Chapter 1 : Refactoring

When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature.

Before you start refactoring, check that you have a solid suite of tests. These tests must be self-checking.

Refactoring changes the programs in small steps. If you make a mistake, it is easy to find the bug.

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Code that communicates its purpose is very important. I often refactor just when I'm reading some code. That way as I gain understanding about the program, I embed that understanding into the code for later so I don't forget what I learned.

Chapter 2. Principles in Refactoring

Refactoring :

a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

Why should you refactor?

- Refactoring Improves the Design of Software
- Refactoring Makes Software Easier to Understand
- Refactoring Helps You Find Bugs
- Refactoring Helps You Program Faster

When Should You Refactor?

Refactoring is something you do all the time in little bursts.

Three strikes and you refactor - The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor.

- Refactor When You Add Function
- Refactor When You Need to Fix a Bug
- Refactor As You Do a Code Review

Why Refactoring Works?

Programs have two kinds of value: what they can do for you today and what they can do for you tomorrow.Most of the time we are focused on today part.

Things that make programs hard to work with are :

- Programs that are hard to read are hard to modify.
- Programs that have duplicated logic are hard to modify.
- Programs that require additional behavior that requires you to change running code are hard to modify.
- Programs with complex conditional logic are hard to modify.

Refactoring is the process of taking a running program and adding to its value, not by changing its behavior but by giving it more of these qualities that enable us to continue developing at speed.

Indirection and Refactoring

Computer Science is the discipline that believes all problems can be solved with one more layer of indirection.

Indirection is a two-edged sword, however. Every time you break one thing into two pieces, you have more things to manage. It also can make a program harder to read as an object delegates to an object delegating to an object. So you'd like to minimize indirection.

But Indirection can pay for itself in one of the ways :

- To enable sharing of logic.
- To explain intention and implementation separately.
- To isolate change
- To encode conditional logic.

Identify indirection that isn't paying for itself and take it out. Often this takes the form of intermediate methods that used to serve a purpose but no longer do. Or it could be a component that you expected to be shared or polymorphic but turned out to be used in only one place. When you find parasitic indirection, take it out. Again, you will have a more valuable program, not because there is more of one of the four qualities listed earlier but because it costs less indirection to get the same amount from the qualities.

Problems with Refactorings

- Databases -

  Most business applications are tightly coupled to the database.

  With nonobject databases a way to deal with this problem is to place a separate layer of software between your object model and your database model. That way you can isolate the changes to the two different models. As you update one model, you don't need to update the other. You just update the intermediate layer. Such a layer adds complexity but gives you a lot of flexibility. Even without refactoring it is very important in situations in which you have multiple databases or a complex database model that you don't have control over. You don't have to start with a separate layer. You can create the layer as you notice parts of your object model becoming volatile. This way you get the greatest leverage for your changes. Object databases both help and hinder.Changing Interfaces
- Changing Interfaces -

If a refactoring changes a published interface, you have to retain both the old interface and the new one, at least until your users have had a chance to react to the change.

Don't publish interfaces prematurely. Modify your code ownership policies to smooth refactoring.

When shouldn’t you refactor?

There are times when you should not refactor at all. The principle example is when you should rewrite from scratch instead.A clear sign of the need to rewrite is when the current code just does not work. A compromise route is to refactor a large piece of software into components with strong encapsulation. Then you can make a refactor-versus-rebuild decision for one component at a time.

The other time you should avoid refactoring is when you are close to a deadline. At that point the productivity gain from refactoring would appear after the deadline and thus be too late.

Refactoring and Design

The problem with building a flexible solution is that flexibility costs. Building flexibility in all the places makes the overall system a lot more complex and expensive to maintain.

Refactoring can lead to simpler designs without sacrificing flexibility. This makes the design process easier and less stressful.

Refactoring and Performance

The purpose of refactoring is to make the software easier to understand and modify. Like refactoring, performance optimization does not usually change the behavior of a component (other than its speed); it only alters the internal structure. However, the purpose is different. Performance optimization often makes code harder to understand, but you need to do it to get the performance you need.

The secret to fast software, in all but hard real-time contexts, is to write tunable software first and then to tune it for sufficient speed.

The budgeting approach is used in hard real-time systems.

The second approach is the constant attention approach. With this approach, every programmer, all the time, does whatever he or she can to keep performance high.

The interesting thing about performance is that if you analyze most programs, you find that they waste most of their time in a small fraction of the code. If you optimize all the code equally, you end up with 90 percent of the optimizations wasted, because you are optimizing code that isn't run much. The time spent making the program fast, the time lost because of lack of clarity, is all wasted time.

The third approach to performance improvement takes advantage of this 90 percent statistic. In this approach you build your program in a well-factored manner without paying attention to performance until you begin a performance optimization stage, usually fairly late in development. During the performance optimization stage, you follow a specific process to tune the program.

You begin by running the program under a profiler that monitors the program and tells you where it is consuming time and space. This way you can find that small part of the program where the performance hot spots lie. Then you focus on those performance hot spots and use the same optimizations you would use if you were using the constant attention approach. But because you are focusing your attention on a hot spot, you are having much more effect for less work. Even so you remain cautious. As in refactoring you make the changes in small steps. After each step you compile, test, and rerun the profiler. If you haven't improved performance, you back out the change. You continue the process of finding and removing hot spots until you get the performance that satisfies your users.

Having a well-factored program helps with this style of optimization in two ways.First, it gives you time to spend on performance tuning. Because you have well-factored code, you can add function more quickly. This gives you more time to focus on performance. (Profiling ensures you focus that time on the right place.) Second, with a well-factored program you have finer granularity for your performance analysis. Your profiler leads you to smaller parts of the code, which are easier to tune. Because the code is clearer, you have a better understanding of your options and of what kind of tuning will work. I've found that refactoring helps me write fast software. It slows the software in the short term while I'm refactoring, but it makes the software easier to tune during optimization.

Chapter 3. Bad Smells in Code

- Duplicated Code
- Long Method - Generally, any method longer than ten lines should make you start asking questions.
- Large Class - When a class is trying to do too much, it often shows up as too many instance variables. More generally, common prefixes or suffixes for some subset of the variables in a class suggest the opportunity for a component.
- Long Parameter List - More than three or four parameters for a method.
- Divergent Change - Divergent change occurs when one class is commonly changed in different ways for different reasons.
- Shotgun Surgery - When every time you make a kind of change, you have to make a lot of little changes to a lot of different classes. Divergent change is one class that suffers many kinds of changes, and shotgun surgery is one change that alters many classes.
- Feature Envy - A method that seems more interested in the class other than the one it actually is in.
- Data Clumps - Bunches of data that hang around together really ought to be made into their own object.
- Primitive Obsession - Use of primitives instead of small objects for simple tasks
- Switch Statements - The problem with switch statements is essentially that of duplication. Often you find the same switch statement scattered about a program in

different places. If you add a new clause to the switch, you have to find all these switch, statements and change them.
- Parallel Inheritance Hierarchies - Parallel inheritance hierarchies is really a special case of shotgun surgery. In this case, every time you make a subclass of one class, you also have to make a subclass of another.
- Lazy Class - A class that isn’t doing enough to pay for itself should be eliminated.
- Speculative Generality - Speculative generality can be spotted when the only users of a method or class are test cases.
- Temporary field - Sometimes you see an object in which an instance variable is set only in certain circumstances.A common case of temporary field occurs when a complicated algorithm needs several variables.
- Message chains - If there are functions which are again calling other functions. Like a()->b()->c()....
- Middle Man - You look at a class's interface and find half the methods are delegating to this other class.
- Inappropriate Intimacy - Sometimes classes become far too intimate and spend too much time delving in each others' private parts.
- Alternative Classes with Different Interfaces
- Incomplete Library Class - Sooner or later, libraries stop meeting user needs. The only solution to the problem – changing the library – is often impossible since the library is read-only.
- Data Class - A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters).
- Refused Bequest - If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter.
- Comments - A method is filled with explanatory comments

Chapter 4 : Building Tests

- Make sure all tests are fully automatic and that they check their own results.
- A suite of tests is a powerful bug detector that decapitates the time it takes to find bugs.
- Run your tests frequently. Localize tests whenever you compile—every test at least every day.
- One of the most useful times to write tests is before you start programming. When you need to add a feature, begin by writing the test.
- When you get a bug report, start by writing a unit test that exposes the bug.
- It is better to write and run incomplete tests than not to run complete tests.
- Think of the boundary conditions under which things might go wrong and concentrate your tests there.
- Don't forget to test that exceptions are raised when things are expected to go wrong.
- Don't let the fear that testing can't catch all bugs stop you from writing the tests that will catch most bugs.

Chapter 6 : Composing Methods

Extract Method

You have a code fragment that can be grouped together. Turn the fragment into a method whose name explains the purpose of the metho

```
void printOwing(double amount) {
    printBanner();

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + amount);
}
```

⇓

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails (double amount) {

   System.out.println ("name:" + _name);
   System.out.println ("amount" + amount);
}
```

Inline Methods

A method's body is just as clear as its name. Put the method's body into the body of its callers and remove the method.

```
int getRating() {
    return (moreThanFiveLateDeliveries()) ?
}
```

```
boolean moreThanFiveLateDeliveries() {
    return _numberOfLateDeliveries > 5;
}
```

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

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

Inline Temp

You have a temp that is assigned to once with a simple expression, and the temp is getting in the way of other refactorings.

Replace all references to that temp with the expression.

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000)
```

```
return (anOrder.basePrice() > 1000)
```

Replace Temp with Query

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

Extract the expression into a method. Replace all references to the temp with the expression.The new method can then be used in other methods.

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```

⇓

```
   if (basePrice() > 1000)
       return basePrice() * 0.95;
   else
       return basePrice() * 0.98;

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

Temps often are used to store summary information in loops. The entire loop can be extracted into a method; this removes several lines of noisy code.

You may be concerned about performance in this case. As with other performance issues, let it slide for the moment. Nine times out of ten, it won't matter. When it does matter, you will fix the problem during optimization. With your code better factored, you will often find more powerful optimizations, which you would have missed without refactoring. If worse comes to worse, it's very easy to put the temp back.

Introduce Explaining Variable

You have a complicated expression. Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.

```
  if ( (platform.toUpperCase().indexOf("MAC") > -1) &&
       (browser.toUpperCase().indexOf("IE") > -1) &&
        wasInitialized() && resize > 0 )
{
      // do something
}
```

⇓

```
    final boolean isMacOs     = platform.toUpperCase().indexOf("MAC") >
-1;
    final boolean isIEBrowser = browser.toUpperCase().indexOf("IE")  >
-1;
    final boolean wasResized  = resize > 0;

