a specific subclass, usually because it only makes sense there. Obviously you often do this when you *Extract Subclass (277).*

Mechanics

- Declare method in all subclasses and copy the body into each subclass
  - ☞ You may need to declare fields as protected for the method to access them. Usually you do this if you intend to push the field down later. Otherwise use an accessor on the superclass. If this accessor was not public you will need to declare it as protected.
- Remove method from superclass
  - ☞ You may have to change callers to use the subclass in variable and parameter declarations.
  - ☞ If it makes sense to access the method through a superclass variable, all other subclasses implement the method, and the superclass is abstract, you can declare the method as abstract in the superclass.
- Compile and test
- Remove method from each subclass that does not need it
- Compile and test

Push Down Field

A field is only used by some subclasses

Move the field to the relevant subclasses

Motivation

Push Down Field (276) is the opposite of *Pull Up Field (270)*. You use it when you don't need a field in the superclass but only in a subclass

Mechanics

- Declare field in all subclasses
- Remove field from superclass
- Compile and test
- Remove field from all subclasses that don't need it
- Compile and test

Extract Subclass

A class has features that are only used by some instances

Create a subclass for that subset of features

Motivation

The main trigger for this refactoring is when you realize you have a class that has some behavior that is only used for some instances of the class and isn't used for others. Sometimes this is signalled by a type code, in which case you *Replace Type Code with Subclasses (224)* or *Replace Type Code with State/Strategy (227)*. But you don't have to have a type code to suggest the use for a subclass.

The main alternative to this refactoring is *Extract Component (166)*. This is a choice between delegation and inheritance. *Extract Subclass (277)* is usually simpler to do, but it has limitations. You can't change the class based behavior of an object once it is created, you can do this with *Extract Component (166)* simply by plugging in different components. You can also only use subclasses for one dimension of variability, so if you are using subclasses for one dimension you have to use delegation for the others.

Mechanics

- Define a new subclass of the source class
- Provide constructors for the new subclass
  - In the simple cases just copy the arguments of the superclass and call the superclass constructor with `super`.
  - If you want to hide the use of the subclass from clients you can use *Replace Constructor with Factory Method (256)*
- Find all calls to constructors of the superclass, if they need the subclass, replace with a call to the new constructor.
  - If the subclass constructor needs different arguments, use *Rename Method (234)* to change it. If some of the superclass's constructor parameters are no longer needed use *Rename Method (234)* on that too.
  - If the superclass can no longer be directly instantiated, declare it abstract.
- One by one use *Push Down Method (276)* and *Push Down Field (276)* to move features onto the subclass.
  - Unlike with *Extract Component (166)* it usually easier to do the methods first and the data last
  - When a public method is pushed, you may need to redefine a caller's variable or parameter type to call the new method. The compiler will catch these cases.
- Look for any field that designates information now indicated by the hierarchy (usually a boolean or type code). Eliminate it by using *Self Encapsulate Field (184)*, and then replacing the getter with polymorphic constant methods. All users of this field should be refactored with *Replace Switch with Polymorphism (147)*.
  - With any methods outside the class that use accessor, consider using *Move Method (160)* to move it into this class and then *Replace Switch with Polymorphism (147)*.
- Compile and test after each push down.

Example

I'll start with a job item class that determines prices for items of work at a local garage.

```
class JobItem ...
  public JobItem (int unitPrice, int quantity, boolean isLabor, Employee employee) {
      _unitPrice = unitPrice;
      _quantity = quantity;
      _isLabor = isLabor;
      _employee = employee;
  }
  public int getTotalPrice() {
      return getUnitPrice() * _quantity;
  }
  public int getUnitPrice(){
      return (_isLabor) ?
```

```
            _employee.getRate():
            _unitPrice;
    }
    public int getQuantity(){
        return _quantity;
    }
    public Employee getEmployee() {
        return _employee;
    }
    private int _unitPrice;
    private int _quantity;
    private Employee _employee;
    private boolean _isLabor;

class Employee...
    public Employee (int rate) {
        _rate = rate;
    }
    public int getRate() {
        return _rate;
    }
    private int _rate;
```

I'll extract a `LaborItem` subclass from this class. I begin by just creating the new class.

```
class LaborItem extends JobItem {}
```

The first thing we need is a constructor for the labor item since job item does not have a no-arg constructor. For the moment I'll just copy the signature of the parent constructor.

```
    public LaborItem (int unitPrice, int quantity, boolean isLabor, Employee employee) {
        super (unitPrice, quantity, isLabor, employee);
    }
```

This is enough to get the new subclass to compile. However the constructor is messy, some arguments are needed by the labor item, some not. We will deal with that in a moment.

