Catalog of Refactorings

Finding References

Many of the refactoring steps below 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.

If your environment does not have a multi-file search and replace then consider using a separate tool. Most platforms have a perl implementation, and it is easy to set perl up to do a multi-file search (TK add perl script here). On Unix, of course, you can just use grep.

In a strongly typed language you can let the compiler do the hunting for you in many cases. 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 overrridden 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 it is better to use the text search alternative.

Smalltalk does not use text files to hold its code, instead it 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. You do have to be careful about methods declared on more than one class. A similar item will get you all references to an instance variable.

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

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.

- ❍ 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 situtation where isValid() fails, the class should not be left in an invalid state. The calling method that makes the moficiation 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.

The principle of Design by Contract [Meyer] says that the invariant should be checked as part of the post-condition of any modifier method. To this create a method called checkInvariant(). This method should do the following tk checkCode

```
Protected void checkInvariant() {
  Assert(isValid());
}

Protected void assert(boolean test) {
  If (! test) throw new AssertionFailedException();
}
```

The AssertionFailedException should be an unchecked exception (it should never appear in production code). CheckInvariant() acts as a debugging aid when developing a class. For production code the above implementation should be replaced by a simple return. It does not replace the actual tests of isValid() that should be made by any method that might alter the invariant.

- ❍ Create a checkInvariant() method that tests isValid() and throws an exception if the result is false. Put a call to checkInvariant() at the end of all public modifier methods.

Add Superclass

You have some classes with similar behavior

Create a superclass of these classes and move common state and behavior to this class

T.K. Make this a summary of some of the other steps

Change Method Signature

A method could be renamed to make its intention clearer

Change the signature of the method

An important part of the code style I am advocating here is using method names to communicate their purpose. Thus it is extremely important that methods are well named. Poor naming is worse than inlining. Therefore you should expect to change the names of methods frequently as you come up with better names. Don't be tempted to not bother. Just 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. Clarity is the most important thing to strive for.

❍ Check to see if this method signature is implemented by a superclass or subclass. If so carry out these steps for each implementation.
❍ Declare a new method with the signature you want to use
❍ Copy the body of the old method over to the new
❍ Delete the old method
  ☞ In Smalltalk use 'senders' to find the old references first, then change them, then delete the old method.
❍ Find all references to the old method signature and change them to refer to the new one
  ☞ In a strongly typed language you can do this by compiling. Otherwise use a text search to the old references
❍ Compile and test.

Check Invariant on Modifiers

A class has an invariant that could be changed by a modifier method

Ensure the invariant is checked by the modifier. If it is the method's responsibility then check and throw an exception if it is false. If it is not the method's responsibility then check it in an assertion.

If the class has an invariant, then this invariant should be checked by each modifier to ensure that it is still correct. If the class does not have an invariant, but should have, then *add the invariant* first.

Whenever you check the invariant on a modifier you need to consider whether it is really the responsibility of the modifier to check the invariant. Ask yourself the question "should my caller have already checked this or is my caller expecting me to check this and notify it of problems?" Consider the common (but unrealistic example) of a bank account that is not supposed to fall below zero. You may have a client that first checks the balance, and only if enough cash is present does it remove money. In this case the caller has the responsibility of the checking. In another case the caller may just try to remove the money and expect the withdraw operation to complain if there is not enough money.

If it is the caller's responsibility to make the check, then the mofier should still do the check, but only as part of an assertion. It only does the check as a debugging aid and the program should not expect the check to be made. In this case use the checkInvariant() method described in the *add invariant* refactoring.

If it is the modifier's responsibility to do the check then it should either be also responsible for cleaning things up if the check fails, or it should throw an exception. In our postive balance bank account case the withdraw operation might withdraw as much as it can and return the amount withdrawn to the caller. That would be cleaning up as best it can. Otherwise it must throw a checked exception (which involves *changing the method's signature*).

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

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.

