Refactoring, a first example

Martin Fowler
fowler@acm.org

Refactoring is a technique to improve the quality of existing code. It works by applying a series of small steps, each of which changes the internal structure of the code, while maintaining its external behavior. You begin with a program that runs correctly, but is not well structured, refactoring improves its structure, making it easier to maintain and extend.

When I describe refactoring to people, one of the most difficult things to get over is the rhythm of refactoring. This is the way you do small step by small step, slowly improving code quality. So it seems that the best way to deal with this is to give an example. As soon as I do this, however, I run into a big problem. If I pick a large program, then describing it and how it is refactored is too complicated for any reader to work through. However if I pick a program that is small enough to be comprehensible, then refactoring does not look like it is worthwhile.

Thus I'm in the classic bind of anyone who wants to describe techniques that are useful for real world programs. Frankly it is not worth the effort to do the refactoring that I'm going to show you on a small program like the one I'm going to use. But if the code I'm showing you is part of a larger system, then the refactoring soon becomes important. So I have to ask you to look at this and imagine in the context of a much larger system.

The Starting Point

The sample program is quite simple. It is a program to calculate and print out 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 identifies the type of 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.

There are several classes that represent various video elements. Here's a class diagram to show them.

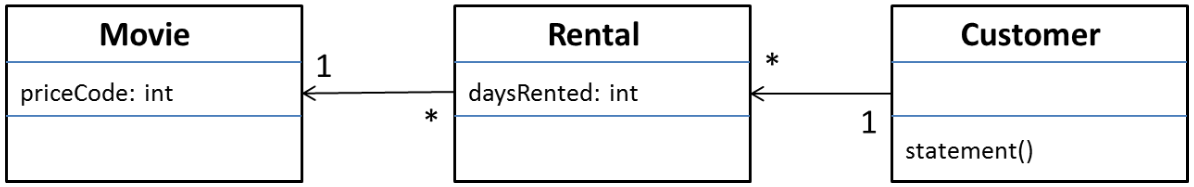
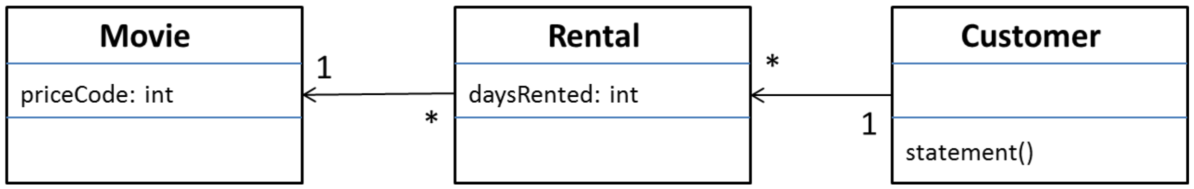


Figure 1. A class diagram of the starting point classes. Only the most important features are shown. The notation is UML

Movie

Movie is just a simple data class.

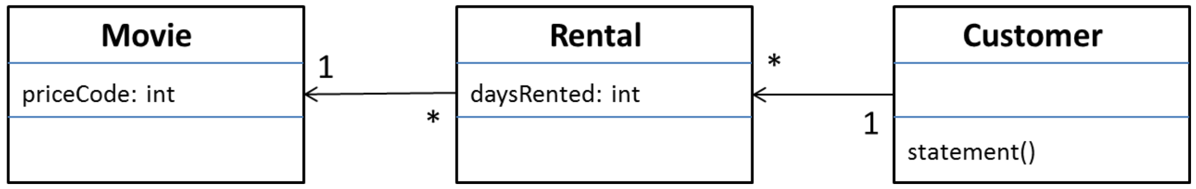
```
public class Movie {

  public static final int  CHILDREN = 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) {
    _title = title;
    _priceCode = priceCode;
  }

  public int getPriceCode() {
    return _priceCode;
  }

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

  public String getTitle() {
    return _title;
  }
}
```



Rental

The rental class represents a customer renting a movie.

