Chapter 1: Refactoring, a first example

How do I begin to talk about refactoring? The traditional way to begin talking about something is to outline the history, broad principles, and the like. When somebody does that at a conference I get slightly sleepy. My mind starts wandering, with a low priority background process polling the speaker until they give an example.

The examples wake me up because it is with examples that I can see what is going on. With principles it is too easy to make broad generalizations, too hard to figure out how to apply things. An example helps make things clear.

So I'm going to start this book with an example of refactoring. During the process I'll tell you a lot about how refactoring works, and give you a sense of the process of refactoring. I can then do the usual principles style introduction.

However with an introductory example, or even with the more detailed examples later on, 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 it in the context of a much larger system.

The Starting Point

The sample program is quite simple. It is a program to print out a statement of a customer's charges at a video store. There are several classes that represent various video elements. Here's a class diagram to show them.

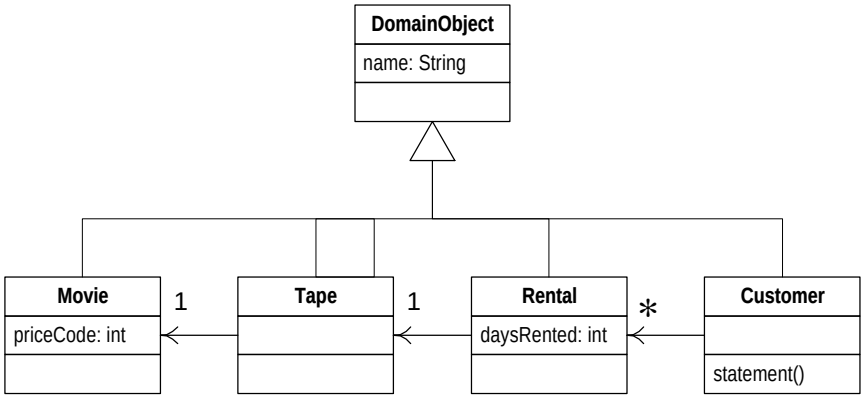


Figure 1.1: A class diagram of the starting point classes. Only the most important features are shown. The notation is UML [Fowler, UML]

I'll describe each of these classes in turn.

Domain Object

DomainObject is a general class that adds a name.

```
public class DomainObject {

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

  public DomainObject ()               {};

  public String name ()                {
      return _name;
  };

  public String toString() {
      return _name;
  };

  protected String _name = "no name";
}
```

Movie

Movie represents the notion of a film. A video store might have several tapes in stock of the same movie

```
public class Movie extends DomainObject {
    public static final int  CHILDRENS = 2;
    public static final int  REGULAR = 0;
    public static final int  NEW_RELEASE = 1;


  private int _priceCode;

  public Movie(String name, int priceCode) {
      _name = name;
      _priceCode = priceCode;
  }

  public int priceCode() {
      return _priceCode;
  }

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

  public void persist() {
      Registrar.add ("Movies", this);
  }

  public static Movie get(String name) {
      return (Movie) Registrar.get ("Movies", name);
  }
}
```

The movie uses a class called a registrar (not shown) as a class to hold instances of movie. It also does this with other classes. It uses the message persist to tell an object to save itself to the registrar. It can then retrieve the object, based on its name, with a get(String) method. This behavior won't be considered in the refactoring, but is used in testing.

Tape

The tape represents a physical tapes. Obviously there can be many tapes in the shop for the same movie.

```
class Tape extends DomainObject
    {
    public Movie movie() {
      return _movie;
    }
    public Tape(String serialNumber, Movie movie) {
      _serialNumber = serialNumber;
      _movie = movie;
    }
    private String _serialNumber;
    private Movie _movie;
    }
```

Rental

The rental class represents a customer renting a movie.

```
class Rental extends DomainObject
    {
    public int daysRented() {
      return _daysRented;
    }
    public Tape tape() {
      return _tape;
    }
    private Tape _tape;
    public Rental(Tape tape, int daysRented) {
      _tape = tape;
      _daysRented = daysRented;
    }
    private int _daysRented;
    }
```

Customer

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

```
class Customer extends DomainObject
    {
    public Customer(String name) {
        _name = name;
    }
    public void addRental(Rental arg) {
      _rentals.addElement(arg);
    }
    public static Customer get(String name) {
      return (Customer) Registrar.get("Customers", name);
    }
    public void persist() {
      Registrar.add("Customers", this);
    }
    private Vector _rentals = new Vector();
    }
```

Customer also has the method that produces a statement. Figure 1.2 shows the interactions for this method, the body for this method is on the facing page.