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. Tk check code

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();
}
```

- ❍ 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*.

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.

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.

Consolidate Conditional Expression

You have a sequence of conditional tests looking with the same outcome

Combine them into a single conditional epression, and extract it

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

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

You can consolidate this into a single logical expression by or-ing the conditions together.

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

This doesn't mean you should always do it. Certainly there are times when the series of tests reads more clearly. You have to judge by the situation which is clearer.

When I do this I nearly always then take the conditions and extract them into a new method.

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

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

If the routine you are looking only tests the condition then you can turn the routine into a single return statement using the tertiary operator

```
Money pay() {
  return (payable()) ?
     normalPayAmount():
     Money.dollars(0);
}
```

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

Consolidate duplicate conditional fragments

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

Move it outside of the expression

This is a very simple refactoring. You have code of the form

```
If (condition) {
  Do something;
  Statement1;
}
else {
  Do something else;
  Statement1;
}
```

This can be refactored to put the bit that varies outside the conditional

```
If (condition) {
  Do something;
}
else {
  Do something else;
}
statement1;
```

If the common code is at the beginning of each conditional block, move it to before the conditional statement; if it is after the block, move it to after the statement

If the common code is not at the beginning or end, then life is a bit more complicated. The first to look at is whether any of the before or after code changes anything. See if you can move the common code within each leg to the beginning or the end of its conditional leg. Once that is done then you can move it as above.

If your common code is more than a single statement then you should extract that code into a method.

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 decorator of the original

If you can't modify the source code, but you want to add some features to a class, then your simplest option is to *Create Foreign Methods*. 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. 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 decorator of the original. Tk discuss how.

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. tk pattern name for this. With the decorator you need to create the original and place it in the tk name instance variable. To get at the original you need do nothing when using subclassing, since the subclass will substitute cleanly for the original. For the decorator you will need an appropriate method to get at the original. tk show this.

- ❍ Create an extension class either as a subclass or decorator of the original
- ❍ Add converting constructors tk pattern name 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.

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

In an ideal world you should never have to do this. Such an ideal world would allow you to always modify the source of the server class. You may be prevented because you don't have the source, there are change control problems, or because of code ownership policy. Foreign Methods are a work-around for these cases.

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.

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 an Extension* instead.

- ❍ Create a method in the client class that does what you need
- ❍ 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

Decompose a Conditional Expression

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

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

A lot of complicated expressions are conditional. When I'm looking at a complicated conditional the first thing I try is to extract each logical part of the conditional: the condition, the then part and the else part.

- ❍ 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 the conditionals.

Decompose a Method

You have a long, complex method that is difficult to understand.

Extract methods from logical blocks of code.

The longer a procedure is, the more difficult it is to understand. Yet there is overhead in switching between procedures. Each time you see a sub procedure call you have to switch context to look and see what the sub-procedure does. As you do this you have to scroll your editor and you lose the place in the original procedure. Also in many languages calling a procedure has an overhead because the new record is needed on the stack.

This has influenced the heuristic for how long you make a subroutine. In my FORTRAN days the heuristic was one page of print out, since we tended to look at code on paper. In my Unix days they were one screenful in the emacs buffer.

Object-oriented code changes the heuristics in a number of ways. Object-oriented languages minimize the overhead of method calls to negligible proportions, indeed you many compilers eliminate them completely with inlineing. Methods in object languages are thus much shorter, Smalltalk often averages at 3 lines of code for a method, C^{++} at around ten.

The problems of context switching for the reader are mitigated by two factors. Firstly modern development environments have browsers where it is easy to open two browsers on the same class at once. This way you can usually fit two methods on the screen at once. Secondly by working to think of good names for methods, you can often remove the need to look at the other method at all. Good naming is a skill, a really important skill that is worth persevering to acquire.

The net effect of this is that you should be much more aggressive about decomposing methods. A heuristic I follow is whenever I feel the need to comment something, I instead write an *Intention Revealing Method*. I may do this on a group of lines, or on as small as a single line of code. I will do this even if the method call is no shorter than the code it replaces, providing the method explains the purpose of the code.

- ❍ Scan a method for chunks of code that seem to make logical units
  - ☞ Look for any lines of code offset by spaces and commented at the top. The comment tells you what the name of the extracted method should be.
  - ☞ Decompose Conditional Expressions
  - ☞ Look for any bit of code that is commented, even single or partial lines. Replace the code with an Intention Revealing Method using the comment to get the name
- ❍ Extract a method from each chunk

Eliminate Duplicate Subclass Behavior

A subclass overrides a method but does the same thing

Remove the subclass method.

You would think this doesn't happen, but sometimes it does. Sometimes it is not obvious that the subclass does the same thing. Sometimes this occurs as a result of combining classes into a hierarchy. Certainly whenever you change a subclass hierarchy you should check any subclass methods for any odd behavior.

Encapsulate Downcast

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

Move to downcast to within the method

Downcasting is one of the most annoying things you have to do in Java. It is annoying because it feels uneccessary, you are telling the compiler something it ought to be able to figure out itself. But since the figuring out is rather complicated, and since we don't have template methods, you often have to do it yourself.

Whenever you downcast after a single method call, consider rewriting the method to include the downcast. This is a common case with container classes. I have a Site class with a Vector of Reading objects. To get the last reading in the vector I could go

