Refactoring workbook

Version 0.1.2

May the source be with you!

Version Management

Versie	Datum
0.1	15-11-08
0.1.1	29-11-08
0.1.2	17-01-09

Content

Start Code

Main class

```
public class Main {

    public static void main(String[] args) {
        Customer customer = new Customer("Theo Dorant");
        customer.addRental(new Rental(new Movie("The Matrix runs on windows", 0), 6));
        customer.addRental(new Rental(new Movie("The Island under water", 1), 2));
        customer.addRental(new Rental(new Movie("The days without time", 0), 14));
        System.out.println(customer.statement());
        System.out.println("--------------------------");
        System.out.println(customer.htmlStatement());
    }
}
```

Rental class

```
public class Rental {

    private Movie _movie;
    private int _daysRented;

    public Rental(Movie movie, int daysRented){
        _movie = movie;
        _daysRented = daysRented;
    }
    public int getDaysRented() {
        return _daysRented;
    }
    public Movie getMovie() {
        return _movie;
    }
}
```

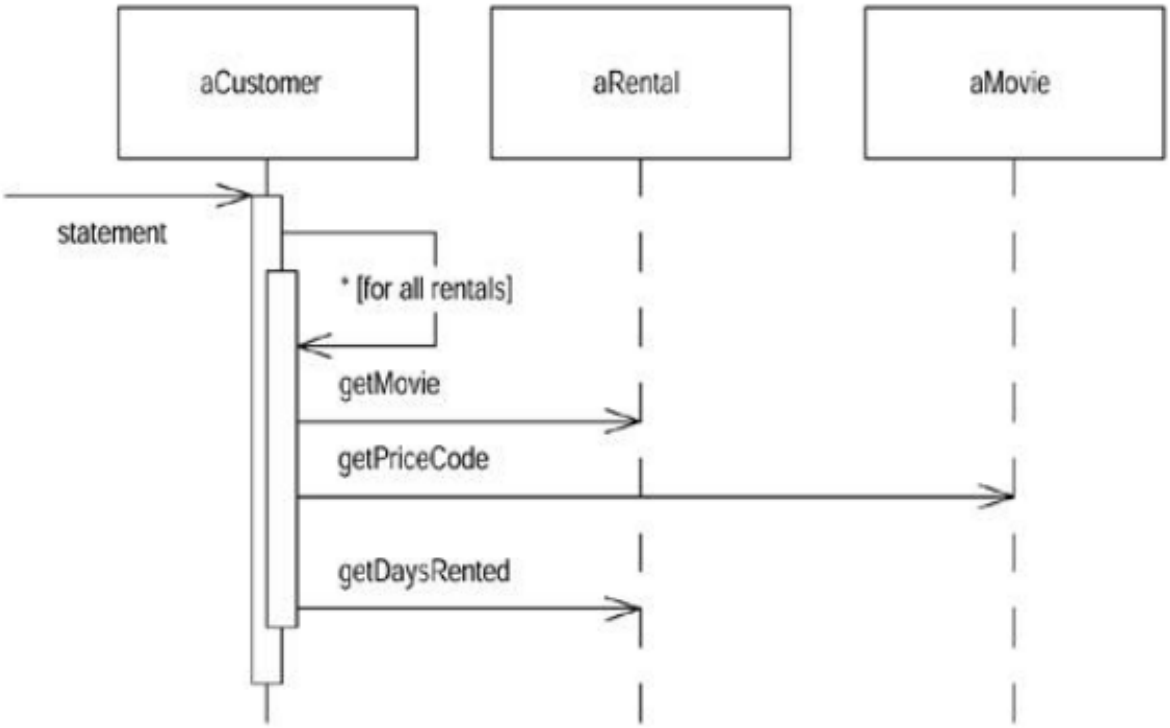
Movie class

```
public class Movie {

    public static final int CHILDRENS = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;

    private String _title;
    private int _priceCode;

    public Movie(String title, int priceCode){
        this._title = title;
        this._priceCode = priceCode;
    }
    public int getPriceCode() {
        return _priceCode;
    }
    public void setPriceCode(int arg) {
        this._priceCode = arg;
    }
    public String getTitle() {
        return _title;
    }
}
```

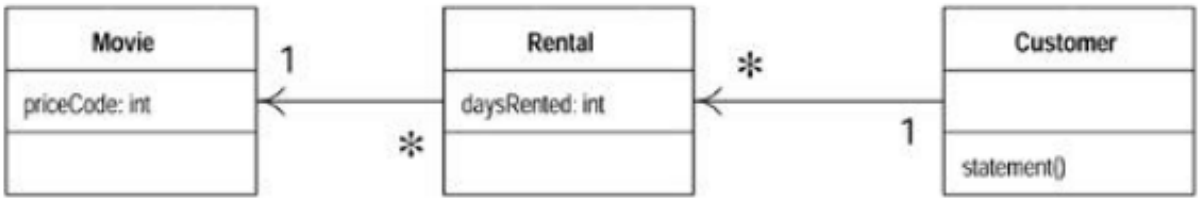
Customer class

```
import java.util.Enumeration;
import java.util.Vector;

public class Customer {

    private String _name;
    private Vector _rentals = new Vector();

    public Customer(String name) {
        _name = name;
    }

    public void addRental(Rental arg) {
        _rentals.addElement(arg);
    }

    public String getName() {
        return _name;
    }

    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while ( rentals.hasMoreElements() ) {
            double thisAmount = 0;
            Rental each = ( Rental ) rentals.nextElement();

            //determine amount for each line
            switch ( each.getMovie().getPriceCode() ) {
                case Movie.REGULAR:
                    thisAmount += 2;
                    if ( each.getDaysRented() > 2 ) {
                        thisAmount += (each.getDaysRented() - 2) * 1.5;
                    }
                    break;
                case Movie.NEW_RELEASE:
                    thisAmount += each.getDaysRented() * 3;
                    break;
                case Movie.CHILDRENS:
                    thisAmount += 1.5;
                    if ( each.getDaysRented() > 3 ) {
                        thisAmount += (each.getDaysRented() - 3) * 1.5;
                    }
                    break;
            }

            // add frequent renter point
            frequentRenterPoints++;
            // add bonus for a two day new release rental
            if ( (each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                                    each.getDaysRented() > 1 ) {
                frequentRenterPoints++;
            }

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle() + "\t" +
                        String.valueOf(thisAmount) + "\n";
            totalAmount += thisAmount;

        }
        //add footer
        result += "Amount owed is:  " + String.valueOf(totalAmount) + ".\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) +
              " frequent renter points.";
        return result;
    }
}
```

UML Before



Interactions for the statement method



Class diagram of the starting-point classes

About the start code

The sample program is very simple. It is a program to calculate and print a statement of a customer's charges at a video store. The program is told which movies a customer has rented and for how long. It then calculates the charges, which depend on how long the movie is rented, and identifies the type movie. There are three kinds of movies: regular, children's, and new releases. In addition to calculating charges, the statement also computes frequent renter points, which vary depending on whether the film is a new release.

What are your impressions about the design of this program? I would describe it as not well designed and certainly not object oriented. For a simple program like this, that does not really matter. There's nothing wrong with a quick and dirty simple program. But if this is a representative fragment of a more complex system, then I have some real problems with this program. That long statement routine in the Customer class does way too much. Many of the things that it does should really be done by the other classes.