The next step is to look for calls to the job item's constructor, and look for cases where the labor item's constructor should be called instead. So statements like

```
        JobItem j1 = new JobItem (0, 5, true, kent);
```

become

```
        JobItem j1 = new LaborItem (0, 5, true, kent);
```

Notice that at this stage I've not changed the type of the variable, only the type of the constructor. This is because I only want to use the new type where I have to. At this point we have no specific interface for the subclass, so I don't want to declare any variations yet.

Now is a good time to clean up the constructor parameter lists. I use *Rename Method (234)* on each of them. I do the superclass first. I create the new constructor, and make the old one private (the subclass still needs it).

```
class JobItem...
  protected JobItem (int unitPrice, int quantity, boolean isLabor, Employee employee) {
      _unitPrice = unitPrice;
      _quantity = quantity;
      _isLabor = isLabor;
      _employee = employee;
  }
  public JobItem (int unitPrice, int quantity) {
      _unitPrice = unitPrice;
      _quantity = quantity;
      _isLabor = false;
  }
```

Calls from outside now use the new constructor.

```
      JobItem j2 = new JobItem (10, 15);
```

Once I've compiled and tested I use *Rename Method (234)* on the subclass constructor.

```
class LaborItem
  public LaborItem (int quantity, Employee employee) {
      super (0, quantity, true, employee);
  }
```

For the moment, I'll still use the protected superclass constructor.

Now I can start pushing down the features of job item. I begin with the methods. I start with using *Push Down Method (276)* on `getEmployee`.

```
class LaborItem...
  public Employee getEmployee() {
      return _employee;
  }
class JobItem...
  protected Employee _employee;
```

Since the employee field will be pushed down later, I declare it as protected for the moment.

Once the employee field is protected I can clean up the constructors so it is only set in the subclass that it is going down to.

```
class JobItem...
  protected JobItem (int unitPrice, int quantity, boolean isLabor) {
      _unitPrice = unitPrice;
      _quantity = quantity;
      _isLabor = isLabor;
  }
class LaborItem ...
  public LaborItem (int quantity, Employee employee) {
      super (0, quantity, true);
       _employee = employee;
  }
```

The field `_isLabor` is used to indicate information that is now inherent in the hierarchy. So I can remove the field. The best way to do this is to first use *Self Encapsulate Field (184)* and then change the accessor to use a polymorphic constant method.

```
class JobItem...
  protected boolean isLabor() {
      return false;
  }

class LaborItem...
  protected boolean isLabor() {
      return true;
  }
```

Then I can get rid of the `isLabor` field.

Now I can look at users of the `isLabor` methods. These should be refactored with *Replace Switch with Polymorphism (147)*. So I take the method

```
class JobItem...
  public int getUnitPrice(){
      return (isLabor()) ?
           _employee.getRate():
           _unitPrice;
  }
```

and replace it with

```
class JobItem...
  public int getUnitPrice(){
      return _unitPrice;
  }
class LaborItem...
  public int getUnitPrice(){
      return  _employee.getRate();
```

```
}
```

Once a group of methods that use some data have been pushed down, I can use *Push Down Field (276)* on the data. If I can't do it because some method uses the data, that is a signal to further work on the methods, either by *Push Down Method (276)* or *Replace Switch with Polymorphism (147)*.

Since the unit price is only used by items that are non-labor (parts job items), I can now use *Extract Subclass (277)* to create a parts item class. When I've done that the job item class will be abstract.

Extract Superclass

You have two subclasses with similar features

Create a superclass and move the common features to the superclass

Motivation

Duplicate code is one of the principal bad things in systems. If you say things in multiple places, then when it comes time to change what you say, you have more things to change than you should.

One form of duplicate code is two classes that do similar things in the same way, or similar things in a different way. Objects provide a built in mechanism to simplify this situation with inheritance. However you often don't notice the commonalities until you have created the subclasses, in which case you need to create the inheritance structure later.

An alternative to this is *Extract Component (166)*. The choice is essentially that between inheritance and delegation. Inheritance is the simpler choice if the two classes share interface as well as behavior. If you made the wrong choice you can always *Replace Inheritance with Delegation (295)* later.

