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.

Currently there is not much written on refactoring. I have been making a start at doing something about this with some presentations and other articles. At some point these may evolve into a book. This document is work in progress, so please be lenient with its inevitable errors!

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 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.

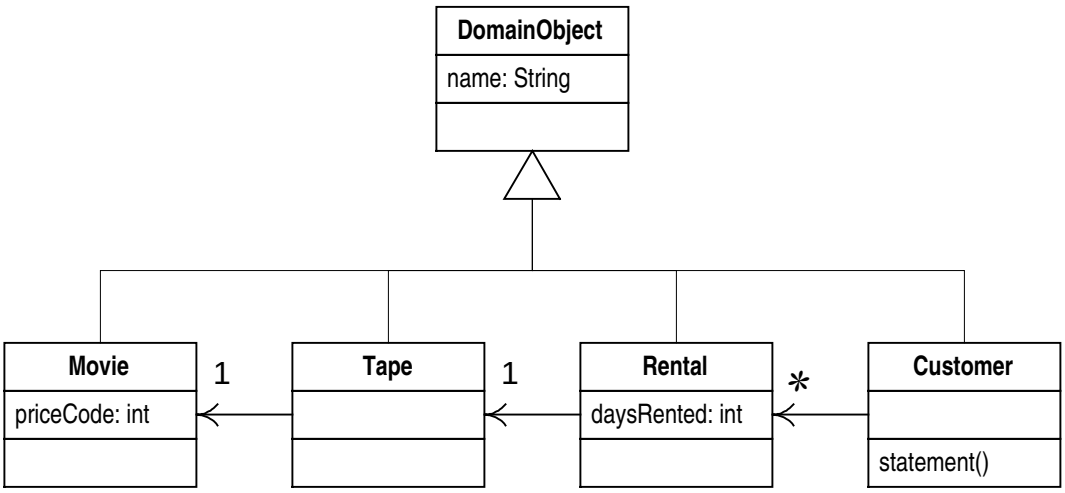


Figure 1. A class diagram of the starting point classes. Only the most important features are shown. The notation is UML [Fowler]

Here is the code for the classes. DomainObject is a general class that does a few standard things, such as hold 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 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 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. I also do this with other classes. I use the message persist to tell an object to save itself to the registrar. I can then retrieve the object, based on its name, with a get(String) method.

The tape class represents a physical tape.

```
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;
    }
```

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;
    }
```

The customer class represents the customer. So far all the classes have been dumb encapsulated data. Customer holds all the behavior for producing a statement in its `statement()` method.

```
class Customer extends DomainObject
    {
    public Customer(String name) {
        _name = name;
    }
    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 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();
    }
```

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 is 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 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.

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

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 (which I'm highlighting 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.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;

    }
```

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 = 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 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.

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

```
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 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. This method is thus on the wrong object, it should be moved to the rental. To *move a method* you first copy the code over to rental, adjust it to fit in its new home and compile.

```
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.

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. In general, however, you need to do a find across all the classes that might be using that 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
            thisAmount = each.charge();
            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;

    }
```

When I've made the change the next thing is to remove the old method. The compiler should then tell me if I missed anything.

There is certainly some more I would like to do to `Rental.charge()` but I will leave it for the moment and return to `Customer.statement()`.

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()) {
            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 thus 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. 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 extract a method from the frequent renter points part of the code (highlighted 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();

            //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.

```
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;

    }

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

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

```
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;

    }

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

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 `charge()` method on 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;
    }
```

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

```
    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;
    }
```

It is worth stopping and thinking a bit about this refactoring. Most refactoring 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". 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.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;

    }
```

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, I would like to move onto the methods I've moved onto rental. Back on with the refactoring hat.

Moving the Rental Calculations to Movie

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 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 due of 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 rather than the movie's type? Its 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.

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

```
Class rental…
    double charge() {
        return _tape.movie().charge(_daysRented);
    }
```

Some people would prefer to remove that chain of calls by having a `charge(int)` message on tape. This would lead to

```
Class rental
    double charge() {
```

```
        return _tape.charge(_daysRented);
    }

Class tape
    double charge() {
        return _movie.charge(_daysRented);
    }
```

You can make that change if you like, I don't tend to worry about message chains providing that they all lie in the same package. If they cross package boundaries, then I'm not so happy, and would add an insulating method.

Having done this with charge amounts, I'm inclined to do the same with frequent renter points. The need is less pressing, but I think it is more consistent to do them both the same way. And again if the movie classifications change it makes it easier to update the code.

```
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;
    }
```

With these two changes I can hide those constants, which is generally a Good Thing. Even constant data should be private.

```
    private static final int  CHILDRENS = 2;
    private static final int  REGULAR = 0;
    private static final int  NEW_RELEASE = 1;
```

To really do this, however, I need to change a couple of other parts of the class. I need to change how we create a movie. I used to create a movie with a message like

```
new Movie ("Ran", Movie.REGULAR);
```

and the constructor

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

To keep this type code hidden I need some creation methods.

