# The Classic Fowler Refactoring of a Video Store

This was one of the original refactorings Martin Fowler presented to illustrate the concept of refactoring. The original version of this Video Rental exercise can be found at:

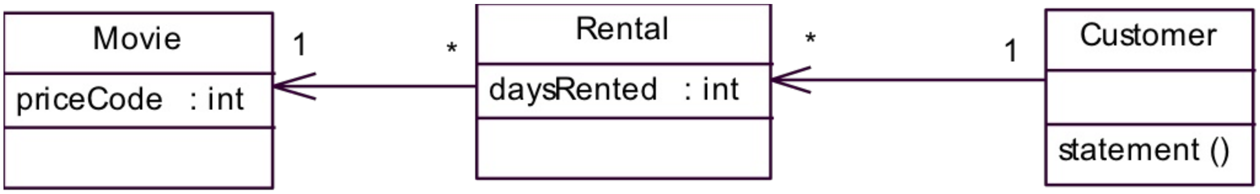
http://www.cs.unc.edu/~stotts/723/refactor/chap1.htm

The problem with using Fowler's original example is that we would have to write all of the tests ourselves for the video store, which is a little bit tedious for this course. So instead, we will be using the updated version of this exercise apparently authored by Rody Middlekoop for his university OOSE class and made available under an MIT license.

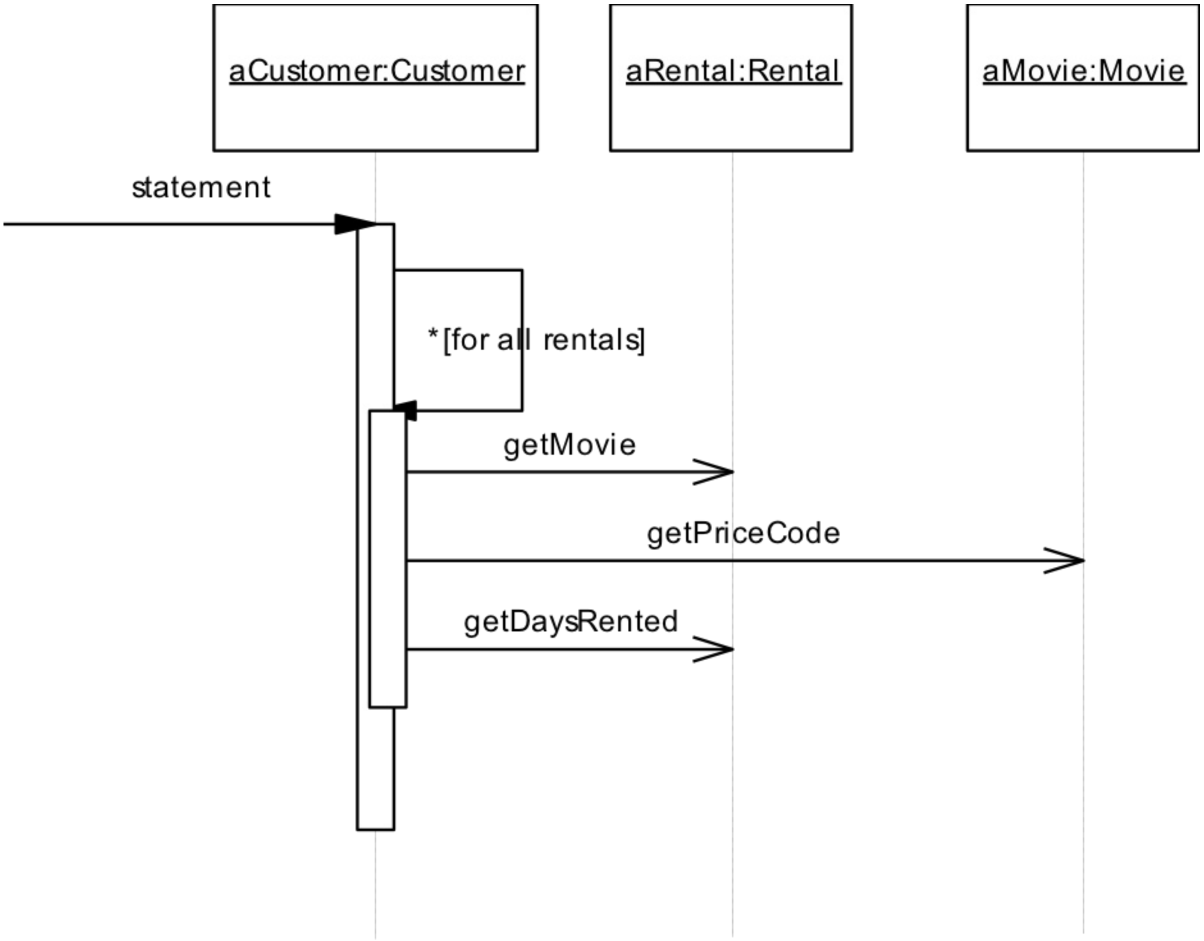
### *The Original Code.*

The code with tests is available in the videostore_start workspace in the lab 7-5 resources folder. Spend some time looking through the code, and run the tests to see that that the tests pass. There are not a lot of tests, so if you want to add a few more, feel free. In the descriptions that follow, various comments are made from these two authors. Comments from Rody Middlekoop are noted with an RM while comments from Martin Fowler are noted with MF.

