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

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

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

To find a person's manager, clients ask:

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

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

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

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

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

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

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

Introduce Foreign Method

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

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

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

```
    Date newStart = nextDay(previousEnd);

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

Motivation

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

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

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

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

Mechanics

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

    ?rarr; *The method should not access any of the features of the client class. If it needs a value, send it in as a parameter.*

- Make an instance of the server class the first parameter.
- Comment the method as "foreign method; should be in *server.*"

    ?rarr; *This way you can use a text search to find foreign methods later if you get the chance to move the method.*

Example

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

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

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

```
  Date newStart = nextDay(previousEnd);

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

Introduce Local Extension

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

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

Motivation

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

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

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

In choosing between subclass and wrapper, I usually prefer the subclass because it is less work. The biggest roadblock to a subclass is that it needs to apply at object-creation time. If I can take over the creation process that's no problem. The problem occurs if you apply the local extension later. Subclassing forces me to create a new object of that subclass. If other objects refer to the old one, I have two objects with the original's data. If the original is immutable, there is no

problem; I can safely take a copy. But if the original can change, there is a problem, because changes in one object won't change the other and I have to use a wrapper. That way changes made through the local extension affect the original object and vice versa.

Mechanics

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

> ?rarr; *A constructor takes the original as an argument. The subclass version calls an appropriate superclass constructor; the wrapper version sets the delegate field to the argument.*

- Add new features to the extension.
- Replace the original with the extension where needed.
- Move any foreign methods defined for this class onto the extension.

Examples

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

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

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

A wrapper uses delegation:

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

Example: Using a Subclass

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

```
class MfDateSub extends Date
```

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

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

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

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

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

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

becomes

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

Example: Using a Wrapper

I start by declaring the wrapping class:

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

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

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

The converting constructor now just sets the instance variable:

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

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

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

   public boolean equals (MfDateWrap arg) {
       return (toDate().equals(arg.toDate()));
   }
```

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

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

becomes

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

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

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

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

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

The purpose of this kind of overriding is to hide the fact I'm using a wrapper from the user of the class. This is good policy because the user of wrapper really shouldn't care about the wrapper and should be able to treat the two equally. However, I can't completely hide this information. The problem lies in certain system methods, such as `equals.` Ideally you would think that you could override `equals` on `MfDateWrap` like this

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

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

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

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

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

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

?rarr; *A good test is to see whether 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.