```
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();
  }
```

- ❍ Look for cases where you have to downcast the result from calling a method
- ❍ Move the downcast into the method

Altering a mehod to return a subclass does alter the signature of the method, but will not break exisiting 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. (tk check conseuqnces of this.)

Hide Method

A Method is not used by any other class

Make the Method private

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.

Generalize Field

Two subclasses have the same field

Move the field to the superclass

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.

- ❍ Inspect all uses of the candidate fields to ensure they are used in the same way
  - ☞ Use references in Smalltalk, or a text search to find each use of the field.
- ❍ 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
- ❍ 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 self-encapsulating the new field

Generalize Method

You have methods with identical results on a subclass

Move them to the superclass

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 paramaterized into a single superclass 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.

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. See the refactoring Unify Subclass Methods for more on this theme.

- ❍ Inspect the methods to ensure they do the same thing
- ❍ 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 superclasses but not the superclass, then declare an abstract method on the superclass.
- ❍ Delete the subclass methods
- ❍ Compile and test.

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

In general I prefer short methods which are named after their intention. This leads to clearly written code. However if a method's body says what it does, just as well as the name, then you can safely remove the method and inline the code when it is used.

Extract a 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.

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 (see *Decompose a Method*). I then turn that fragment of code into its own method.

In the simplest case this merely is a cut and paste of the fragment into a new method. The complication comes from locally scoped variables: temps and parameters.

- ❍ 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 usually local variables and parameters to the methods
- ❍ 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 do a Temporary Variable Split or a Parameter Copy and try again. You can eliminate temporary variables by replacing a temp with a query.
  - ☞ If you are returning a value to deal with a modified temp you have three different cases of how to deal with it. If the temp is not modified from its initial value before the new method gets its hands on it, and the initialization is an obvious one (such as 0) then you can assign the returned value to the temp (e.g. temp = extractedMethod()). However if there is some use of the variable beforehand, or the initialization is not an obvious one, then you need to see if your extracted method reads from the temp in its body. If it does you must pass in the temp as a parameter and return the result by assignment (e.g. temp = extractedMethod(temp). If you never read the temp, just update it with an appending assignment (e.g. temp = temp + extractedMethod())
- ❍ 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.

Refactory will do this as a menu item.

Often I extract a method simply to make its purpose clearer. (tk ref to utils chapter at the beginning). While some might consider it pointless to do so for such a small method, I belive the key is not the size of what is being extracted, but 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.

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

---

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.

- ❍ 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.
- ❍ Create a field in the target class
- ❍ Add Getting and Setting Methods for the target field
- ❍ 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.

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

---

Moving methods is the bread and butter of refactoring. I do this because classes are collaborating too much and are too highly coupled. By moving methods around to reduce this I can make the classes simpler and they end up being a more crisp implementation of a set of responsibilities.

Deciding which methods to move is a process in itself. 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. I choose according to instinct, after all I can always change it again later.

- ❍ Examine all the methods that the source method calls that are defined on the source class. Consider whether they should be moved too.
  - ☞ In Smalltalk, use the messages menu item.
  - ☞ If a called method is only used by the method you are about to move, then you might as well move it too. If the called method is used by other methods consider moving them. 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

- ❍ Copy the code from the source method to the target. Adjust the method to make it work in its new home and compile.
  - ☞ 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.
- ❍ 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 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.
- ❍ Decide whether to remove the source method, or turn it into a Delegating Method.
  - ☞ Turning the source into a delegating method is easier if you have many references. Often you can turn it into a delegating method first, compile and test, and then remove the references as a later step.
- ❍ 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.
- ❍ If you delegate, then all you need to do is replace the body of the source method with a call to the target method
- ❍ Compile and test.

Paramaterize a 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

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 *= raise Amount;
}
```

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

A less obvious case is

```
  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);
  }
```

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

- ❍ Create a parameterized method that can be substitued for each repetitive fragment
- ❍ Replace one fragment with a call to the new method
- ❍ Compile and test
- ❍ Repeat for all the fragments.