The sample program 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 rented and for how long. It then calculates the charges, which depend on how long the movie is rented and the category of the 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. Below you'll find the class diagram of the starting point classes. Only most important features are shown. (RM)



Sequence diagram for the statement method: (RM)



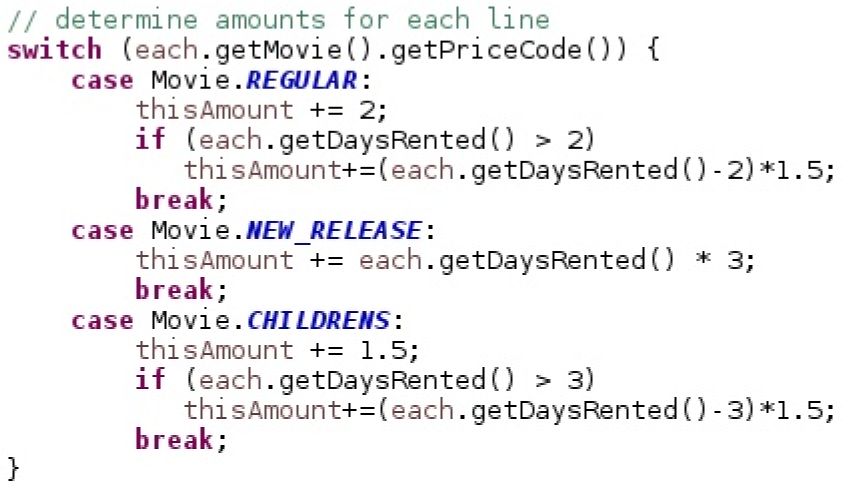
This is really brought out by a new requirement, just in from the users, they want a similar statement in html. As you look at the code you can see that it is impossible to reuse any of the behavior of the current statement() method for an htmlStatement(). Your only recourse is to write a whole new method 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. So the lack of design does not do too much to hamper the writing of htmlStatement(), (although it might be tricky to figure out exactly where to do the changes). But what happens when the charging rules change? You have to fix both statement() and htmlStatement(), and ensure the fixes are consistent. The problem from cut and pasting code comes when you have to change it later. Thus 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. (MF)

But you still have to write the htmlStatement() program. You may feel that you should not touch the existing statement() method, after all it works fine. Remember the old engineering adage: "if it ain't broke, don't fix it". statement() may not be broke, but it does hurt. It is making your life more difficult to write the htmlStatement() method. (MF)

So this is where refactoring comes in. 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; then first refactor the program to make it easy to add the feature, then add the feature. (RM)

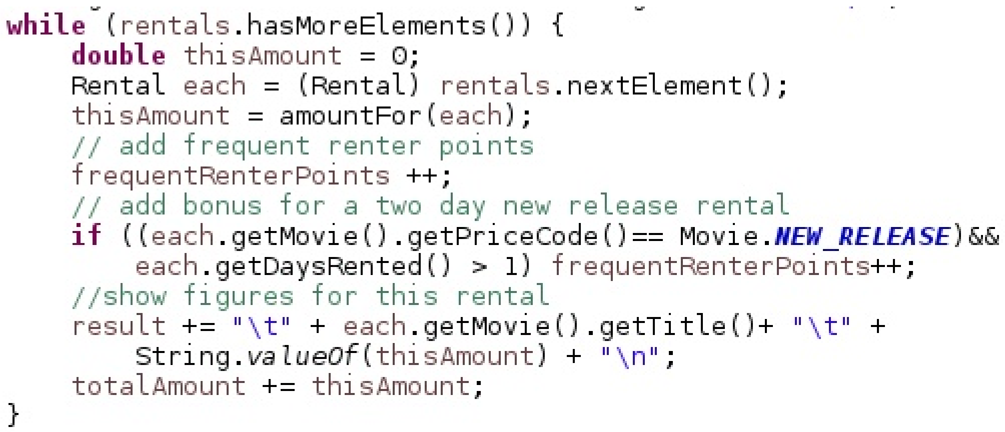
### *Extracting the Amount Calculation*

The obvious first target of my attention is the overly long statement() method. When I look at a long method like that, I am looking to take a chunk of the code an extract a method from it. Extracting a method is taking the chunk of code and making a method out of it. An obvious piece here is the switch statement (MF)

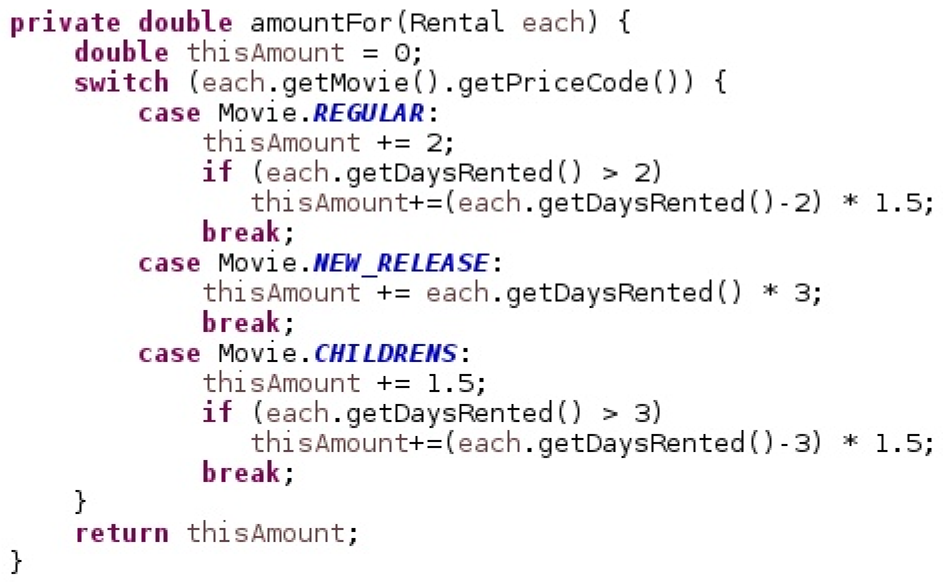


```
// determine amounts for each line
switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
        thisAmount += 2;
        if (each.getDaysRented() > 2)
           thisAmount+=(each.getDaysRented()-2)*1.5;
        break;
    case Movie.NEW_RELEASE:
        thisAmount += each.getDaysRented()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

This looks like it would make a good chunk to extract into its own method. When we extract a method, we need to look in the fragment for any variables that are local in scope to the method we are looking at, that 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 we can pass in as a parameter. Modified variables need more care. If there is only one we can return it. The temp is initialized to 0 each time round the loop, and not altered until the switch gets its hands on it. So we can just assign the result. The extraction looks like this. (MF)

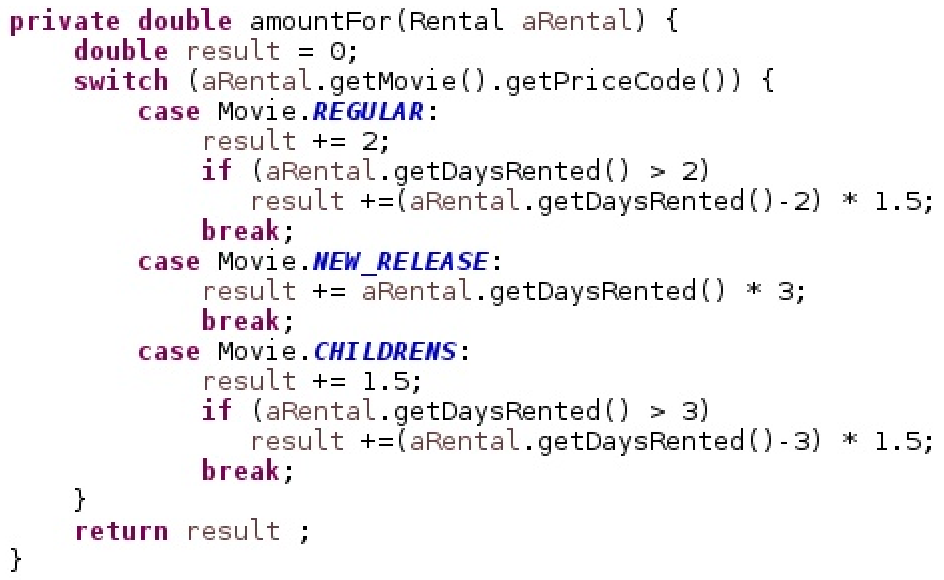