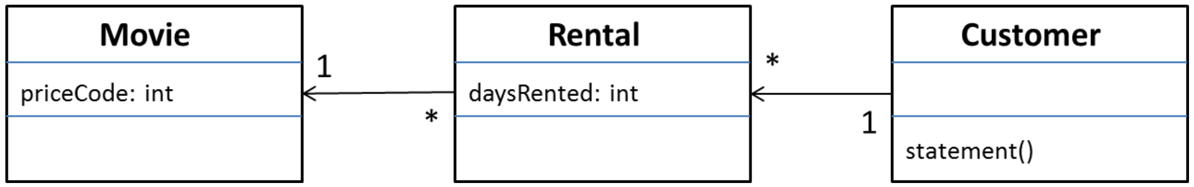
```
class Rental {

  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;
  }
}
```



Customer

The customer class represents the customer of the store. Like the other classes it has data and accessors.

```
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;
  }
```



So far all the classes have been dumb encapsulated data. Customer also has the method that produces a statement. See next page.

```
public String statement() {

 double totalAmount = 0;
 int frequentRenterPoints = 0;
 Enumeration rentals = _rentals.elements();
 String result = "Rental Record for " + name() + "\n";

 while (rentals.hasMoreElements()) {
   double thisAmount = 0;
   Rental each = (Rental) rentals.nextElement();

   //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.CHILDREN:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
          thisAmount += (each.getDaysRented() - 3) * 1.5;
     break;
   }

   // 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;

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

What are your impressions about the design of this program?

I would describe as not well designed, and certainly not object-oriented. For a simple program as this, that does not really matter. There's nothing wrong with a quick and dirty *simple* program. But if we imagine this as a fragment of a more complex system, then I have some real problems with this program.

That long statement routine in the Customer does far too much. Many of the things that it does should really be done by the other classes.

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 complicated. 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.

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.

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 <u>first refactor</u> the program to make it easy to add the feature, <u>then add the feature</u>.

Whenever I do refactoring, the first step is always the same. I need to build a solid set of testsfor that section of code. The tests are essential because even though I follow refactorings structured to avoid most of the opportunities for introducing bugs, I'm human and make mistakes. Thus I need solid tests.

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

Self-checking = example: when comparing strings, tests either say 'OK' if strings compared to a reference string are equal, or they print a list of failures, i.e., strings that are not equal (although they should).

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 (highlighted red below).

```
 public String statement() {

   double totalAmount = 0;
   int frequentRenterPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = "Rental Record for " + name() + "\n";

   while (rentals.hasMoreElements()) {
      double thisAmount = 0;
      Rental each = (Rental) rentals.nextElement();

      //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.CHILDREN:
             thisAmount += 1.5;
             if (each.getDaysRented() > 3)
               thisAmount += (each.getDaysRented() - 3) * 1.5;
        break;
      }

      // 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;

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

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.

```
public String statement() {

  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + name() + "\n";

  while (rentals.hasMoreElements()) {
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement();

    //determine amounts for each line
    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;

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

Extracted method:

```
private int amountFor(Rental each) {

  int 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.CHILDREN:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
            thisAmount += (each.getDaysRented() - 3) * 1.5;
     break;
  }
  return thisAmount;
}
```

Question: Will all tests pass as expected?

When I did this, the tests blew up. A couple of the test figures gave me the wrong answer. I was puzzled for a few seconds then realized what I had done. Foolishly I had made the return type of amountFor() int instead of double.

```
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.CHILDREN:
      thisAmount += 1.5;
      if (each.getDaysRented() > 3)
         thisAmount += (each.getDaysRented() - 3) * 1.5;
    break;
  }
  return thisAmount;
}
```

It's the kind of silly mistake that I often make, and it can be a pain to track down as Java converts ints to doubles without complaining (but merrily rounding). Fortunately it was easy to find in this case, because the change was so small. Here is the essence of the refactoring process illustrated by accident. Because each change is so small, any errors are very easy to find. You don't spend long debugging, even if you are as careless as I am.

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

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 amountFor() and this is a good place to change them.

Original code:

```
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.CHILDREN:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
            thisAmount += (each.getDaysRented() - 3) * 1.5;
     break;
  }
  return thisAmount;
}
```

Renamed code:

```
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.CHILDREN:
        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.

