# Java Project

MyJavaProject’s refactoring package has some example that I tried for refactoring long method having switch-cases.

# Why Refactoring is required?

Refactoring changes the programs in small steps. If you make a mistake, it is easy to find the bug.  
  
Refactoring certainly will make software go more slowly, but it also makes the software more amenable to performance tuning. 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.  
  
Break big method into smaller chunks of methods  
Improve variable names in new methods  
  
I prefer short, well-named methods for several reasons. First, it increases the chances that other methods can use a method when the method is finely grained. Second, it allows the higher-level methods to read more like a series of comments. Overriding also is easier when the methods are finely grained.

Move newly created method to appropriate class

See what data a new method is using and move that method to a new class accordingly  
 e.g. a new method getCharge is using Rental.  
 So, this method should be moved to Rental class.  
 rename method names as you move to other classes.  
  
Replace local/temp variables with method calls

Methods are reusable and gives cleaner design than variables.

## What's the problem with local/temp variables?

The problem is local variables: parameters passed into the original method and temporaries declared within the original method. Local variables are only in scope in that method, so when I use Extract Method, these variables cause me extra work. In some cases they even prevent me from doing the refactoring at all.

## When to use local/temp variables?

You can introduce local/temp variable to break very complex expressions which are hard to read.  
 e.g.  
if ( (platform.toUpperCase().indexOf("MAC") > -1) && (browser.toUpperCase().indexOf("IE") > -1) && wasInitialized() && resize > 0 ) {...}  
  
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) { ... }

Replacing the CONDITIONAL Logic (if-else if/switch-case) with Inheritance and Polymorphism

## You can replace conditional logic with

* state/strategy pattern
* chain of responsibility  
  - Chain of responsibility actually moves through entire chain even though one member of a chain could process the request (this is good for if-if-if conditions)  
  - You can improve a pattern a bit by breaking a chain as soon as one member a chain processed the request (this is good for if-elseif-elseif conditions)

Read design patterns to understand the difference between state and strategy patterns.

## Reviewing MyJavaProject’s refactoring example:

See moved Rental's getCharge(...) . it has switch-case.

You have 2 choices to break this switch-case using design pattern

* either create an inheritance of using Movie and its children RegularMovie, NewReleaseMove, ChildrenMovie  
   or
* create an inheritance of PriceCode with its children RegularPriceCode, NewReleasePriceCode, ChildrenPriceCode ane inject PriceCode into Movie.

You need to be very careful while choosing the inheritance for the right object. Which one is more vulnerable towards increase in subclasses (Movie or PriceCode).

If you see switch-case, it is based on a property PriceCode of a Movie class. So, inheritance should be created for PriceCode and not movie.

As behaviour of getCharge method is changing due to change in Movie class' property PriceCode (state change of Movie class), we will end up with creating a state pattern (and not strategy pattern).  
Create inheritance of PriceCode with its children RegularPriceCode, NewReleasePriceCode, ChildrenPriceCode ane inject PriceCode into Movie.  
  
Here, I have done wrong in my example. I have created an inheritance of Movie instead of PriceCode.

Remove Assignments to Parameters

## Pass by value vs Pass by ref

With pass by value, any change to the parameter is not reflected in the calling routine.  
  
 int discount (int inputVal, int quantity, int yearToDate) {  
 if (inputVal > 50) inputVal -= 2 }  
  
 change to  
  
 int discount (int inputVal, int quantity, int yearToDate) {  
 int result = inputVal;  
 if (inputVal > 50) result -= 2;  
 }  
  
 void aMethod(Object foo) {  
 foo.modifyInSomeWay(); // that's OK  
 foo = anotherObject; // trouble - eventhough foo is a reference it is actually passed by value. changing foo will not change caller's foo.  
 }

Replace Method with Method Object  
  
 class Order {  
 double price() {  
 double primaryBasePrice;  
 double secondaryBasePrice;  
 double tertiaryBasePrice;  
 // long computation;  
 ...  
 }

You have a long method that uses local variables in such a way that you cannot apply 'Extract Method - breaking long method in smaller methods'.