```
while (rentals.hasMoreElements()) {
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement();
    thisAmount = amountFor(each);
    // add frequent renter points
    frequentRenterPoints ++;
    // add bonus for a two day new release rental
    if ((each.getMovie().getPriceCode()== Movie.NEW_RELEASE)&&
        each.getDaysRented() > 1) frequentRenterPoints++;
    //show figures for this rental
    result += "\t" + each.getMovie().getTitle()+ "\t" +
        String.valueOf(thisAmount) + "\n";
    totalAmount += thisAmount;
}
```



```
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;
}
```

This refactoring has taken a large method and broken it down into two much more manageable chunks. We can now consider the chunks a bit better. I don't like some of the variables names in amountOf() and this is a good place to change them. (MF)

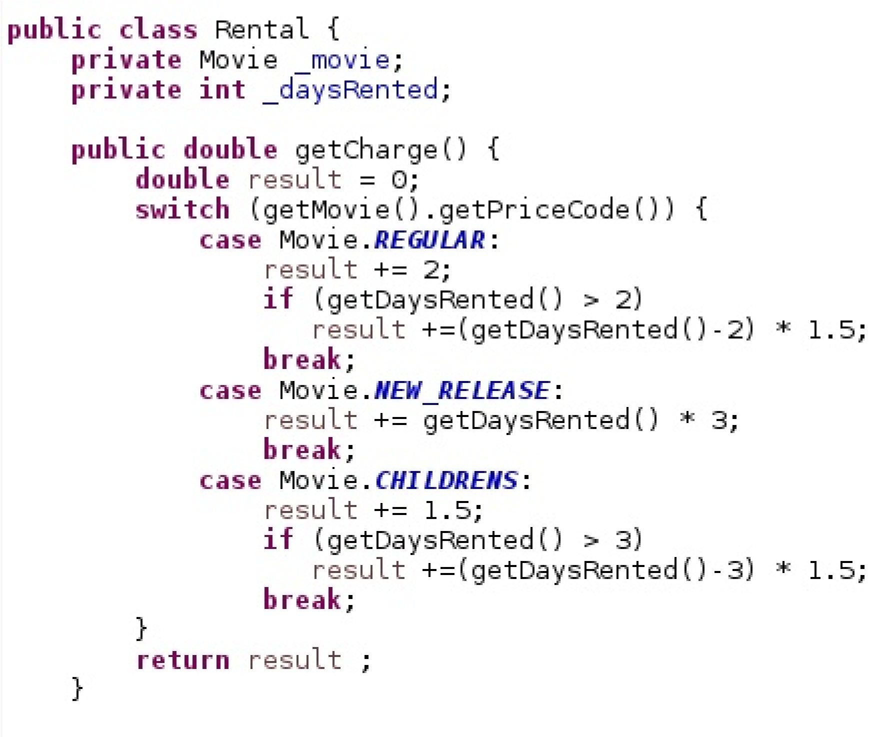


```
private double amountFor(Rental aRental) {
    double result = 0;
    switch (aRental.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (aRental.getDaysRented() > 2)
               result +=(aRental.getDaysRented()-2) * 1.5;
            break;
     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

Is that renaming worth the effort? Absolutely. Good code should communicate what it is doing clearly, and variable names are key to clear code. Never be afraid to change the names to 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. (MF)

### *Moving the amount calculation*

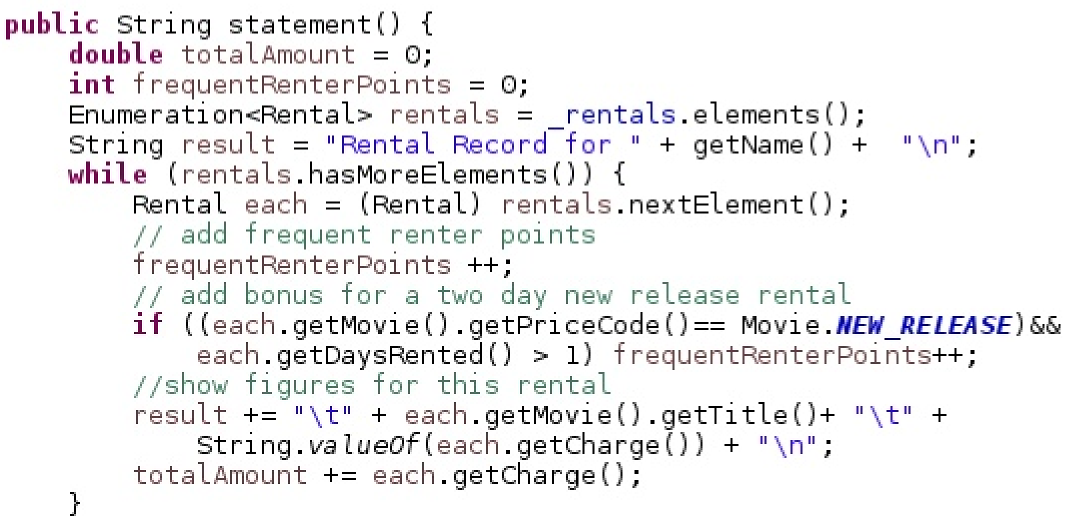
As I look at amountOf, I can see that it uses information from the rental, but does not use information from the customer. This method is thus on the wrong object, it should be moved to the rental. To move a method you first copy the code over to rental, adjust it to fit in its new home and compile. (MF)



```
public class Rental {
    private Movie _movie;
    private int _daysRented;

    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 ;
    }
```