The compiler doesn't care whether the code is ugly or clean. But when we change the system, there is a human involved, and humans do care. A poorly designed system is hard to change. Hard because it is hard to figure out where the changes are needed. If it is hard to figure out what to change, then there is a big chance that the programmer will make mistakes and introduce bugs.

In this case we have a change that the users would like to make. First they want a statement printed in HTML so that the statement can be Web enabled and fully buzzword compliant. Consider the impact this change would have. As you look at the code you can see that it is impossible to reuse any of the behavior of the current statement method for an HTML statement. Your only recourse is to write a whole new method that duplicates much of the behavior of statement. Now, of course, this is not too onerous. You can just copy the statement method and make whatever changes you need.

But what happens when the charging rules change? You have to fix both statement and html Statement and ensure that the fixes are consistent. The problem with copying and pasting code comes when you have to change it later. If you are writing a program that you don't expect to change, then cut and paste is fine. If the program is long lived and likely to change, then cut and paste is a menace.

This brings me to a second change. The users want to make changes to the way they classify movies, but they haven't yet decided on the change they are going to make. They have a number of changes in mind. These changes will affect both the way renters are charged for movies and the way that frequent renter points are calculated. As an experienced developer you are sure that whatever scheme users come up with, the only guarantee you're going to have is that within six months they will change it again.

You may be tempted to make the fewest possible changes to the program; after all, it works fine. Remember the old engineering adage: "if it ain't broke, don't fix it." The program may not be broken, but it does hurt. It is making your life more difficult because you find it hard to make the changes your users want. This is where refactoring comes in.

The first step

Whenever I refactor, the first step is always the same. I need to build a solid set of tests for that section of code. The tests are essential because even though I refactor in a structured manner in order to avoid introducing bugs, I'm still human and prone to making mistakes.

Thus I need solid tests.

The obvious first target of my attention is the overly long statement method. When I look at a long method like that, I am trying to decompose the method into smaller pieces. Smaller pieces of code tend to make things more manageable. They are easier to work with and move around.

The first phase of the refactoring is to show how I split up the long method and move the pieces to better classes. My aim is to make it easier to write an HTML statement method with much less duplication of code.

Decomposing and Redistributing the Statement Method

My first step is to find a logical clump of code and use **Extract Method** (In Netbeans it is called “**Introduce Methode**”). An obvious piece here is the switch statement. This looks like it would make a good chunk to extract into its own method.

First I need to look in the fragment for any variables that are local in scope to the method we are looking at, the local variables and parameters. This segment of code uses two: *each* and *thisAmount*. Of these each is not modified by the code but *thisAmount* is modified. Any non- modified variable I can pass in as a parameter. Modified variables need more care. If there is only one, I can return it. The temp is initialized to 0 each time around the loop and is not altered until the switch gets to it. So I can just assign the result.

The next page shows the code before and after refactoring. The before code is on the left, the resulting code on the right. The code I'm extracting from the original and any changes in the new code that I don't think are immediately obvious are in *italic* and **boldface** type. As I continue with this workbook, I'll continue with this left-right convention.

Code Before

class Customer . . . . .

```
public String statement() {
   double totalAmount = 0;
   int frequentRenterPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = "Rental Record for " + getName() + "\n";
   while ( rentals.hasMoreElements() ) {
      double thisAmount = 0;
      Rental each = ( Rental ) rentals.nextElement();

      //determine amount for each line
      switch ( each.getMovie().getPriceCode() ) {
         case Movie.REGULAR:
            thisAmount += 2;
            if ( each.getDaysRented() > 2 ) {
               thisAmount += (each.getDaysRented() - 2) * 1.5;
            }
            break;
         case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;
         case Movie.CHILDRENS:
            thisAmount += 1.5;
            if ( each.getDaysRented() > 3 ) {
               thisAmount += (each.getDaysRented() - 3) * 1.5;
            }
            break;
      }

      // add frequent renter point
      frequentRenterPoints++;          // add bonus for a two day new release rental
      if ( (each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                                 each.getDaysRented() > 1 ) {
         frequentRenterPoints++;
      }
      //show figures for this rental
      result += "\t" + each.getMovie().getTitle() + "\t" +
                        String.valueOf(thisAmount) + "\n";
      totalAmount += thisAmount;
   }
   result += "Amount owed is:  " + String.valueOf(totalAmount) + ".\n";
   result += "You earned " + String.valueOf(frequentRenterPoints) +
         " frequent renter points.";
   return result;
}
```

Code After

class Customer . . . . .

```
public String statement() {
   double totalAmount = 0;
   int frequentRenterPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = "Rental Record for " + getName() + "\n";
   while (rentals.hasMoreElements()) {
      double thisAmount = 0;
      Rental each = (Rental) rentals.nextElement();

      thisAmount = amountFor(each);
      frequentRenterPoints++;          // add bonus for a two day new release rental
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                                 each.getDaysRented() > 1) {
         frequentRenterPoints++;
      }
      //show figures for this rental
      result += "\t" + each.getMovie().getTitle() + "\t" +
                        String.valueOf(thisAmount) + "\n";
      totalAmount += thisAmount;
   }
   result += "Amount owed is:  " + String.valueOf(totalAmount) + ".\n";
   result += "You earned " + String.valueOf(frequentRenterPoints) +
         " frequent renter points.";
   return result;
}

private double amountFor(Rental each) {
   double thisAmount = 0;
   switch (each.getMovie().getPriceCode()) {
      case Movie.REGULAR:
         thisAmount += 2;
         if (each.getDaysRented() > 2) {
            thisAmount += (each.getDaysRented() - 2) * 1.5;
         }
         break;
      case Movie.NEW_RELEASE:
         thisAmount += each.getDaysRented() * 3;
         break;
      case Movie.CHILDRENS:
         thisAmount += 1.5;
         if (each.getDaysRented() > 3) {
            thisAmount += (each.getDaysRented() - 3) * 1.5;
         }
         break;
   }
   return thisAmount;
}
```

Now that I've broken the original method down into chunks, I can work on them separately. I don't like some of the variable names in amountFor, and this is a good place to change them.

Code Before

class Customer . . . . .

```
private double amountFor(Rental each) {
  double thisAmount = 0;
  switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
      thisAmount += 2;
      if (each.getDaysRented() > 2) {
        thisAmount += (each.getDaysRented() - 2) * 1.5;
      }
      break;
    case Movie.NEW_RELEASE:
      thisAmount += each.getDaysRented() * 3;
      break;
    case Movie.CHILDRENS:
      thisAmount += 1.5;
      if (each.getDaysRented() > 3) {
        thisAmount += (each.getDaysRented() - 3) * 1.5;
      }
      break;
  }
  return thisAmount;
}
```

Code After

class Customer . . . . .

```
private double amountFor(Rental aRental) {
  double result = 0;
  switch (aRental.getMovie().getPriceCode()) {
    case Movie.REGULAR:
      result += 2;
      if (aRental.getDaysRented() > 2) {
        result += (aRental.getDaysRented() - 2) * 1.5;
      }
      break;
    case Movie.NEW_RELEASE:
      result += aRental.getDaysRented() * 3;
      break;
    case Movie.CHILDRENS:
      result += 1.5;
      if (aRental.getDaysRented() > 3) {
        result += (aRental.getDaysRented() - 3) * 1.5;
      }
      break;
  }
  return result;
}
```

Once I've done the renaming, I compile and test to ensure I haven't broken anything. Is renaming worth the effort? Absolutely. Good code should communicate what it is doing clearly, and variable names are a key to clear code. Never be afraid to change the names of things to improve clarity. With good find and replace tools, it is usually not difficult. Strong typing and testing will highlight anything you miss.