Preserve Whole Object

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

Send the whole object instead

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. Preserving the Whole Object 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 moving the method instead.

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.

- ❍ Create a new parameter for the whole object
- ❍ Adjust the calling method to pass in the whole object.
- ❍ Determine which parameters should be obtained from the whole object
- ❍ For each parameter, find the references to the parameter in the called method body, and replace by invoking the appropriate method on the whole object parameter.
  - ☞ To make the steps smaller, you can test after each parameter is replaced
- ❍ Delete the parameters that are now obtained from the whole object.
- ❍ Compile and Test
- ❍ 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

Reduce Parameter List

You have a method with many parameters

Reduce the amount of parameters by Replacing Parameters with Methods and Preserving Whole Objects

In my early programming days I was always taught to pass everything a routine needed in with the parameter. This was understandable because the alternative was global data, and global data is trouble. Objects change this situation because if you don't have something you need, you can always ask another object to get it for you. Thus with objects you don't pass in everything the method needs, enough so that the method can get to everything it needs. A lot of what a method needs is available on method's host class. Thus in object-oriented programs parameter lists tend to be much smaller than on traditional programs.

Remove Assignments to Parameters

The code assigns to a parameter

Use a temporary variable instead.

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 problem with assigning to a parameter is that the semantics of doing so depend on whether the parameter is passed by reference or passed by value. Languages differ as to which of these they do, and even a single language may do one for some kind of variables and something else for others. As a result it's very hard for a human to figure out what happens when you do this. By using a temporary variable instead, you make the whole picture much more clear.

Sometimes, of course, a method will do this to indicate changes. If only one parameter is altered, then that really ought to be the return value of the method. If there is more than one parameter involved, then the issue is more complicated. The values should be fields in some object and you should then send in the necessary objects and send commands to these objects to change their values.

- ❍ 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 the temporary variable.
- ❍ 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.
- ❍ If only one parameter is used in this way, and the method does not currently return a value, then recast the method to pass the new reference back as its return value.
- ❍ If more than one parameter is involved then (T.K further discussion here)

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

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. You can get around this in several ways.

The most obvious way is to use a break (if your programming language has the construct) or a go to (if you don't have a break). That way you get

```
while false
  if (condition)
     do something
     break
  next step of loop
```

Another alternative is to extract the loop into its own method and use a return to act as a break

```
while false
  if (condition)
     do something
     return
  next step of loop
```

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.

Remove Nested Conditionals

You have nested conditionals that are hard to follow

Unwind the nested conditionals into a single path using Guard Clauses

Remove Setting Method

A field should be set at creation time and never altered

Remove any Getting Method for that field

Providing a Setting Method indicates that that field may be changed. If you don't want that to happen then don't provide the indication and anyone reading your code will not get confused.

This situation often occurs with people who blindly use Indirect Variable Access. Such people then use Setting Methods 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.

The most tempting time to do this is when the initialization involves some computation on a provided value

```
Class Foo {
  foo (Object value) {
     setField(value);
  }

  setField (Object value) {
     _field = aMethod(value);
  }
```

This makes a certain design sense, particularly if there is more than one constructor. But while the basic idea of putting the manipulations in one place is sensible, the approach suffers from poor naming. I would prefer

```
Class Foo {
  Foo (Object value) {
    _field = intialFieldValue (value);
  }

  initialFieldValue (Object value) {
    return aMethod(value);
  }
```

tk better example

Replace Type Code with Methods

You have a type code and want to hide it

Replace the type code with explicit methods.

A type code is some coding scheme that alters the behavior of an object. Consider a pen that can write in black, red, or blue. Often you would find the colors defined as constants

```
public static final BLACK = 0;
public static final RED = 1;
public static final BLUE = 2;
```

Type codes a common element in non-OO programming, but they are much less important in OO programming. In non-OO programming you often find case statements switching behavior based on some type code from somewhere. In OO programs such case statements are frowned upon. If they do exist they only exist within the class that declares the type code (in this case the pen). Often *type codes are replaced with subclasses*.