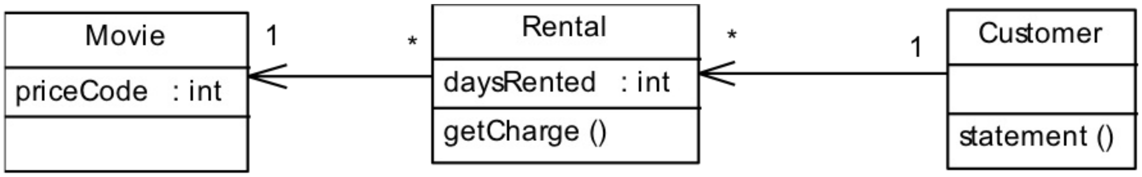
When I've made the change the next thing is to remove the old method. The compiler should then tell me if I missed anything. There is certainly some more I would like to do to Rental.charge() but I will leave it for the moment and return to Customer.statement(). (MF)

The next thing that strikes me is that thisAmount() is now pretty redundant. It is set to the result of each.charge() and not changed afterwards. Thus I can eliminate thisAmount by replacing a temp with a query. (MF)

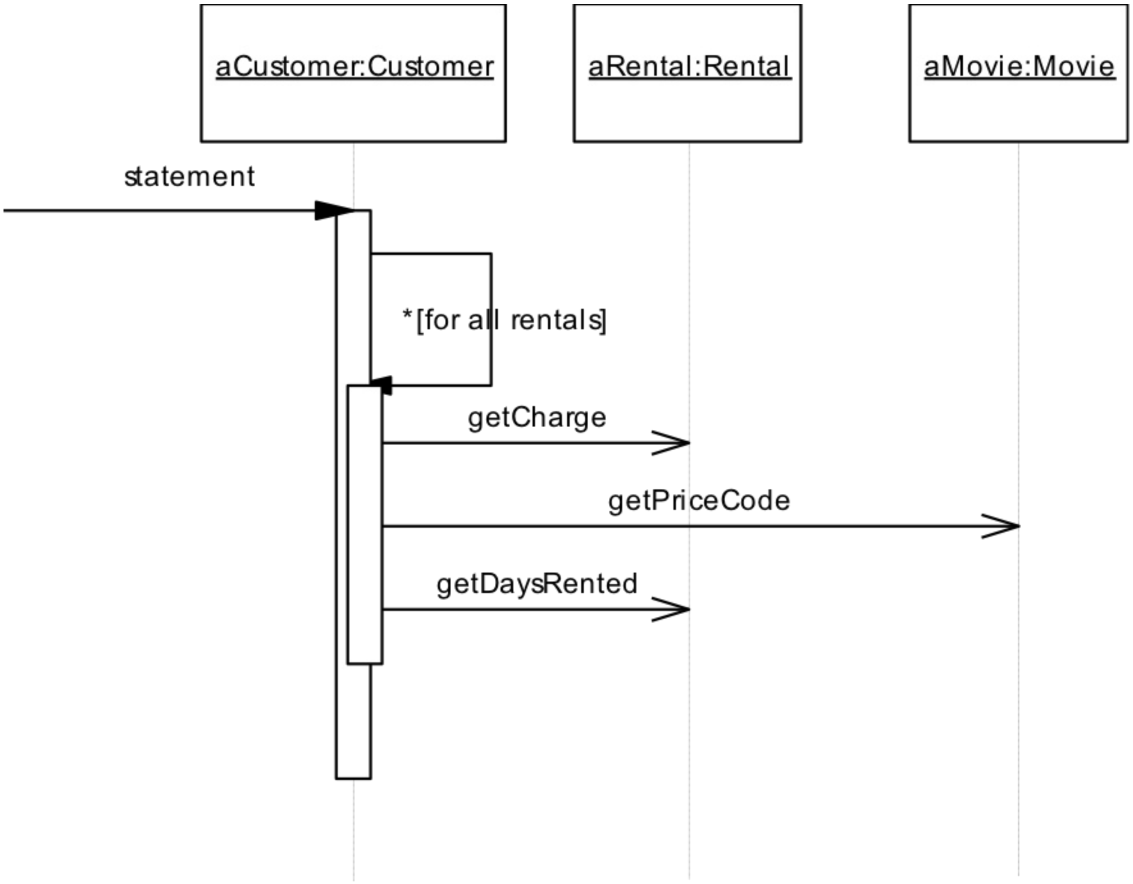


```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration<Rental> rentals = _rentals.elements();
    String result = "Rental Record for " + getName() +  "\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();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

After moving the charge, the class diagram looks like this:



And the sequence diagram looks like this:



The refactoring up to this point is available in workspace VideoStore_1 in the lab 7 solutions folder.

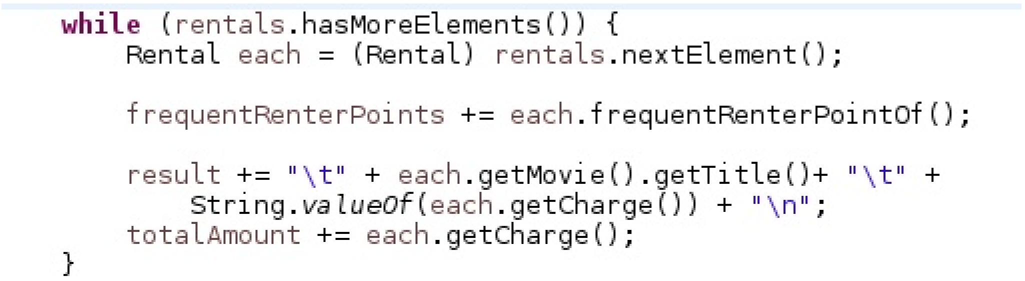
## *Exercise 1.*

The next few refactorings are for you to try on your own.

Perform the following two refactorings, similar to the refactorings 1 and 3 on charges. (RM)

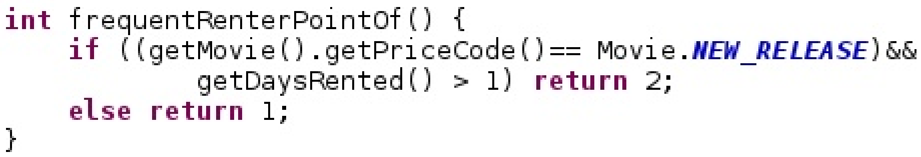
The next step is to do a similar thing for the frequent renter points. It seems reasonable to put the responsibility on the rental. First we need to extract a method from the frequent renter points part of the code. (MF)

1. Apply Extract Method on the frequent renter computation. Call the new method getFrequentRenterPoints. (RM)



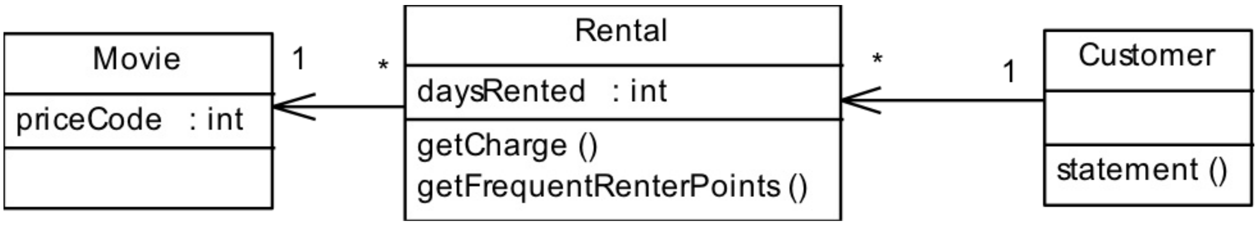
```
while (rentals.hasMoreElements()) {
    Rental each = (Rental) rentals.nextElement();