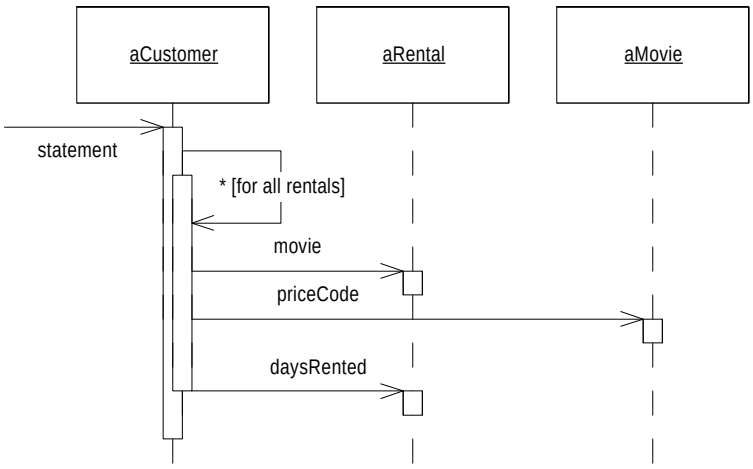


Figure 1.2: Interactions for the statement method

```
    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.tape().movie().priceCode()) {
                case Movie.REGULAR:
                    thisAmount += 2;
                    if (each.daysRented() > 2)
                        thisAmount += (each.daysRented() - 2) * 1.5;
                    break;
                case Movie.NEW_RELEASE:
                    thisAmount += each.daysRented() * 3;
                    break;
                case Movie.CHILDRENS:
                    thisAmount += 1.5;
                    if (each.daysRented() > 3)
                        thisAmount += (each.daysRented() - 3) * 1.5;
                    break;

            }
            totalAmount += thisAmount;

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
           if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) && each.daysRented() > 1)
frequentRenterPoints ++;

            //show figures for this rental
              result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(thisAmount) +
"\n";

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


  }
```

Comments on the starting program

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 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.

Even so the program works, is this not just an aesthetic judgement, a dislike of ugly code? It is until we want to change the system. 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 need to be made. And if it is hard to figure out what to change, there is a strong chance that the programmer will make a mistake, and introduce bugs.

In this case we have a number of changes that the users would like to make. First they want a statement printed in HTML so that they can be web-enabled and fully buzzword compliant. Consider what 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 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.

This brings me to the second change. The users want to make changes to way they classify movies. Now they haven't yet decided on the change they are going to make. They have a number of changes in

mind. This change will affect both the way movies are charged for, and they way that frequent renter points are calculated. As an experienced developer you are sure that whatever scheme they come up with, the only guarantee you're going to have is that they will change it again within six months.

Again that statement method is where the changes need to be made to deal with changes in classification and charging rules. But if we copy the statement to html statement we need to ensure that any changes are completely consistent. Furthermore as the rules grow in complexity it's going to be harder to figure out where to make the changes, harder to do them without making a mistake.

You may be tempted to make the minimum changes you can 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 broke, but it does hurt. It is making your life more difficult to make the changes your users want.

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; first refactor the program to make it easy to add the feature, then add the feature.

The First Step in Refactoring

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

Since the statement result produces a string, what I do is create a few customers, give each customer a few rentals of various kinds of films, and generate the statement strings. I then do a string comparison between the new string and some reference strings that I have hand checked. I set up all of these tests so I can run them from one java command on the command line. The tests take only a few seconds to run, and as you will see, I run them often.

An important part of the tests is the way they report their results. They either say "OK", meaning that all the strings are identical to the reference strings, or they print a list of failures: those lines that turned out differently. The tests are thus self-checking. It is vital to make tests self checking. If you don't you end up spending time hand checking some numbers from the test against some numbers of a desk pad, and that slows you down.

As we do the refactoring we will lean on the tests. I'm going to be relying on the tests to tell me if I introduce a bug. Thus it is essential for refactoring that you have good tests. But it's worth spending the time building the tests, because the tests give you the security you need to change the program later. This is such an important part of refactoring that I do into more detail on testing in Chapter 4.

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

Decomposing and Redistributing the Statement Method

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 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 refactorings in this chapter will show me splitting up the long method, and then moving the pieces to better classes. My aim is to make it easier to write an html statement method, with much less duplication of code.

My first step is to find a logical clump of code and use *Extract Method (114).* An obvious piece here is the switch statement. This looks like it would make a good chunk to extract into its own method.

When I extract a method, as in any refactoring, I need to know what can go wrong. If I do the extraction badly, I could introduce a bug into the program. So before I do the refactoring I need to figure out how to do it safely. I've done this refactoring a few times before, so I've written down the safe steps in the catalog.

First I 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 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 round the loop, and is not altered until the switch gets its hands on it. So I can just assign the result.

The next two pages show the code before and after the refactoring. The before code is on the left, the resulting code on the right. I've highlighted the code I'm extracting on the original, and highlighting any changes on the new code that I don't think is immediately obvious. As I continue with this chapter I'll continue with this left/right convention. *(Note to Reviewers: You'll need to make sure the facing pages are correct. The next two pages should be left and right respectively)*

```
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.tape().movie().priceCode()) {
                case Movie.REGULAR:
                    thisAmount += 2;
                    if (each.daysRented() > 2)
                        thisAmount += (each.daysRented() - 2) * 1.5;
                    break;
                case Movie.NEW_RELEASE:
                    thisAmount += each.daysRented() * 3;
                    break;
                case Movie.CHILDRENS:
                    thisAmount += 1.5;
                    if (each.daysRented() > 3)
                        thisAmount += (each.daysRented() - 3) * 1.5;
                    break;
            }
            totalAmount += thisAmount;

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) && each.daysRented() > 1)
frequentRenterPoints ++;

            //show figures for this rental
              result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(thisAmount) +
"\n";

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