    if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {
        // do something
    }
```

I prefer to use Extract Method, because now these methods are available to any other part of the object that needs them. Initially I make them private, but I can always relax that if another object needs them.

So when do I use Introduce Explaining Variable? The answer is when Extract Method is more effort.

Split Temporary Variable

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

Make a separate temporary variable for each assignment.

Using a temp for two different things is very confusing for the reader.

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

⇓

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

Remove Assignments to Parameters

The code assigns to a parameter. Use a temporary variable instead.

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

⇓

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

I have no problems with doing something to the object that was passed in; I do that all the time.

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

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

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

Replace Method with Method Object

You have a long method that uses local variables in such a way that you cannot apply Extract Method. Turn the method into its own object so that all the local variables become fields on that object. You can then decompose the method into other methods on the same object.

Substitute Algorithm

You want to replace an algorithm with one that is clearer.

Replace the body of the method with the new algorithm.

If you find a clearer way to do something, you should replace the complicated way with the clearer way. Refactoring can break down something complex into simpler pieces, but sometimes you just reach the point at which you have to remove the whole algorithm and replace it with something simpler.

When you have to take this step, make sure you have decomposed the method as much as you can. Substituting a large, complex algorithm is very difficult.

Chapter 7 : Moving Features Between Objects

One of the most fundamental, if not the fundamental, decision in object design is deciding where to put responsibilities.

Move Method

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

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

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

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

When I need to use a feature of the source class I can do one of four things:

- move this feature to the target class as well,
- create or use a reference from the target class to the source
- pass the source object as a parameter to the method.
- if the feature is a variable, pass it in as a parameter.

Move Field

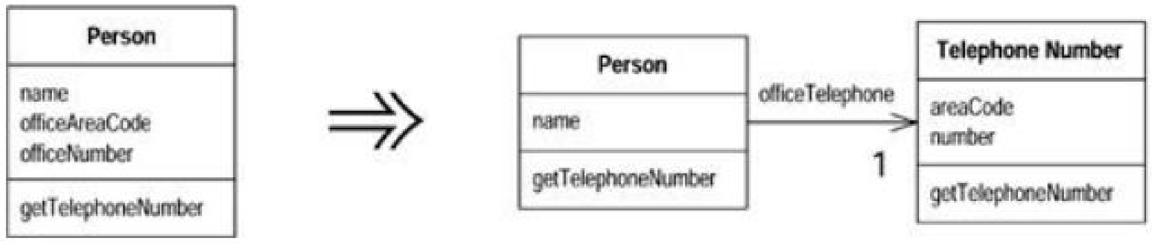
A field is, or will be, used by another class more than the class on which it is defined. Create a new field in the target class, and change all its users.

If you are likely to be moving the methods that access it frequently or if a lot of methods access the field, you may find it useful to use Self Encapsulate Field.

Extract Class

You have one class doing work that should be done by two.

Create a new class and move the relevant fields and methods from the old class into the new class.



The decision then is how much to expose the new class to my clients.

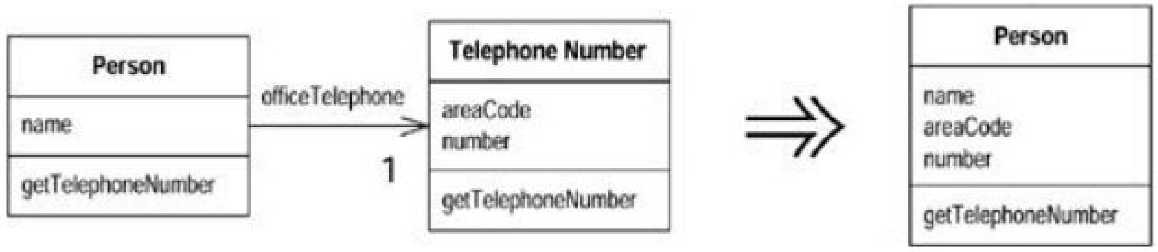
I have the following options:

- I accept that any object may change any part of the telephone number. This makes the telephone number a reference object, and I should consider Change Value to Reference. In this case the person would be the access point for the telephone number.
- I don't want anybody to change the value of the telephone number without going through the person. I can either make the telephone number immutable, or I can provide an immutable interface for the telephone number.
- Another possibility is to clone the telephone number before passing it out. But this can lead to confusion because people think they can change the value. It

also may lead to aliasing problems between clients if the telephone number is passed around a lot.

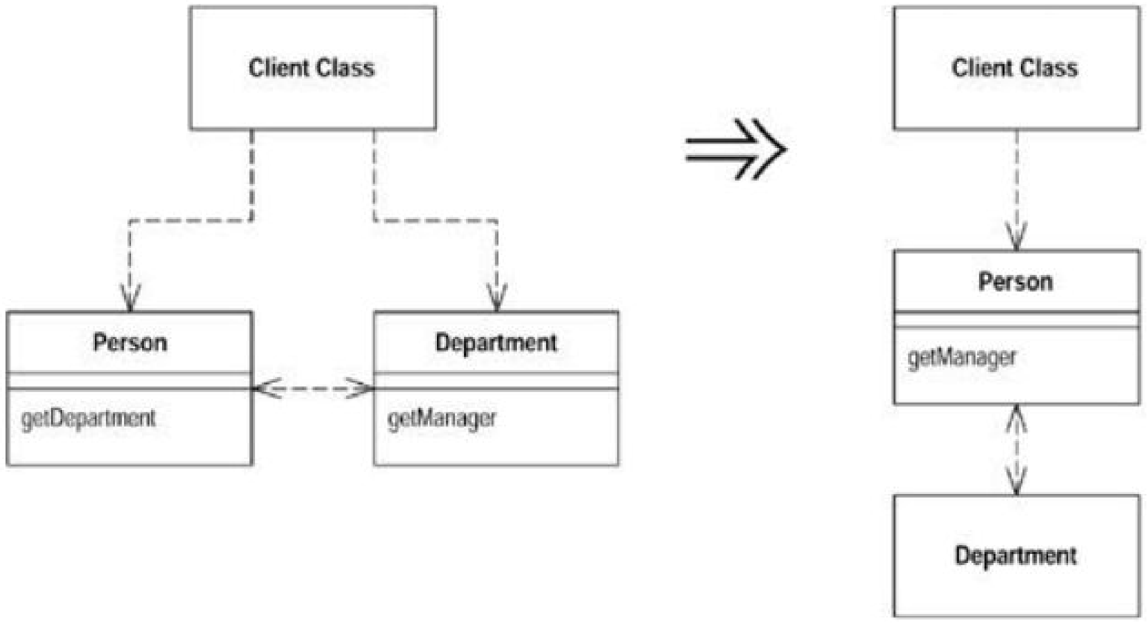
Inline Class

A class isn't doing very much.Move all its features into another class and delete it.



Hide Delegate

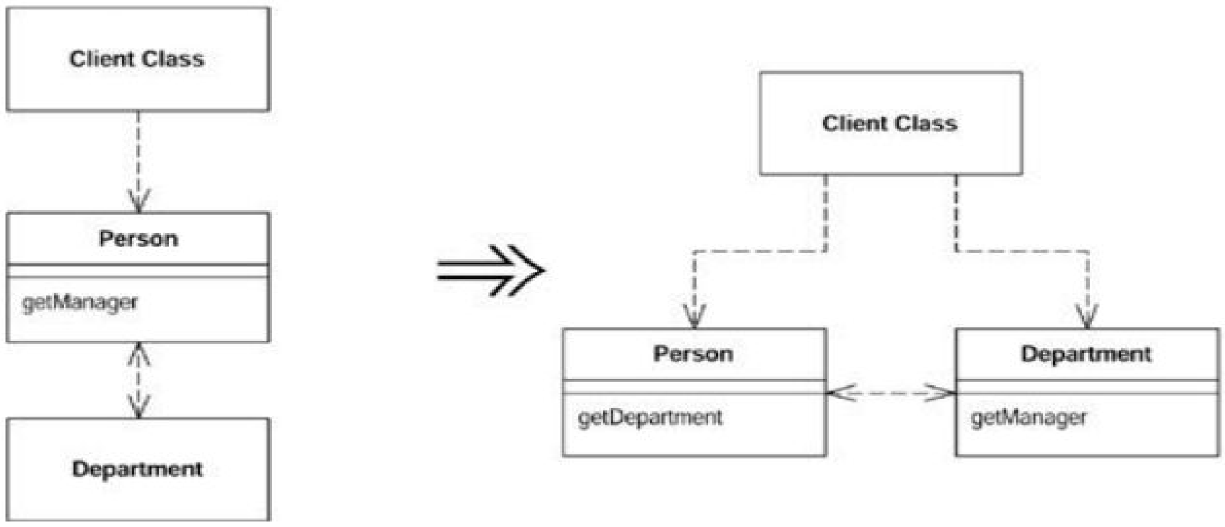
A client is calling a delegate class of an object.
Create methods on the server to hide the delegate.



Encapsulation means that objects need to know less about other parts of the system. Then when things change, fewer objects need to be told about the change—which makes the change easier to make.

Remove Middle Man

A class is doing too much simple delegation.
Get the client to call the delegate directly.



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);
```

$\Downarrow$

```
    Date newStart = nextDay(previousEnd);

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

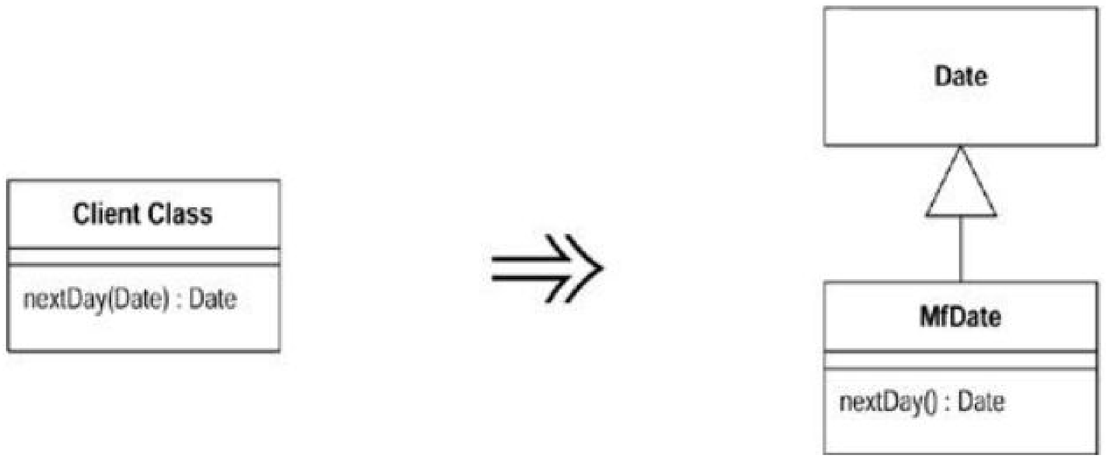
If you find yourself creating many foreign methods on a server class, or you find many of your classes need the same foreign method, you should use Introduce Local Extension instead.

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.

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.



A local extension is a separate class, but it is a subtype of the class it is extending.

By using the local extension you keep to the principle that methods and data should be packaged into well-formed units.

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.

A particular problem with using wrappers is how to deal with methods that take an original as an argument.I can avoid testing the type of unknown objects by providing versions of this method for both Date and MfDateWrap.

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.

Chapter 8 : Organizing Data

Self Encapsulate Field

You are accessing a field directly, but the coupling to the field is becoming awkward. Create getting and setting methods for the field and use only those to access the field.

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

⇓

```
private int _low, _high;
boolean includes (int arg) {
    return arg >= getLow() && arg <= getHigh();
}
int getLow() {return _low;}
int getHigh() {return _high;}
```

Essentially the advantages of indirect variable access are that it allows a subclass to override how to get that information with a method and that it supports more flexibility in managing the data, such as lazy initialization, which initializes the value only when you need to use it.

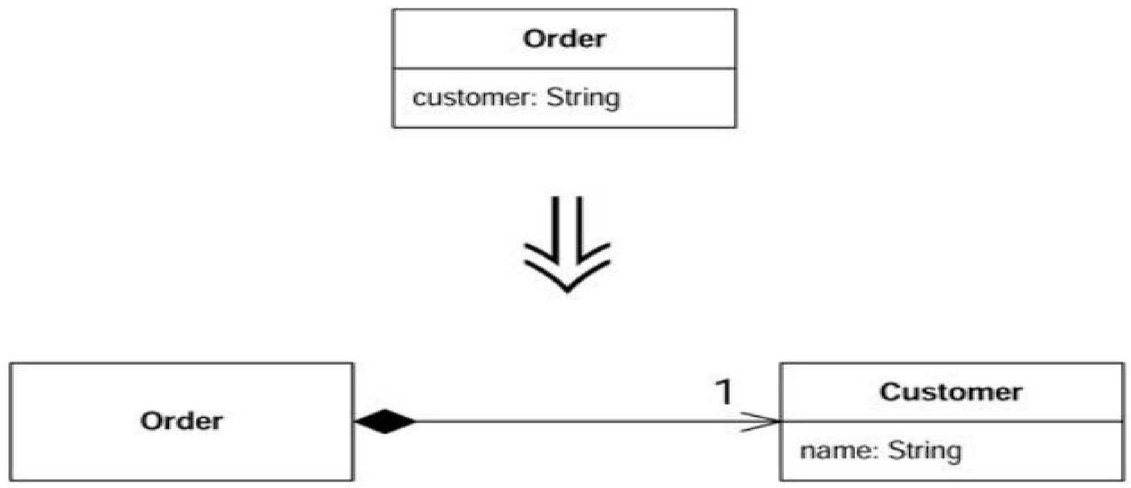
When you are using self-encapsulation you have to be careful about using the setting method in the constructor. Often it is assumed that you use the setting method for changes after the object is created, so you may have different behavior in the setter than you have when initializing. In cases like this I prefer using either direct access from the constructor or a separate initialization method.

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

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

Replace Data Value with Object

You have a data item that needs additional data or behavior.
Turn the data item into an object.

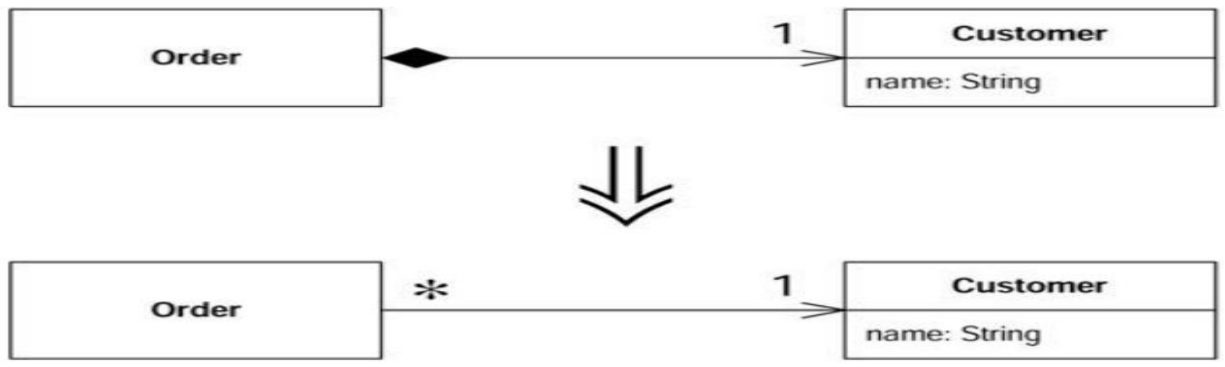


After applying this refactoring technique, it’s wise to apply Change Value to Reference on the field that contains the object. This allows storing a reference to a single object that corresponds to a value instead of storing dozens of objects for one and the same value.

Usually this approach is needed for real-world objects like Customer, not for objects like date.

Change Value to Reference

You have a class with many equal instances that you want to replace with a single object.Turn the object into a reference object.



Reference objects are things like customer or account. Each object stands for one object in the real world, and you use the object identity to test whether they are equal. Value objects are things like date or money.

Decide what object is responsible for providing access to the objects. This may be a static dictionary or a registry object.

Change Reference to Value

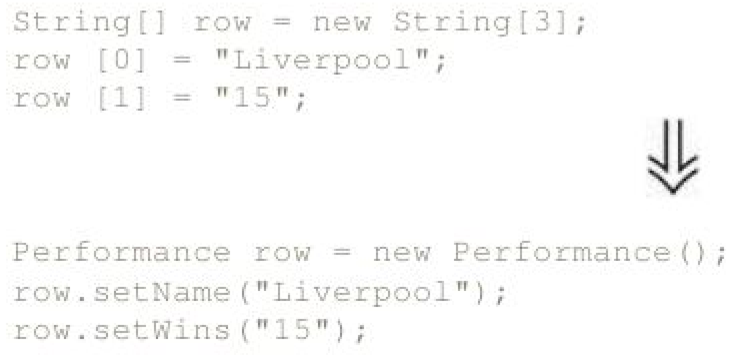
You have a reference object that is small, immutable, and awkward to manage. Turn it into a value object.

An important property of value objects is that they should be immutable. Any time you invoke a query on one, you should get the same result. If this is true, there is no problem having many objects represent the same thing. If the value is mutable, you have to ensure that changing any object also updates all the other objects that represent the same thing. That's so much of a pain that the easiest thing to do is to make it a reference object.

It's important to be clear on what immutable means. If you have a money class with a currency and a value, that's usually an immutable value object. That does not mean your salary cannot change. It means that to change your salary, you need to replace the existing money object with a new money object rather than changing the amount on an exisiting money object. Your relationship can change, but the money object itself does not.

Replace Array with Object

You have an array in which certain elements mean different things. Replace the array with an object that has a field for each element