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

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. In most cases a method should be on the object whose data it uses. This method is thus on the wrong object, it should be moved to the rental.

```
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.CHILDREN:
      result += 1.5;
      if (aRental.getDaysRented() > 3)
         result += (aRental.getDaysRented() - 3) * 1.5;
    break;
  }
  return result;
 }
```

To *move a method* you first copy the code over to rental, adjust it to fit in its new home and compile.

```
class Rental...

 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.CHILDREN:
      result += 1.5;
      if (getDaysRented() > 3)
         result += (getDaysRented() - 3) * 1.5;
    break;
  }
  return result;
 }
```

In this case fitting into the new home means to remove the parameter aRental. I also renamed the method to getCharge() 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.

```
class Customer...

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

The next step is to find every reference to the old method, and adjusting the reference to use the new method. In this case, this step is easy as we just created the method and it is in only one place:

```
class Customer...

 public String statement() {

  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + name() + "\n";

  while (rentals.hasMoreElements()) {
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement();

    //determine amounts for each line
    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;

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

In general, however, you need to do a 'find' across all the classes that might be using that method.

I change amountFor(each) to each.getCharge().

```
class Customer...

 public String statement() {

   double totalAmount = 0;
   int frequentRenterPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = "Rental Record for " + name() + "\n";

   while (rentals.hasMoreElements()) {
      double thisAmount = 0;
      Rental each = (Rental) rentals.nextElement();

      //determine amounts for each line
      thisAmount = each.getCharge();

      // 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;

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

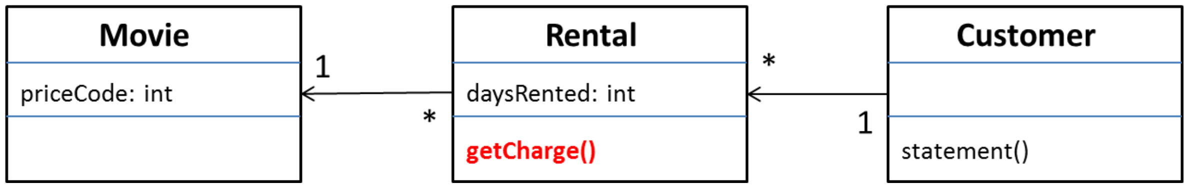


Figure 2. State of classes after moving the charge method.

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(), e.g., get rid of the switch statement, but I will leave it for the moment and return to Customer.statement().

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*.

```
 public String statement() {

  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + name() + "\n";

  while (rentals.hasMoreElements()) {
    ~~double thisAmount = 0;~~
    Rental each = (Rental) rentals.nextElement();

    ~~//determine amounts for each line~~
    ~~thisAmount = each.getCharge();~~

    // 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();

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

I like to get rid of temporary variables like thisAmount as much as possible. Temps are often a problem in that they cause a lot of parameters to get passed around when they don't need to. You can easily lose track of what they are there for. They are particularly insidious in long methods.

Of course there is a small 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 refactored.

Extracting Frequent Renter Points

The next step is to do a similar thing for the frequent renter points (i.e., their calculation shouldn't be defined in class Customer but in class Rental). Again the rules vary with the type, although there is less variation than with the charging. But 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 (highlighted red below).

```
public String statement() {

  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + name() + "\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();

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

Again we look at the use of locally scoped variables. Again it uses each, which can be passed in as a parameter. The other temp used is frequentRenterPoints. In this case frequentRenterPoints does have a value beforehand. The body of the extracted method doesn't read the value, however, so we don't need to pass it in as a parameter as long as we use an appending assignment.