```
public String _name
```

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

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, other objects can change and access data values without the owning object's 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, it is easier to change the code, because the changed code is in one place rather than scattered all over the program.

Encapsulate Field 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 Encapsulate Field I look for methods that use the new methods to see whether they fancy packing their bags and moving to the new object with a quick Move Method.

Mechanics

- Create getting and setting methods for the field.
- Find all clients outside the class that reference the field. If the client uses the value, replace the reference with a call to the getting method. If the client changes the value, replace the reference with a call to the setting method.

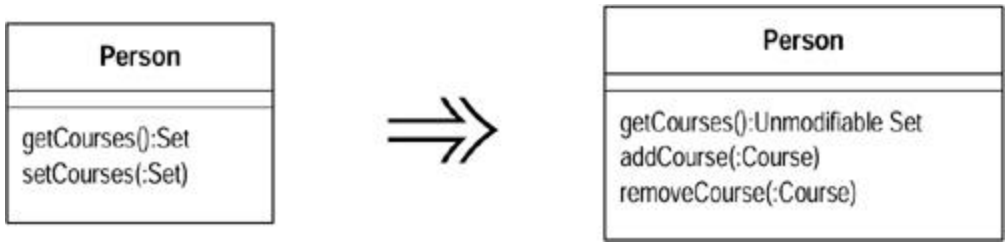
  ?rarr; *If the field is an object and the client invokes a modifier on the object, that is a use. Only use the setting method to replace an assignment.*

- Compile and test after each change.
- Once all clients are changed, declare the field as private.
- Compile and test.

Encapsulate Collection

A method returns a collection.

Make it return a read-only view and provide add/remove methods.



Motivation

Often a class contains a collection of instances. This collection might be an array, list, set, or vector. Such cases often have the usual getter and setter for the collection.

However, collections should use a protocol slightly different from that for other kinds of data. The getter should not return the collection object itself, because that allows clients to manipulate the contents of the collection without the owning class's knowing what is going on. It also reveals too much to clients about the object's internal data structures. A getter for a multivalued attribute should return something that prevents manipulation of the collection and hides unnecessary details about its structure. How you do this varies depending on the version of Java you are using.

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 and remove operations or have the clients call those operations instead.

  ?rarr; *Setters are used in two cases: when the collection is empty and when the setter is replacing a nonempty collection.*

  ?rarr; *You may wish to use* Rename Method *to rename the setter. Change it from set to initialize or replace.*

- Compile and test.
- Find all users of the getter that modify the collection. Change them to use the add and remove methods. Compile and test after each change.

- When all uses of the getter that modify have been changed, modify the getter to return a read-only view of the collection.

  ?rarr; *In Java 2, this is the appropriate unmodifiable collection view.*

  ?rarr; *In Java 1.1, you should return a copy of the collection.*

  Compile and test.

- Find the users of the getter. Look for code that should be on the host object. Use Extract Method and Move Method to move the code to the host object.

For Java 2, you are done with that. For Java 1.1, however, clients may prefer to use an enumeration. To provide the enumeration:

- Change the name of the current getter and add a new getter to return an enumeration. Find users of the old getter and change them to use one of the new methods.

  ?rarr; *If this is too big a jump, use Rename Method on the old getter, create a new method that returns an enumeration, and change callers to use the new method.*

- Compile and test.

Examples

Java 2 added a whole new group of classes to handle collections. It not only added new classes but also altered the style of using collections. As a result the way you encapsulate a collection is different depending on whether you use the Java 2 collections or the Java 1.1 collections. I discuss the Java 2 approach first, because I expect the more functional Java 2 collections to displace the Java 1.1 collections during the lifetime of this book.

Example: Java 2

A person is taking courses. Our course is pretty simple:

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

I'm not going to bother with anything else on the course. The interesting class is the person:

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

With this interface, clients adds courses with code such as

```
            Person kent = new Person();
            Set s = new HashSet();
            s.add(new Course ("Smalltalk Programming", false));
            s.add(new Course ("Appreciating Single Malts", true));
            kent.setCourses(s);
            Assert.equals (2, kent.getCourses().size());
            Course refact = new Course ("Refactoring", true);
            kent.getCourses().add(refact);
            kent.getCourses().add(new Course ("Brutal Sarcasm",
false));
            Assert.equals (4, kent.getCourses().size());
            kent.getCourses().remove(refact);
            Assert.equals (3, kent.getCourses().size());
```

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

```
            Iterator iter = person.getCourses().iterator();
            int count = 0;
            while (iter.hasNext()) {
               Course each = (Course) iter.next();
               if (each.isAdvanced()) count ++;
            }
```

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

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

Life will be easier if I initialize the field as well:

```
    private Set _courses = new HashSet();
```

I then look at the users of the setter. If there are many clients and the setter is used heavily, I need to replace the body of the setter to use the add and remove operations. The complexity of this process depends on how the setter is used. There are two cases. In the simplest case the client uses the setter to initialize the values, that is, 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(Set arg) {
        Assert.isTrue(_courses.isEmpty());
```

```
        Iterator iter = arg.iterator();
        while (iter.hasNext()) {
            addCourse((Course) iter.next());
        }
     }
```

After changing the body this way, it is wise to use Rename Method to make the intention clearer.

```
    public void initializeCourses(Set arg) {
        Assert.isTrue(_courses.isEmpty());
        Iterator iter = arg.iterator();
        while (iter.hasNext()) {
            addCourse((Course) iter.next());
        }
    }
```

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 (as general cases often do).