```
String[] row = new String[3];
row [0] = "Liverpool";
row [1] = "15";
```

```
Performance row = new Performance();
row.setName("Liverpool");
row.setWins("15");
```

Duplicate Observed Data

You have domain data available only in a GUI control, and domain methods need access. Copy the data to a domain object. Set up an observer to synchronize the two pieces of data.

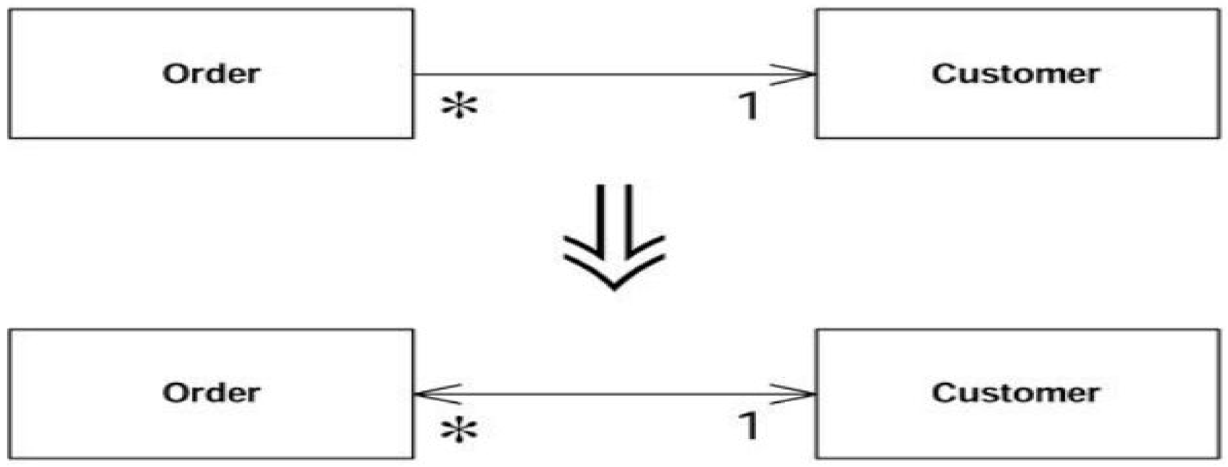
A well-layered system separates code that handles the user interface from code that handles the business logic.

This refactoring technique, which in its classic form is performed using the Observer template, isn’t applicable for web apps, where all classes are recreated between queries to the web server.

Duplicate Observed Data also applies if you use event listeners instead of observer/observable.

Change Unidirectional Association to Bidirectional

You have two classes that need to use each other's features, but there is only a one-way link.Add back pointers, and change modifiers to update both sets

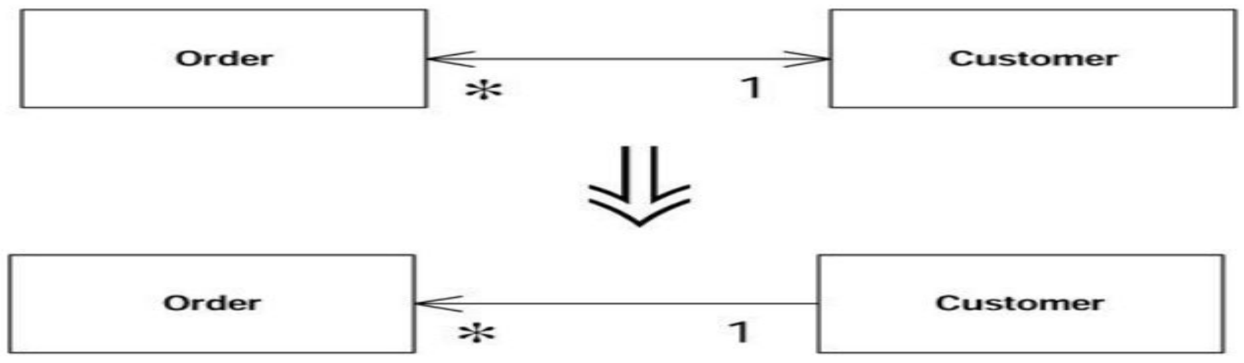


Now I need to decide which class will take charge of the association. I prefer to let one class take charge because it keeps all the logic for manipulating the association in one place. My decision process runs as follows:

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

Change Bidirectional Association to Unidirectional

You have a two-way association but one class no longer needs features from the other. Drop the unneeded end of the association.



Replace Magic Number with Symbolic Constant

You have a literal number with a particular meaning.
Create a constant, name it after the meaning, and replace the number with it.

```
double potentialEnergy(double mass, double height) {
    return mass * 9.81 * height;
}
```

```
double potentialEnergy(double mass, double height) {
    return mass * GRAVITATIONAL_CONSTANT * height;
}
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

Encapsulate Field

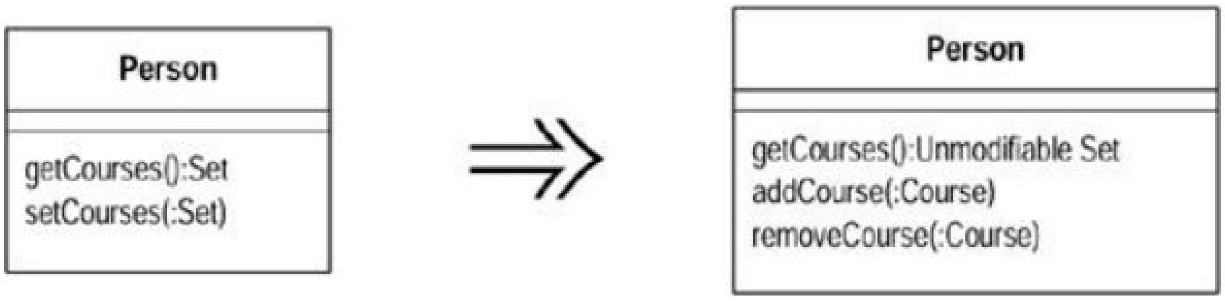
There is a public field.
Make it private and provide accessors.

```
public String _name
```

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

Encapsulate Collection

A method returns a collection.
Make it return a read-only view and provide add/remove methods.



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.

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.

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

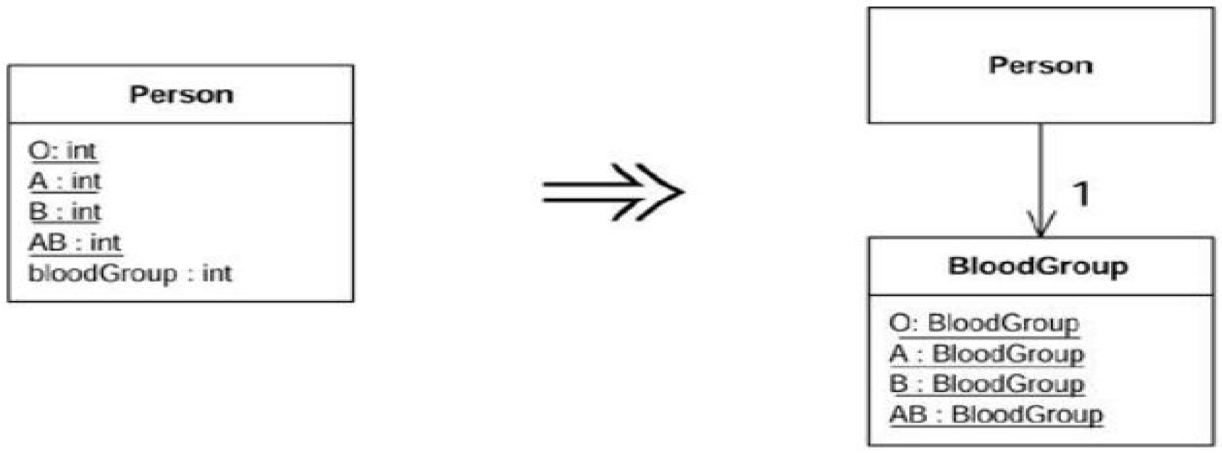
You may wish to use Rename Method to rename the setter.Change it from set to initialize or replace.

Replace Record with Data Class

You need to interface with a record structure in a traditional programming environment.
Make a dumb data object for the record.

Replace Type Code with Class

A class has a numeric type code that does not affect its behavior.
Replace the number with a new class.



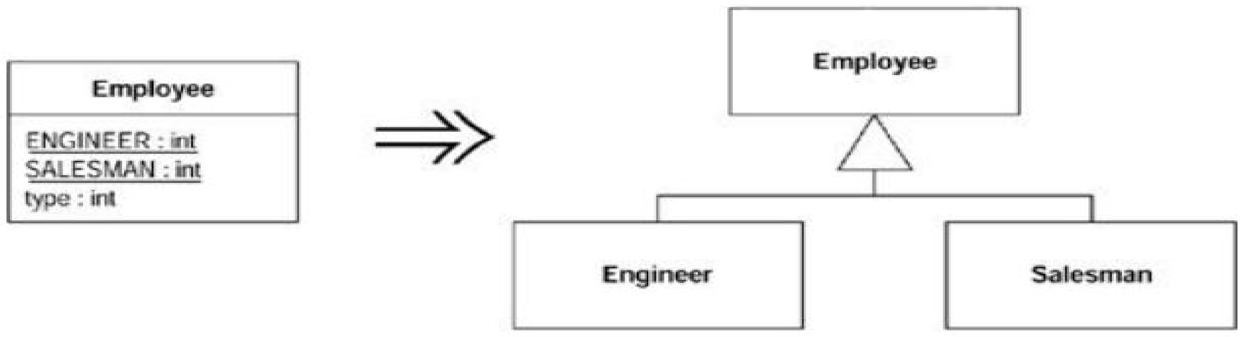
If you replace the number with a class, the compiler can type check on the class. By providing factory methods for the class, you can statically check that only valid instances are created and that those instances are passed on to the correct objects.