Remember

> “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.

Moving the Amount Calculation

As I look at amountFor, I can see that it uses information from the rental, but does not use information from the customer.
This immediately raises my suspicions that the method is on the wrong object. In most cases a method should be on the object whose data it uses, thus the method should be moved to the rental. To do this I use Move Method. However Netbeans does not support this refactor method yet, so we have to do this one on our own. With this you first copy the code over to rental, adjust it to fit in its new home, and compile as follows:

Code Before

class Customer . . . . .

```
private double amountFor(Rental aRental) {
    double result = 0;

    switch (aRental.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (aRental.getDaysRented() > 2) {
                result += (aRental.getDaysRented() - 2) * 1.5;
            }
            break;
        case Movie.NEW_RELEASE:
            result += aRental.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (aRental.getDaysRented() > 3) {
                result += (aRental.getDaysRented() - 3) * 1.5;
            }
            break;
    }
    return result;
}
```

Code After

class Rental . . . . .

```
public double getCharge() {
    double result = 0;

    switch (getMovie().getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (getDaysRented() > 2) {
                result += (getDaysRented() - 2) * 1.5;
            }
            break;
        case Movie.NEW_RELEASE:
            result += getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (getDaysRented() > 3) {
                result += (getDaysRented() - 3) * 1.5;
            }
            break;
    }
    return result;
}
```

class Customer . . . . .

```
private double amountFor(Rental aRental) {
    return aRental.getCharge();
}
```

In this case fitting into its new home means removing the parameter. I also renamed the method as I did the move.
I can now test to see whether this method works. To do this I replace the body of "*Customer.amountFor"* to delegate to the new method.

Now we can find every reference to the old method and adjust the reference to use the new method, as follows:

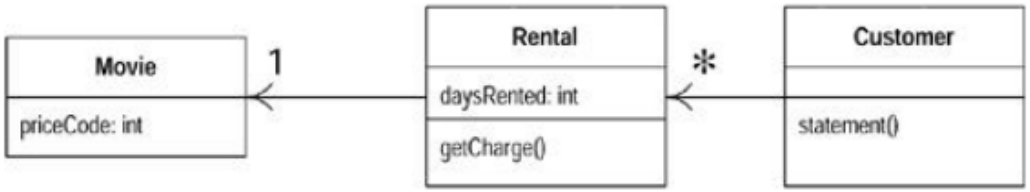
Code Before	Code After

class Customer . . . . .

```
public String statement() {
 . . . . .
    while (rentals.hasMoreElements()) {
       double thisAmount = 0;
       Rental each = (Rental) rentals.nextElement();

       thisAmount = amountFor(each);

       // add frequent renter point
       frequentRenterPoints++;
 . . . . .

private double amountFor(Rental each) {
    return each.getCharge();
}
```

class Customer . . . . .

```
public String statement() {
 . . . . .
    while (rentals.hasMoreElements()) {
       double thisAmount = 0;
       Rental each = (Rental) rentals.nextElement();

       thisAmount = each.getCharge();

       // add frequent renter point
       frequentRenterPoints++;
 . . . . .
```



After this is done you can remove the old method *amountFor* by using the refactor function "**Safe delete**".
Sometimes I leave the old method to delegate to the new method. This is useful if it is a public method and I don't want to change the interface of the other class.

I can now compile and test to see whether I've broken anything.

The next thing that strikes me is that thisAmount is now redundant. It is set to the result of each.charge and not changed afterwards. Thus I can eliminate thisAmount by using **Replace Temp with Query**. However, once again, netbeans does not support this refactor method so we wil have to do this one as well.

Code Before

```
class Customer . . . . .

    public String statement() {
    . . . . .
        while (rentals.hasMoreElements()) {
            double thisAmount = 0;
            Rental each = (Rental) rentals.nextElement();
            thisAmount = amountFor(each);

            // add frequent renter point
            frequentRenterPoints++;
            // add bonus for a two day new release rental
            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
each.getDaysRented() > 1) {
                frequentRenterPoints++;
            }

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle() + "\t" +
                      String.valueOf(thisAmount) + "\n";
            totalAmount += thisAmount;
        }
    . . . . .
```

Code After

```
class Customer . . . . .

    public String statement() {
    . . . . .
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();

            // add frequent renter point
            frequentRenterPoints++;
            // add bonus for a two day new release rental
            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
each.getDaysRented() > 1) {
                frequentRenterPoints++;
            }

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle() + "\t" +
                      String.valueOf(each.getCharge()) + "\n";
            totalAmount += each.getCharge();
        }
    . . . . .
```

I like to get rid of temporary variables such as this as much as possible. Temps are often a problem in that they cause a lot of parameters to be passed around when they don't have to be. You can easily lose track of what they are there for. They are particularly insidious in long methods. Of course there is a performance price to pay; here the charge is now calculated twice. But it is easy to optimize that in the rental class, and you can optimize much more effectively when the code is properly factored.

Extracting Frequent Renter Points

The next step is to do a similar thing for the frequent renter points. The rules vary with the tape, although there is less variation than with charging. It seems reasonable to put the responsibility on the rental. First we need to use **Extract Method** (In Netbeans it is called "**Introduce Methode**") on the frequent renter points part of the code:

Code Before

class Customer . . . . .

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

        // add frequent renter point
        frequentRenterPoints++;
        // add bonus for a two day new release rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
each.getDaysRented() > 1) {
            frequentRenterPoints++;
        }

        //show figures for this rental
        result += "\t" + each.getMovie().getTitle() + "\t" +
                  String.valueOf(each.getCharge()) + "\n";
        totalAmount += each.getCharge();
    }