```
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 = amountOf(each);
            totalAmount += thisAmount;

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) && each.daysRented() > 1)
frequentRenterPoints ++;

            //show figures for this rental
              result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(thisAmount) +
"\n";

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

    }

    private int amountOf(Rental each) {
        int thisAmount = 0;
        switch (each.tape().movie().priceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.daysRented() > 2)
                    thisAmount += (each.daysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.daysRented() * 3;
                break;
            case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (each.daysRented() > 3)
                    thisAmount += (each.daysRented() - 3) * 1.5;
                break;
        }
        return thisAmount;
    }
```

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 `amountOf` int instead of `double`.

```
private double amountOf(Rental each) {
    double thisAmount = 0;
    switch (each.tape().movie().priceCode()) {
        case Movie.REGULAR:
            thisAmount += 2;
            if (each.daysRented() > 2)
                thisAmount += (each.daysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            thisAmount += each.daysRented() * 3;
            break;
        case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (each.daysRented() > 3)
                thisAmount += (each.daysRented() - 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 in this case Java converts doubles to ints without complaining, but merrily rounding (see [Java Spec] §15.25.2). Fortunately it was easy to find in this case, because the change was so small and I had a good set of tests. 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: so if you make a mistake, it is easy to find where the bug is.

If I were doing this in Smalltalk, with the Refactoring Browser, then this refactoring is very simple. I just highlight the code, pick “extract method” from the menus, type in a method name, and it’s done. Furthermore it’s a tool that does it, and it doesn’t make silly mistakes like I do. I’m looking forward to a Java version!

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 `amountOf` and this is a good place to change them.

Here's the original code...

```
private double amountOf(Rental each) {
    double thisAmount = 0;
    switch (each.tape().movie().priceCode()) {
        case Movie.REGULAR:
            thisAmount += 2;
            if (each.daysRented() > 2)
                thisAmount += (each.daysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            thisAmount += each.daysRented() * 3;
            break;
        case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (each.daysRented() > 3)
                thisAmount += (each.daysRented() - 3) * 1.5;
            break;

    }
    return thisAmount;
}
```

... and here is the renamed code.

```
private double amountOf(Rental aRental) {
        double result = 0;
        switch (aRental.tape().movie().priceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (aRental.daysRented() > 2)
                    result += (aRental.daysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += aRental.daysRented() * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (aRental.daysRented() > 3)
                    result += (aRental.daysRented() - 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 a 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.*

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.

```
class Customer...
  private double amountOf(Rental aRental) {
        double result = 0;
        switch (aRental.tape().movie().priceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (aRental.daysRented() > 2)
                    result += (aRental.daysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += aRental.daysRented() * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (aRental.daysRented() > 3)
                    result += (aRental.daysRented() - 3) * 1.5;
                break;

        }
        return result;
    }
```

This immediately raises my suspicions that the method is on the wrong object; it should be moved to the rental. To do this I use *Move Method (160)*. With this you first copy the code over to rental, adjust it to fit in its new home and compile.

```
Class Rental
    double charge() {
        double result = 0;
        switch (tape().movie().priceCode()) {
            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;
    }
```

In this case fitting into its new home means removing the parameter. I also renamed the method as I did the move.

The next step is to find every reference to the old method, and adjusting the reference to use the new method.

```
class Customer
  public String statement() {
        [snip]

            //determine amounts for each line
            thisAmount = amountOf(each);
            totalAmount += thisAmount;
       [snip]
```

In this case this step is easy as we just created the method and it is in only one place. In general, however, you need to do a find across all the classes that might be using that method.

```
class Customer
  public String statement() {
        [snip]

            //determine amounts for each line
            thisAmount = each.charge();
            totalAmount += thisAmount;
          [snip]
```

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.

Sometimes you leave the old method there, but replace its body so that it just delegates to the new method. This is useful if it is a public method and you don't want to change the interface of the other class.

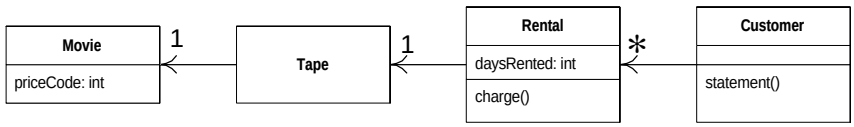


Figure 1.3: State of classes after moving the charge method (I'll ignore the domain object superclass in these diagrams from now on)

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`.

```
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 = amountOf(each);
            totalAmount += thisAmount;

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
           if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) && each.daysRented() > 1)
frequentRenterPoints ++;

            //show figures for this rental
              result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(thisAmount) +
"\n";

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

    }
```

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 using *Inline Temp (121)*.

```
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();

            //determine amounts for each line
            totalAmount += each.charge();

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) && each.daysRented() > 1)
frequentRenterPoints ++;

            //show figures for this rental
             result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(each.charge()) +
"\n";

        }
        //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 this 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 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. I'll talk more about that issue later in "Refactoring and Performance" on page 74.