Mechanics

- Create a blank abstract superclass, make the original classes sub-

classes of this superclass

- One by one, use *Pull Up Field (270)*, *Pull Up Method (271)* and *Pull up Constructor Body (273)* to move common elements to the superclass.
  - ☞ It's usually easier to move the fields first
  - ☞ If you have subclass methods that have different signatures but the same purpose, use *Rename Method (234)* to get them to the same name and then *Pull Up Method (271)*.
  - ☞ If you have methods with the same signature but different bodies, declare the common signature as an abstract method on the superclass.
- Compile and test after each pull
- Examine the methods left on the subclasses. See if there are common parts, if so you can use *Extract Method (114)* followed by *Pull Up Method (271)* on the common parts. If the overall flow is similar you may be able to use *Form Template Method (289)*
- After pulling up all the common elements, check each client of the subclasses. If they only use the common interface you can change the required type to the superclass.

Example

For this case I'll use an employee and a department.

```
class Employee...
  public Employee (String name, String id, int annualCost) {
      _name = name;
      _id = id;
      _annualCost = annualCost;
  }
  public int getAnnualCost() {
      return _annualCost;
  }
  public String getId(){
      return _id;
  }
  public String getName() {
      return _name;
  }
  private String _name;
  private int _annualCost;
  private String _id;
import java.util.*;

public class Department...
  public Department (String name) {
      _name = name;
  }
  public int getTotalAnnualCost(){
      Enumeration e = getStaff();
```

```
        int result = 0;
        while (e.hasMoreElements()) {
            Employee each = (Employee) e.nextElement();
            result += each.getAnnualCost();
        }
        return result;
    }
    public int getHeadCount() {
        return _staff.size();
    }
    public Enumeration getStaff() {
        return _staff.elements();
    }
    public void addStaff(Employee arg) {
        _staff.addElement(arg);
    }
    public String getName() {
        return _name;
    }
    private String _name;
    private Vector _staff = new Vector();
```

There are a couple of areas of commonality here. Firstly both employees and departments have names. Secondly they both have annual costs, although the methods for calculating them is slightly different. I'll extract a superclass for both of these features. The first step is to create the new superclass and define the existing superclasses to be subclasses of this superclass.

```
abstract class Party {}
class Employee extends Party...
class Department extends Party...
```

Now I'll begin to pull features up to the superclass. It is usually easier to *Pull Up Field (270)* first.

```
class Party...
  protected String _name;
```

Then I can use *Pull Up Method (271)* on the getters.

```
class Party {

  public String getName() {
      return _name;
  }
```

I like to make the field private, to do this I need to *Pull up Constructor Body (273)* to assign the name.

```
class Party...
  protected Party (String name) {
      _name = name;
  }
  private String _name;

class Employee...
  public Employee (String name, String id, int annualCost) {
      super (name);
      _id = id;
      _annualCost = annualCost;
  }

class Department...
  public Department (String name) {
      super (name);
  }
```

The methods Department.getTotalAnnualCost and Employee.getAnnualCost, do carry out the same intention, hence they should be named the same. So I first use *Rename Method (234)* to get them to the same name.

```
class Department extends Party {
  public int getAnnualCost(){
      Enumeration e = getStaff();
      int result = 0;
      while (e.hasMoreElements()) {
          Employee each = (Employee) e.nextElement();
          result += each.getAnnualCost();
      }
      return result;
  }
```

Their bodies are still different, so I cannot use *Pull Up Method (271)*, however I can declare an abstract method on the superclass.

```
abstract public int getAnnualCost()
```

Once I've done the obvious changes I look at the clients of the two classes to see if I can change any of them to use the new superclass. Once client of these classes is the department class itself, which holds a collection of Employees. The `getAnnualCost` method only uses the annual cost method, which is now declared on the party.

```
class Department...
  public int getAnnualCost(){
      Enumeration e = getStaff();
      int result = 0;
```

```
        while (e.hasMoreElements()) {
            Party each = (Party) e.nextElement();
            result += each.getAnnualCost();
        }
        return result;
    }
```

This behavior suggests a new possibility. We could treat department and employee as a *Composite* [Gang of Four]. This would allow us to let a department include another department. This would be new functionality, so is not strictly a refactoring. If this were wanted we would do it by changing the name of the staff field to better represent the picture, making a corresponding change in the name `addStaff` and altering the parameter to be a party. The final change would be to make the `headCount` method recursive. We could do this by creating a headcount method for employee that just returns 1 and using *Substitute Algorithm (132)* on the department's headcount to sum the headcounts of it's components.