    frequentRenterPoints += each.frequentRenterPointOf();

    result += "\t" + each.getMovie().getTitle()+ "\t" +
        String.valueOf(each.getCharge()) + "\n";
    totalAmount += each.getCharge();
}
```

2. Apply Move Method on the method getFrequentRenterPoints() to place it in the Rental class. Notice that we can eliminate the paramter (RM)

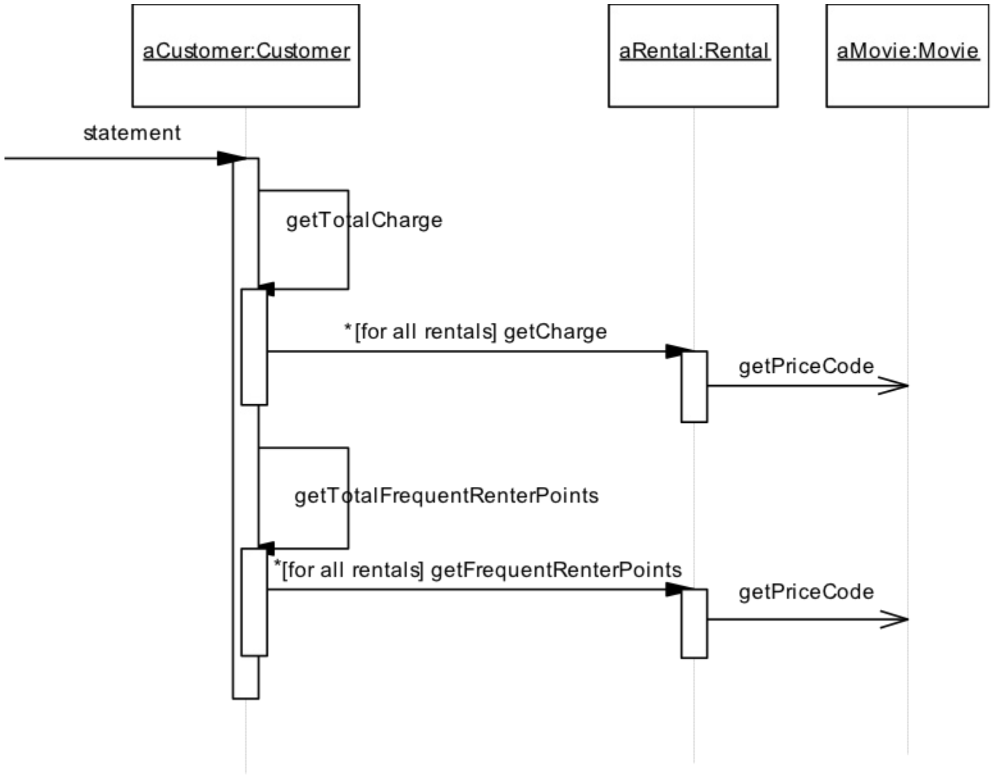