    //add footer
    result += "Amount owed is:  " + String.valueOf(totalAmount) + ".\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
        " frequent renter points.";
    return result;
}
```

Code After

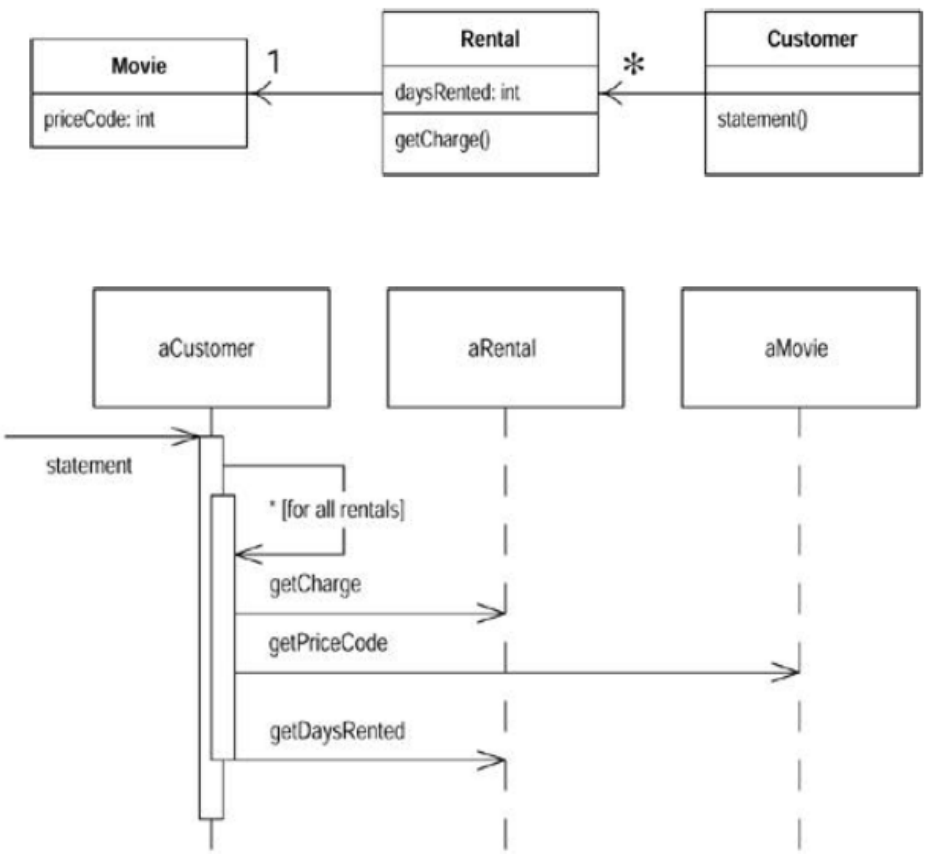
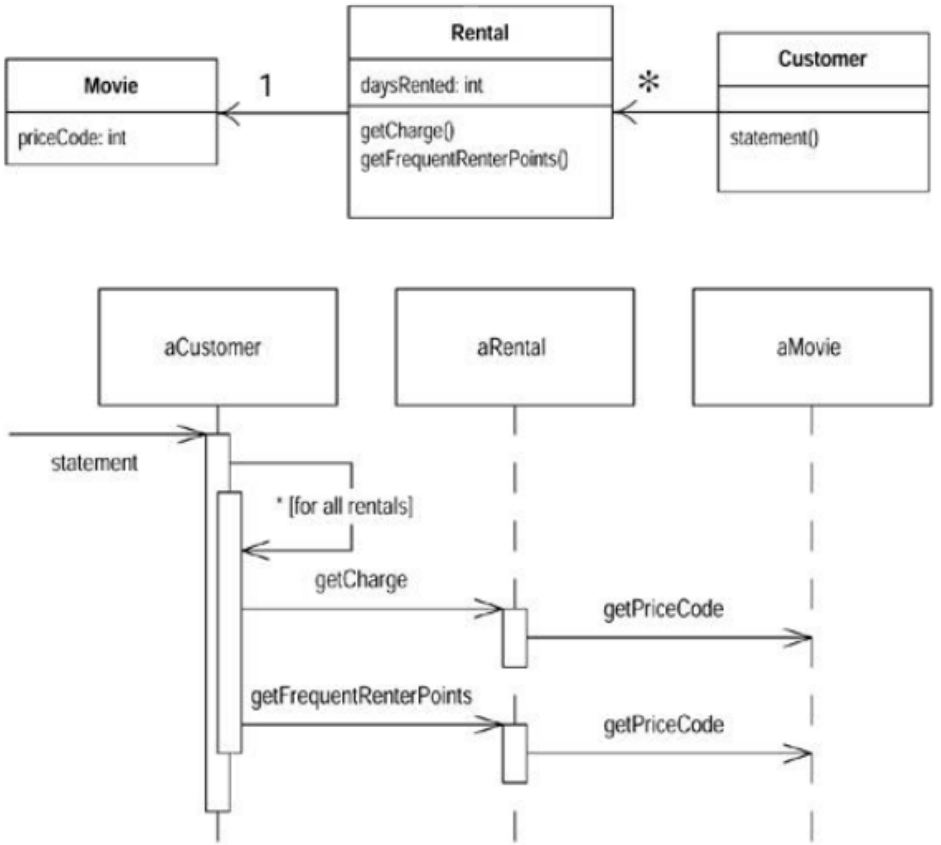
class Customer . . . . .

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";
    while ( rentals.hasMoreElements() ) {
        Rental each = ( Rental ) rentals.nextElement();

        frequentRenterPoints += each.getFrequentRenterPoints();

        //show figures for this rental
        result += "\t" + each.getMovie().getTitle() + "\t" +
                  String.valueOf(each.getCharge()) + "\n";
        totalAmount += each.getCharge();

    }
    //add footer
    result += "Amount owed is:  " + String.valueOf(totalAmount) + ".\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
        " frequent renter points.";
    return result;
}
}
```

class Rental . . . . .

```
public int getFrequentRenterPoints() {
    if ( (getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                getDaysRented() > 1 ) {
        return 2;
    } else {
        return 1;
    }
}
```

I'll summarize the changes I just made with some before-and-after Unified Modeling Language (UML) diagrams. Again the diagrams on the left are before the change; those on the right are after the change.

UML Before	UML After
	
Class & sequence diagrams before extraction and movement of the frequent renter points calculation	*Class & sequence diagrams after extraction and movement of the frequent renter points calculation*

Removing Temps

As I suggested before, temporary variables can be a problem. They are useful only within their own routine, and thus they encourage long, complex routines. In this case we have two temporary variables, both of which are being used to get a total from the rentals attached to the customer. Both the ASCII and HTML versions require these totals. I like to use **Replace Temp with Query** to replace "*totalAmount*" and "*frequentRentalPoints*" with query methods. Queries are accessible to any method in the class and thus encourage a cleaner design without long, complex methods.

I began by replacing "*totalAmount*" with a charge method on customer:

Code Before

```
class Customer . . . . .

    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints();

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle() + "\t" +
                      String.valueOf(each.getCharge()) + "\n";
            totalAmount += each.getCharge();
        }
        //add footer
        result += "Amount owed is:  " + String.valueOf(totalAmount) + ".\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) +
            " frequent renter points.";
        return result;
    }
. . . . .
```

Code After

```
class Customer . . . . .

    public String statement() {
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints();

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle() + "\t" +
                      String.valueOf(each.getCharge()) + "\n";
        }
        //add footer
        result += "Amount owed is:  " + String.valueOf(getTotalCharge()) + ".\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) +
            " frequent renter points.";
        return result;
    }

    private double getTotalCharge() {
        double result = 0;
        Enumeration rentals = _rentals.elements();
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += each.getCharge();
        }
        return result;
    }
```

This isn't the simplest case of Replace Temp with Query totalAmount was assigned to within the loop, so I have to copy the loop into the query method.

After compiling and testing that refactoring, I did the same for "*frequentRenterPoints*".