Extracting Frequent Renter Points

The next step is to do a similar thing for the frequent renter points. Again the rules vary with the tape, although there is less variation than with the charging. But it seems reasonable to put the responsibility on the rental. First we need to use *Extract Method (114)* on the frequent renter points part of the code (highlighted below).

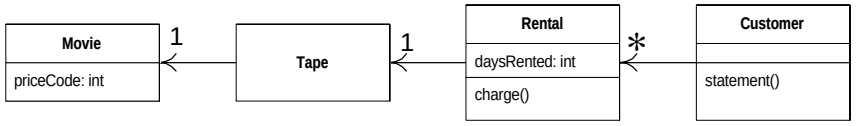
```
 statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + name() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();

            //determine amounts for each line
            totalAmount += each.charge();

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) && each.daysRented() > 1)
frequentRenterPoints ++;

            //show figures for this rental
             result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(each.charge()) +
"\n";

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

    }
```

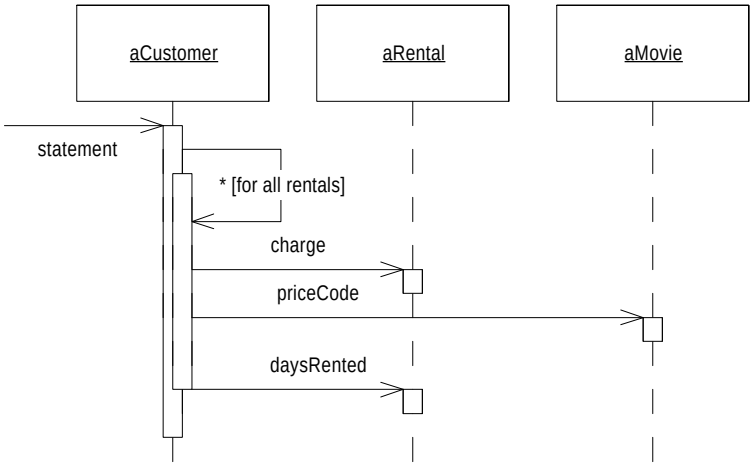
Again we look at the use of locally scoped variables. Again it uses each, which can be passed in as a parameter. The other temp used is `frequentRenterPoints`. In this case `frequentRenterPoints` does have a value beforehand. The body of the extracted method doesn't read the value, however, so we don't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

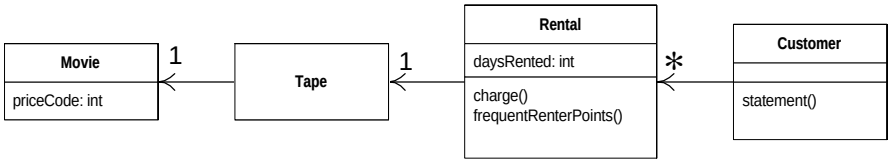
I did the extraction, compiled and tested, and then did a move, and compiled and tested again. With refactoring small steps are the best, that way less tends to go wrong.

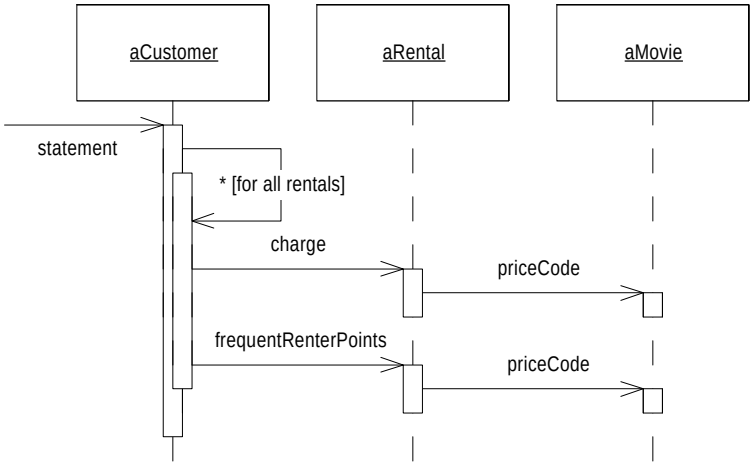
```
class Customer...
  public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + name() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each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

            //determine amounts for each line
            totalAmount += each.charge();

            // add frequent renter points
            frequentRenterPoints += each.frequentRenterPoints();

            //show figures for this rental
             result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(each.charge()) +
"\n";

        }
        //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 frequentRenterPoints() {
        if ((tape().movie().priceCode() == Movie.NEW_RELEASE) && daysRented() > 1) return 2;
        else return 1;
    }
```

I'll summarize the changes I just made by some before and after UML diagrams. Again the diagrams on the left are before the change, those on the right show after the change.