```
int frequentRenterPointOf() {
    if ((getMovie().getPriceCode()== Movie.NEW_RELEASE)&&
            getDaysRented() > 1) return 2;
    else return 1;
}
```

At the end of this step, the class diagram looks like:



And the sequence diagram looks like:


aCustomer:Customer
aRental:Rental
aMovie:Movie
statement
getTotalCharge
*[for all rentals] getCharge
getPriceCode
getTotalFrequentRenterPoints
*[for all rentals] getFrequentRenterPoints
getPriceCode

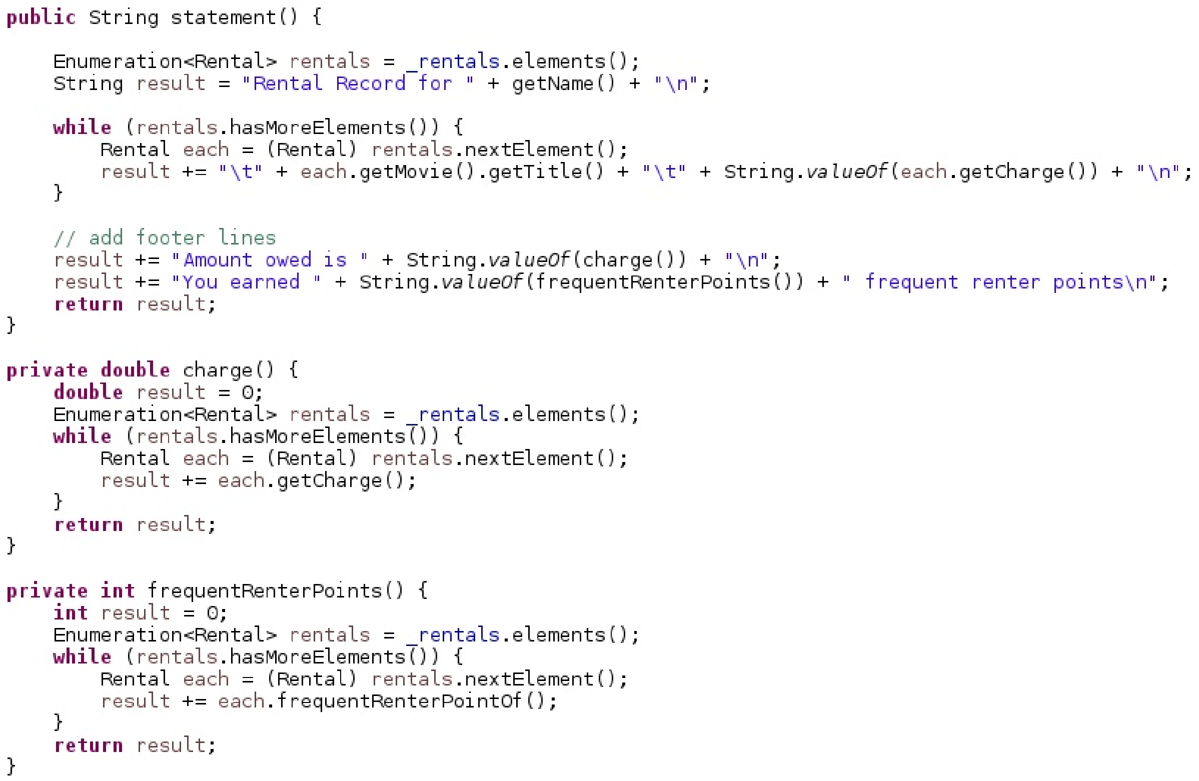
The completed code for this section is in VideoStore_exercise1

## *Exercise 2.*

Perform the following two refactorings (RM)

1. Apply the refactoring Replace Temp with Query. Replace temp variable totalAmount with query method getTotalCharge.
2. Apply the refactoring Replace Temp with Query. Replace temp variable frequentRenterPoints with query method getTotalFrequentRenterPoints().

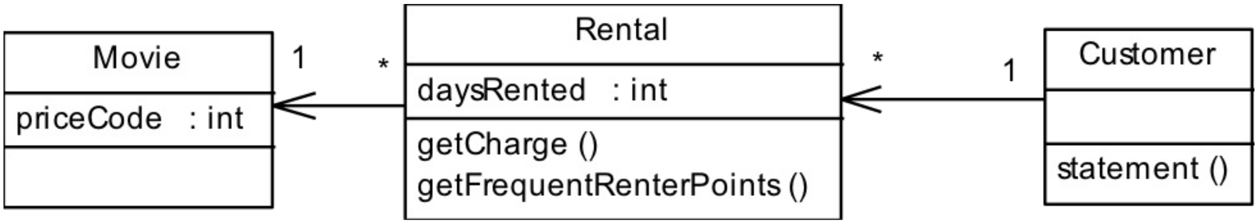
The resulting code now looks like this:



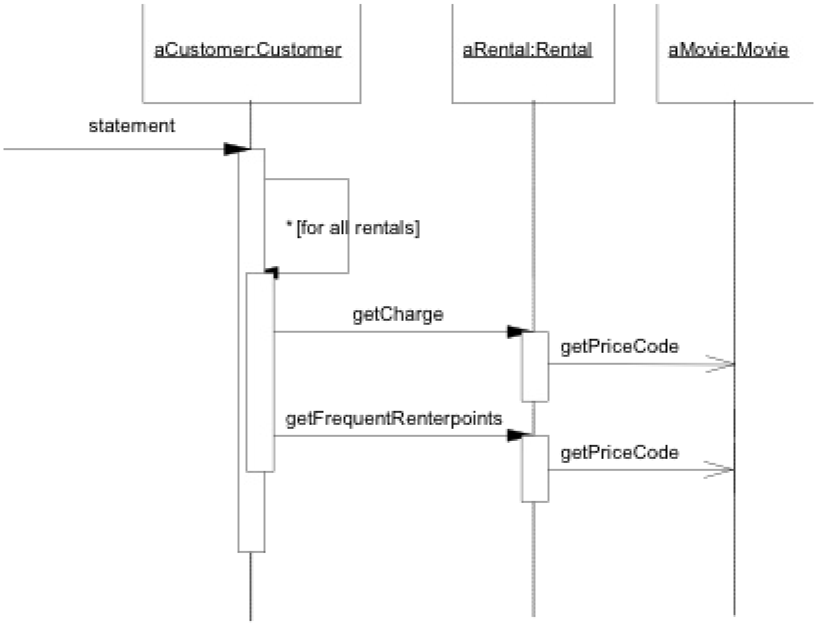
```
public String statement() {

    Enumeration<Rental> rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        result += "\t" + each.getMovie().getTitle() + "\t" + String.valueOf(each.getCharge()) + "\n";
    }