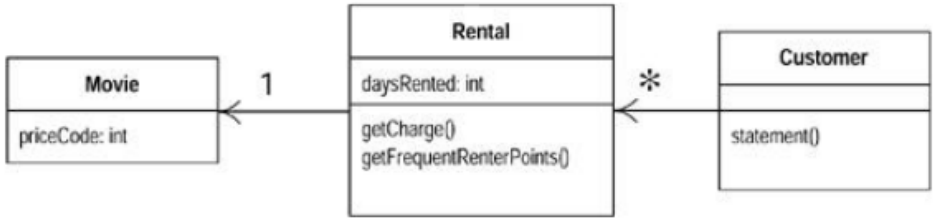
Code Before

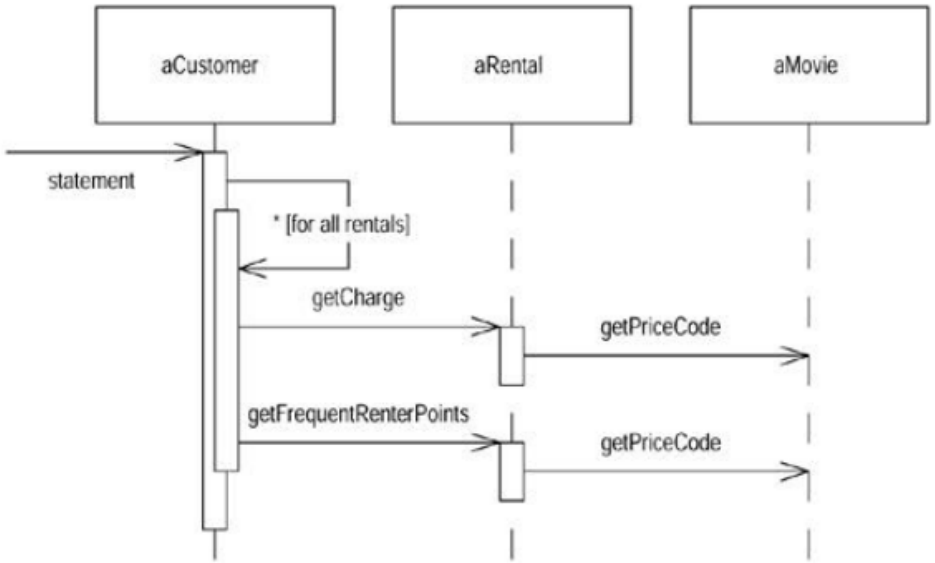
```
class Customer . . . . .

    public String statement() {
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints();

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle() + "\t" +
                      String.valueOf(each.getCharge()) + "\n";
        }
        //add footer
        result += "Amount owed is:  " + String.valueOf(getTotalCharge()) + ".\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) +
            " frequent renter points.";
        return result;
    }
. . . . .
```

Code After

```
class Customer . . . . .

    public String statement() {
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle() + "\t" +
                      String.valueOf(each.getCharge()) + "\n";
        }
        //add footer
        result += "Amount owed is:  " + String.valueOf(getTotalCharge()) + ".\n";
        result += "You earned " + String.valueOf(getTotalFrequentRenterPoints())
+ " frequent renter points.";
        return result;
    }

    private int getTotalFrequentRenterPoints() {
        int result = 0;
        Enumeration rentals = _rentals.elements();
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += each.getFrequentRenterPoints();
        }
        return result;
    }
. . . . .
```

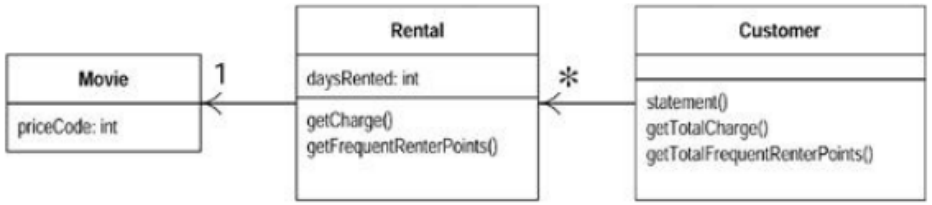
UML Before

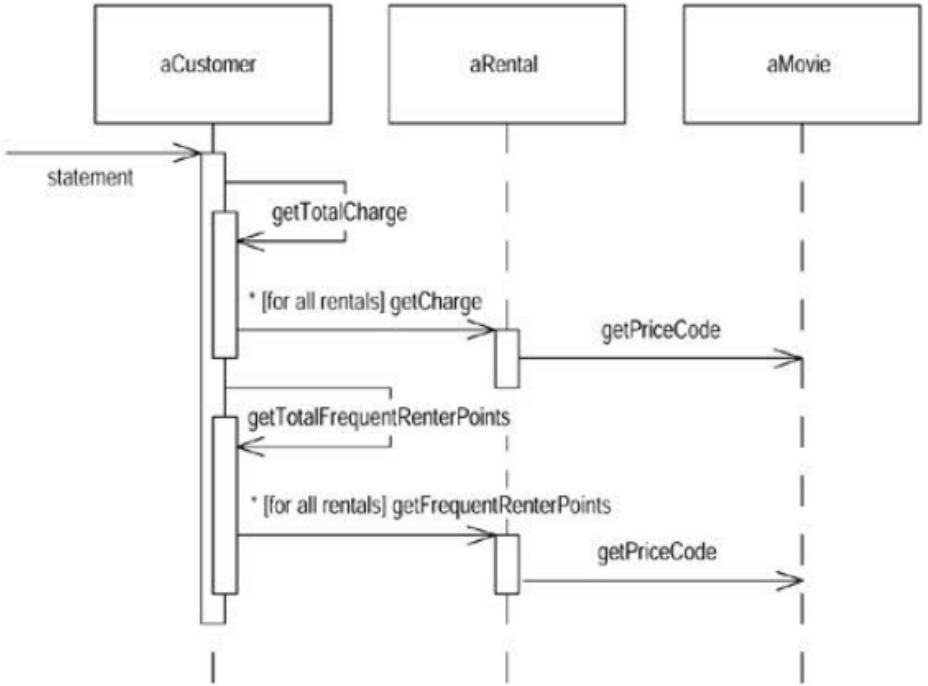




Class & sequence diagram before extraction of the totals

UML After





Class & sequence diagram after extraction of the totals

It is worth stopping to think a bit about the last refactoring. Most refactorings reduce the amount of code, but this one increases it.

The concern with this refactoring lies in performance. The old code executed the "while" loop once, the new code executes it three times. A while loop that takes a long time might impair performance. Many programmers would not do this refactoring simply for this reason. But note the words if and might. Until I profile I cannot tell how much time is needed for the loop to calculate or whether the loop is called often enough for it to affect the overall performance of the system. Don't worry about this while refactoring. When you optimize you will have to worry about it, but you will then be in a much better position to do something about it, and you will have more options to optimize effectively .

You can see the difference immediately with the *"htmlStatement"*. I am now at the point where I take off my refactoring hat and put on my adding function hat. I can write "*htmlStatement"* as follows and add appropriate tests .

New Code

```
class Customer . . . . .

    public String htmlStatement() {
        Enumeration rentals = _rentals.elements();
        String result = "<H1>Rentals for <EM>" + getName() + "</EM></H1><P>\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();

            //show figures for each rental
            result += each.getMovie().getTitle() + ": " +
                      String.valueOf(each.getCharge()) + "<BR>\n";
        }
        //add footer lines
        result += "<P>You owe <EM>" + String.valueOf(getTotalCharge()) + "</EM><P>\n";
        result += "On this rental you earned <EM>" +
            String.valueOf(getTotalFrequentRenterPoints()) +
            "</EM> frequent renter points<P>";
        return result;
    }
. . . . .
```

By extracting the calculations I can create the htmlStatement method and reuse all of the calculation code that was in the original statement method. I didn't copy and paste, so if the calculation rules change I have only one place in the code to go to. Any other kind of statement will be really quick and easy to prepare. The refactoring did not take long. I spent most of the time figuring out what the code did, and I would have had to do that anyway.

Replacing the Conditional Logic on Price Code with Polymorphism

The first part of this problem is that switch statement. It is a bad idea to do a switch based on an attribute of another object. If you must use a switch statement, it should be on your own data, not on someone else's.

This implies that *"getCharge"* should move onto movie.