Extract Interface

Several clients use the same subset of a class's interface or two classes have part of their interfaces in common

Extract that subset into an interface

Motivation

Classes use each other in several ways. Often use of a class means ranging over the whole area of responsibilities of a class. Another case is where only a particular subset of a class's responsibilities are used by a group of clients. Or it may be that a class needs to work with any class that can handle certain requests.

In these cases it is often useful to make this subset of responsibilities a thing in its own right, so that it can be made clear in the use of the system. That way it is easier to see how the responsibilities divide. If new classes need to support the subset, it is easier to see exactly what fits in the subset.

In many object-oriented languages. this capability is supported by multiple inheritance. You create a class for each segment of behavior. and then combine them all in an implementation. Java has single inheritance, but allows you to state and implement this kind of requirement using interfaces. Interfaces have had a big influence on the way people design Java programs. Indeed even Smalltalkers think interfaces are a step forward!

There is some similarity between *Extract Superclass (282)* and *Extract Interface (286)*. *Extract Interface (286)* can only bring out common interface, not common code. So using *Extract Interface (286)* can lead to smelly duplicate code. You can reduce this by using *Extract Component (166)* to put the behavior into a component and delegating to it. If there is substantial common behavior *Extract Superclass (282)* is simpler, but you do only get to have one superclass.

Interfaces are good to use whenever a class has distinct roles in different situations. Use *Extract Interface (286)* for each role. Another useful case is when you want to describe the outbound interface of a class, i.e. what operations the class makes on its server. If you want to allow other kinds of server in the future, then all they need do is implement the interface.

Mechanics

- Create an empty interface
- Declare the common operations in the interface
- Declare the relevant class(es) as implementing the interface
- Adjust client type declarations to use the interface

Example

Consider a timesheet class that generates charges for employees. In order to do this it needs to know what the rate of the employee is and whether the employee has a special skill.

```
double charge(Employee emp, int days) {
    int base =  emp.getRate() * days;
    if (emp.hasSpecialSkill())
        return base * 1.05;
    else return base;
}
```

The employee has many other aspects to it than the charge rate and the special skill information, but those are only pieces that this application needs. I can highlight the fact that I only this subset by defining an interface for it.

```
interface Billable {
  public int getRate();
  public boolean hasSpecialSkill();
}
```

I then declare the employee as implementing the interface

```
class Employee implements Billable ...
```

With that done I can change the declaration of charge to show only this part of the employee's behavior is used.

```
double charge(Billable emp, int days) {
    int base =  emp.getRate() * days;
    if (emp.hasSpecialSkill())
        return base * 1.05;
    else return base;
}
```

At the moment the gain is a modest gain in documentability. Such a gain would not be worthwhile for one method, but if several classes used the billable interface on person, then that would be useful. The big gain appears when I want to bill computers too. In order to make them billable I know that all I have to do is implement the billable interface, and I can put computers on timesheets.

Collapse Hierarchy

A superclass and subclass are not very different

Merge them together

Motivation

If you have been working for a while with a class hierarchy it can easily get too tangled for its own good. Refactoring the hierarchy often involves pushing methods and fields up and down the hierarchy.

After you've done this you could well find you have a subclass that isn't adding any value, so you need to merge the classes together.

Mechanics

- Choose which class is going to be removed: the superclass or the subclasses
- Use *Pull Up Field (270)* and *Pull Up Method (271)* or *Push Down Method (276)* and *Push Down Field (276)* to move all the behavior and data of the removed class to the class it is being merged with.
- Compile and test with each move
- Adjust references to the removed class to use the merged class: this will affect variable declarations, parameter types, and constructors.
- Remove the empty class
- Compile and Test

Form Template Method

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

Give each step the same signature, so that the original methods become the same. Then you can pull them up.

Motivation

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

A common case is where we see two methods that seem to carry out broadly similar steps in the same sequence, but the steps are not the same. In this case we can move the sequence to the superclass, while allowing polymorphism to play its role in ensuring the different steps do their thing differently.

Mechanics

- Decompose the methods so that all the extracted methods are either identical or completely different.
- Use *Pull Up Method (271)* to pull the identical methods into the superclass.
- For the different methods use *Rename Method (234)* so the signatures for all the methods at each step are the same.
- Compile and test after each signature change.
- This will make the original methods the same, in that they all issue the same set of method calls, but the subclasses handle the calls differently.
- Use *Pull Up Method (271)* on one of the original methods. Define the signatures as abstract methods on the superclass
- Compile and test
- Remove the other methods, compile and test after each one.