```
class Customer...

 public String statement() {

   double totalAmount = 0;
   int frequentRenterPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = "Rental Record for " + name() + "\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 lines
   result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
   result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";
   return result;
 }


class Rental...

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

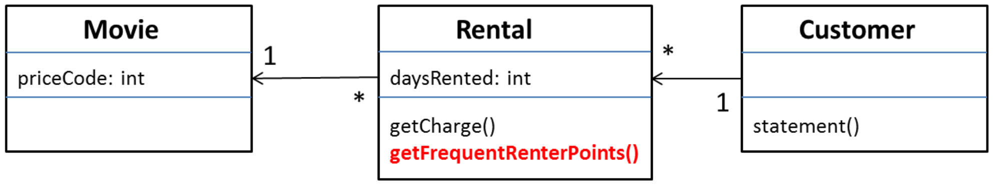


Figure 3. State of classes after extraction and moving of the frequent renter points calculation.

Removing Temps

As I suggested before, temporary variables can be a problem. They are only useful 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 will require these totals.

I like to *replace temps with queries*. 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 getTotalCharge() method on customer.

```
class Customer...

 public String statement() {

  ~~double totalAmount = 0;~~
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + name() + "\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(getCharge()) + "\n";

    ~~totalAmount += each.getCharge();~~

  }
  //add footer lines
  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 had to copy the loop into the query method.

After compiling and testing that refactoring, I then did the same for frequentRenterPoints, i.e., I change frequentRenterPoints to getTotalFrequentRenterPoints().

```
class Customer...

 public String statement() {

  ~~int frequentRenterPoints = 0;~~
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + name() + "\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(getCharge()) + "\n";

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

 private double getTotalFrequentRenterPoints(){

  double result = 0;
  Enumeration rentals = _rentals.elements();

  while (rentals.hasMoreElements()) {
    Rental each = (Rental) rentals.nextElement();
    result += each.charge();
  }
  return result;
 }
```

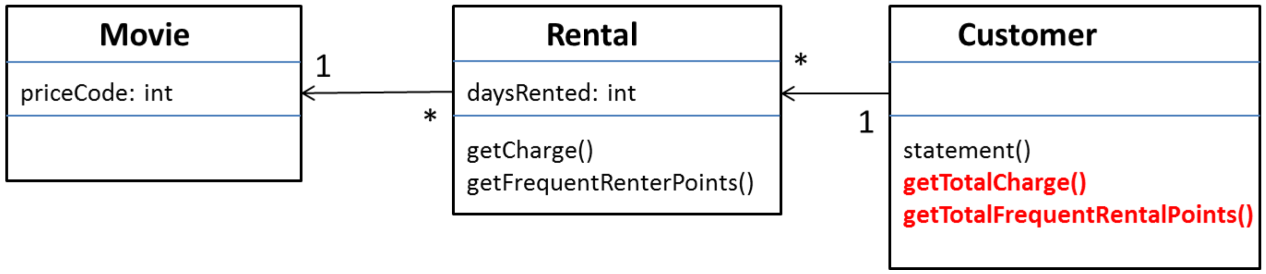


Figure 4. State of classes after extraction of the totals.

It is worth stopping and thinking a bit about the last refactoring. Most refactorings reduce the amount of code, but this one increases it. That's because Java requires a lot of statements to set up a summing loop. Even a simple summing loop with one line of code per element needs six lines of support around it. It's an idiom that is obvious to any programmer but it is a lot of lines all the same.

The other concern with this refactoring lies in performance. The old code executed the while loop once, the new code executes it three times. If the while loop takes time, this might significantly 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 to affect the overall performance of the system. While some loops do cause performance issues, most do not. So while refactoring don't worry about this. 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. (For a good discussion on why it is better to write clearly first and then optimize, see [McConnell, Code Complete].

These queries are now available to any code written in the customer class. Indeed they can easily be added to the interface of the class should other parts of the system need this information. Without queries like these, other methods need to deal with knowing about the rentals and building the loops. In a complex system that will lead to much more code to write and maintain.

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() like this (and add an appropriate test).