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

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

private int frequentRenterPoints() {
    int result = 0;
    Enumeration<Rental> rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        result += each.frequentRenterPointOf();
    }
    return result;
}
```

(Correction, the name in the class diagram for getFrequentRenterPointOf() is incorrect)

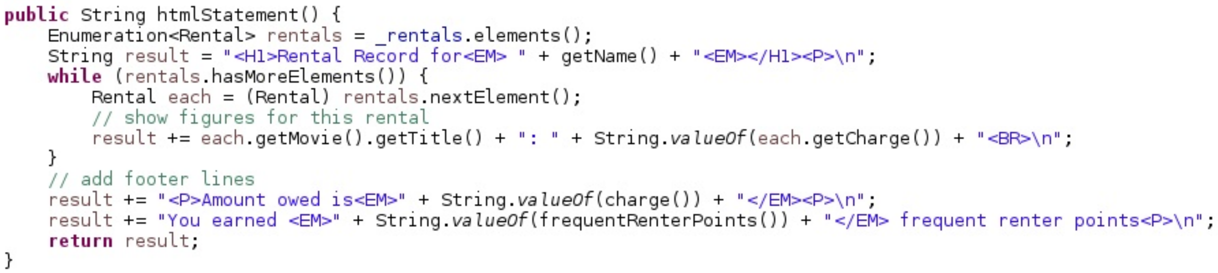
At the end of this step, the class diagram looks like:



And the sequence diagram looks like:



The requested method (htmlStatement) can now be added with minimal code duplication. (RM)



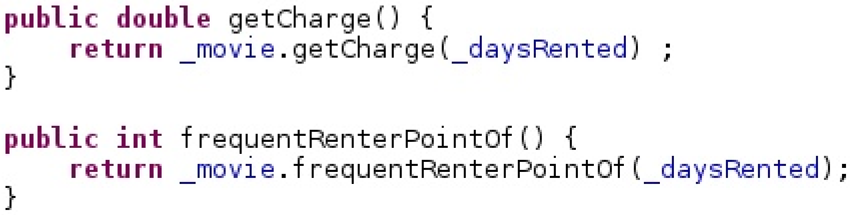
```java
public String htmlStatement() {
    Enumeration<Rental> rentals = _rentals.elements();
    String result = "<H1>Rental Record for<EM> " + getName() + "<EM></H1><P>\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        // show figures for this rental
        result += each.getMovie().getTitle() + ": " + String.valueOf(each.getCharge()) + "<BR>\n";
    }
    // add footer lines
    result += "<P>Amount owed is<EM>" + String.valueOf(charge()) + "</EM><P>\n";
    result += "You earned <EM>" + String.valueOf(frequentRenterPoints()) + "</EM> frequent renter points<P>\n";
    return result;
}
```

The solution up to the end of this exercise is in the workspace VideoStore_exercise2

### *Exercise 3.*

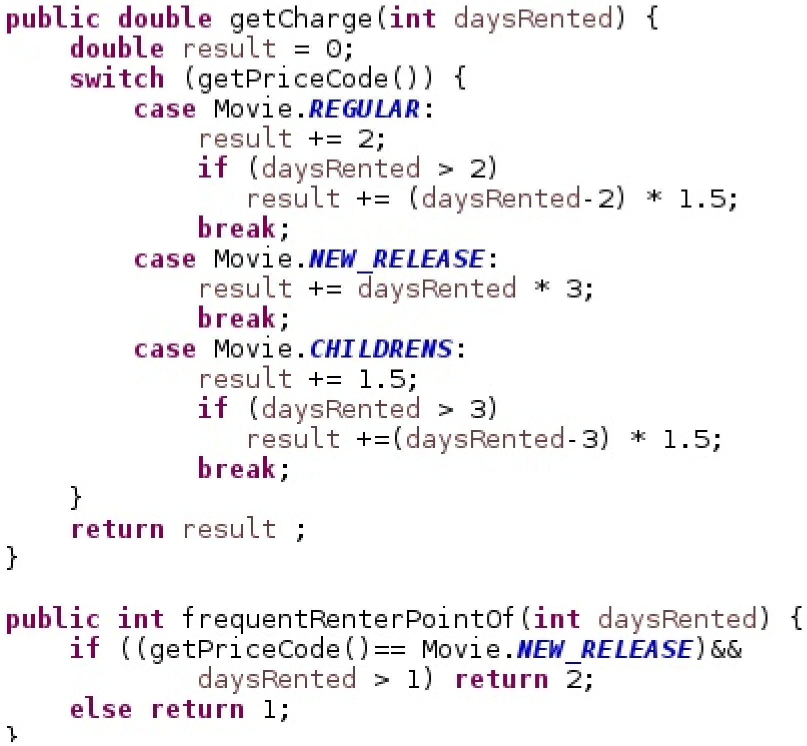
Perform the following two refactorings (RM)

1. Move the getCharge() and getFrequentRenterPointsOf() methods from Rental to movie.
2. Have the old methods in the Rental class now forward or delegate the request the methods in the movie class.



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

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

The resulting code now looks like this:



```
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 ;
}

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

The solution up to the end of this exercise is in the workspace VideoStore_exercise3

### *Exercise 4.*

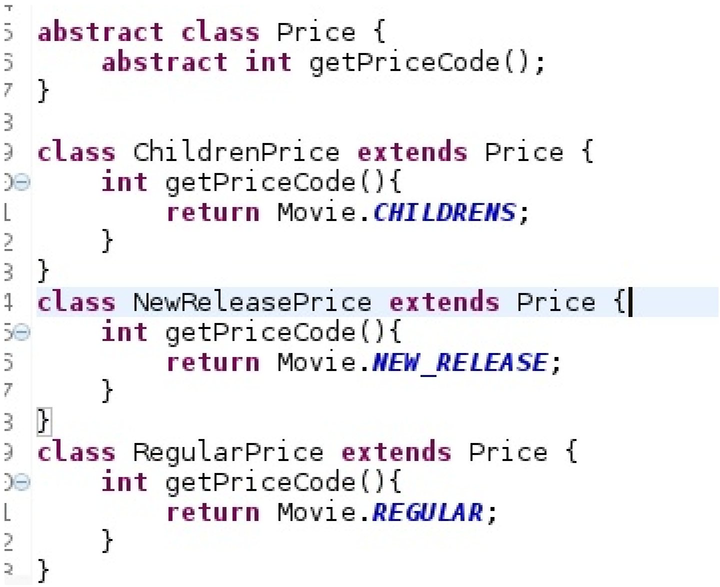
Notice that we still have blocks of conditionals for the different types of movies.  This is reminiscent of the status codes we had in our bank account status case.