Example

For this example, I'll finish off where I left the example in Chapter 1.

At the end of Chapter 1 I had a customer class with two methods for printing statements. The statement method prints statements in ascii.

```
public String statement() {
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + name() + "\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        //show figures for each rental
        result += "\t" + each.tape().movie().name()+ "\t" +
                    String.valueOf(each.charge()) + "\n";
    }
    //add footer lines
    result +=  "Amount owed is " + String.valueOf(charge()) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints()) +
                " frequent renter points";
    return result;
}
```

While the `htmlStatement` does them in HTML.

```
public String htmlStatement() {
    Enumeration rentals = _rentals.elements();
    String result = "<H1>Rentals for <EM>" + name() + "</EM></H1><P>\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
```

```
            //show figures for each rental
            result += each.tape().movie().name()+ ": " +
                        String.valueOf(each.charge()) + "<BR>\n";
        }
        //add footer lines
        result +=  "<P>You owe <EM>" + String.valueOf(charge()) + "</EM><P>\n";
        result += "On this rental you earned <EM>" +
              String.valueOf(frequentRenterPoints()) +
              "</EM> frequent renter points<P>";
        return result;
    }
```

Before I can use *Form Template Method (289)* I need to arrange things so that the two methods are subclasses of some common superclass. I'll do this by using the strategy pattern [Gang of Four] to create a separate strategy hierarchy for printing the statements.

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

Now I use *Move Method (160)* to move the two statement methods over to the subclasses.

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

class TextStatement {
    public String value(Customer customer) {
        Enumeration rentals = customer.rentals();
        String result = "Rental Record for " + customer.name() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            //show figures for each rental
            result += "\t" + each.tape().movie().name()+ "\t" +
                        String.valueOf(each.charge()) + "\n";
        }
        //add footer lines
        result +=  "Amount owed is " +
          String.valueOf(customer.charge()) + "\n";
        result += "You earned " +
          String.valueOf(customer.frequentRenterPoints()) +
          " frequent renter points";
        return result;
    }
class HtmlStatement {
```

```
public String value(Customer customer) {
    Enumeration rentals = customer.rentals();
    String result = "<H1>Rentals for <EM>" + name() + "</EM></H1><P>\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        //show figures for each rental
        result += each.tape().movie().name()+ ": " +
                    String.valueOf(each.charge()) + "<BR>\n";
    }
    //add footer lines
    result +=  "<P>You owe <EM>" + String.valueOf(customer.charge()) + "</EM><P>\n";
    result += "On this rental you earned <EM>" +
          String.valueOf(customer.frequentRenterPoints()) +
          "</EM> frequent renter points<P>";
    return result;
}
```

As I moved them I renamed them to better fit the strategy. I named them the same as the difference between the two now lies in the class rather than the method.

Now I have two similar methods on subclasses I can start to *Form Template Method (289)*. The key to this refactoring is to separate the varying code from the similar code by using *Extract Method (114)* to extract the pieces which are different between the two methods. Each time I extract I create methods with different bodies but the same signature.

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

```
class TextStatement...
    public String value(Customer customer) {
        Enumeration rentals = customer.rentals();
        String result = headerString(customer);
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            //show figures for each rental
            result += "\t" + each.tape().movie().name()+ "\t" +
                        String.valueOf(each.charge()) + "\n";
        }
        //add footer lines
        result +=  "Amount owed is " +
          String.valueOf(customer.charge()) + "\n";
        result += "You earned " +
          String.valueOf(customer.frequentRenterPoints()) +
          " frequent renter points";
```

```
        return result;
    }
  protected String headerString (Customer customer) {
      return "Rental Record for " + customer.name() + "\n";
  }

class HtmlStatement...
    public String value(Customer customer) {
        Enumeration rentals = customer.rentals();
        String result = headerString(customer);
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            //show figures for each rental
            result += each.tape().movie().name()+ ": " +
                        String.valueOf(each.charge()) + "<BR>\n";
        }
        //add footer lines
        result +=  "<P>You owe <EM>" + String.valueOf(customer.charge()) + "</EM><P>\n";
        result += "On this rental you earned <EM>" +
              String.valueOf(customer.frequentRenterPoints()) +
              "</EM> frequent renter points<P>";
        return result;
    }
  protected String headerString (Customer customer) {
      return "<H1>Rentals for <EM>" + customer.name() + "</EM></H1><P>\n";
  }
```

I compile and test, and then continue with the other different elements. I did it one at a time, but I'll show the result.