Movie
priceCode: int
1
Tape
1
Rental
daysRented: int
charge()
frequentRenterPoints()
*
Customer
statement()


aCustomer
aRental
aMovie
statement
* [for all rentals]
charge
priceCode
frequentRenterPoints
priceCode

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 use *Inline Temp (121)* 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.

```
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();

            //determine amounts for each line
            totalAmount += each.charge();

            // add frequent renter points
            frequentRenterPoints += frequentRenterPointOf(each);

            //show figures for this rental
             result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(each.charge()) +
"\n";

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

    }
```

I began by replacing `totalAmount` with a `charge` 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();

            // add frequent renter points
            frequentRenterPoints += each.frequentRenterPoints();

            //show figures for this rental
             result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(each.charge()) +
"\n";

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

    }

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

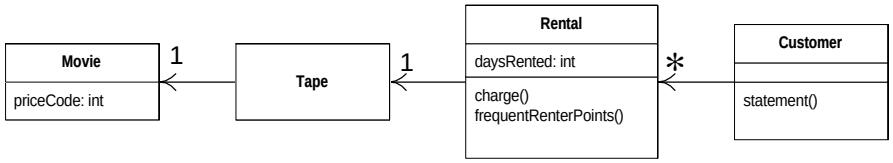
This isn't the simplest case of *Inline Temp (121)*. `totalAmount` is assigned to within the loop, so I have to copy the loop into the query method.

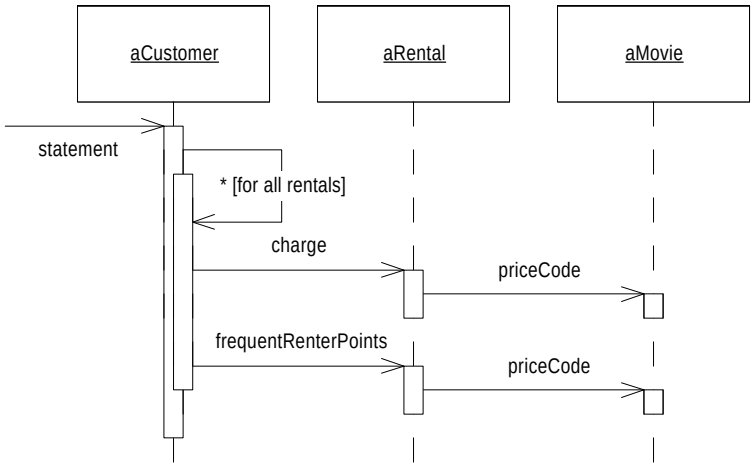
After compiling and testing that refactoring, I then did the same for `frequentRenterPoints`.

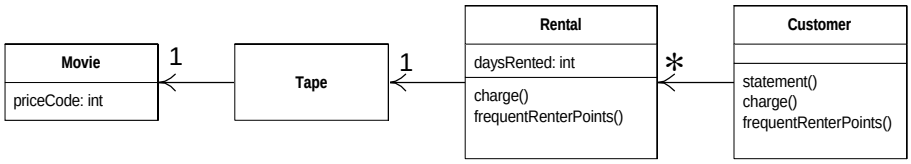
```
class Customer...
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + name() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each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

            // add frequent renter points
            frequentRenterPoints += each.frequentRenterPoints();

            //show figures for this rental
             result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(each.charge()) +
"\n";

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

    }
```

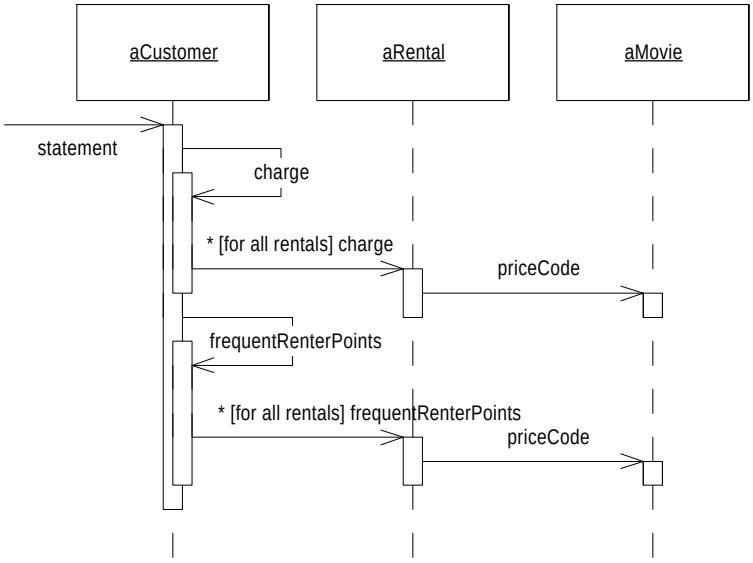
```
public String statement() {
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + name() + "\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        //show figures for each rental
        result += "\t" + each.tape().movie().name()+ "\t" +
                    String.valueOf(each.charge()) + "\n";
    }
    //add footer lines
    result +=  "Amount owed is " + String.valueOf(charge()) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints()) +
                " frequent renter points";
    return result;
}

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

The diagrams show the change for these refactorings in the class diagrams and the interaction diagram for the statement method.






Movie
priceCode: int
1
Tape
1
Rental
daysRented: int
charge()
frequentRenterPoints()
*
Customer
statement()
charge()
frequentRenterPoints()


aCustomer
aRental
aMovie
statement
charge
* [for all rentals] charge
priceCode
frequentRenterPoints
* [for all rentals] frequentRenterPoints
priceCode

It is worth stopping and thinking a bit about this 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 noise that hides what the intent of the loop is. As Java develops and builds up its ability to handle block closures in the style of Smalltalk, I expect that overhead to decrease, probably to the single line that such an expression would take in Smalltalk.

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 the loop takes to calculate, or whether the loop gets called often enough for it to affect the overall performance of the system. 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, see the discussion on page 74.

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 appropriate tests).