Code Before

```
class Rental . . . . .

   public double getCharge() {
      double thisAmount = 0;
      //determine amount for each line
      switch (getMovie().getPriceCode()) {
         case Movie.REGULAR:
            thisAmount += 2;
            if (getDaysRented() > 2) {
               thisAmount += (getDaysRented() - 2) * 1.5;
            }
            break;
         case Movie.NEW_RELEASE:
            thisAmount += getDaysRented() * 3;
            break;
         case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (getDaysRented() > 3) {
               thisAmount += (getDaysRented() - 3) * 1.5;
            }
            break;
      }
      return thisAmount;
   }
. . . . .
```

Code After

```
class Movie . . . . .

   public double getCharge(int daysRented) {
      double result = 0;
      switch (getPriceCode()) {
         case Movie.REGULAR:
            result += 2;
            if (daysRented > 2) {
               result += (daysRented - 2) * 1.5;
            }
            break;
         case Movie.NEW_RELEASE:
            result += daysRented * 3;
            break;
         case Movie.CHILDRENS:
            result += 1.5;
            if (daysRented > 3) {
               result += (daysRented - 3) * 1.5;
            }
            break;
      }
      return result;
   }
}

class Rental . . . . .

   public double getCharge() {
      return _movie.getCharge(_daysRented);
   }
. . . . .
```

For this to work I had to pass in the length of the rental, which of course is data from the rental. The method effectively uses two pieces of data, the length of the rental and the type of the movie. Why do I prefer to pass the length of rental to the movie rather than the movie type to the rental?

It's because the proposed changes are all about adding new types. Type information generally tends to be more volatile. If I change the movie type, I want the least ripple effect, so I prefer to calculate the charge within the movie.

Once I've moved the *"getCharge"* method, I'll do the same with the frequent renter point calculation. That keeps both things that vary with the type together on the class that has the type .

Code Before

class Rental . . . . .

```
  public int getFrequentRenterPoints() {
    if ((getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                getDaysRented() > 1) {
      return 2;
    } else {
      return 1;
    }
  }
. . . . .
```

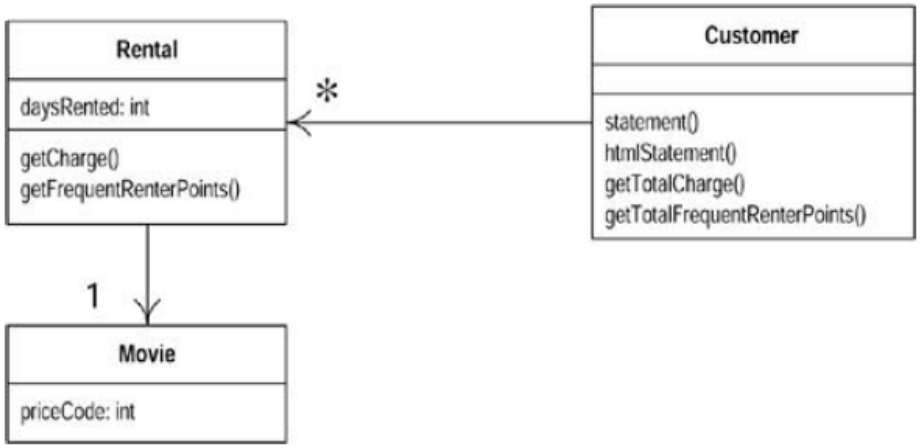
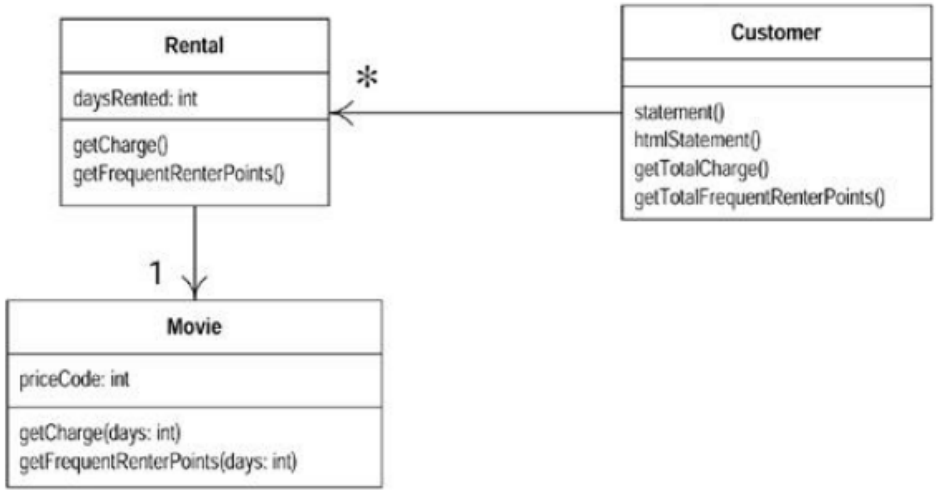
Code After

class Movie . . . . .

```
  public int getFrequentRenterPoints(int daysRented) {
    if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1) {
      return 2;
    } else {
      return 1;
    }
  }
. . . . .
```

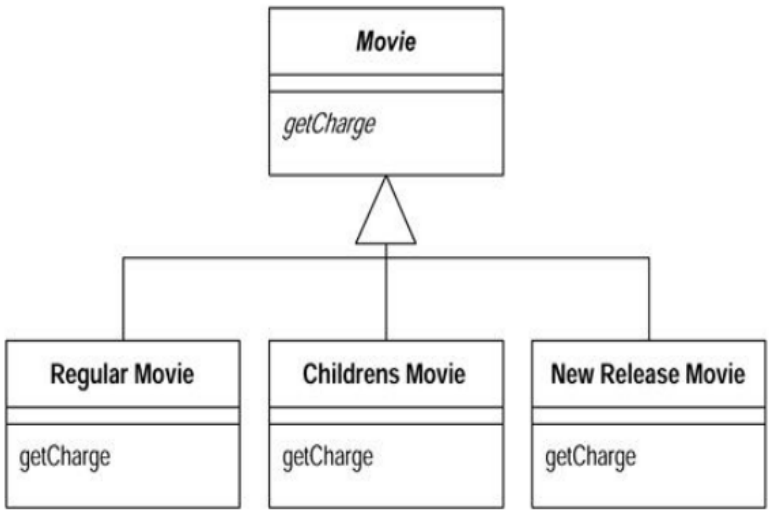
class Rental . . . . .

```
  public int getFrequentRenterPoints(int daysRented) {
    return _movie.getFrequentRenterPoints(_daysRented);
  }
. . . . .
```