If I know that I don't have any additional behavior when adding elements as I initialize, I can remove the loop and use `addAll`.

```
    public void initializeCourses(Set arg) {
        Assert.isTrue(_courses.isEmpty());
        _courses.addAll(arg);
    }
```

I can't just assign the set, even though the previous set was empty. If the client simply create a set and use the setter, I can get them to use the add were to modify the set after passing it in, that would violate encapsulation. I have to make a copy.

If the clients simply create a set and use the setter, I can get them to use the add and remove methods directly and remove the setter completely. Code such as

```
            Person kent = new Person();
            Set s = new HashSet();
            s.add(new Course ("Smalltalk Programming", false));
            s.add(new Course ("Appreciating Single Malts", true));
            kent.initializeCourses(s);
```

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 is cases in which someone uses the getter to modify the underlying collection, for example:

```
kent.getCourses().add(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 an unmodifiable view:

```
public Set getCourses() {
    return Collections.unmodifiableSet(_courses);
}
```

At this point I've encapsulated the collection. No one can change the elements of collection except through methods on the person.

Moving Behavior into the Class

I have the right interface. Now I like to look at the users of the getter to find code that ought to be on person. Code such as

```
        Iterator iter = person.getCourses().iterator();
        int count = 0;
        while (iter.hasNext()) {
           Course each = (Course) iter.next();
           if (each.isAdvanced()) count ++;
        }
```

is better moved to person because it uses only person's data. First I use Extract Method on the code:

```
int numberOfAdvancedCourses(Person person) {
     Iterator iter = person.getCourses().iterator();
     int count = 0;
     while (iter.hasNext()) {
         Course each = (Course) iter.next();
         if (each.isAdvanced()) count ++;
     }
     return count;
 }
```

And then I use Move Method to move it to person:

```
class Person...
```

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

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

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

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

Remove Setting Method

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

Remove any setting method for that field.

Motivation

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

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

Mechanics

- Compile and test.
- Check that the setting method is called only in the constructor, or in a method called by the constructor.
- Modify the constructor to access the variables directly.

> ?rarr; *You cannot do this if you have a subclass setting the private fields of a superclass. In this case you should try to provide a protected superclass method (ideally a constructor) to set these values. Whatever you do, don't give the superclass method a name that will confuse it with a setting method.*

- Compile and test.
- Remove the setting method and make the field final.
- Compile.

Example

A simple example is as follows:

```
class Account {

  private String _id;

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

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

which can be replaced with

```
class Account {

  private final String _id;

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

The problems come in some variations. First is the case in which you are doing 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, I can make the change in the constructor. If the change is complex or I need to call it from separate methods, I need to provide a method. In that case I need to name the method to make its intention clear:

```
class Account {
  private final String _id;
  Account (String id) {
      initializeId(id);
  }

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

An awkward case lies with subclasses that initialize private superclass variables:

```
class InterestAccount extends Account...

  private double _interestRate;

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

The problem is that I cannot access `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, a well-named method is the best thing to use:

```
class InterestAccount...

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

Another case to consider is 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.

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 in which 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 whether they can be hidden. If it doesn't, you should make this check at regular intervals.

A particularly common case is hiding getting and setting methods as you work up a richer interface that provides more behavior. This case is most common when you are starting with a class that is little more than an encapsulated data holder. As more behavior is built into the class, you may 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, you can remove the method.

Mechanics

- Check regularly for opportunities to make a method more private.

  ?rarr; *Use a lint-style tool, do manual checks every so often, and check when you remove a call to a method in another class.*

  ?rarr; *Particularly look for cases such as this with setting methods.*

- Make each method as private as you can.
- Compile after doing a group of hidings.

?rarr; *The compiler checks this naturally, 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.

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

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

Motivation

The most obvious motivation for *Replace Constructor with Factory Method* comes with replacing a type code with subclassing. You have an object that often is created with a type code but now needs subclasses. The exact subclass is based on the type code. However, constructors can only return an instance of the object that is asked for. So you need to replace the constructor with a factory method [Gang of Four].