```
public String htmlStatement() {

   Enumeration rentals = _rentals.elements();
   String result = "<H1>Rentals for <EM>" + name() + "</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.

There is still some code copied from the ASCII version, but that is mainly due to setting up the loop. Further refactoring could clean that up further, extracting methods for header, footer, and detail line are one route I could take.

But that isn't where I want to spend my time now, because now the users are clamoring again. They are getting ready to change the classification of the movies in the store. It's still not clear what changes they want to make, but it sounds new classifications will be introduced, and the existing ones could well be changed. The charges and frequent renter point allocations are to be decided. At the moment, these kinds of changes are awkward. I have to get into the charge and frequent renter point methods and alter the conditional code (switch statement) to make changes to movie classifications. I would like to move onto the methods I've moved onto rental.

Back on with the refactoring hat …

Replacing the Conditional Logic on Price Code with Polymorphism

Yes it's that switch statement that is bugging me. 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 the charge should move onto movie.

```
Class Movie …

  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.CHILDREN:
        result += 1.5;
        if (daysRented > 3)
          result += (daysRented - 3) * 1.5;
        break;
    }
    return result;
  }
```

For this to work I have to pass in the length of the rental, which of course is data due of the rental. The method effectively uses two pieces of data, the length (number of days) of the rental and the type of the movie.

Why do I prefer to pass the length of rental rather than the movie's type? It's because type information tends to be more volatile. I can easily imagine new types of videos appearing. If I change the movie's type I want the least ripple effect, so I prefer to calculate the charge within the Movie class.

I compiled the method into movie and then adjusted the charge method on rental to use the new method.

```
Class Rental…

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

Once I have moved the getCharge() method from class Rental to class Movie, I do the same with the frequent renter points calculation. That keeps both things that vary with the movie type together on the class that has the type.

Thus …

```
class Rental...