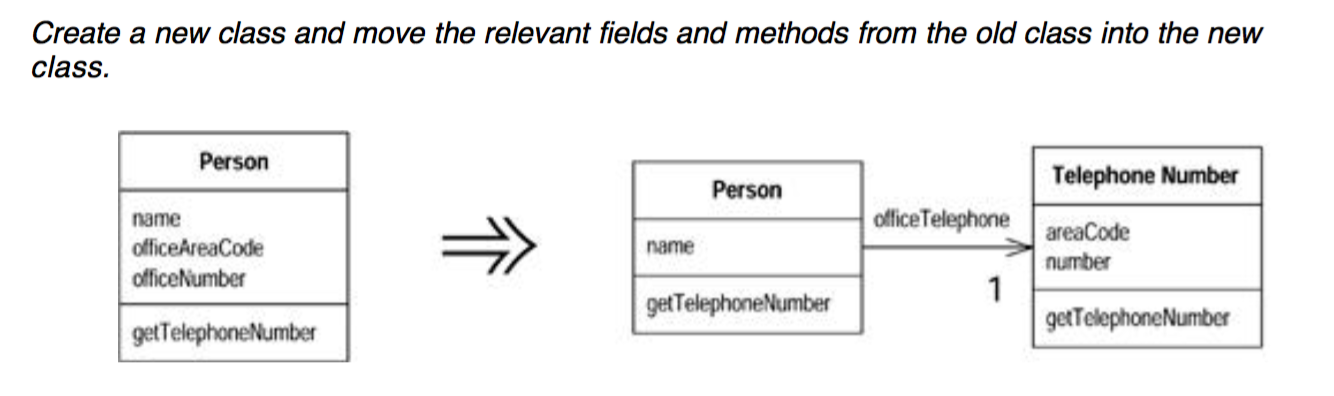
Turn the method into its own object so that all the local variables become fields on that object. You can then decompose the method into other methods on the same object.  
  
 class Order {  
 double price() {  
 return new PriceCalculator(...).calculate()  
 }  
 }  
  
 class PriceCalculator {  
 double primaryBasePrice;  
 double secondaryBasePrice;  
 double tertiaryBasePrice;  
 // long computation;  
 ...  
 PriceCalculator(double primaryBasePrice, ......) {  
 this.primaryBasePrice = primaryBasePrice;  
 ....  
 }  
  
 double calculate() {...}  
 }  
  
- Create a new class with a name from method name.

* Make all method variables as newly created class variables.
* Create a constructor and set these variables using constructor.
* Add a method to do the actual calculation.

Extract class or Inline class  
  
One class should have only one responsibility. If you see a lot of methods in a class, probably it's time to move those method in other existing classes or creating new classes.

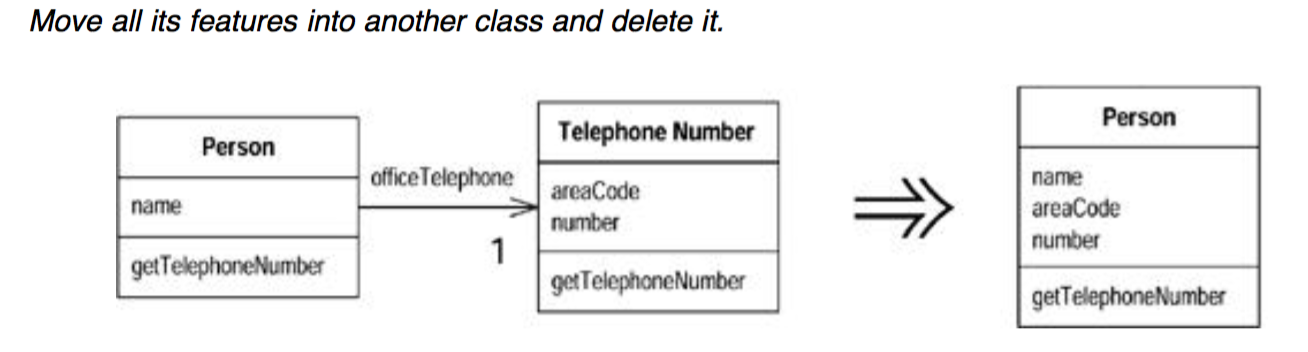
## When to extract a class?

A class that is too big to understand easily. You need to consider where it can be split, and you split it. A good sign is that a subset of the data and a subset of the methods seem to go together. Other good signs are subsets of data that usually change together or are particularly dependent on each other.



## When to inline a class?

A class isn't doing very much.



Substitute Algorithm (Replace current algorithm by a new and better one)

Replace the body of the method with the new algorithm.  
  