```
class TextStatement …
    public String value(Customer customer) {
        Enumeration rentals = customer.rentals();
        String result = headerString(customer);
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += eachRentalString (each);
        }
        result +=  footerString(customer);
        return result;
    }
  protected String eachRentalString (Rental rental) {
      return "\t" + rental.tape().movie().name()+ "\t" + rental.charge() + "\n";
  }

  protected String footerString (Customer customer) {
      return      "Amount owed is " + customer.charge() + "\n" +
          "You earned " + customer.frequentRenterPoints() + " frequent renter points";
  }
class HtmlStatement…
```

```
public String value(Customer customer) {
      Enumeration rentals = customer.rentals();
      String result = headerString(customer);
      while (rentals.hasMoreElements()) {
          Rental each = (Rental) rentals.nextElement();
          result += eachRentalString (each);
      }
      result +=  footerString(customer);
      return result;
}
protected String eachRentalString (Rental rental) {
    return rental.tape().movie().name() + ": " + rental.charge() + "<BR>\n";
}
protected String footerString (Customer customer) {
    return "<P>You owe <EM>" + customer.charge() + "</EM><P>\n" +
        "On this rental you earned <EM>" + customer.frequentRenterPoints() +
        "</EM> frequent renter points<P>";
}
```

Once these changes are done, the two value methods look remarkably similar. So I use *Pull Up Method (271)* on one of them, picking the text version at random. When I pull up I need to declare the subclass methods as abstract.

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

  abstract protected String headerString (Customer customer);
  abstract protected String eachRentalString (Rental rental);
  abstract protected String footerString (Customer customer);
```

I remove the value method from text statement, compile and test. When that works I remove the value method from the html statement, compile and test again.

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

Replace Inheritance with Delegation

You have implemented a class with inheritance, and now regret it

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

Motivation

Inheritance is a wonderful thing, but sometimes it isn't what you want. Often you start inheriting from a class, but then find that many of the superclass operations aren't really true of the subclass. In this case you have an interface that's not a true reflection of what the class does. Or you may find that you are inheriting a whole load of data which is not appropriate for the subclass. Or you may find that there are protected superclass methods that don't make much sense with the subclass.

You can live with it and just use convention to say that although it is a subclass, it's only using part of the superclass function. But that results the code saying one thing and your intention being something else, the essential confusion that you should remove.

By using delegation instead, you make it clear that you are making only partial use of the delegated class. You control what aspects of the interface to take, and what to ignore. The cost is extra delegating methods that are boring to write, but are too simple to go wrong.

Mechanics

- Create a field in the subclass that refers to an instance of the superclass. Initialize it to `this`.
- For each method defined in the subclass, change it to use the delegate field. Compile and test after changing each method.
  - ☞ You won't be able to replace any methods that use super with a method that is defined on the subclass or they may get into an infinite recurse. These methods can only be replaced after you have broken the inheritance.
- Remove the subclass declaration and replace the delegate assignment with an assignment to a new object.
- For each superclass method used by a client, add a simple delegating method.

- Compile and test

Example

One of the classic examples of inappropriate inheritance is making a stack a subclass of vector. Of course java does this in its utilities (naughty boys!) but in this case I'll use a simplified form of stack.

```
class MyStack extends Vector {

    public void push(Object element) {
        insertElementAt(element,0);
    }

    public Object pop() {
        Object result = firstElement();
        removeElementAt(0);
        return result;
    }
}
```

Looking at the users of the class I realize that clients only do four things with stack: push, pop, size, and isEmpty. The latter two are inherited from Vector.

I begin the delegation by creating a field for the delegated vector. I link this to myself, so that I can mix delegation and inheritance.

```
private Vector _vector = this;
```

Now I start replacing methods to get them to use the delegation. I begin with push.

```
    public void push(Object element) {
      _vector.insertElementAt(element,0);
    }
```

I can compile and test here, everything will still work. Now pop.

```
    public Object pop() {
      Object result = _vector.firstElement();
      _vector.removeElementAt(0);
      return result;
    }
```

Once I've done these subclass methods, I now need to break the link to the superclass.

```
class MyStack ~~extends Vector~~ //tbd check strike thru on extends Vector
```

```
private Vector _vector = new Vector();
```

I then add simple delegating methods for those superclass methods that are used by clients.

```
  public int size() {
    return _vector.size();
}

public boolean isEmpty() {
    return _vector.isEmpty();
}
```

Now I can compile and test. If I forgot to add a delegating method, the compilation will tell me.