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

… becomes …

```
 class Rental…

  int getFrequentRenterPoints() {
    return _movie.getFrequentRenterPoints(_daysRented);
  }


 class Movie…

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

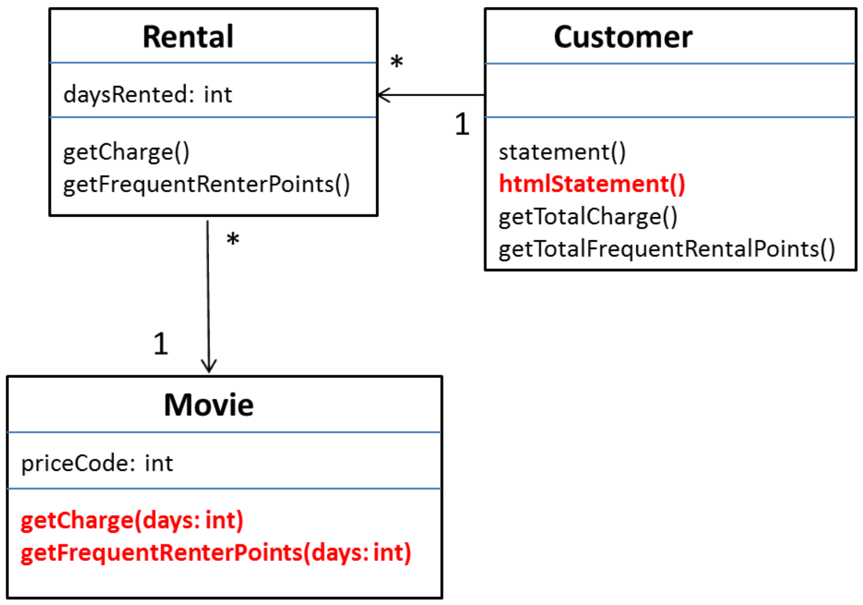


Figure 5. Class diagram after functional enhancement and after moving methods to class Movie.

At last… inheritance

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.

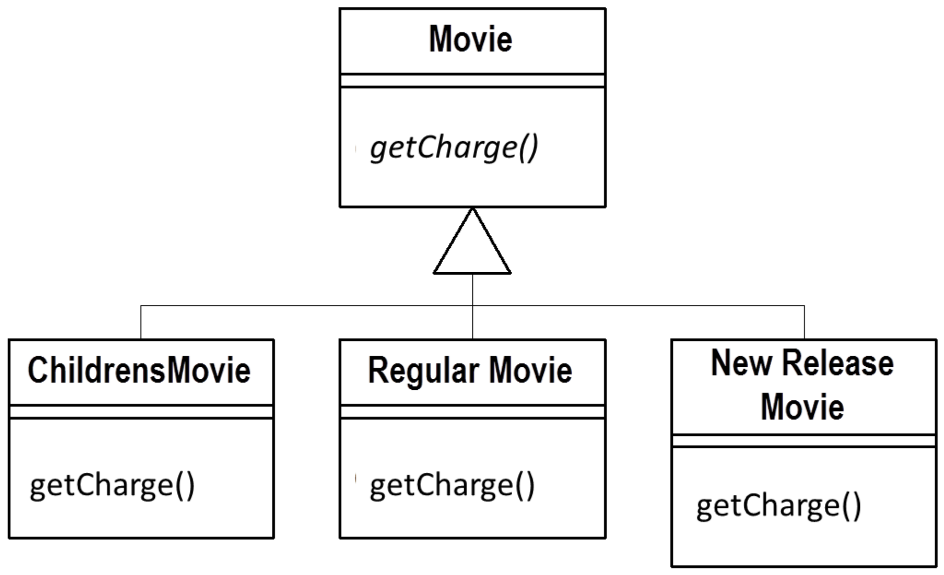


Figure 6. Using inheritance on class Movie.

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* [Gang of Four].

With the state pattern the classes look like this.

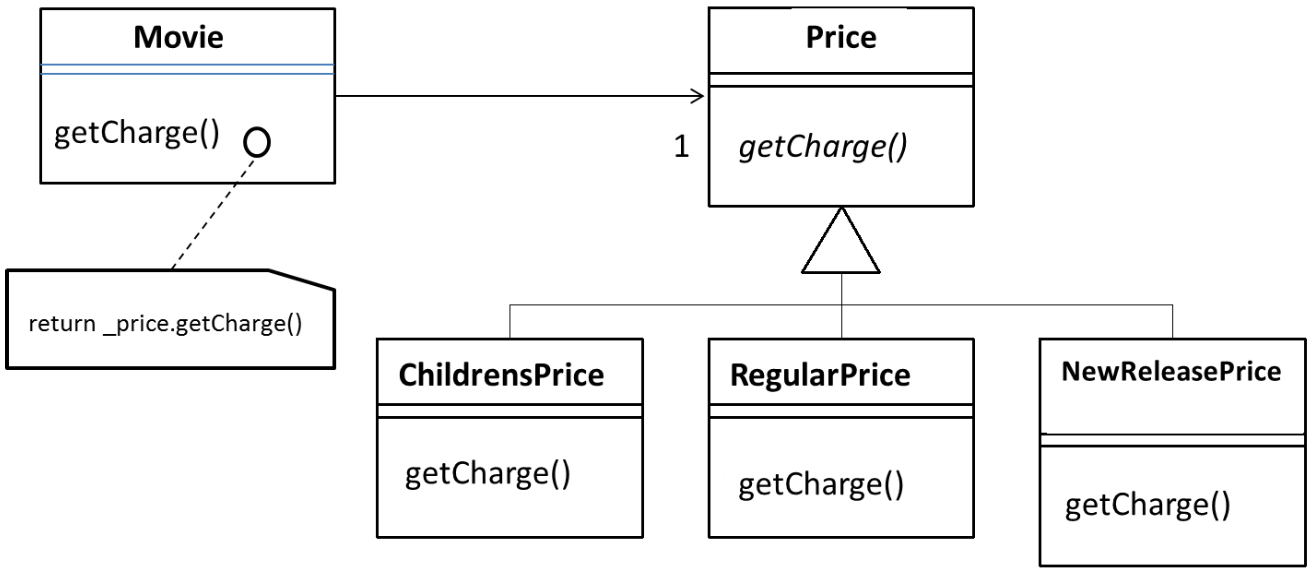


Figure 7. Using the State pattern on class Movie.

By adding the indirection we can do the sub-classing from the price code object, changing the price whenever we need to.

If you are familiar with the Gang of Four design patterns, you may wonder, 'Is this a State, or is it a Strategy?'. Does the class Price represent an algorithm for calculating the price 8in which case I prefer to call it Pricer or PricingStrategy), or does it represent a state of the movie (Movie XYZ is a new release). At this stage the choice of pattern (and name) reflects how you want to think about the structure. At the moment, I am thinking about this as a state of movie. If I later decide a strategy communicates my intention better, I will refactor to do this by changing the names.

To introduce the State pattern I will use three refactorings:

1. Move the type code behavior into the State pattern with *Replace Type Code with State/Strategy*.
2. Use *Move Method* to move the switch statement into the class Price.
3. Use *Replace Conditional with Polymorphism* to eliminate the switch statement.

Replace Type Code with State/Strategy:

To do this, I first 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 use the getting method. However, the constructors do access the Price code:

```
public class Movie…

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

I can use the setting method instead.

```
public class Movie…

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

I compile and test to make sure I didn't break anything.

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 sub-classes.

```
abstract class Price {
 abstract int getPriceCode() {
}
class ChildrensPrice extends Price {
 int getPriceCode() {
   Return Movie.CHILDREN;
 }
}
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:

```
private int _priceCode;

 public int getPriceCode() {
    return _priceCode;
 }

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

This means replacing the price code field , and changing the accessors:

```
class Movie…

 private Price _price;

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

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

Move Method:

Now I apply Move Method to getCharge().

```
Class Movie …

  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.CHILDREN:
        result += 1.5;
        if (daysRented > 3)
         result += (daysRented - 3) * 1.5;
        break;
    }
    return result;
  }
```

… becomes …

```
Class Movie …

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


Class Price …

 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.CHILDREN:
       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*.

Replace Conditional with Polymorphism:

```
Class Price …

  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.CHILDREN:
        result += 1.5;
        if (daysRented > 3)
          result += (daysRented - 3) * 1.5;
        break;
    }
    return result;
  }
```

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

```
Class RegularPrice…

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

This will override the parent case statement, which I just leave as it is. I compile and test for this case, then take the next leg, compile and test….

```
Class ChildrensPrice…

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

```
Class NewReleasePrice…

 double charge(int daysRented){
   return daysRented * 3;
 }
```

When I've done that with all the legs, I declare the Price.getCharge() method abstract.

```
Class Price…

    abstract double getCharge(int daysRented);
```

I can now do the same procedure with getFrequentRenterPoints().

```
class Movie…

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

First I move the method over to Price.

```
class Movie…

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


class Price…

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

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

```
Class NewReleasePrice…

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

Class Price…

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

Now I have removed all the methods that needed a price code. So I can get rid of the price code methods and data on both `Movie` and `Price`.

Putting in the state pattern was quite an effort. Was it worth it? The gain is now that should I change any of price's behavior, add new prices, or add extra price dependent behavior; it will be much easier to change. 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. But 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 like this, but not once did I have to open the debugger. So the process actually flowed quite quickly.

I've now completed the second major refactoring. 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.