Before you do Replace Type Code with Class, however, you need to consider the other type code replacements. Replace the type code with a class only if the type code is pure data, that is, it does not cause different behavior inside a switch statement. For a start Java can only switch on an integer, not an arbitrary class, so the replacement will fail. More important than that, any switch has to be removed with Replace Conditional with Polymorphism. In order for that refactoring, the type code first has to be handled with Replace Type Code with Subclasses or Replace Type Code with State/Strategy.

Even if a type code does not cause different behavior depending on its value, there might be behavior that is better placed in the type code class, so be alert to the value of a Move Method or two.

Replace Type Code with Subclasses

You have an immutable type code that affects the behavior of a class.
Replace the type code with subclasses.



If you have a type code that does not affect behavior, you can use Replace Type Code with Class. However, if the type code affects behavior, the best thing to do is to use polymorphism to handle the variant behavior.

This situation usually is indicated by the presence of case-like conditional statements. These may be switches or if-then-else constructs. In either case they test the value of the type code and then execute different code depending on the value of the type code. Such conditionals need to be refactored with Replace Conditional with Polymorphism.

The simplest way to establish this structure is Replace Type Code with Subclasses. You take the class that has the type code and create a subclass for each type code.

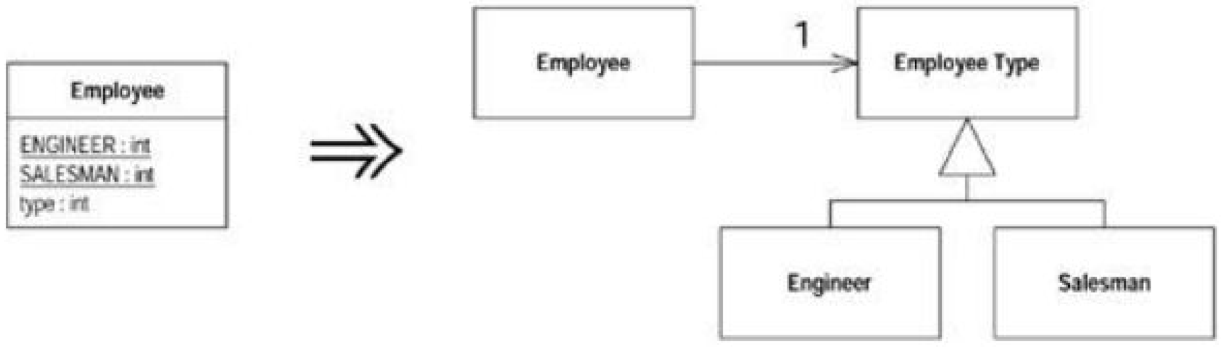
If the type code is passed into the constructor, you need to replace the constructor with a factory method.

The advantage of Replace Type Code with Subclasses is that it moves knowledge of the variant behavior from clients of the class to the class itself. If I add new variants, all I need to do is add a subclass. Without polymorphism I have to find all the conditionals and change those.

However, there are cases in which you can't do this. In the first the value of the type code changes after the object is created. In the second the class with the type code is already subclassed for another reason. In either of these cases you need to use Replace Type Code with State/Strategy.

Replace Type Code with State/Strategy

You have a type code that affects the behavior of a class,but you cannot use subclassing. Replace the type code with a state object.

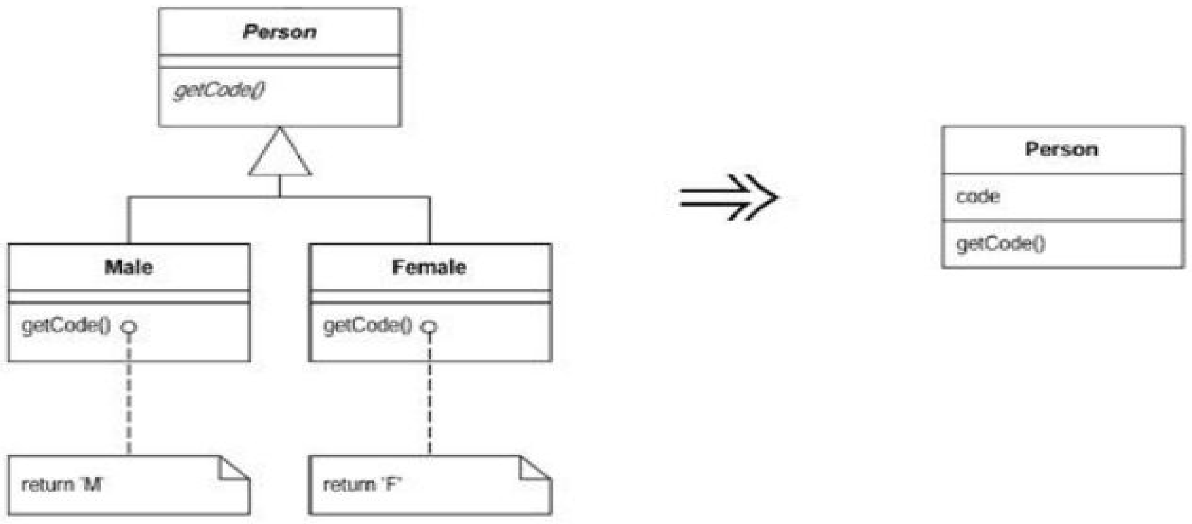


This is similar to Replace Type Code with Subclasses, but can be used if the type code changes during the life of the object or if another reason prevents subclassing.

State and strategy are very similar, so the refactoring is the same whichever you use, and it doesn't really matter. Choose the pattern that better fits the specific circumstances. If you are trying to simplify a single algorithm with Replace Conditional with Polymorphism, strategy is the better term. If you are going to move state-specific data and you think of the object as changing state, use the state pattern.

Replace Subclass with Fields

You have subclasses that vary only in methods that return constant data. Change the methods to superclass fields and eliminate the subclasses.



Although constant methods are useful, a subclass that consists only of constant methods is not doing enough to be worth existing. You can remove such subclasses completely by putting fields in the superclass. By doing that you remove the extra complexity of the subclasses.

Chapter 9. Simplifying Conditional Expressions

Decompose Conditional

You have a complicated conditional (if-then-else) statement.
Extract methods from the condition, then part, and else parts.

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

$\Downarrow$

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

Consolidate Conditional Expression

You have a sequence of conditional tests with the same result.
Combine them into a single conditional expression and extract it.

```
double disabilityAmount() {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
    if (_isPartTime) return 0;
    // compute the disability amount
```

⇓

```
double disabilityAmount() {
    if (isNotEligableForDisability()) return 0;
    // compute the disability amount
```

If you think the checks are really independent and shouldn't be thought of as a single check, don't do the refactoring. Your code already communicates your intention.

Consolidate Duplicate Conditional Fragments

The same fragment of code is in all branches of a conditional expression.
Move it outside of the expression.

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

⇓

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

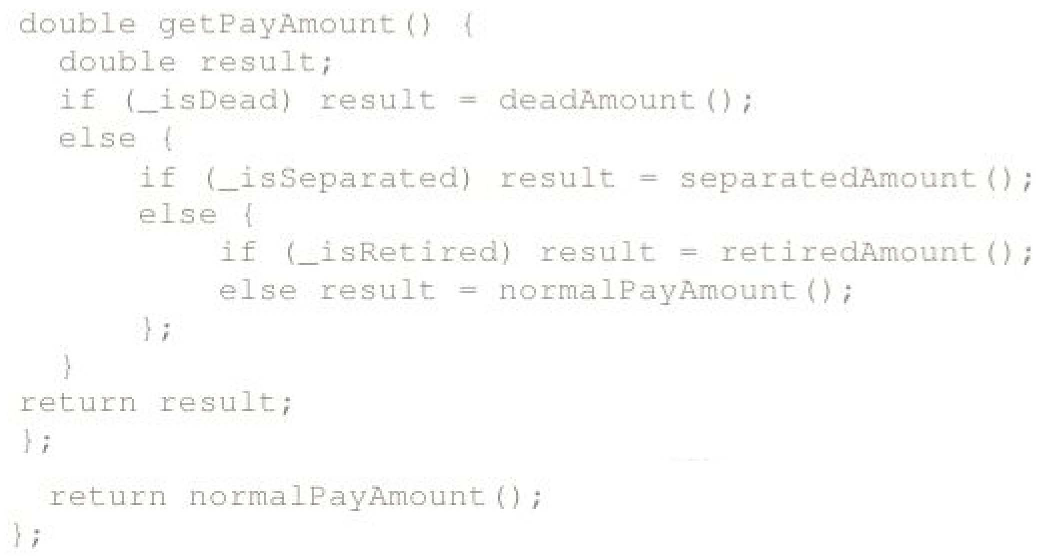
Remove Control Flag

You have a variable that is acting as a control flag for a series of boolean expressions.
Use a break or return instead.

Control flags are more trouble than they are worth. They come from rules of structured programming that call for routines with one entry and one exit point. I agree with (and modern languages enforce) one entry point, but the one exit point rule leads you to very convoluted conditionals with these awkward flags in the code. This is why languages have break and continue statements to get out of a complex conditional.

Replace Nested Conditional with Guard Clauses

A method has conditional behavior that does not make clear the normal path of execution.
Use guard clauses for all the special cases.



```
double getPayAmount() {
   double result;
   if (_isDead) result = deadAmount();
   else {
       if (_isSeparated) result = separatedAmount();
       else {
           if (_isRetired) result = retiredAmount();
           else result = normalPayAmount();
       };
   }
return result;
};

  return normalPayAmount();
};
```

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 form is a situation in which one answer from the conditional indicates normal behavior and the other indicates an unusual condition.

These kinds of conditionals have different intentions, and these intentions should come through in the code. If both are part of normal behavior, use a condition with an if and an else leg. If the condition is an unusual condition, check the condition and return if the condition is true. This kind of check is often called a guard clause.The guard clause either returns, or throws an exception.

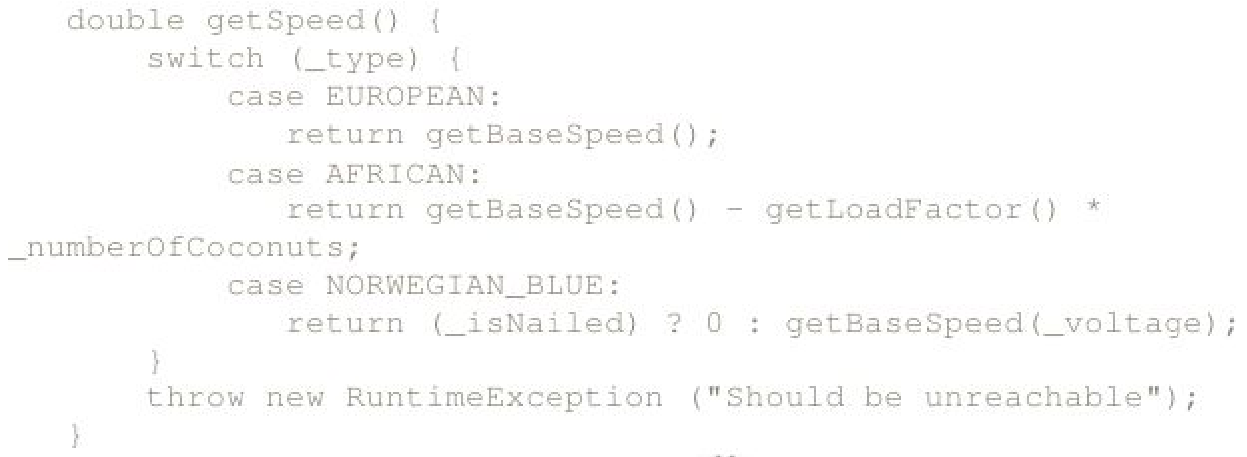
The key point about Replace Nested Conditional with Guard Clauses is one of emphasis. If you are using an if-then-else construct you are giving equal weight to the if leg and the else leg. This communicates to the reader that the legs are equally likely and important. Instead the guard clause says, "This is rare, and if it happens, do something and get out."

The particular cases that cause an exit of the method would be placed at the beginning of the method and act as guards in a way that avoids continuing through the satisfactory flow of the method.