```
    public static Movie newNewRelease(String name){
        return new Movie (name, NEW_RELEASE);
    }
    public static Movie newRegular(String name){
        return new Movie (name, REGULAR);
    }
    public static Movie newChildrens(String name) {
        return new Movie (name, CHILDRENS);
    }
```

Now I create a new movie with

```
    Movie.newRegular("Monty Python and the Holy Grail");
```

Movies can change their classification. I change a movie's classification with

```
    aMovie.setPriceCode(Movie.REGULAR);
```

I will need to add a bunch of methods to handle the changes of classification.

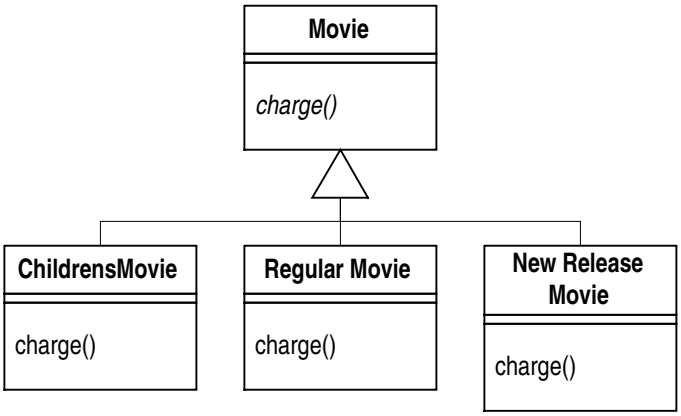
```
    public void beRegular() {
        _priceCode = REGULAR;
    }

    public void beNewRelease() {
        _priceCode = NEW_RELEASE;
    }
```

```
public void beChildrens() {
    _priceCode = CHILDRENS;
}
```

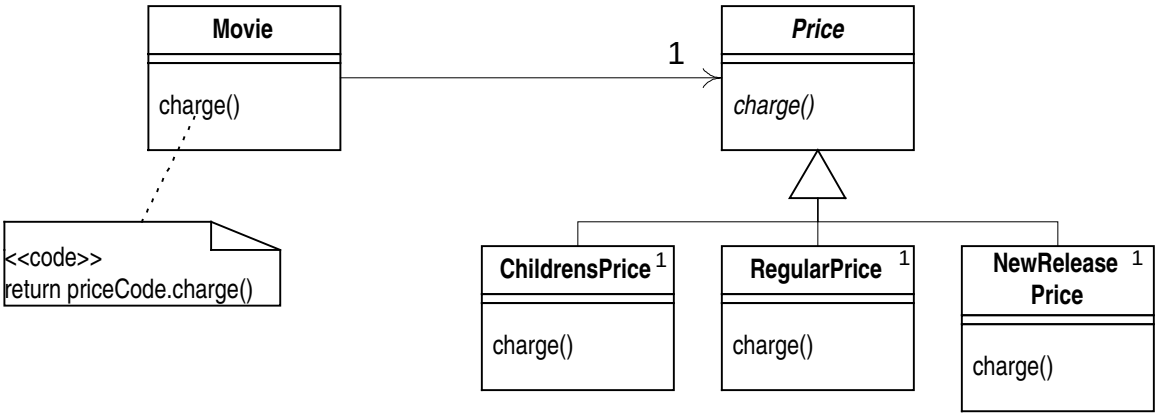
It's a bit of effort to set up these methods, but they are a much more explicit interface then the type codes. Just looking at the name of the method tells you what kind of movie you are getting. This makes the code more understandable. The trade off is that each time I add a price code I have to add a creation and update method. If I had lots of price codes this would hurt (so I wouldn't do it). If I have a few, however, then it's quite reasonable.

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.



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.



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

With a complex class you have to move data and methods around in small pieces to avoid errors, it seems slow but it is the quickest because you avoid debugging. For this case I could probably move the data and methods

in one go as the whole thing is not too complicated. However I'll do it the bit by bit way, so you can see how it goes. Just remember to do it one small bit at a time if you do this to a complicated class.

The first step is to create the new classes. Then I need to sort out how they are managed. As the diagram shows they are all singletons. It seems sensible to get hold of them via the superclass with a method like `Price.regular()`. I can do this by getting the superclass to manage the instances of the subclasses.

```
abstract class Price {
    static Price regular() {
        return _regular;
    }

    static Price childrens() {
        return _childrens;
    }
    static Price newRelease() {
        return _newRelease;
    }

    private static Price _childrens = new ChildrensPrice();
    private static Price _newRelease = new NewReleasePrice();
    private static Price _regular = new RegularPrice();
}
```

Now I can begin to move the data over. The first piece of data to move over is the price code. Of course I'm not actually going to use the price code within the Price object, but I will give it the illusion of doing so. That way the old methods will still work. They key is to modify those methods that access and update the price code value within Movie. My first step is to *self-encapsulate 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, I can use the setting methods instead.

```
public static Movie newNewRelease(String name){
       Movie result = new Movie (name);
       result.beNewRelease();
       return result;
}
public static Movie newRegular(String name){
       Movie result = new Movie (name);
       result.beRegular();
       return result;
}
public static Movie newChildrens(String name) {
       Movie result = new Movie (name);
       result.beChildrens();
       return result;
}

private Movie(String name) {
   _name = name;
}
```

After compiling and testing I now change getting and setting methods to use the new class.

```
    public void beRegular() {
        _price = Price.regular();
    }

    public void beNewRelease() {
        _price = Price.newRelease();
    }

    public void beChildrens() {
       _price = Price.childrens();
    }
    public int priceCode() {
        return _price.priceCode();
    }
```

And provide the `priceCode` methods on `Price` and its subclasses.

```
Class Price…
   abstract int priceCode();

Class RegularPrice…
    int priceCode(){
        return Movie.REGULAR;
```

```
    }
```

To do this I need to make the constants non-private again. This is fine, I don't mind them having a little fame before they bite the dust.

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

After moving the data I can now start moving methods. My prime target is the `charge()` method. 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 *replacing the case statement with inheritance*. 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()`. 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;
    }
```

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.

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