 String foundPerson(String[] people) {  
 for (int i = 0; i < people.length; i++) {  
 if (people[i].equals("Don")) {  
 return "Don";  
 }  
 if (people[i].equals("John")) {  
 return "John";  
 }  
 if (people[i].equals("Kent")) {  
 return "Kent";  
 }  
 }  
 return "";  
 }  
  
  
 String foundPerson(String[] people) {  
 List candidates = Arrays.asList(new String[]{"Don", "John", "Kent"}); // see this..... instead of using so many if conditions, we converted code that uses just one if condition  
  
 for (int i = 0; i < people.length; i++)  
 if (candidates.contains(people[i])) return people[i];  
  
 return "";  
 }

Moving fields and methods from one class to another class

Moving methods is the bread and butter of refactoring. I move methods when classes have too much behavior or when classes are collaborating too much and are too highly coupled. By moving methods around, I can make the classes simpler and they end up being a more crisp implementation of a set of responsibilities.

I usually look through the methods on a class to find a method that seems to reference another object more than the object it lives on. A good time to do this is after I have moved some fields.  
  
 e.g.  
 getCharge(Rental rental) method was originally in Customer class, but it is using Rental object mainly, so I moved this method to Rental class.  
 Similarly, you should think for fields also.  
  
 class Account {  
 private AccountType accountType;  
 private int daysOverdrawn;  
  
 double overdraftCharge() {  
 if (accountType.isPremium()) { // see this. behavior of the method is changing based on AccountType's field. So, this should be moved to AccountType class. You may pass Account object to AccountType to use its daysOverdrawn property.  
 double result = 10;  
 if (daysOverdrawn > 7) result += (daysOverdrawn - 7) \*  
 0.85;  
 return result;  
 }  
 else return daysOverdrawn \* 1.75;  
 }  
  
 double bankCharge() {  
 double result = 4.5;  
 if (daysOverdrawn > 0) result += overdraftCharge(); {  
 return result;  
 }  
 }  
 }  
  
 Improved code  
  
 class AccountType {  
 double overdraftCharge(Account account) {  
 if (isPremium()) {  
 double result = 10;  
 if (account.getDaysOverdrawn() > 7)  
 result += (account.getDaysOverdrawn() - 7) \* 0.85;  
 return result;  
 } else return account.getDaysOverdrawn() \* 1.75;  
 }  
 }

# When to Hide Delegate and When to Keep?

## When to Hide Delegate?

**class** Person {  
 Department **department**;  
  
 **public** Department getDepartment() {  
 **return department**;  
 }  
  
 **public void** setDepartment(Department dept) {  
 **department** = dept;  
 }  
}

**class** Department {  
 **private** String **chargeCode**;  
 **private** Person **manager**;  
  
 **public** Department(Person manager) {  
 **this.manager** = manager;  
 }  
  
 **public** Person getManager() {  
 **return manager**;  
 }  
}

If a client wants to know a person's manager, it needs to get the department first:

manager = person.getDepartment().getManager();

Bad - This reveals to the client how the department class works and that the department is responsible to tracking the manager. I can reduce this coupling by hiding the department class from the client. I do this by creating a simple delegating method on person:

Class Person {

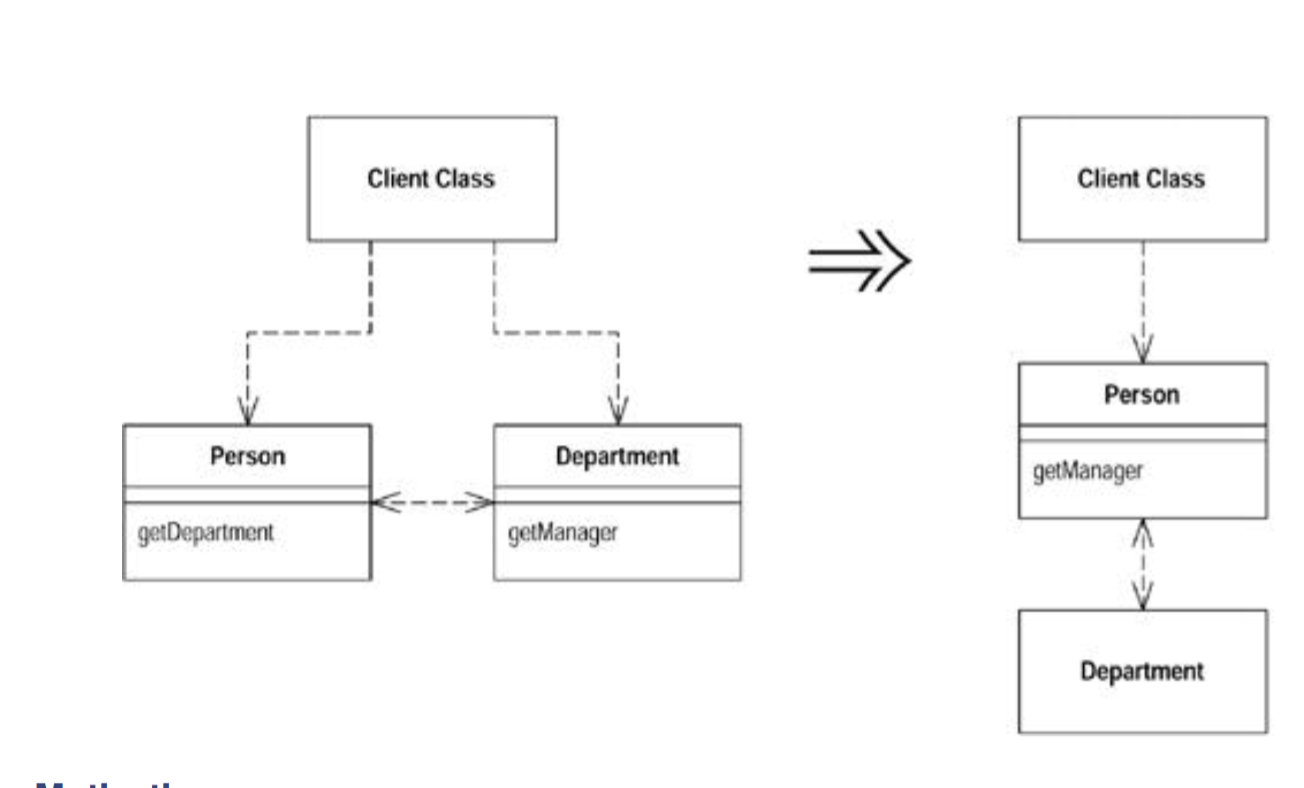
…

**public** Person getManager() {  
 **return** department.getManager();  
}

}

I now need to change all clients of person to use this new method:

manager = person.getManager();

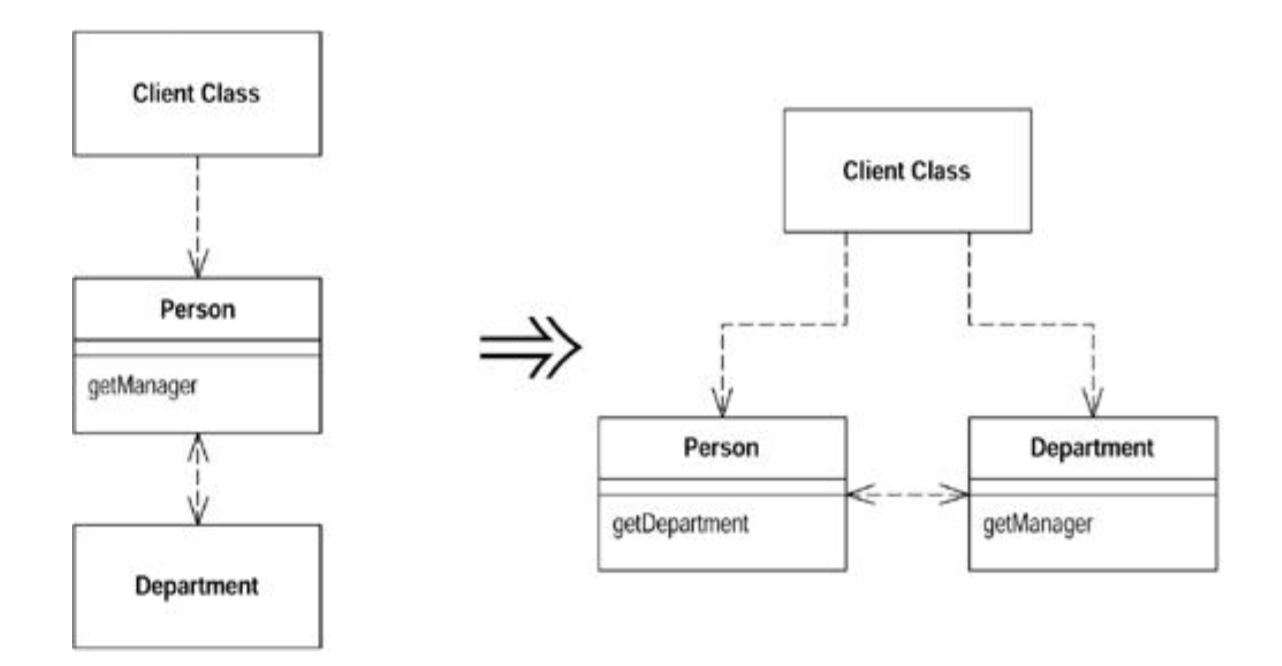


## When to Keep Delegate?

A class is doing too much simple delegation.