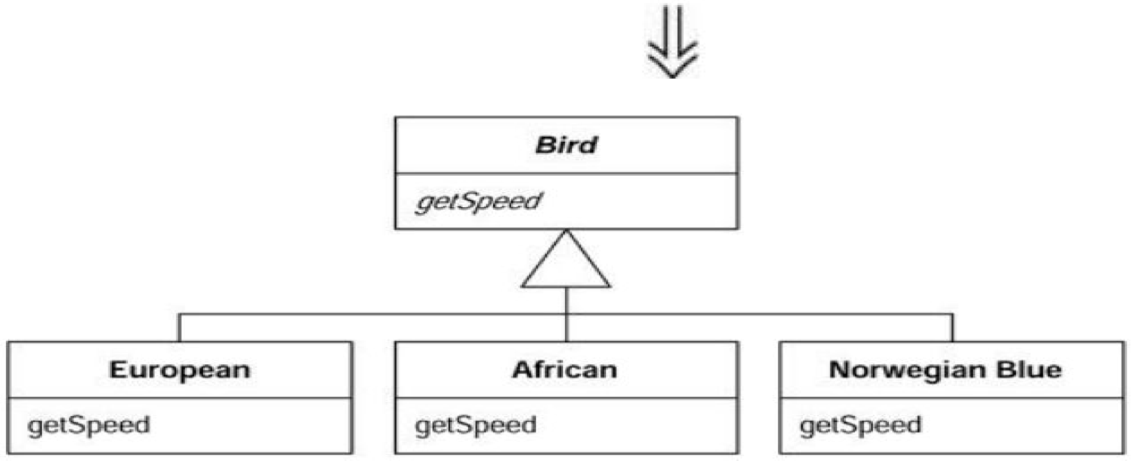
In this way, the method is easy to read since the particular cases are at the beginning of the same and the case of satisfactory flow use is the body of the method.

Replace Conditional with Polymorphism

You have a conditional that chooses different behavior depending on the type of an object.
Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.



```
   double getSpeed() {
       switch (_type) {
           case EUROPEAN:
              return getBaseSpeed();
           case AFRICAN:
              return getBaseSpeed() - getLoadFactor() *
_numberOfCoconuts;
           case NORWEGIAN_BLUE:
              return (_isNailed) ? 0 : getBaseSpeed(_voltage);
       }
       throw new RuntimeException ("Should be unreachable");
   }
```



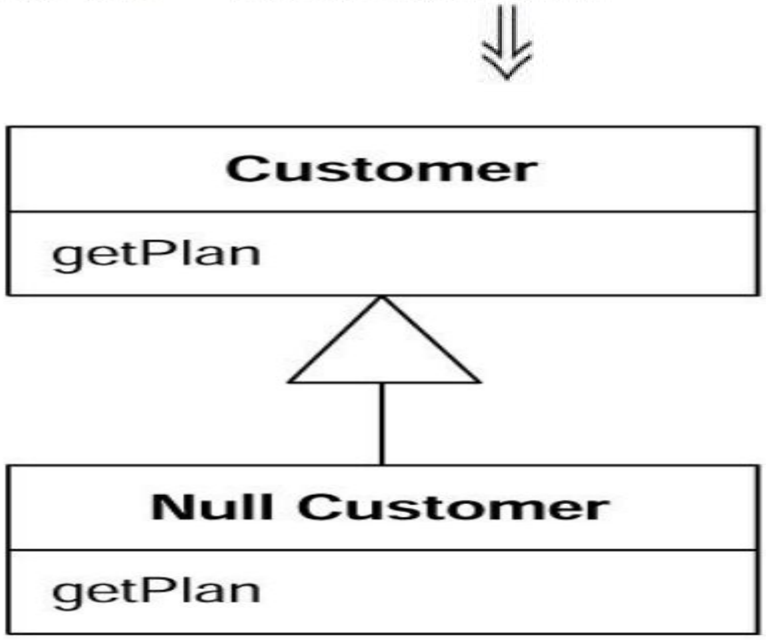
To create the inheritance structure you have two options: Replace Type Code with Subclasses and Replace Type Code with State/Strategy. Subclasses are the simplest option, so you should use them if you can. If you update the type code after the object is created, however, you cannot use subclassing and have to use the state/strategy pattern. You also need to use the state/strategy pattern if you are already subclassing this class for another reason. Remember that if several case statements are switching on the same type code, you only need to create one inheritance structure for that type code.

You may need to make some private members of the superclass protected in order to do this.

Introduce Null Object

You have repeated checks for a null value.
Replace the null value with a null object.

```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```



The essence of polymorphism is that instead of asking an object what type it is and then invoking some behavior based on the answer, you just invoke the behavior. The object, depending on its type, does the right thing.

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

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

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.

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();
}
```

Indeed assertions usually are removed for production code. It is therefore important to signal something is an assertion.

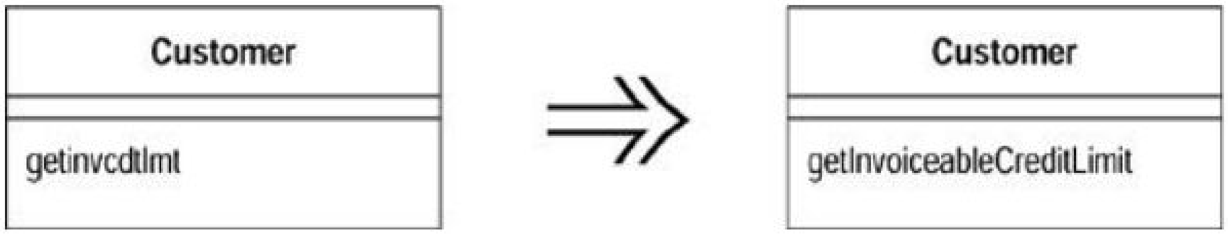
Assertions should be easily removable.Have an assert class that you can use for assertion behavior.

Beware of overusing assertions. Use assertions only to check things that need to be true.Always ask whether the code still works if an assertion fails. If the code does work, remove the assertion.

Chapter 10. Making Method Calls Simpler

Rename Method

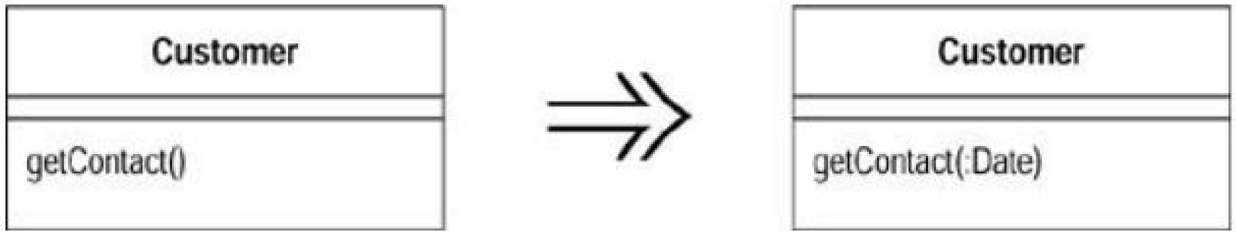
The name of a method does not reveal its purpose.
Change the name of the method.



Remember , your code is for human first and computer second.

Add Parameter

A method needs more information from its caller.
Add a parameter for an object that can pass on this information.



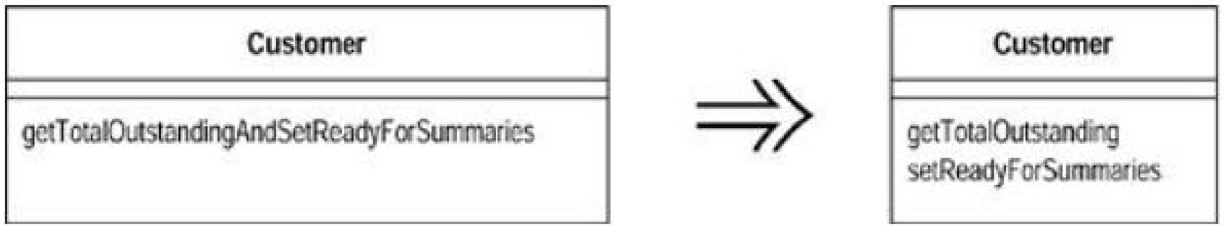
The motivation is simple. You have to change a method, and the change requires information that wasn't passed in before, so you add a parameter.

Remove Parameter

A parameter is no longer used by the method body.
Remove it.

Separate Query from Modifier

You have a method that returns a value but also changes the state of an object.
Create two methods, one for the query and one for the modification.



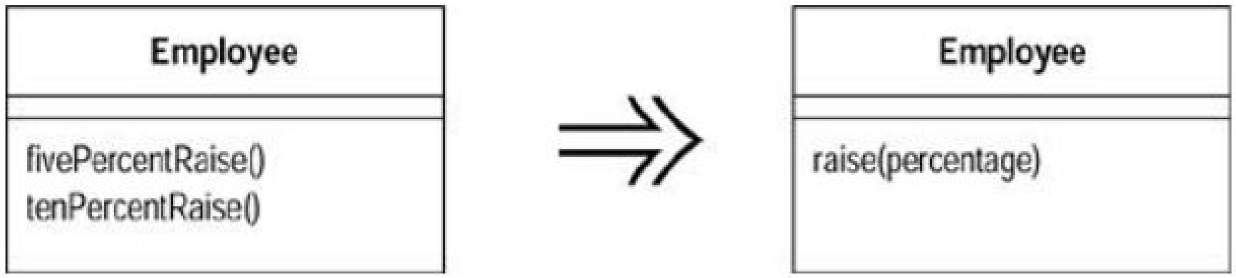
When you have a function that gives you a value and has no observable side effects, you have a very valuable thing. You can call this function as often as you like. You can move the call to other places in the method. In short, you have a lot less to worry about.

A good rule to follow is to say that any method that returns a value should not have observable side effects. Some programmers treat this as an absolute rule [Meyer]. I'm not 100 percent pure on this (as on anything), but I try to follow it most of the time, and it has served me Well.

You'll note I use the phrase observable side effects. A common optimization is to cache the value of a query in a field so that repeated calls go quicker. Although this changes the state of the object with the cache, the change is not observable.

Parameterize Method

Several methods do similar things but with different values contained in the method body. Create one method that uses a parameter for the different values.



Replace Parameter with Explicit Methods

You have a method that runs different code depending on the values of an enumerated parameter.
Create a separate method for each value of the parameter.

```
void setValue (String name, int value) {
    if (name.equals("height"))
        _height = value;
    if (name.equals("width"))
        _width = value;
    Assert.shouldNeverReachHere();
}
```

⇓

```
void setHeight(int arg) {
    _height = arg;
}
void setWidth (int arg) {
    _width = arg;
}
```

Preserve Whole Object

You are getting several values from an object and passing these values as parameters in a method call.
Send the whole object instead.

```
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

⇓

```
withinPlan = plan.withinRange(daysTempRange());
```

```
int low = daysTempRange().getLow();
```

Preserve Whole Object often makes code more readable.

There is a down side. When you pass in values, the called object has a dependency on the values, but there isn't any dependency to the object from which the values were extracted. Passing in the required object causes a dependency between the required object and the called object. If this is going to mess up your dependency structure, don't use Preserve Whole Object.

That a called method uses lots of values from another object is a signal that the called method should really be defined on the object from which the values come. When you are considering Preserve Whole Object, consider Move Method as an alternative.

You may not already have the whole object defined. In this case you need Introduce Parameter Object.

Replace Parameter with Method

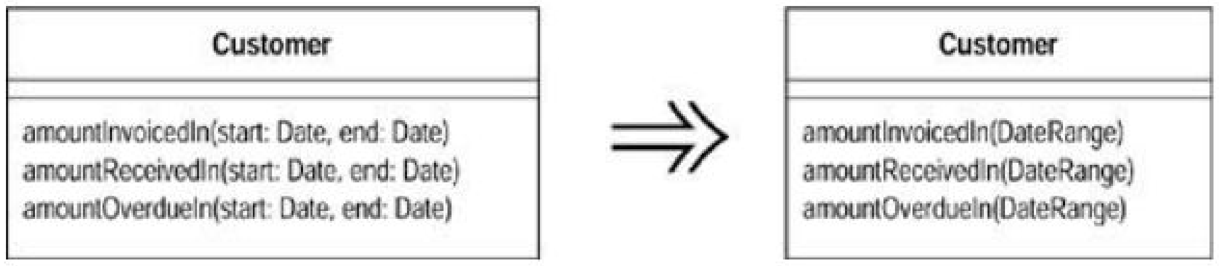
An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method.
Remove the parameter and let the receiver invoke the method.

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice, discountLevel);
```

```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice (basePrice);
```

Introduce Parameter Object

You have a group of parameters that naturally go together.
Replace them with an object.


Customer
amountInvoicedIn(start: Date, end: Date)
amountReceivedIn(start: Date, end: Date)
amountOverdueIn(start: Date, end: Date)
Customer
amountInvoicedIn(DateRange)
amountReceivedIn(DateRange)
amountOverdueIn(DateRange)

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

Remove Setting Method

A field should be set at creation time and never altered.
Remove any setting method for that field.

Hide Method

A method is not used by any other class.
Make the method private.