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

This would allow me to replace the switch statement by using polymorphism. Sadly it has one slight flaw: it doesn't work. A move can change its classification during its lifetime. An object cannot change its class during its lifetime. There is a solution however, the state pattern  By adding the indirection we can do the subclassing from the price code object, changing the price whenever we need to. (MF)

With a complex class you have to move data and methods around in small pieces to avoid errors, it seems slow but it is the quickest because you avoid debugging. For this case I could probably move the data and methods in one go as the whole thing is not too complicated. However I'll do it the bit by bit way, so you can see how it goes. Just remember to do it one small bit at a time if you do this to a complicated class. (MF)

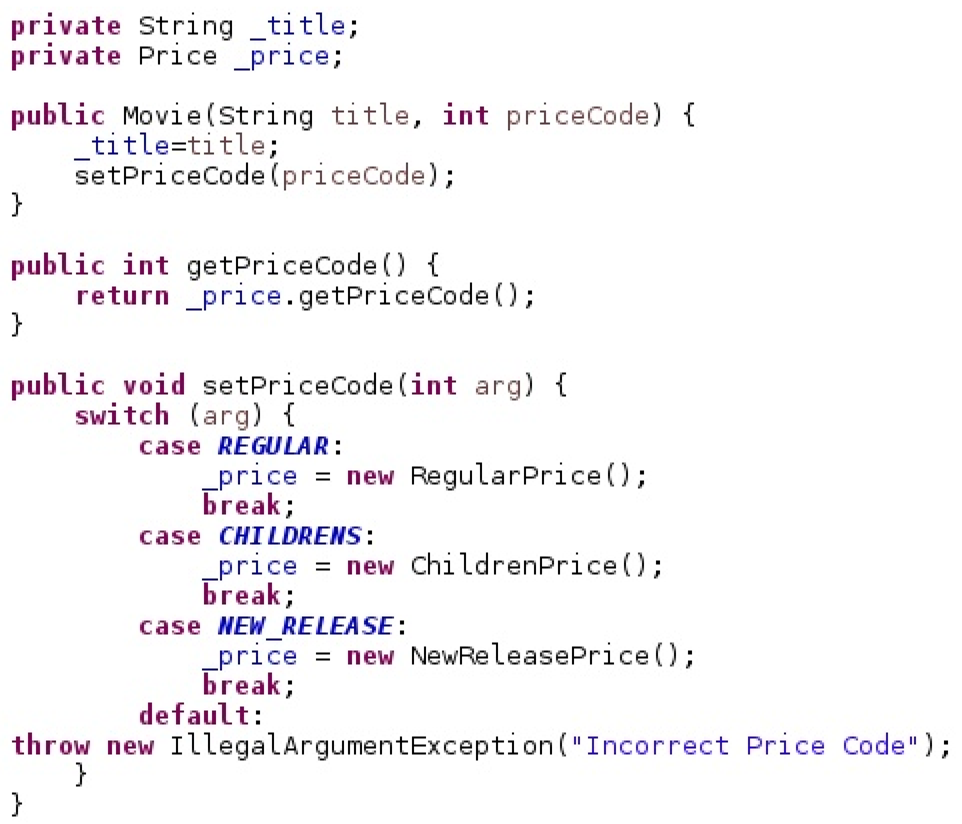
The first step is to create the new classes. (MF)  In this case we are going to use an abstract class called Price instead of an interface.



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

class ChildrenPrice 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;
    }
}
```

1. Move the type code behavior into the state pattern with Replace Type Code with State/Strategy. (RM)
2. Then we replace the price code with a price field and change the accessors (getPriceCode and setPriceCode):

The resulting code now looks like this:

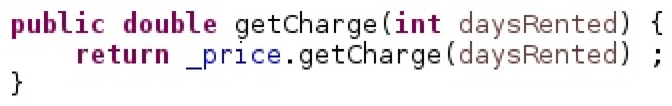


```
private String _title;
private Price _price;

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

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

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

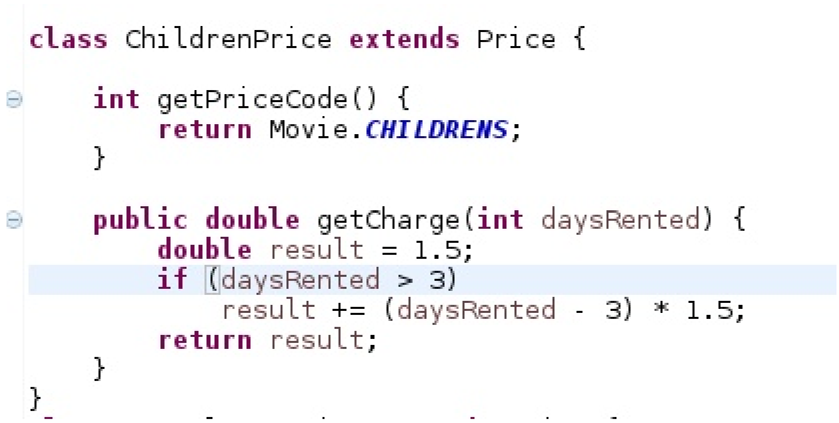
The solution up to the end of this exercise is in the workspace VideoStore_exercise4

### *Exercise 5.*

All that remains to be done is to move the getCharge() methods from the rather big switch statement in Movie into the individual movie types and replace the getCharge() with a forwarded method to the appropriate class.



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



```
class ChildrenPrice extends Price {

    int getPriceCode() {
        return Movie.CHILDRENS;
    }

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

The solution up to the end of this exercise is in the workspace VideoStore_exercise5