impossible to reuse any of the behavior of the current statement method for an HTML statement. 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.

But what happens when the charging rules change? You have to fix both `statement` and `htmlStatement` and ensure the fixes are consistent. The problem with copying and pasting code comes when you have to change it later. 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 a second change. The users want to make changes to the way they classify movies, but they haven't yet decided on the change they are going to make. They have a number of changes in mind. These changes will affect both the way renters are charged for movies and the way that frequent renter points are calculated. As an experienced developer you are sure that whatever scheme users come up with, the only guarantee you're going to have is that they will change it again within six months.

The statement method is where the changes have to be made to deal with changes in classification and charging rules. If, however, we copy the statement to an 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 and harder to make them without making a mistake.

You may be tempted to make the fewest possible changes 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 broken, but it does hurt. It is making your life more difficult because you find it hard to make the changes your users want. This is where refactoring comes in.

Tip

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

Because the statement result produces a string, I 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: 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 whether I introduce a bug. It is essential for refactoring that you have good tests. It's worth spending the time to build 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 go into more detail on testing in Chapter 4.

Tip

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 show how I split up the long method and move 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. 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, the 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 around the loop and is not altered until the switch gets to it. So I can just assign the result.

The next two pages show the code before and after refactoring. The before code is on the left, the resulting code on the right. The code I'm extracting from the original and any changes in the new code that I don't think are immediately obvious are in boldface type. As I continue with this chapter, I'll continue with this left-right convention.

```
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;

    }
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();

          thisAmount = amountFor(each);

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

          //show figures for this rental
```

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

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

    }
}
private int amountFor(Rental each) {
    int thisAmount = 0;
    switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            thisAmount += 2;
            if (each.getDaysRented() > 2)
                thisAmount += (each.getDaysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented() - 3) * 1.5;
            break;

    }
    return thisAmount;
}
```

Whenever I make a change like this, I compile and test. I didn't get off to a very good start—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'd made the return type `amountFor int` instead of `double`:

```
   private double amountFor(Rental each) {
       double thisAmount = 0;
       switch (each.getMovie().getPriceCode()) {
           case Movie.REGULAR:
               thisAmount += 2;
               if (each.getDaysRented() > 2)
                   thisAmount += (each.getDaysRented() - 2) * 1.5;
              break;
           case Movie.NEW_RELEASE:
               thisAmount += each.getDaysRented() * 3;
               break;
           case Movie.CHILDRENS:
               thisAmount += 1.5;
               if (each.getDaysRented() > 3)
                   thisAmount += (each.getDaysRented() - 3) * 1.5;
```

```
                break;
        }
        return thisAmount;
    }
```

It's the kind of silly mistake that I often make, and it can be a pain to track down. In this case Java converts doubles to ints without complaining but merrily rounding [Java Spec]. 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 a long time debugging, even if you are as careless as I am.

Tip

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

Because I'm working in Java, I need to analyze the code to figure out what to do with the local variables. With a tool, however, this can be made really simple. Such a tool does exist in Smalltalk, the Refactoring Browser. With this tool 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, the tool doesn't make silly mistakes like mine. 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 `amountFor,` and this is a good place to change them.

Here's the original code:

```
    private double amountFor(Rental each) {
        double thisAmount = 0;
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                    thisAmount += (each.getDaysRented() - 3) * 1.5;
                break;
        }
        return thisAmount;
    }
```

Here is the renamed code:

```
private double amountFor(Rental aRental) {
    double result = 0;
    switch (aRental.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (aRental.getDaysRented() > 2)
                result += (aRental.getDaysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            result += aRental.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (aRental.getDaysRented() > 3)
                result += (aRental.getDaysRented() - 3) * 1.5;
            break;
    }
    return result;
}
```

Once I've done the renaming, I compile and test to ensure I haven't broken anything.

Is 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 of 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

> **Tip**
>
> *Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*

Code that communicates its purpose is very important. I often refactor just when I'm reading some code. That way as I gain understanding about the program, I embed that understanding into the code for later so I don't forget what I learned.

Moving the Amount Calculation

As I look at `amountFor`, I can see that it uses information from the rental, but does not use information from the customer.

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