You can use factory methods for other situations in which constructors are too limited. Factory methods are essential for Change Value to Reference. They also can 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 private.
- Compile.

Example

A quick but wearisome and belabored example is the employee payment system. I have the following employee:

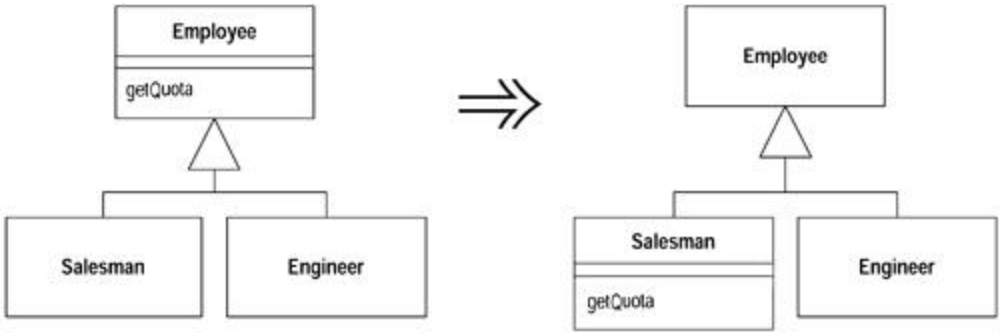
```
class Employee {

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

Push Down Method

Behavior on a superclass is relevant only for some of its subclasses.

Move it to those subclasses.



Motivation

Pull Down Method is the opposite of Pull Up Method. I use it when I need to move behavior from a superclass to a specific subclass, usually because it makes sense only there. You often do this when you use Extract Subclass.

Mechanics

- Declare a method in all subclasses and copy the body into each subclass.

  ?rarr; *You may need to declare fields as protected for the method to access them. Usually you do this if you intend to push down the field later. Otherwise use an accessor on the superclass. If this accessor is not public, you need to declare it as protected.*

- Remove method from superclass.

  ?rarr; *You may have to change callers to use the subclass in variable and parameter declarations.*

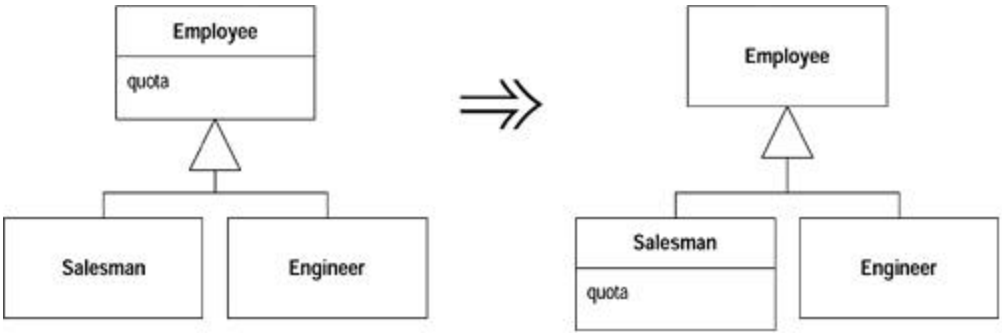
  ?rarr; *If it makes sense to access the method through a superclass variable, you don't intend to remove the method from any subclasses, and the superclass is abstract, you can declare the method as abstract, in the superclass.*

- Compile and test.
- Remove the method from each subclass that does not need it.
- Compile and test.

Push Down Field

A field is used only by some subclasses.

Move the field to those subclasses.



Motivation

Push Down Field is the opposite of Pull Up Field. Use it when you don't need a field in the superclass but only in a subclass.

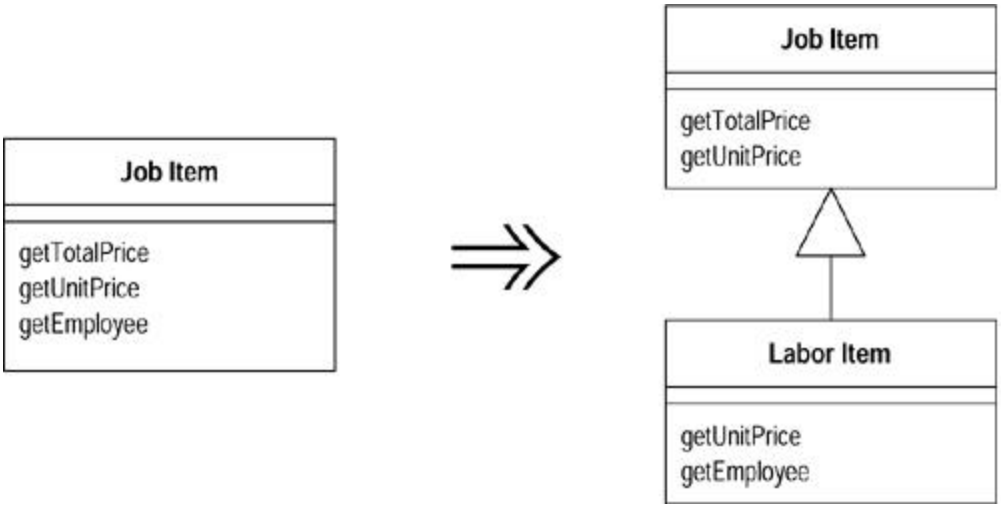
Mechanics

- Declare the field in all subclasses.
- Remove the field from the superclass.
- Compile and test.
- Remove the field from all subclasses that don't need it.
- Compile and test.

Extract Subclass

A class has features that are used only in some instances.

Create a subclass for that subset of features.



Motivation

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

You often find that null objects return other null objects.

Example: Testing Interface

The testing interface is an alternative to defining an isNull method. In this approach I create a null interface with no methods defined:

```
  interface Null {}
```

I then implement null in my null objects:

```
  class NullCustomer extends Customer implements Null...
```

I then test for nullness with the instanceof operator:

```
  aCustomer instanceof Null
```

I normally run away screaming from the `instanceof` operator, but in this case it is okay to use it. It has the particular advantage that I don't need to change the customer class. This allows me to use the null object even when I don't have access to customer's source code.

Other Special Cases

When carrying out this refactoring, 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 there is an 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 actually can carry data, such as usage records for the unknown customer, so that we can bill the customers when we find out who they are.

In essence there is a bigger pattern here, called *special case.* A special case class is a particular instance of a class with special behavior. So `UnknownCustomer` and `NoCustomer` would both be special cases of `Customer.` You often see special cases with numbers. Floating points in Java have special cases for positive and negative infinity and for not a number (NaN). The value of special cases is that they help reduce dealing with errors. Floating point operations don't throw exceptions. Doing any operation with NaN yields another NaN in the same way that accessors on null objects usually result in other null objects.

Introduce Assertion

A section of code assumes something about the state of the program.

Make the assumption explicit with an assertion.

```
   double getExpenseLimit() {
       // should have either expense limit or a primary project
       return (_expenseLimit != NULL_EXPENSE) ?
           _expenseLimit:
           _primaryProject.getMemberExpenseLimit();
   }
```