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);
 }
```

The most obvious motivation for Replace Constructor with Factory Method comes with replacing a type code with subclassing.

Encapsulate Downcast

A method returns an object that needs to be downcasted by its callers.
Move the downcast to within the method.

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

⇓

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

Downcasting may be a necessary evil, but you should do it as little as possible.

Replace Error Code with Exception

A method returns a special code to indicate an error.
Throw an exception instead.

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

⇓

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

If the exception is unchecked, adjust the callers to make the appropriate check before calling the method. Compile and test after each such change.

If the exception is checked, adjust the callers to call the method in a try block.

Replace Exception with Test

You are throwing a checked exception on a condition the caller could have checked first. Change the caller to make the test first.

```
double getValueForPeriod (int periodNumber) {
    try {
        return _values[periodNumber];
    } catch (ArrayIndexOutOfBoundsException e) {
        return 0;
    }
}
```

⇓

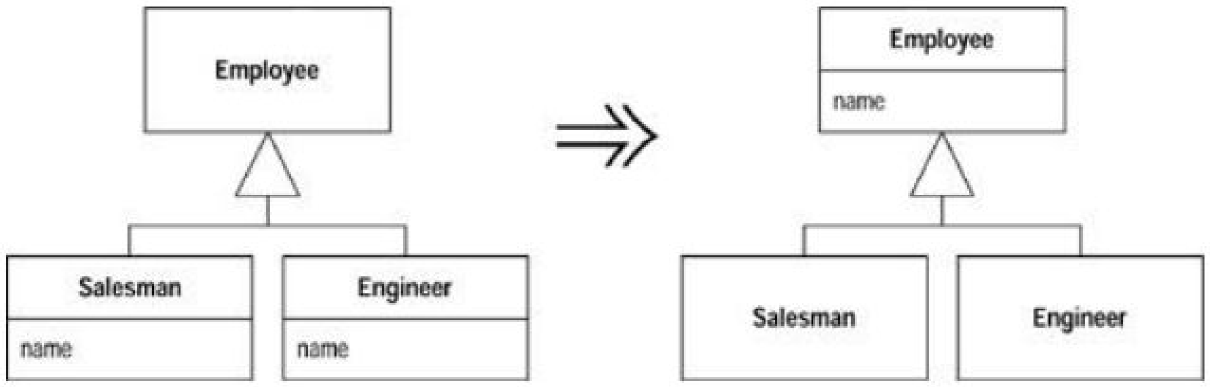
```
double getValueForPeriod (int periodNumber) {
    if (periodNumber >= _values.length) return 0;
    return _values[periodNumber];
}
```

Exceptions should be used for exceptional behavior—behavior that is an unexpected error. They should not act as a substitute for conditional tests. If you can reasonably expect the caller to check the condition before calling the operation, you should provide a test, and the caller should use it.

Chapter 11 : Dealing with Generalization

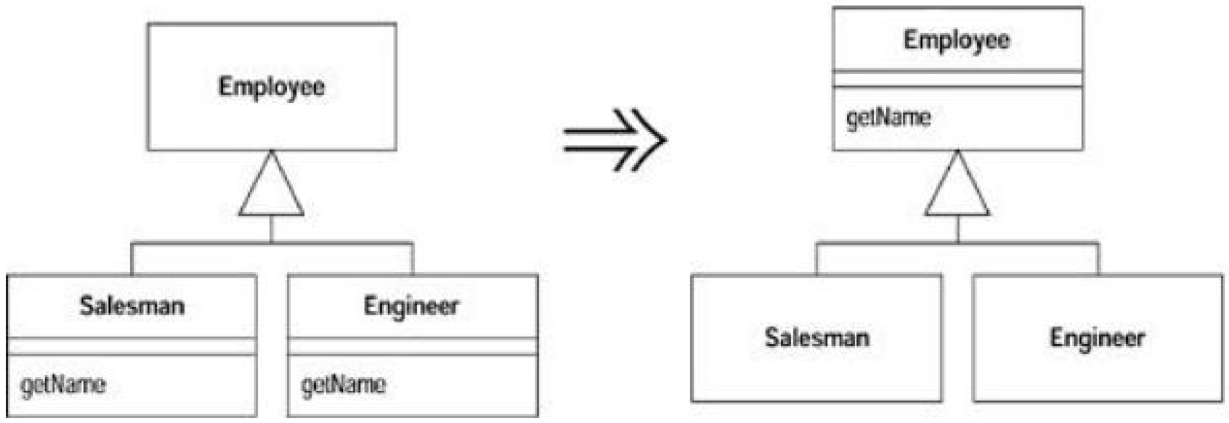
Pull Up Field

Two subclasses have the same field.
Move the field to the superclass.



Pull Up Method

You have methods with identical results on subclasses.
Move them to the superclass.



Pull Up Constructor Body

You have constructors on subclasses with mostly identical bodies.
Create a superclass constructor; call this from the subclass methods.

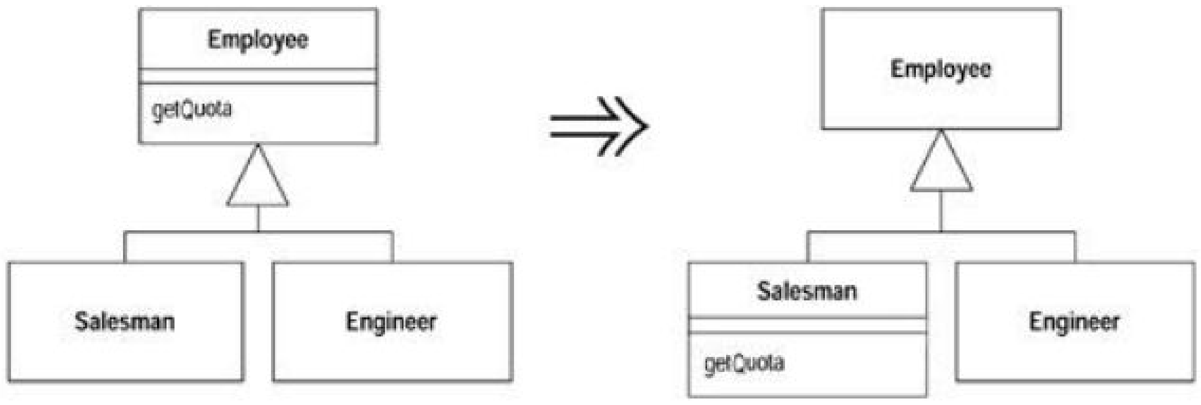
```
class Manager extends Employee...
  public Manager (String name, String id, int grade) {
      _name = name;
      _id = id;
      _grade = grade;
  }
```

⇓