But often clients do need to know something about the type code that an object has. Someone has to choose the pen color, and may want to change it. A GUI widget may want to display the color on the screen. Certainly the type code field on the instance should be encapsulated.

But a further move can be made to hide the existence of the type code entirely. This way there is no public declaration of a series of constants. Instead the interface of the class reflects the type code explicitly. Instead of finding out if the pen is set to black with `aPen.getPenColor() == Pen.BLACK` you would say `aPen.isBlack()`. Instead of setting it to red with `aPen.setPenColor(Pen.RED)`; you would go `aPen.beRed()`.

This is effective if you don't have too many values for your type code, and if you don't frequently add new type codes. If either of those circumstances appear you should *replace the type code with a class* instead.

- ❍ For each value of the type code, create a boolean query method. Name the method `is`*`TypeCodeValue`*`()`. So in the above example we would have `isBlack(), isRed(), isBlue()`. Replace all calls to the old type code query with the new queries and remove the old query.
- ❍ Compile and test.
- ❍ If the type code is mutable, for each value of the type code create a boolean modifier. Name the method `be`*`TypeCodeName`*`()` (eg `beBlack(), beRed()`). Replace all calls to the old modifier and remove the old modifier.
  - ☞ If there are rules about which changes can occur in what order, these should be checked by the modifiers
- ❍ Compile and test
- ❍ If the constructor takes a type code as an argument, then create static methods for each value of the type code. Name the constructors newTypeCodeName() (eg newBlack(), newRed()). Create a new private constructor with all the arguments of the old constructor except the type code. Use the new constructor and the boolean modifiers to implement the static methods. Replace all calls to the old constructor with calls to the new static methods and remove the old constructor.
  - ☞ If the modifiers are checking rules that are not applicable during creation then you won't be able to use the modifiers, either use direct variable access or create a private setting method for the type code.
- ❍ Compile and test
- ❍ Make the type codes private

❍ Compile and test

Replace Type Code with Class

You have a type code with many values

Replace the type code with a class. Create one instance for each value of the type code.

Replace Parameter with Method

You have a series of methods with the same parameter passed around between them

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

If a method can get a value that is passed in as parameter by another means, then it should. The parameter is unnecessary and should be removed. Leaving it there is only adding clutter to the method and often to a whole bunch of related methods too. By getting rid of it you make the calling path easier, and also make it easier to refactor later on.

In some cases the parameter may be there for a future parameterizsation 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 painful. Stable interfaces are good, but freezing a poor interface is a problem.

❍ Create a method that gets the object that is used in the parameter
❍ Replace references to the parameter in method bodies with references to the method.
  ☞ To keep the steps as small as possible, do the replacement one method at a time, working from the bottom of the call graph, and compile and test after each method.
❍ Remove the parameter from method calls and declarations
❍ Compile and test.

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 a group of procedures. Turn each procedure into a method of that class. Global variables become fields on the class. Treat the class as a Singleton.

Just copying a procedural program over to an object-oriented environment and refactoring it until it is right can certainly work, but it may well be more efficient to redesign from scratch. This is particularly so if it will take a while to get the old procedure to work properly in its native form. Debugging these things is tough, which is why we avoid them in ObjectLand.

Replace Type Code with Subclasses

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

Replace the type code with subclasses

Type codes can often result in different behavior. You may have a case statement of the form:

```
Switch (enum)
  Case value1:
     DoSomething1
  Case value2:
     DoSomething2
  Case value3:
     DoSomething3
```

Or a chain of if then else clauses:

```
If (enum) doSomething;
Else doSomethingElse;
```

Not all of these are contenders for subclasses. Some alteration of behavior based on the properties of the object is quite reasonable. However when you have several methods that are behaving differently based on the same value, particularly where it is of the multi-legged case statement variety, then it is time to consider using polymorphism instead.

There are two ways you can use polymorphism. The first is to subclass the existing class directly. This is the simplest way to do it. However if you already have subclasses for some other reason, or if the value is mutable (that is it may change during the lifetime of the object) then direct subclasses does not work. Instead you need to *replace the type code with a state object*.

- ❍ *Self-encapsulate* the type code
- ❍ 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.
- ❍ If the type code is passed into the constructor, you will need to *replace the constructor with creation methods*.
- ❍ Compile and test.