```
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.tape().movie().name()+ ": " +
                    String.valueOf(each.charge()) + "<BR>\n";
    }
    //add footer lines
    result +=  "<P>You owe <EM>" + String.valueOf(charge()) + "</EM><P>\n";
    result += "On this rental you earned <EM>" + String.valueOf(frequentRenterPoints()) +
                "</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 cut and paste, so if the calculation rules change I have only one place in the code to go to. Any other kinds of statement will be really quick and easy to prepare. The refactoring did not take long, most of the time was figuring out what the code did - and I would have to do that anyway.

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 (and you can see how to do this on page 289). But now the users are clamoring again. They intend to change the classification of the movies in the store. It's not clear what changes they want to make yet, but it sounds like new classifications will be introduced and the existing ones could well be changed. The charges and frequent renter point allocations for these classifications are still to be decided. At the moment, making these kind of changes will be awkward. I have to get into the charge and frequent renter point methods and alter the conditional code to make changes to film classifications.

Back on with the refactoring hat.

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.

```
class Rental...
    double charge() {
        double result = 0;
        switch (tape().movie().priceCode()) {
            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;
    }
```

This implies that the charge should move onto movie

```
class Movie …
    double charge(int daysRented) {
        double result = 0;
        switch (priceCode()) {
            case REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case NEW_RELEASE:
                result += daysRented * 3;
                break;
            case CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
      }
        return result;
    }
```

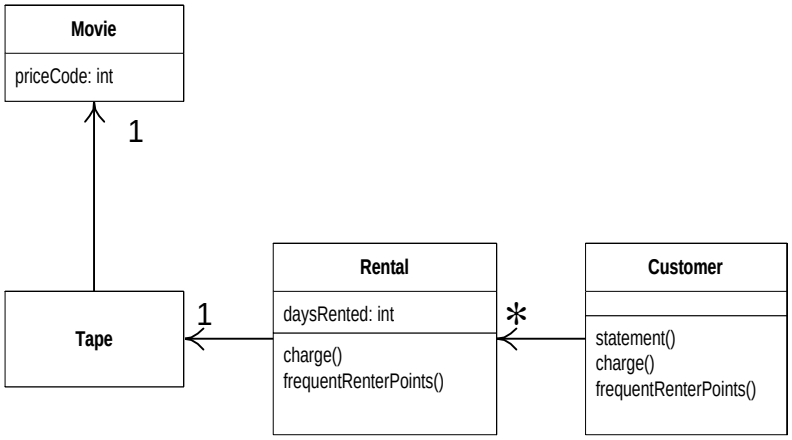
For this to work I have to pass in the length of the rental, which of course is data from the rental. The method effectively uses two pieces of data, the length of the rental and the type of the movie. Why do I prefer to pass the length of rental to the movie rather than the movie's type to the rental? It's because the proposed changes are all about adding new types. Indeed generally type information tends to be more volatile. If I change the movie's type I want the least ripple effect, so I prefer to calculate the charge within the movie.

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

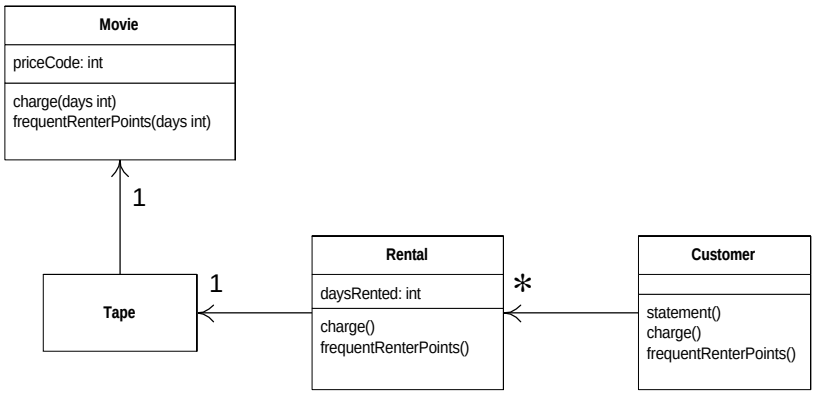
```
class Rental…
    double charge() {
        return _tape.movie().charge(_daysRented);
    }
```

Once I've moved the with charge methods, 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.

```
class Rental...
  int frequentRenterPoints() {
       if ((tape().movie().priceCode() == Movie.NEW_RELEASE) && daysRented() > 1) return 2;
       else return 1;
    }
```



```
Class rental…
    int frequentRenterPoints() {
        return _tape.movie().frequentRenterPoints(_daysRented);
    }

class movie…
    int frequentRenterPoints(int daysRented){
        if ((priceCode() == NEW_RELEASE) && daysRented > 1) return 2;
        else return 1;
    }
```



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.