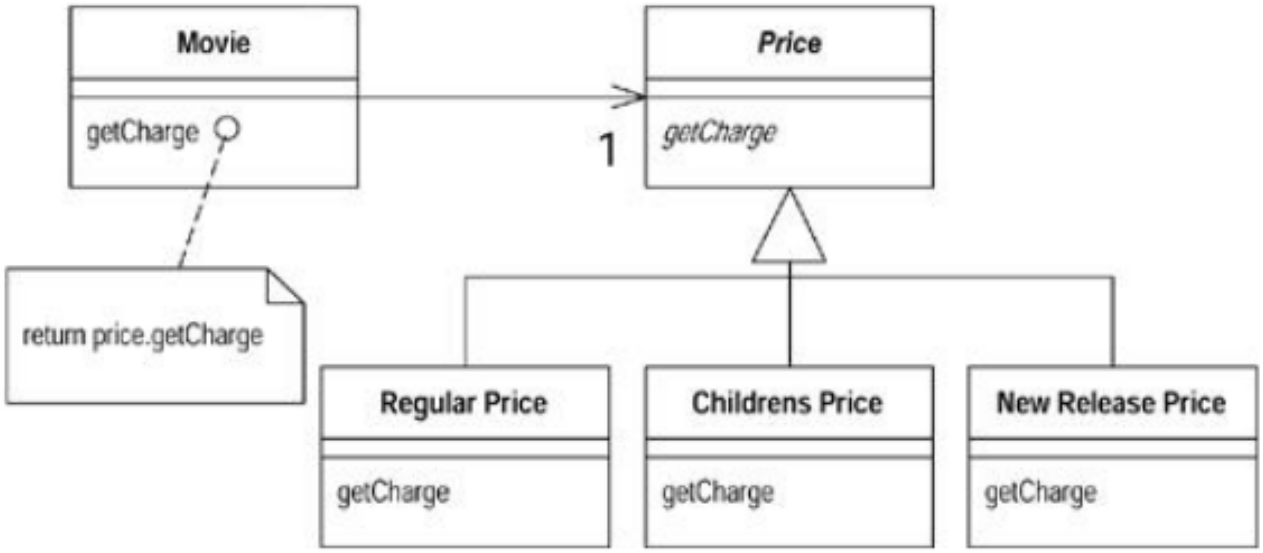
UML Before	UML After
	
Class diagram before moving methods to movie	*Class diagram after moving methods to movie*

At last ... Inheritance

We have several types of movie that have different ways of answering the same question. This sounds like a job for subclasses. We can have three subclasses of movie, each of which can have its own version of charge



This allows me to replace the switch statement by using polymorphism. Sadly it has one slight flaw —it doesn't work. A movie can change its classification during its lifetime. An object cannot change its class during its lifetime. There is a solution, however, the State pattern.



The first step is to use **Self Encapsulate Field** on the type code to ensure that all uses of the type code go through getting and setting methods. Because most of the code came from other classes, most methods already use the getting method. However, the constructors do access the price code.

Code Before

```
class Movie . . . . .

    public Movie(String title, int priceCode) {
        this._title = title;
        this._priceCode = priceCode;
    }
. . . . .
```

Code After

```
class Movie . . . . .

    public Movie(String title, int priceCode) {
        this._title = title;
        setPriceCode(priceCode);
    }
. . . . .
```

Now I add the new classes. I provide the type code behavior in the price object. I do this with an abstract method on price and concrete methods in the subclasses .

New Code

```
abstract class Price {
    abstract int getPriceCode();
}

class ChildrensPrice extends Price {
    int getPriceCode() {
        return Movie.CHILDRENS;
    }
}

class NewReleasePrice extends Price {
    int getPriceCode() {
        return Movie.NEW_RELEASE;
    }
}

class RegularPrice extends Price {
    int getPriceCode() {
        return Movie.REGULAR;
    }
}
```

Now I need to change movie's accessors for the price code to use the new class. This means replacing the price code field with a price field, and changing the accessors.

Code Before

```
class Movie . . . . .

    public int getPriceCode() {
        return _priceCode;
    }

    public void setPriceCode(int arg) {
        this._priceCode = arg;
    }
. . . . .
```

Code After

```
class Movie . . . . .

    public int getPriceCode() {
        return _price.getPriceCode();
    }

    public void setPriceCode(int arg) {
        switch (arg) {
            case REGULAR:
                _price = new RegularPrice();
                break;
            case CHILDRENS:
                _price = new ChildrensPrice();
                break;
            case NEW_RELEASE:
                _price = new NewReleasePrice();
                break;
            default:
                throw new IllegalArgumentException("Incorrect Price Code");
        }
    }
. . . . .
```

I can now compile and test.

Now I apply **Move Method** to getCharge.

Code Before

class Movie . . . . .

```
public double getCharge(int daysRented) {
    double result = 0;
    switch (getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (daysRented > 2) {
                result += (daysRented - 2) * 1.5;
            }
            break;
        case Movie.NEW_RELEASE:
            result += daysRented * 3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (daysRented > 3) {
                result += (daysRented - 3) * 1.5;
            }
            break;
    }
    return result;
}
```

Code After

class Movie . . . . .

```
public double getCharge(int daysRented) {
    return _price.getCharge(daysRented);
}
```

. . . . .

class Price . . . . .

```
public double getCharge(int daysRented) {
    double result = 0;
    switch (getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (daysRented > 2) {
                result += (daysRented - 2) * 1.5;
            }
            break;
        case Movie.NEW_RELEASE:
            result += daysRented * 3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (daysRented > 3) {
                result += (daysRented - 3) * 1.5;
            }
            break;
    }
    return result;
}
```

Once it is moved I can start using **Replace Conditional with Polymorphism**.

I do this by taking one leg of the case statement at a time and creating an overriding method. I start with *"RegularPrice"*:

Code Before	Code After

```
class Price . . . . .

   public double getCharge(int daysRented) {
      double result = 0;
      switch (getPriceCode()) {
         case Movie.REGULAR:
            result += 2;
            if (daysRented > 2) {
               result += (daysRented - 2) * 1.5;
            }
            break;
         case Movie.NEW_RELEASE:
            result += daysRented * 3;
            break;
         case Movie.CHILDRENS:
            result += 1.5;
            if (daysRented > 3) {
               result += (daysRented - 3) * 1.5;
            }
            break;
      }
      return result;
   }
```

```
class RegularPrice . . . . .

   double getCharge(int daysRented) {
      double result = 2;
      if (daysRented > 2) {
         result += (daysRented - 2) * 1.5;
      }
      return result;
   }

class ChildrensPrice . . . . .

   double getCharge(int daysRented) {
      double result = 1.5;
      if (daysRented > 3) {
         result += (daysRented - 3) * 1.5;
      }
      return result;
   }

class NewReleasePrice . . . . .

   double getCharge(int daysRented) {
      return daysRented * 3;
   }

class Price . . . . .

   abstract double getCharge(int daysRented);
```

When I've done that with all the legs, make sure the *"Price.getCharge"* method is abstract.

I can now follow the same procedure with *“getFrequentRenterPoints”.*

Code Before

class Movie . . . . .

```
public int getFrequentRenterPoints(int daysRented) {
   if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1) {
      return 2;
   } else {
      return 1;
   }
}
```

Code After

class Movie . . . . .

```
public int getFrequentRenterPoints(int daysRented) {
   return _price.getFrequentRenterPoints(daysRented);
}
```

class Price . . . . .

```
int getFrequentRenterPoints(int daysRented) {
   if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1) {
      return 2;
   } else {
      return 1;
   }
}
```

In this case, however, I don't make the superclass method abstract. Instead I create an overriding method for new releases and leave a defined method (as the default) on the superclass .

Code Before

class Price . . . . .

```
int getFrequentRenterPoints(int daysRented) {
   if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1) {
      return 2;
   } else {
      return 1;
   }
}
```

Code After

class Price . . . . .