Replace Type Code with State Object

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.

This is similar to *replacing a type 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 the state pattern [Gang of Four].

- ❍ *Self encapsulate* the type code.
- ❍ Create a new class, name it after the purpose of the type code. This is the state object.
- ❍ Create a field in the old class for the new state object.
- ❍ Add subclasses of the state object, one for each type code.
- ❍ 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.
- ❍ 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.

Replace Case Statement with Inheritance

You have a case statement that switches on a type code

Move each leg of the case statement to a subclass

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. Statements look like this

```
Switch (enum)
  Case value1:
     DoSomething1
  Case value2:
     DoSomething2
  Case value3:
     DoSomething3
```

You can replace these statements by subclasses and polymorphism. Often several case statements will switch on the same type code. Each of these like case statements can be replaced by polymorphic methods.

- ❍ *Replace the type code with subclasses or state objects.*
- ❍ If the case statement is one part of a larger method then take the case statement part and *extract it into a method*
- ❍ If the type code was replaced by a state object, *move the case statement method* to the state object
- ❍ Create a subclass method that overrides the case statement method. Copy the body of that leg of the case 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
  - ☞ You may find it useful to test. Then if all is fine introduce a deliberate error into the subclass method, retest and check that the test blows up. This checks that the subclass code executed properly.
- ❍ Remove that leg of the case statement, compile and test.
- ❍ Repeat with each leg of the case statement until all legs are turned into subclass methods.
- ❍ Make the superclass method abstract

Replace Record With Data Class

You have a record structure in a traditional programming environment. This might be a C struct, a COBOL record, or a relational database table. How do you begin to make this object oriented?

Make a class for the record. For each field in the record make a field in the class. Set the protection for all the fields to private or protected. Provide a Getting Method and a Setting Method for each field.

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

Replace Temp with Query

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

Extract the expression into a method. Replace all references to the temp with a reference to the method.

- ❍ Look for a temporary variable that is set once with an assignment. If a temp is not set once consider Splitting the Temporarys Variable.
- ❍ 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.
- ❍ Find all references to the temp and replace them with a call to the new method.
- ❍ Remove the declaration and the assignment of the temp.
- ❍ Compile and test.

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 any method in the class can get at that information. That helps a lot in coming up with cleaner code in the class.

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 mulitple values. In this 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 something 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.

Self Encapsulate Field

You are using accessing a field directly, but the coupling to the field is becoming awkward. This is particularly common when there are several subclasses

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

The choice between Direct and Indirect Variable Access is one of the most heated ones in object design. People get very involved in saying which one is always right. I'm always in too 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.

- ❍ 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.

Spin a Satellite

You have a class that seems too complex

Create a new class and move some of the features over to it

Classes can easily get too big, this is especially true when the class represents an important concept in the problem domain.

- ❍ Decide which fields and which principal methods you want to move over.
- ❍ Create a field in the source class that refers to the target. Create the target object at the time that you fill the fields that you are planning to move over.
  - ☞ Most of the time you do this when you create the source object.
- ❍ Move the fields to the new class.
  - ☞ If you aren't intending to move any fields, then you can make the new object a *flyweight*.

- ❍ Examine the methods of the source class and decide which to move over. Once you have moved them, look again to see if you should move some more.
  - ☞ If several methods you use need to reference the source class then put a reference to the source in a field of the target. Build this reference when you create the target.

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.

Temporary variables are made for various uses. Some of these uses lead to the temp being assigned to several times. Loop Variables change for each run around a loop. Collecting Temporary Variables 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.

Temps should have a single responsibility within a method. Any variable with more than one responsibility should be replaced by a temp for each responsibility.

- ❍ At each assignment declare a new temporary variable and assign the right hand side of assignment to it.
  - ☞ 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.
- ❍ Remove the original temporary variable declaration.
  - ☞ You can still keep the original if its name makes very good sense for one of the new temporary variables. I prefer to remove the original because then the compiler will spot if I miss replacing a reference to it.
- ❍ Find each use of the temporary variable and replace it with the last declared replacement.
- ❍ Compile and test.

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.

Unify Subclass Methods

You have two similar, but not identical subclass methods

Decompose each method so that the pieces are either the same or completely different. If the order of the pieces is the same, then you have a template method.

TK Steps for finding things to refactor