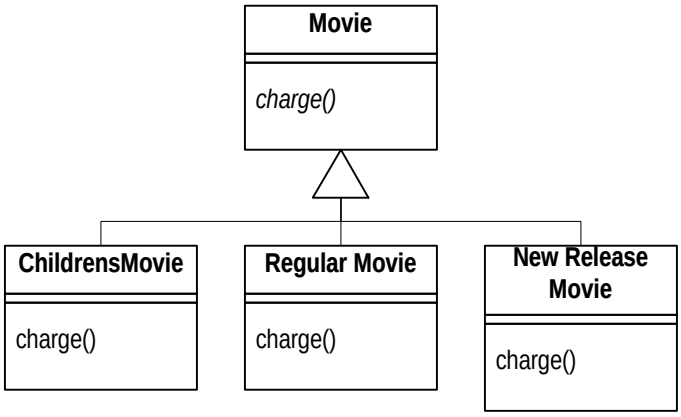


Figure 1.4: Using subclasses for the kinds of movies

This would allow 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* [Gang of Four]. With the state pattern the classes look like this.

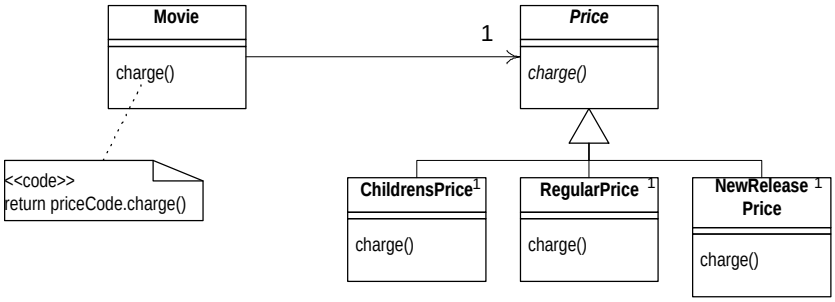


Figure 1.5: Using the state pattern for the type of movie.

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

If you are familiar with the Gang of Four patterns, you may wonder "is this a state or is it a strategy?". Does the price class represent an algorithm for calculating the price (in which case I would prefer to call it Pricer or PricingStragegy), or does it represent a state of the movie (Star Trek X 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'm 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. First I'll move the type code behavior into the state pattern with *Replace Type Code with State/Strategy (227).* Then I can use *Move Method (160)* to move the switch statement into the price class. Finally I'll use *Replace Switch with Polymorphism (147)* to get rid of the switch statement.

So I begin with *Replace Type Code with State/Strategy (227)*. It's first step is to use *Self Encapsulate Field (184)* on the type code. ensuring that all uses of the type code go though getting and setting methods. Since most of the code came from other classes, most methods already use the getting method. However the constructors do access the price code.

```
class Movie...
    public Movie(String name, int priceCode) {
      _name = name;
      _priceCode = priceCode;
    }
```

I can use the setting method instead.

```
class Movie
    public Movie(String name, int priceCode) {
      _name = name;
      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.

```
abstract class Price {
  abstract int priceCode();
}
class ChildrensPrice extends Price {
  int priceCode() {
      return Movie.CHILDRENS;
  }
class NewReleasePrice extends Price {
  int priceCode() {
      return Movie.NEW_RELEASE;
  }
class RegularPrice extends Price {
  int priceCode() {
      return Movie.REGULAR;
  }
```

I can compile the new classes at this point.

Now I need to change movie's accessors for the price code to use the new class.

```
public int priceCode() {
    return _priceCode;
}
public setPriceCode (int arg) {
    _priceCode = arg;
}
private int _priceCode;
```

This means replacing the price code field with a price field, and changing the accessors.

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

I can now compile and test and the more complex methods don't realize the world has changed.

Now I apply *Move Method (160)* to `charge()`.

```
class Movie …
    double charge(int daysRented) {
        double result = 0;
        switch (priceCode()) {
            case REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case NEW_RELEASE:
                result += daysRented * 3;
                break;
            case CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
      }
        return result;
    }
```

It is simple to move.

```
class Movie…
    double charge(int daysRented) {
        return _price.charge(daysRented);
    }

class Price…
    double charge(int daysRented) {
        double result = 0;
        switch (priceCode()) {
            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 Switch with Polymorphism (147)*.

```
class Price…
    double charge(int daysRented) {
        double result = 0;
        switch (priceCode()) {
            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;
    }
```

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 charge(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…. (To make sure I'm executing the subclass code, I like to throw in a deliberate bug and run it to ensure the tests blow up. Not that I'm paranoid or anything.)

```
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.charge()` method abstract.

```
Class Price…
    abstract double charge(int daysRented);
```

I can now do the same procedure with `frequentRenterPoints()`.

```
class Rental...
  int frequentRenterPoints() {
        if ((tape().movie().priceCode() == Movie.NEW_RELEASE) && daysRented() > 1) return 2;
        else return 1;
    }
```

First I move the method over to Price.

