Chapter 1. Refactoring, a First Example

How do I begin to write about refactoring? The traditional way to begin talking about something is to outline the history, broad principles, and the like. When someone does that at a conference, I get slightly sleepy. My mind starts wandering with a low-priority background process that polls the speaker until he or she gives 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 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 provide the usual principles-style introduction.

With an introductory example, however, I run into a big problem. If I pick a large program, describing it and how it is refactored is too complicated for any reader to work through. (I tried and even a slightly complicated example runs to more than a hundred pages.) However, if I pick a program that is small enough to be comprehensible, refactoring does not look like it is worthwhile.

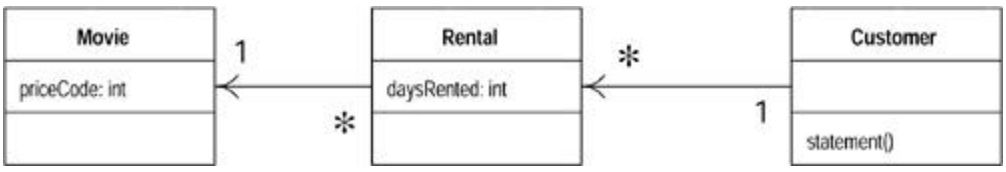
Thus I'm in the classic bind of anyone who wants to describe techniques that are useful for real-world programs. Frankly it is not worth the effort to do the refactoring that I'm going to show you on a small program like the one I'm going to use. But if the code I'm showing you is part of a larger system, 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 very simple. It is a program to calculate and print a statement of a customer's charges at a video store. The program is told which movies a customer rented and for how long. It then calculates the charges, which depend on how long the movie is rented, and identifies the type movie. There are three kinds of movies: regular, children's, and new releases. In addition to calculating charges, the statement also computes frequent renter points, which vary depending on whether the film is a new release.

Several classes represent various video elements. Here's a class diagram to show them (Figure 1.1).

Figure 1.1. Class diagram of the starting-point classes. Only the most important features are shown. The notation is Unified Modeling Language UML [Fowler, UML].



I'll show the code for each of these classes in turn.

Movie

Movie is just a simple data class.

```
public class Movie {

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

  private String _title;
  private int _priceCode;

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

  public int getPriceCode() {
      return _priceCode;
  }

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

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

Rental

The rental class represents a customer renting a movie.

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

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

Customer

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

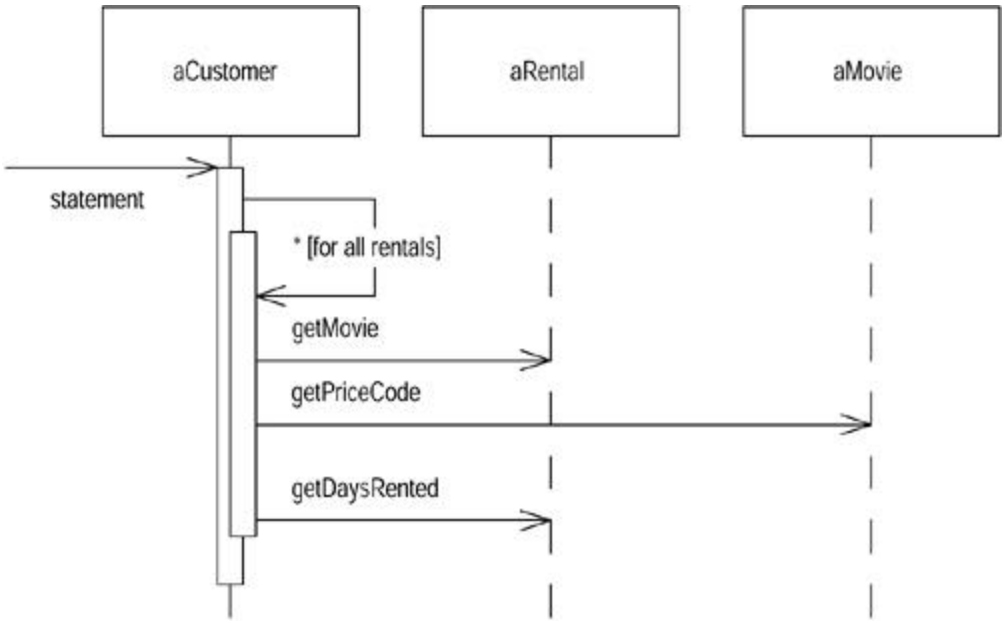
```
class Customer {
  private String _name;
  private Vector _rentals = new Vector();

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

  public void addRental(Rental arg) {
    _rentals.addElement(arg);
  }
  public String getName (){
      return _name;
  };
```

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 " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            double thisAmount = 0;
            Rental each = (Rental) rentals.nextElement();

            //determine amounts for each line
            switch (each.getMovie().getPriceCode()) {
                case Movie.REGULAR:
                    thisAmount += 2;
                    if (each.getDaysRented() > 2)
```

```
                        thisAmount += (each.getDaysRented() - 2) * 1.5;
                    break;
                case Movie.NEW_RELEASE:
                    thisAmount += each.getDaysRented() * 3;
                    break;
                case Movie.CHILDRENS:
                    thisAmount += 1.5;
                    if (each.getDaysRented() > 3)
                        thisAmount += (each.getDaysRented() - 3) * 1.5;
                    break;

            }

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

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

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

    }
```

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 this is a representative fragment of a more complex system, then I have some real problems with this program. That long statement routine in the Customer class does 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 judgment, 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 are needed. 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 change that the users would like to make. First they want a statement printed in HTML so that the statement can be Web enabled and fully buzzword compliant. Consider the impact this change would have. As you look at the code you can see that it is