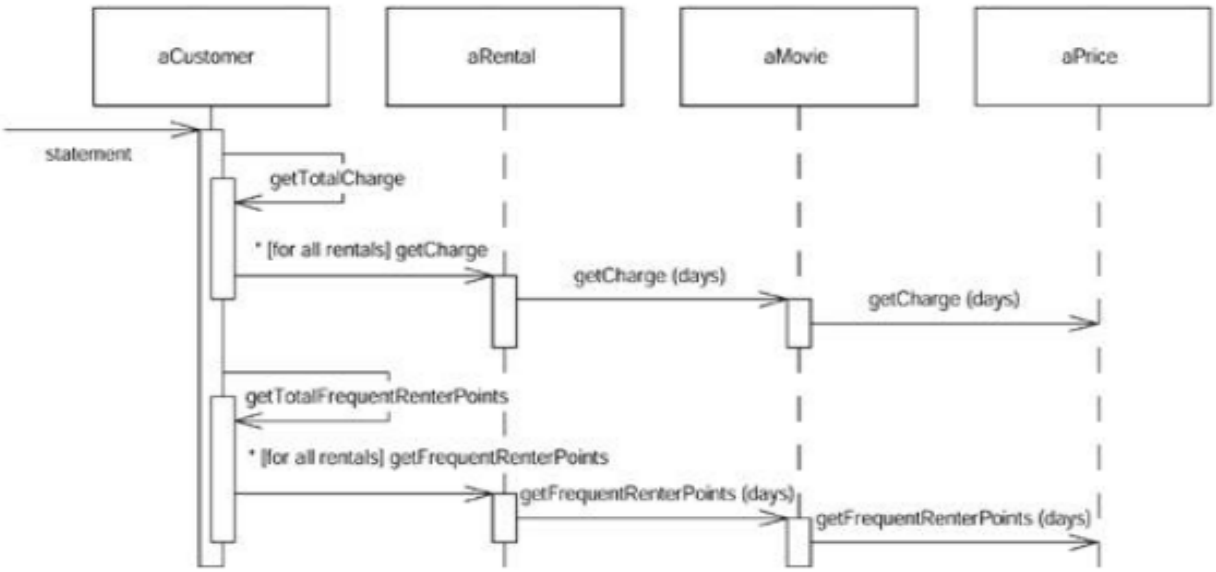
```
int getFrequentRenterPoints(int daysRented) {
   return 1;
}
```

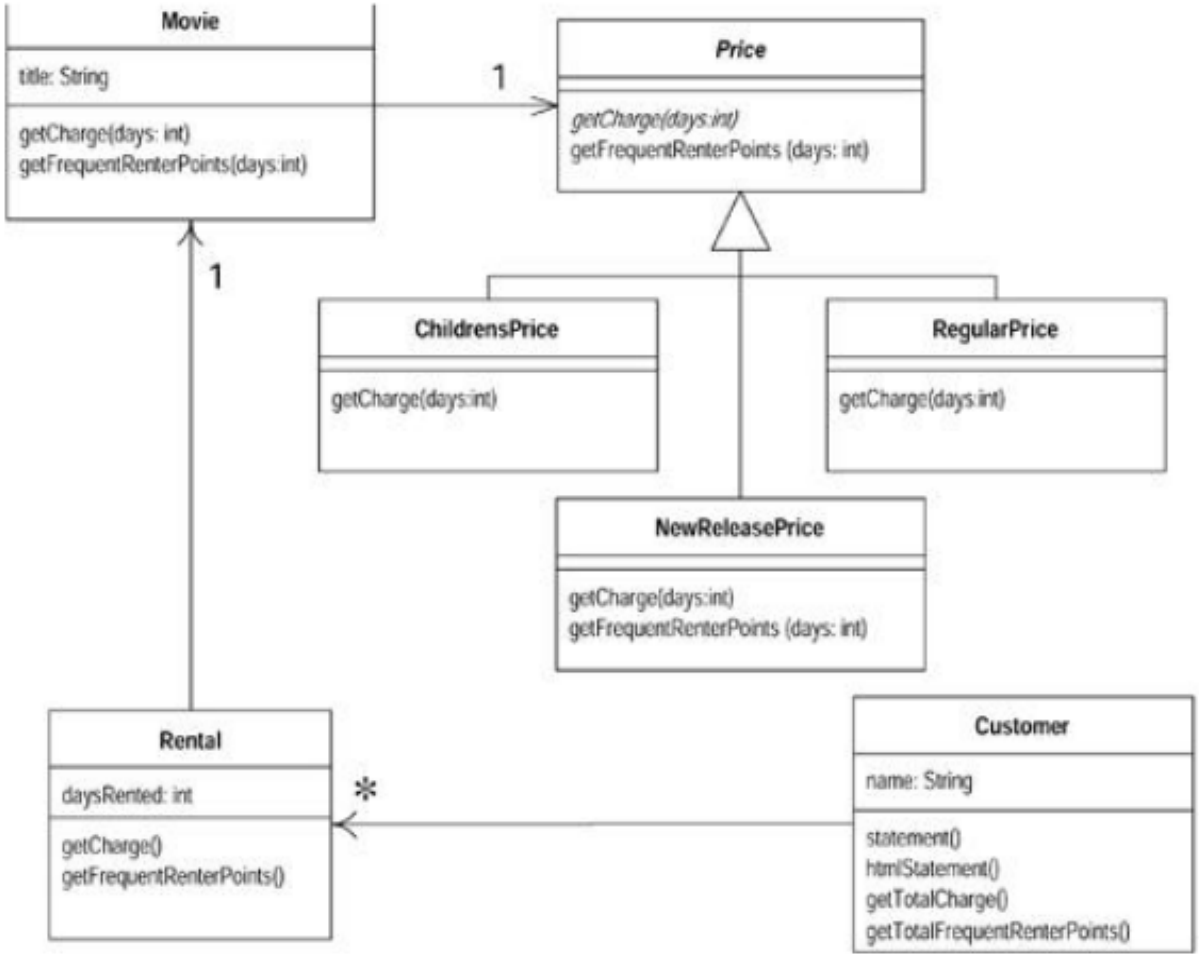
class NewReleasePrice . . . . .

```
int getFrequentRenterPoints(int daysRented) {
   return (daysRented > 1) ? 2 : 1;
}
. . . . .
```

Putting in the state pattern was quite an effort. Was it worth it? The gain is that if I change any of price's behavior, add new prices, or add extra price-dependent behavior, the change will be much easier to make. The rest of the application does not know about the use of the state pattern. For the tiny amount of behavior I currently have, it is not a big deal. In a more complex system with a dozen or so price-dependent methods, this would make a big difference. All these changes were small steps. It seems slow to write it this way, but not once did I have to open the debugger, so the process actually flowed quite quickly. It is going to be much easier to change the classification structure of movies, and to alter the rules for charging, and the frequent renter point system.

UML After


aCustomer
aRental
aMovie
aPrice
statement
getTotalCharge
* [for all rentals] getCharge
getCharge (days)
getCharge (days)
getTotalFrequentRenterPoints
* [for all rentals] getFrequentRenterPoints
getFrequentRenterPoints (days)
getFrequentRenterPoints (days)


Movie
title: String
getCharge(days: int)
getFrequentRenterPoints(days:int)
1
Price
getCharge(days:int)
getFrequentRenterPoints (days: int)
1
ChildrensPrice
getCharge(days:int)
RegularPrice
getCharge(days:int)
NewReleasePrice
getCharge(days:int)
getFrequentRenterPoints (days: int)
Rental
daysRented: int
getCharge()
getFrequentRenterPoints()
*
Customer
name: String
statement()
htmlStatement()
getTotalCharge()
getTotalFrequentRenterPoints()

Final Thoughts

This is a simple example, yet I hope it gives you an idea of what refactoring is like. I've used several refactorings, including Extract Method, Move Method, and Replace Conditional with Polymorphism. All these lead to better-distributed responsibilities and code that is easier to maintain.

The most important lesson from this example is the rhythm of refactoring: test, small change, test, small change, test, small change. It is that rhythm that allows refactoring to move quickly and safely.

Credits

Martin Fowler: Refactoring: Improving the Design of Existing Code.

Ingmar Hendriks, member of the Refactoring Interest Group.