```
Class Movie…
    int frequentRenterPoints(int daysRented){
        return _price.frequentRenterPoints(daysRented);
    }
Class Price…
    int frequentRenterPoints(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 frequentRenterPoints(int daysRented){
        return (daysRented > 1) ?
            2:
            1;
    }

Class Price…
    int frequentRenterPoints(int daysRented){
        return 1;
    }
```

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. It took me longer to write this section of the book than it did to change the code.

I've now completed the second major refactoring and the system is now is a state where 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 1.6 and Figure 1.7 show how the state pattern works with price information.

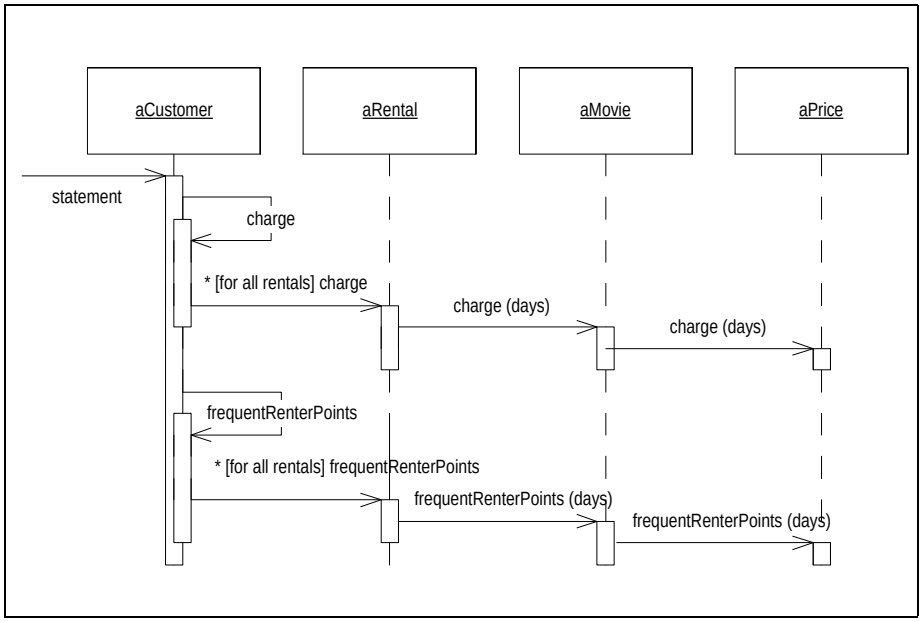


Figure 1.6: Interactions using the state pattern

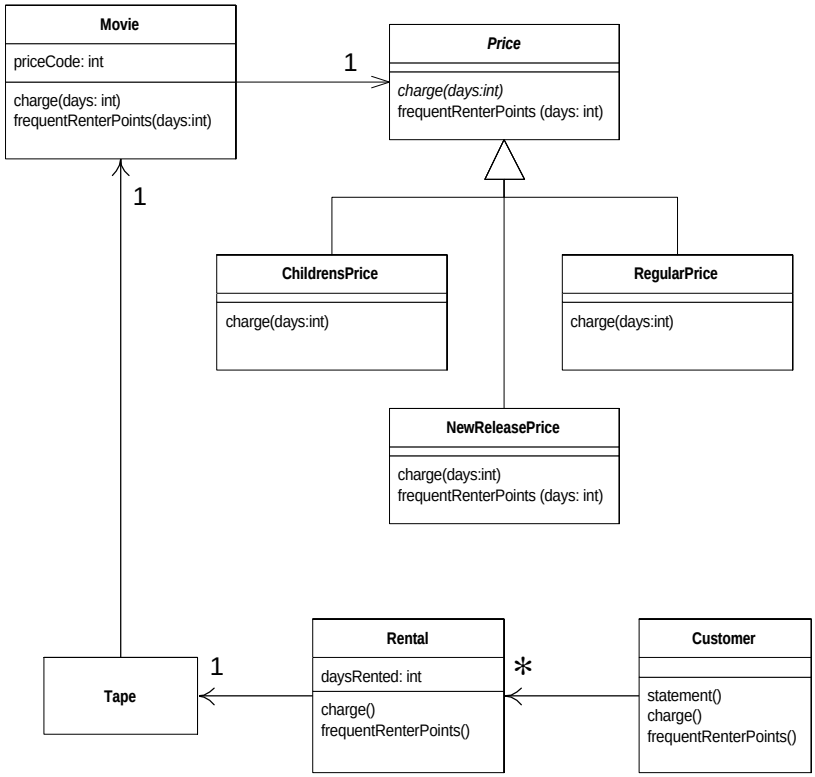


Figure 1.7: Class Diagram after adding the state pattern

Final Thoughts

This is a simple example, yet I hope it gives you the feeling of what refactoring is like. I've used several refactorings including: *Extract Method (114)*, *Move Method (160)*, and *Replace Switch with Polymorphism (147)*. All these lead to 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.

If you're with me this far you should now understand what refactoring is all about. We can now move onto some background, principles, and theory. (Although not too much!)