```
   double getExpenseLimit() {
       Assert.isTrue (_expenseLimit != NULL_EXPENSE || _primaryProject
!= null);
       return (_expenseLimit != NULL_EXPENSE) ?
           _expenseLimit:
           _primaryProject.getMemberExpenseLimit();
   }
```

Motivation

Often sections of code work only if certain conditions are true. This may be as simple as a square root calculation's working only on a positive input value. With an object it may be assumed that at least one of a group of fields has a value in it.

Such assumptions often are not stated but can only be decoded by looking through an algorithm. Sometimes the assumptions are stated with a comment. A better technique is to make the assumption explicit by writing an assertion.

An assertion is a conditional statement that is assumed to be always true. Failure of an assertion indicates programmer error. As such, assertion failures should always result in unchecked exceptions. Assertions should never be used by other parts of the system. Indeed assertions usually are removed for production code. It is therefore important to signal something is an assertion.

Assertions act as communication and debugging aids. In communication they help the reader understand the assumptions the code is making. In debugging, assertions can help catch bugs closer to their origin. I've noticed the debugging help is less important when I write self-testing code, but I still appreciate the value of assertions in communciation.

Mechanics

Because assertions should not affect the running of a system, adding one is always behavior preserving.

- *When you see that a condition is assumed to be true, add an assertion to state it.*

?rarr; *Have an assert class that you can use for assertion behavior.*

Beware of overusing assertions. Don't use assertions to check everything that you think is true for a section of code. Use assertions only to check things that *need* to be true. Overusing assertions can lead to duplicate logic that is awkward to maintain. Logic that covers an assumption is good because it forces you to rethink the section of the code. If the code works without the assertion, the assertion is confusing rather than helpful and may hinder modification in the future.

Always ask whether the code still works if an assertion fails. If the code does work, remove the assertion.

Beware of duplicate code in assertions. Duplicate code smells just as bad in assertion checks as it does anywhere else. Use Extract Method liberally to get rid of the duplication.

Example

Here's a simple tale of expense limits. Employees can be given an individual expense limit. If they are assigned a primary project, they can use the expense limit of that primary project. They don't have to have an expense limit or a primary project, but they must have one or the other. This assumption is taken for granted in the code that uses expense limits:

```
  class Employee...
    private static final double NULL_EXPENSE = -1.0;
    private double _expenseLimit = NULL_EXPENSE;
    private Project _primaryProject;
   double getExpenseLimit() {
       return (_expenseLimit != NULL_EXPENSE) ?
           _expenseLimit:
           _primaryProject.getMemberExpenseLimit();
   }
   boolean withinLimit (double expenseAmount) {
       return (expenseAmount <= getExpenseLimit());
   }
```

This code contains an implicit assumption that the employee has either a project or a personal expense limit. Such an assertion should be clearly stated in the code:

```
   double getExpenseLimit() {
       Assert.isTrue (_expenseLimit != NULL_EXPENSE || _primaryProject
!= null);
       return (_expenseLimit != NULL_EXPENSE) ?
           _expenseLimit:
           _primaryProject.getMemberExpenseLimit();
   }
```

This assertion does not change any aspect of the behavior of the program. Either way, if the condition is not true, I get a runtime exception: either a null pointer exception in `withinLimit` or a runtime exception inside `Assert.isTrue.` In some circumstances the assertion helps find the bug, because it is closer to where things went wrong. Mostly, however, the assertion helps to communicate how the code works and what it assumes.

I often find I use Extract Method on the conditional inside the assertion. I either use it in several places and eliminate duplicate code or use it simply to clarify the intention of the condition.

One of the complications of assertions in Java is that there is no simple mechanism to putting them in. Assertions should be easily removable, so they don't affect performance in production code. Having a utility class, such as `Assert,` certainly helps. Sadly, any expression inside the assertion parameters executes whatever happens. The only way to stop that is to use code like:

```
   double getExpenseLimit() {
       Assert.isTrue (Assert.ON &&
           (_expenseLimit != NULL_EXPENSE || _primaryProject != null));
       return (_expenseLimit != NULL_EXPENSE) ?
           _expenseLimit:
           _primaryProject.getMemberExpenseLimit();
   }
```

or

```
   double getExpenseLimit() {
       if (Assert.ON)
           Assert.isTrue (_expenseLimit != NULL_EXPENSE ||
_primaryProject != null);
       return (_expenseLimit != NULL_EXPENSE) ?
           _expenseLimit:
           _primaryProject.getMemberExpenseLimit();
   }
```

If `Assert.ON` is a constant, the compiler should detect and eliminate the dead code if it is false. Adding the clause is messy, however, so many programmers prefer the simpler use of `Assert` and then use a filter to remove any line that uses assert at production time (using perl or the like).

The `Assert` class should have various methods that are named helpfully. In addition to `isTrue`, you can have `equals`, and `shouldNeverReachHere`.