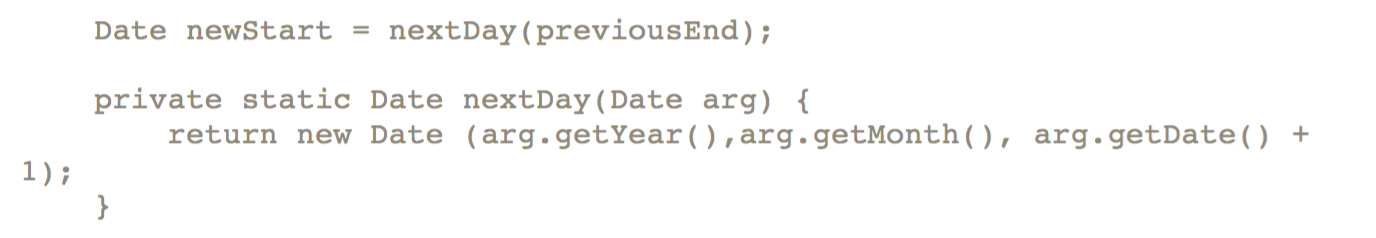
In the motivation for *Hide Delegate,* I talked about the advantages of encapsulating the use of a delegated object. There is a price for this. The price is that every time the client wants to use a new feature of the delegate, you have to add a simple delegating method to the server. After adding features for a while, it becomes painful. The server class is just a middle man, and perhaps it's time for the client to call the delegate directly.

It's hard to figure out what the right amount of hiding is. Fortunately, with *Hide Delegate* and *Keep Delegate* it does not matter so much. You can adjust your system as time goes on. As the system changes, the basis for how much you hide also changes.



# Introduce Foreign Method

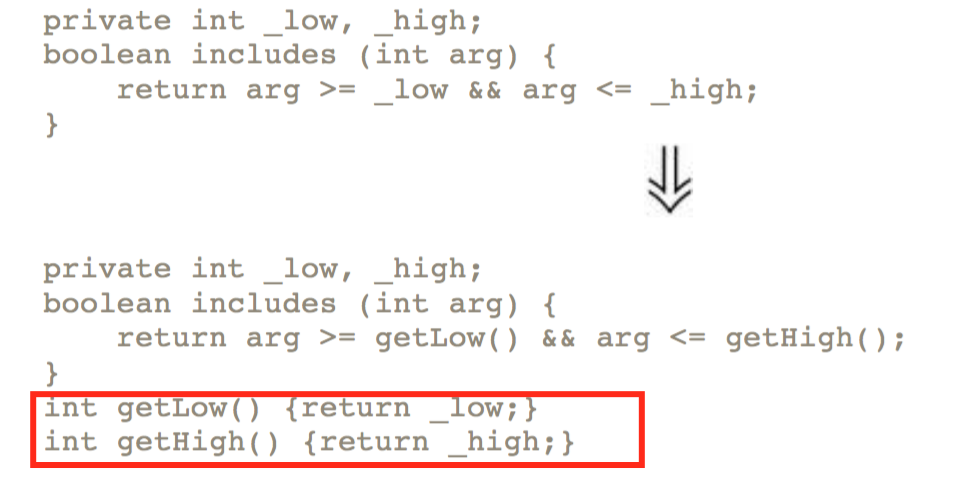




Using Self-Encapsulation (Access variables using getter)

When it comes to accessing fields, there are two schools of thought. One is that within the class where the variable is defined, you should access the variable freely (direct variable access). The other school is that even within the class, you should always use accessors (indirect variable access).

When to use getters?



Advantage :

Advantage is truly when you want to override the variables in subclass. In Java, you can’t override variables, but you can override methods. So, getters gives benefit in this case.

It’s ok to access variables directly, until you need to override it.

# Replace Data Value with Object (pg 141)