Replace Delegation with Inheritance

You're using delegation and would prefer inheritance

Make the delegating class a subclass of the delegate

Motivation

This is the flip side of *Replace Delegation with Inheritance (297)*. If you find yourself using all the method of the delegate, and sick of writing all those simple delegating methods, then you can switch back to inheritance pretty easily.

There are a couple of caveats to bear in mind. If you aren't using all the methods of class you are delegating too, then you shouldn't do this, because a subclass should always follow the interface of the superclass. If the delegating methods are tiresome, you should consider providing an accessor for the delegate so clients can call it directly. This will reveal the delegate to clients, of course, but sometimes it's best to remove the middle man.

Another caveat is if the delegate is shared by more than one delegate object and has mutable data. In this case you can't replace it with

inheritance because you'll no longer share the data. Data sharing is a responsibility that cannot be transferred back to inheritance.

Mechanics

- Make the delegating object a subclass of the delegate.
- Compile
  - ☞ You may get some method clashes at this point; where methods have the same name, but vary on return type, exceptions, or visibility. Use *Rename Method (234)* to fix these.
- Set the delegate field to be the object itself
- Remove the simple delegation methods
- Compile and test
- Replace all other delegations with calls to the object itself
- Remove the delegate field.

Example

I'll start with a simple employee delegating to a simple person.

```
class Employee {
  Person _person = new Person();

  public String getName() {
      return _person.getName();
  }
  public void setName(String arg) {
       _person.setName(arg);
  }
  public String toString () {
      return "Emp: " + _person.getLastName();
  }
}


class Person {
  String _name;

  public String getName() {
      return _name;
  }
  public void setName(String arg) {
      _name = arg;
  }
  public String getLastName() {
      return _name.substring(_name.lastIndexOf(' ')+1);
  }
}
```

The first step is just to declare the subclass

```
class Employee extends Person
```

Compiling at this point alerts me to any method clashes. These occur if methods with the name have different return types, or throw different exceptions. Any such problems need to be fixed with *Rename Method (234)*. This simple example is free of such encumbrances.

The next step is to make the delegate field refer to the object itself. Then I must remove all simple delegation methods such as `getName` and `setName`. If I leave any in, I will get a stack overflow error due to infinite recursion. In this case this means removing `getName` and `setName` from Employee.

Once I've got the class working I can then change those methods that make use of the delegate methods, switching them instead to use calls directly.

```
public String toString () {
    return "Emp: " + getLastName();
}
```

Once I've got rid of all those, I can get rid of the _person field.

Chapter 12: Others

Okay, I'll admit it. I couldn't think of where to put these.

Often classes expect some condition to be true for all instances of the class, but this invariant doesn't make it into the code. If you use *Add Invariant (301)* to put it there, you embed that design knowledge into the code, making it clearer, which is the purpose of refactoring.

Sometimes you may have to start with a program that isn't object-oriented at all. In which case your first step is *Replace Program With Class (305).*

Add Invariant

The class has an unstated assumption of an invariant

Create a method that checks the invariant. Call it as an assertion after any modifier

Motivation

Invariants (or constraint rules) are conditions about an object that should always be true. An order processing system might have a rule that says that priority orders can only be made by important customers in good credit standing. Such a rule, often referred to as a Business Rule, captures some important knowledge about the domain, and should be represented clearly in the appropriate classes. Often such rules are scattered about the system, typically on various modification actions. You might find some checks present during construction of an order, some during marking the order as priority, some on reassignment of an order between customers.

You can make things more explicit by writing a method that specifically checks for the invariant. A boolean method `isValid` can contain the code to make the checks and be called from every point that does some checking. In particular you want to move rule checking code away from interface classes into domain classes in these sorts of cases, in order to remove duplication of checks.

You may well need more than one kind of validity check, and you may find things are better if you allow a certain degree of invalidity in your objects. You may have a series of rules about taking orders, but an order entry person may wish to put some information into the system at one time with the intention of doing more work later. In this case you may want more than two levels of validity, with some operations requiring the stronger level.

Mechanics

- Create an `isValid` method that does all the constraint checks on the class
- Find all situations where a modifier checks some part of the constraint rule. Replace this check with a call to `isValid`.
- Find all situations where an interface class checks some part of a domain class's constraint rules, and replace by calls to `isValid`.
  - ☞ In any situation where `isValid` fails, the class should not be left in an invalid state. The calling method that makes the modification to the class should be responsible for cleaning up the domain class to return it to a legal state, or throw an appropriate checked exception.
- Compile and test after each move of checking code. Make sure the tests probe for a failure at each use of the constraint checks.
- If possible, apply *Replace Exception With Test (264)* to allow up-front tests of the invariant.
- Examine other modifiers and ensure they also check the invariant.