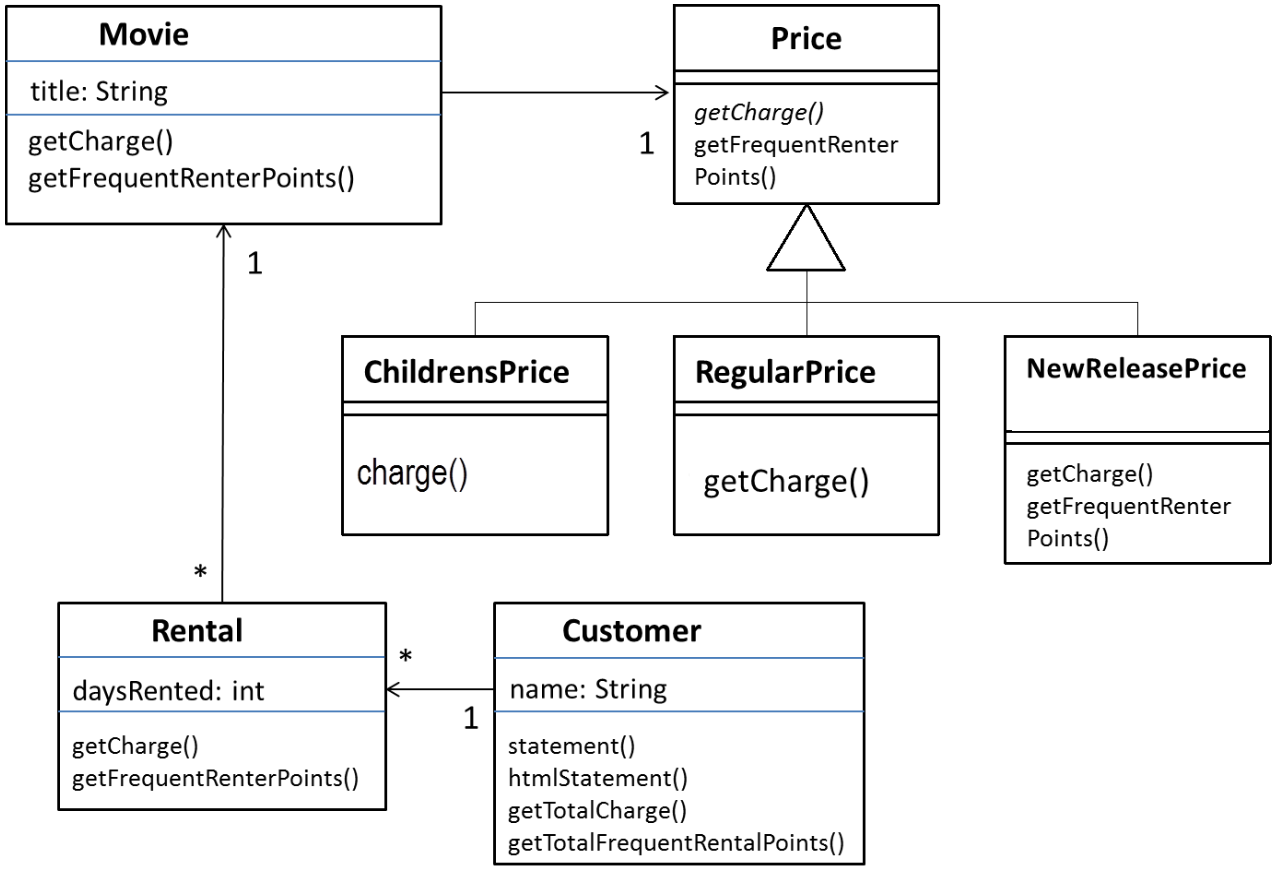


Figure 8. Class diagram after addition of the state pattern.

Final Thoughts

This is a simple example, yet I hope it gives you the feeling of what refactoring is like. I've used several refactoring techniques: moving behavior, replacing case statements, method extraction. All these lead to a better distributed responsibilities, and code that is easier to maintain. It does look rather different to procedural style code, and that does take some getting used to. But once you are used to it, it is hard to go back to procedural programs.

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.

References

[Fowler]
Fowler M with Scott K, *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley, Reading MA, 1997

[Gang of Four]
Gamma E, Helm R, Johnson R, and Vlissides J, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, Reading MA, 1995

[McConnell, Code Complete]
McConnell Steve, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, 1993