```
  public Manager (String name, String id, int grade) {
      super (name, id);
      _grade = grade;
  }
```

Push Down Method

Behavior on a superclass is relevant only for some of its subclasses.
Move it to those subclasses.

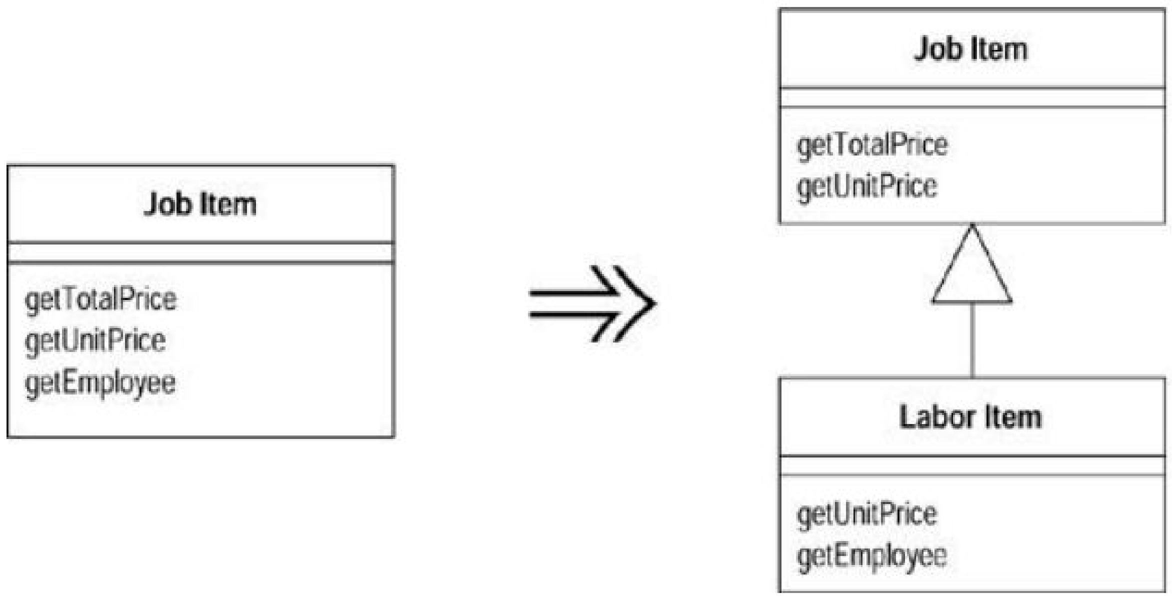


Push Down Field

A field is used only by some subclasses.
Move the field to those subclasses.

Extract Subclass

A class has features that are used only in some instances.
Create a subclass for that subset of features.



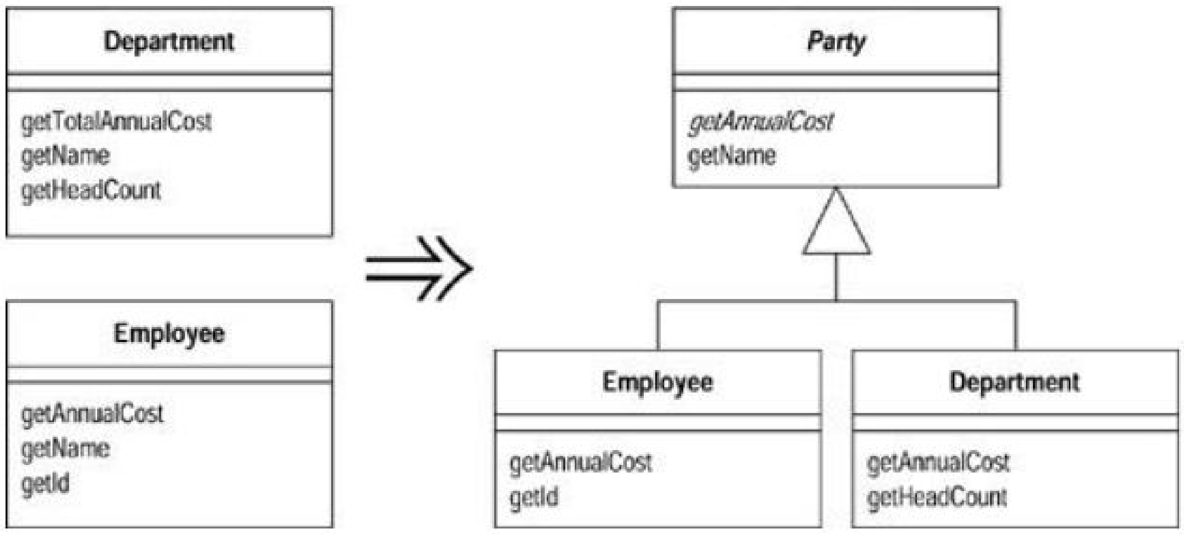
The main trigger for use of Extract Subclass is the realization that a class has behavior used for some instances of the class and not for others.

The main alternative to Extract Subclass is Extract Class. This is a choice between delegation and inheritance. Extract Subclass is usually simpler to do, but it has limitations. You can't change the class-based behavior of an object once the object is created. You can change the class-based behavior with Extract Class simply by plugging in different components.

Extract Superclass

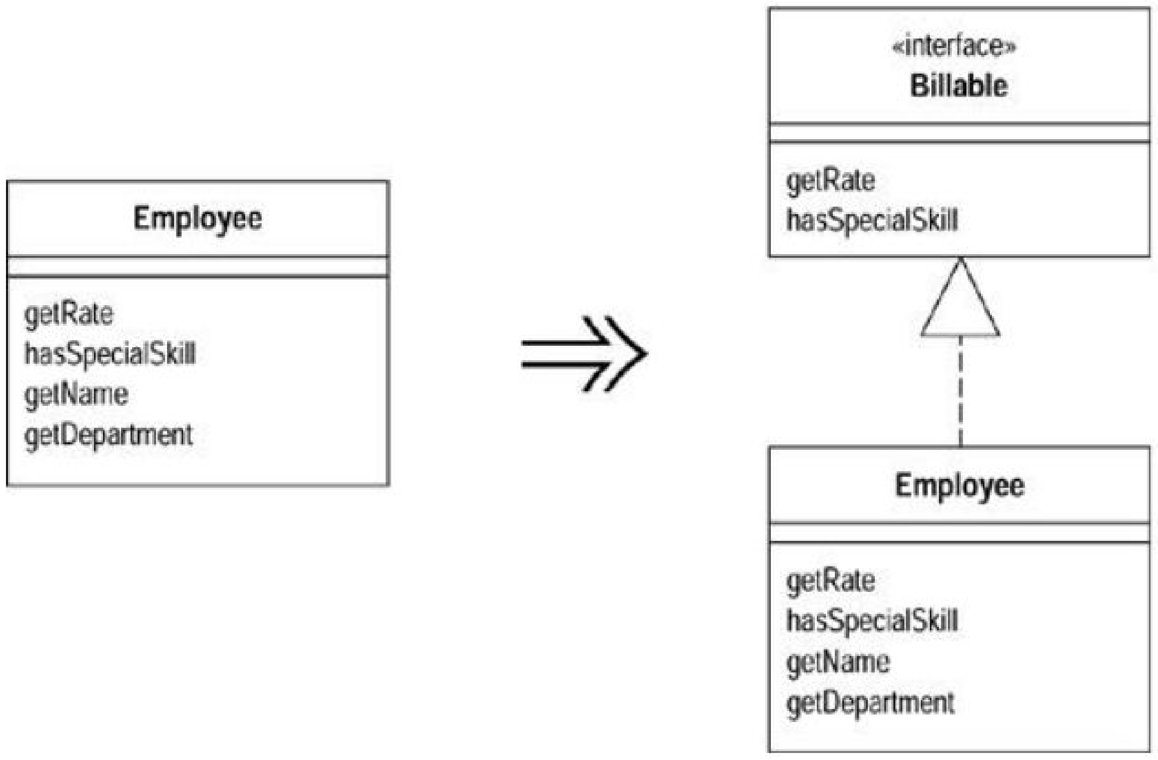
You have two classes with similar features.

Create a superclass and move the common features to the superclass.



Extract Interface

Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common.
Extract the subset into an interface.

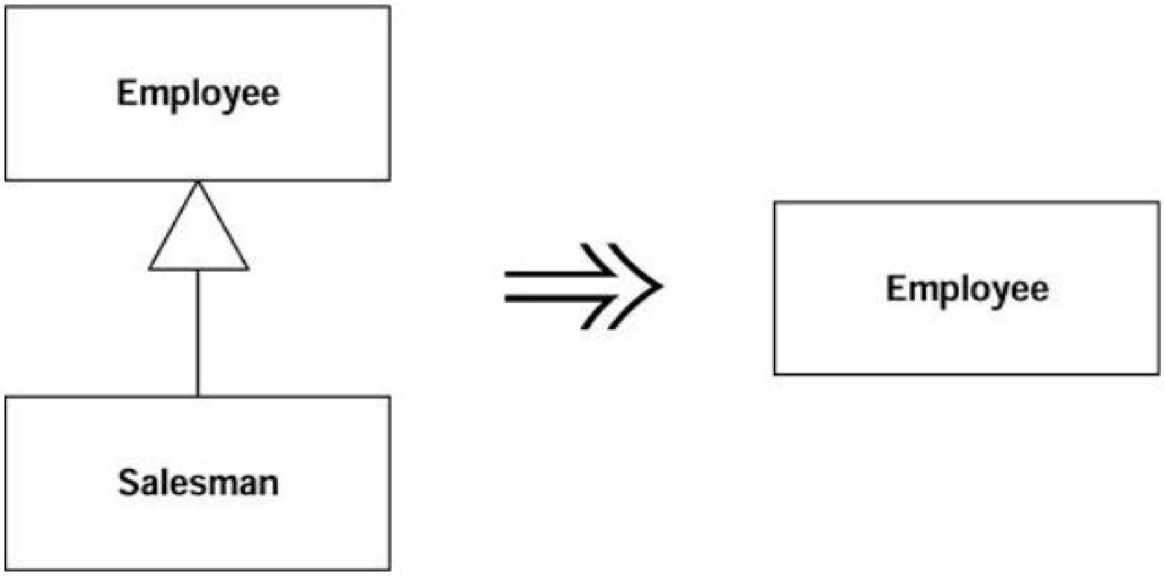


There is some similarity between Extract Superclass and Extract Interface. Extract Interface can only bring out common interfaces, not common code. Using Extract Interface can lead to smelly duplicate code. You can reduce this problem by using Extract Class to put the behavior into a component and delegating to it. If there is substantial common behavior Extract Superclass is simpler, but you do only get to have one superclass.

Interfaces are good to use whenever a class has distinct roles in different situations. Use Extract Interface for each role.

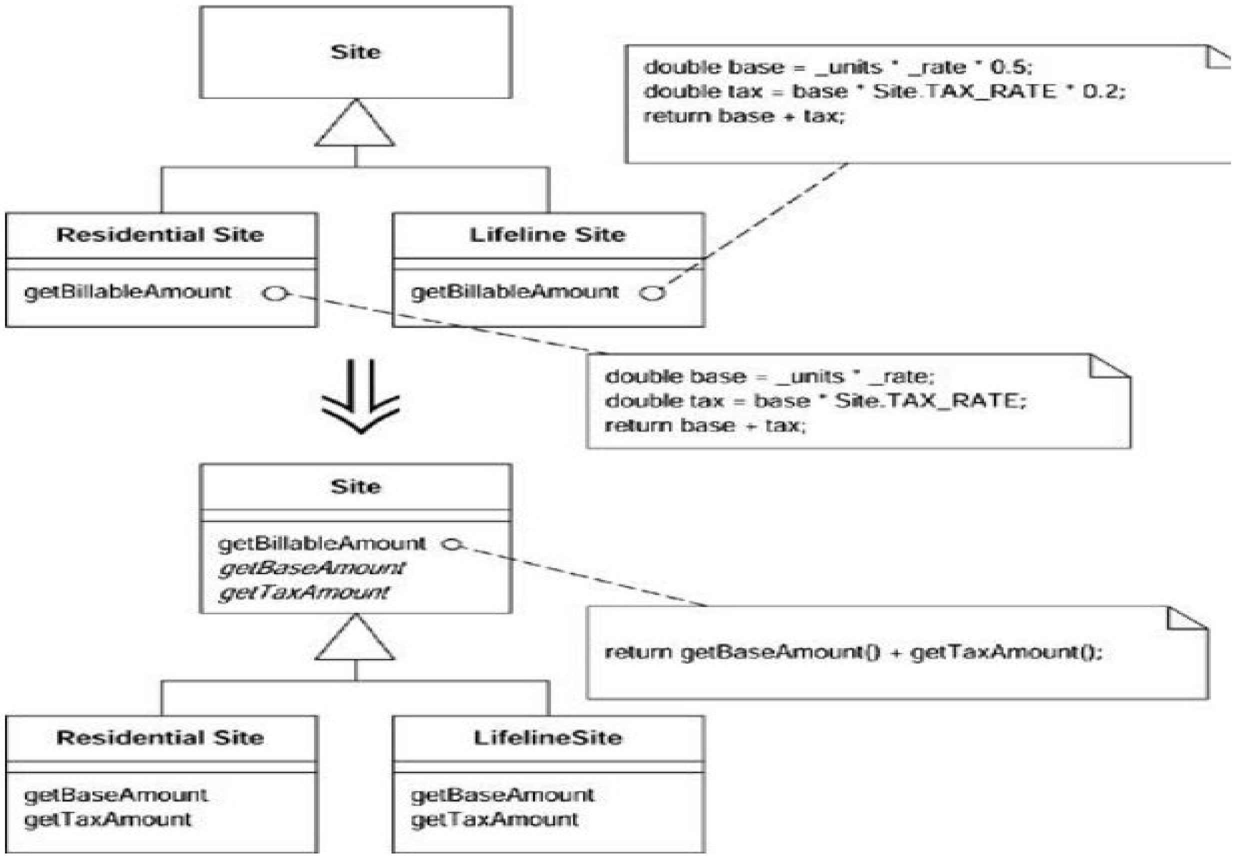
Collapse Hierarchy

A superclass and subclass are not very different.
Merge them together.



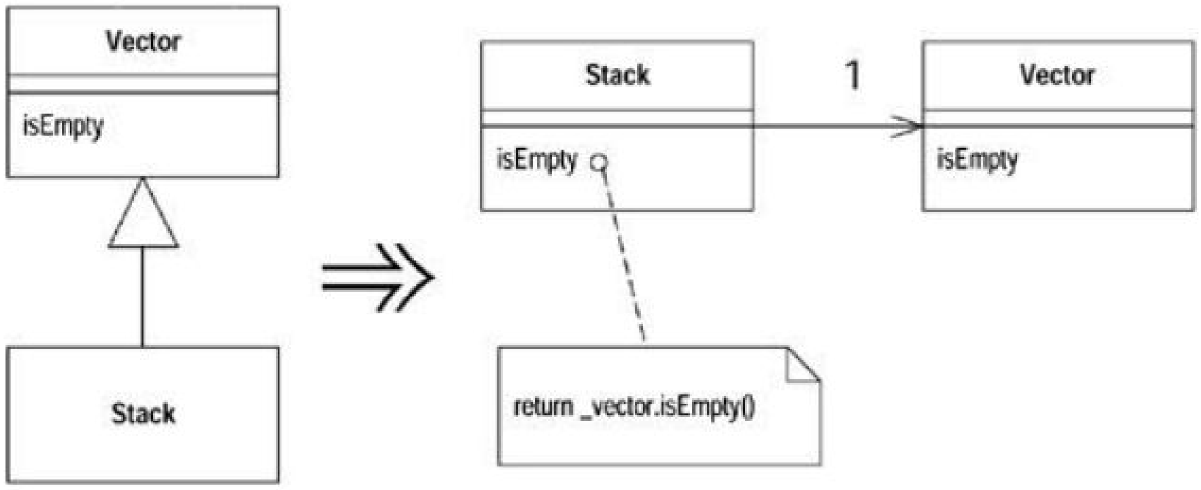
Form Template Method

You have two methods in subclasses that perform similar steps in the same order, yet the steps are different. Get the steps into methods with the same signature, so that the original methods become the same. Then you can pull them up.



Replace Inheritance with Delegation

A subclass uses only part of a superclasses interface or does not want to inherit data. Create a field for the superclass, adjust methods to delegate to the superclass, and remove the Subclassing.

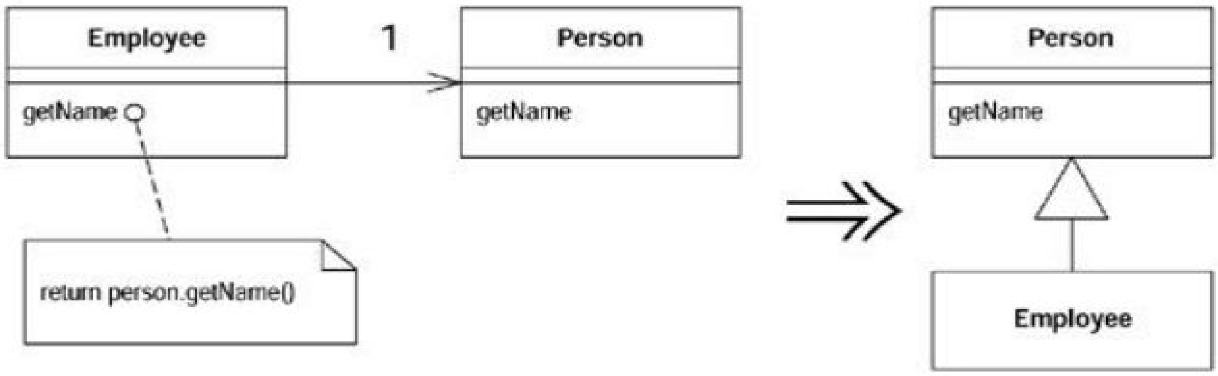


You may find that there are protected superclass methods that don't make much sense with the subclass.