Example

Here is some code that makes creates a transfer object to transfer between accounts

```
int transferAmount = 450;
if ((cust1.getBalance() > transferAmount) &&
            !cust1.isStopped() && !cust2.isStopped())
    transfer = new Transfer(cust1, cust2, transferAmount);
```

The client code is checking that the conditions on the transfer are legal before doing the transfer. The transfer should be responsible for this kind of check

```
class transfer…
  public Transfer (CustomerAccount from, CustomerAccount to, int amount)
  throws InvalidTransferException {
      _from = from;
      _to = to;
      _amount = amount;
      if (!isValid()) throw new InvalidTransferException();
  }

  public boolean isValid() {
      return ((_from.getBalance() > _amount) &&
          !_from.isStopped() &&
          !_to.isStopped());
  }

  private CustomerAccount _from;
  private CustomerAccount _to;
  private int _amount;
```

The client now creates transfers and checks for a problem

```
      try {
          transferAmount = 450;
          transfer = new Transfer(cust1, cust2, transferAmount);
      } catch (InvalidTransferException e) {
          //handle error
      }
```

This style of checking is done after the transfer is created: do the work, then see if it worked. An alternative is to check up-front, look to see if things will be valid and if so go ahead. We can use *Replace Exception With Test (264)*. With this the caller is expected to do something like

```
      int transferAmount = 450;
      if (Transfer.wouldBeValid(cust1, cust2, transferAmount))
          transfer = new Transfer (cust1, cust2, transferAmount);
```

To do this the transfer class needs to provide a helper routine, to test for validity.

```
  public static boolean wouldBeValid
          (CustomerAccount from, CustomerAccount to, int amount)
  {
      return (new Transfer(from, to, amount, true).isValid());
  }
```

```
private Transfer (CustomerAccount from, CustomerAccount to,
                  int amount, boolean specialAccess)
{
    _from = from;
    _to = to;
    _amount = amount;
}
```

The implementation here is a little curious. The best way to do the check with minimum redundancy is to create the transfer and ask it if it is valid. But you shouldn't allow clients to create invalid transfers. One way around this is to set up a basic private constructor with a dummy argument that is there only for the method dispatching. This dummy argument isn't ever referenced by the constructor's body, so it can take any value; but it's presence forces the private constructor to be called rather than the public one. The validity checker can use this method, as can the public constructor.

```
public  Transfer  (CustomerAccount from, CustomerAccount to, int amount) {
    this (from, to, amount, true);
    Assert.isTrue (isValid());
}
```

(You can also use *Replace Constructor with Factory Method (256)* and use the factory publicly and the constructor privately.)

The check in the public constructor is now an assertion rather than a checked exception. This reflects that calling for check is the now the client's job.

The `wouldBeValid` function could not call the public constructor and catch the exception. This is because assertions would be removed in production code and thus should not be used for condition checking.

I prefer the up-front style if it is at all possible. It makes for a cleaner interface and less reliance on exceptions. I like exceptions to signal something has gone wrong which could not have been predicted in advance.

Replace Program With Class

You have a program in a traditional programming environment.
How do you begin to make this object oriented?

Make a class for the program.

Motivation

In the ideal world of software engineering textbooks, we always start from a blank sheet of paper. In the real world, of course, there are legacy systems. They get involved in various ways, the way I'm thinking about here is when we are replacing some part of a legacy system. In this case we may want to take parts of the legacy system, such as particular algorithms, and duplicate them in a new system.

Can refactoring help us with this task? So far the evidence is limited. Certainly you can take a procedural algorithm, recode it in an OO language in a simple minded way, and then refactor it into good shape. The question is whether this is quicker than starting from scratch. The biggest limitation is when the legacy algorithm uses a lot of global data that needs to be simulated somehow in the object world. Often the hardest part of this effort is just getting the simple recoded version to work – debugging these things is tough.

Mechanics

- Make a class for the program that you want to move. Make each procedure a method of that class
- Make global variables fields of that class
- Use *Replace Record with Data Class (218)* on any records
- Treat the class as a singleton
  - ☞ That way it is easier to reinitialize the class during testing.