By using delegation instead, you make it clear that you are making only partial use of the delegated class. You control which aspects of the interface to take and which to ignore. The cost is extra delegating methods that are boring to write but are too simple to go wrong.

Replace Delegation with Inheritance

You're using delegation and are often writing many simple delegations for the entire interface.
Make the delegating class a subclass of the delegate.



If you find yourself using all the methods of the delegate and are sick of writing all those simple delegating methods, you can switch back to inheritance pretty easily.

If you aren't using all the methods of the class to which you are delegating, you shouldn't use Replace Delegation with Inheritance, because a subclass should always follow the interface of the superclass. If the delegating methods are tiresome, you have other options. You can let the clients call the delegate themselves with Remove Middle Man.

Another situation to beware of is that in which the delegate is shared by more than one object and is mutable. In this case you can't replace the delegate with inheritance because you'll no longer share the data. Data sharing is a responsibility that cannot be transferred back to inheritance.When the object is immutable, data sharing is not a problem, because you can just copy and nobody can tell.

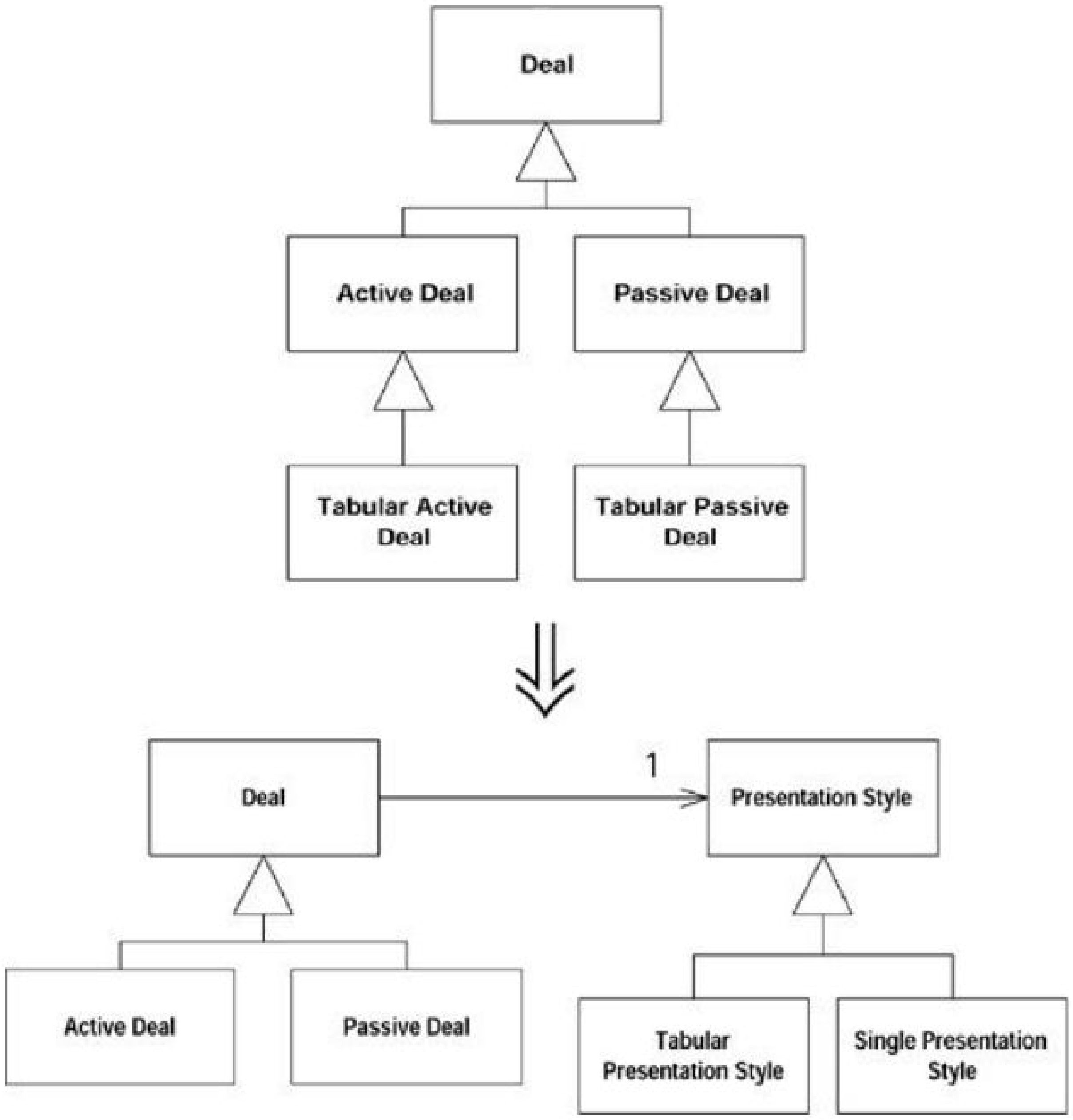
Chapter 12. Big Refactorings

The big refactorings require a degree of agreement among the entire programming team that isn't needed with the smaller refactorings. The big refactorings set the direction for many, many changes. The whole team has to recognize that one of the big refactorings is "in play" and make their moves accordingly.

Tease Apart Inheritance

You have an inheritance hierarchy that is doing two jobs at once.

Create two hierarchies and use delegation to invoke one from the other.

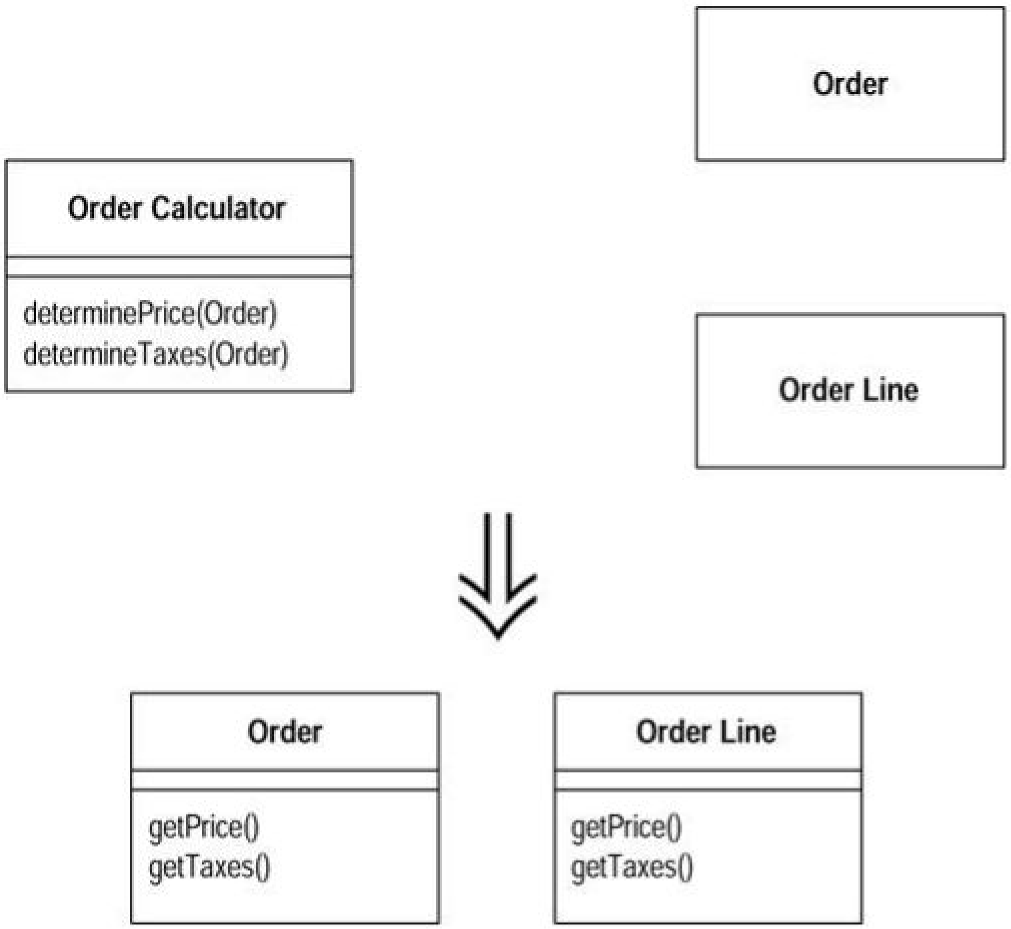


You can easily spot a single inheritance hierarchy that is doing two jobs. If every class at a certain level in the hierarchy has subclasses that begin with the same adjective, you probably are doing two jobs with one hierarchy.

Convert Procedural Design to Objects

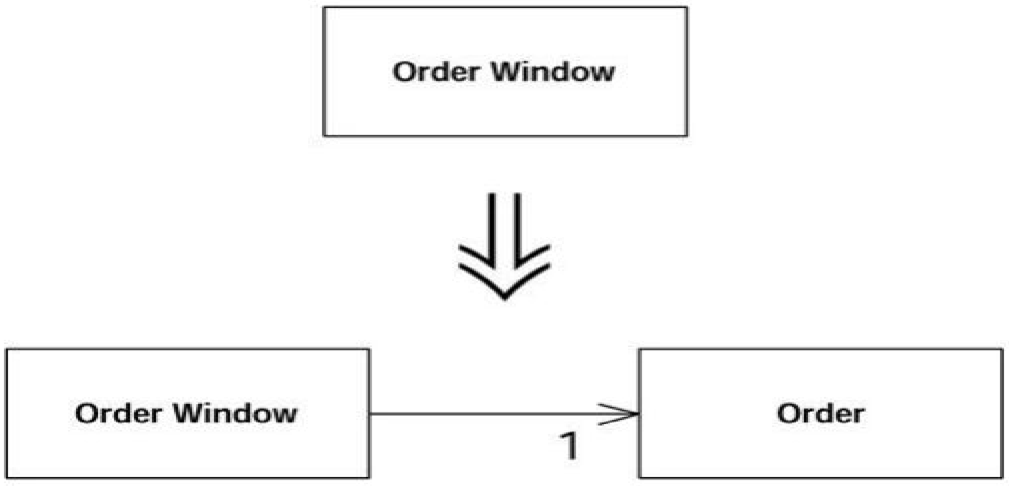
You have code written in a procedural style.

Turn the data records into objects, break up the behavior, and move the behavior to the objects.



Separate Domain from Presentation

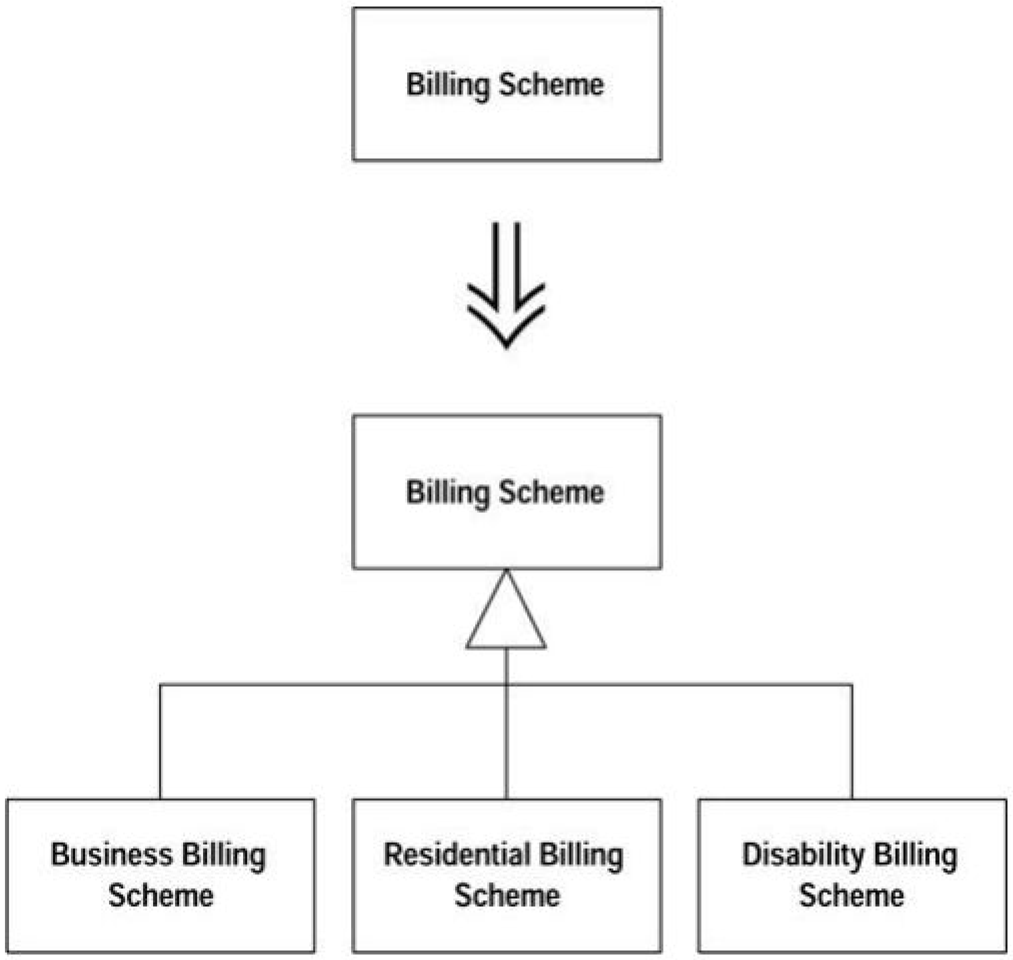
You have GUI classes that contain domain logic.
Separate the domain logic into separate domain classes



Extract Hierarchy

You have a class that is doing too much work, at least in part through many conditional statements.

Create a hierarchy of classes in which each subclass represents a special case.



The strategy here works only if your conditional logic remains static during the life of the object. If not, you may have to use Extract Class before you can begin separating the cases from each other.