Refactoring

Executive Summary

This lab is based on running examples from the book Refactoring: Improving the Design of Existing Code. You will see and perform several refactorings. You will also get a sense of Eclipse's support for refactoring. For the submission, you will be required to refactor the example program and submit your solutions.

This tutorial is based on materials from Dr. Christine Julien.

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 rental store. The program is told which movies a customer rented and for how long. It then calculates the charges, which depend on how long the movie is rented, and identifies the type of movie.

There are three kinds of movies: regular, children's, and new releases. In addition to calculating charges, the statement also computes frequent renter points, which vary depending on whether the film is a new release.

Please checkout code from https://subversion.assembla.com/svn/dp-refactoring/. There are two classes, Rental and Customer.
The Rental class represents a customer renting a movie.The Customer class represents the customer of the store. Like the other classes it has data and accessors. Note that, you need to test your program after each refactoring using the provided JUnit test.

Comments on the Starting Program

There's nothing wrong with a quick and dirty simple program. But if this is a representative fragment of a more complex system, then we have some real problems with this program. The 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.

For example, suppose 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 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. 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 us 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 thestatement to an htmlSatement, 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.

The First Step in Refactoring

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

Because the statement result produces a string, we create a few customers, give each customer a few rentals of various kinds of films, and generate the statement strings. We then do a string comparison between the new string and some reference strings that we have hand checked. We set up all of these tests so we 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, we 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. In this lab, we provide some test cases, if your refactoring does not lead to any bug, the output of your program will show congratulation messages; otherwise, it will show the error messages.

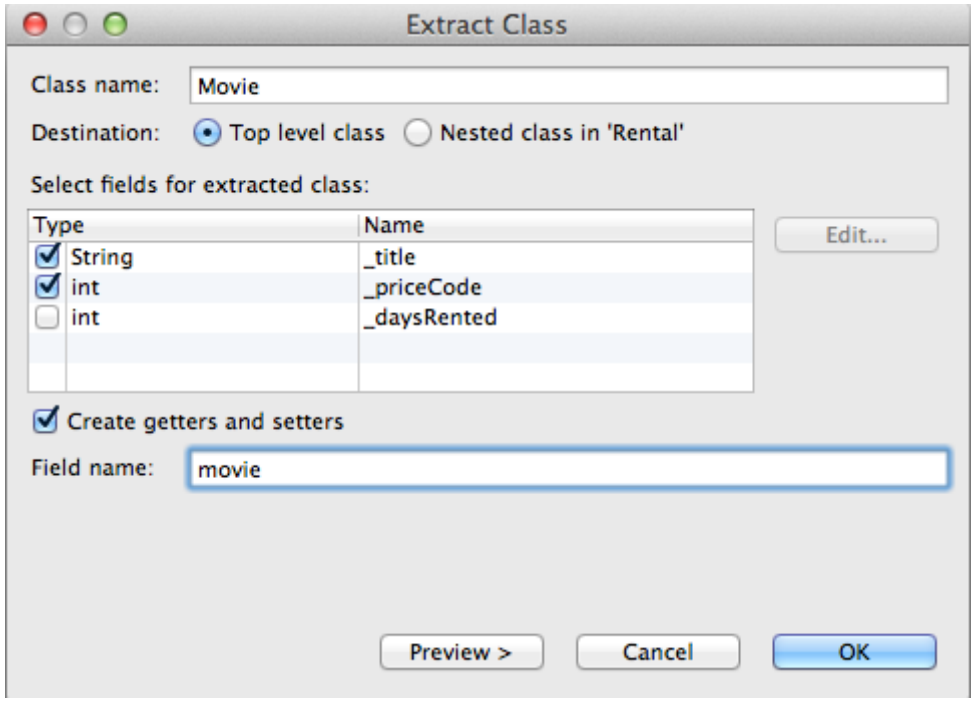
Decomposing and Redistributing the statement method

We are now going to effectively redo in Eclipse the refactoring that we did in class. The obvious first target of our attention is the overly long statement method. When we look at a long method like that, we are 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 show how we split up the long method and move the pieces to better classes. Our aim is to make it easier to write an htmlStatement method with much less duplication of code.

Extract Class

Our first step is to use Extract Class that creates a new class and move the relevant fields and methods from the old class into the new class. Such a class is one with many methods and quite a lot of data. A class that is too big to understand easily. You need to consider where it can be split, and you split it. In this example, we can separate the movie behaviors into its own class from the Rental class.

Eclipse includes a refactoring engine. Right click and select “Refactor” -> “Extract Class”.



Check the _title and _priceCode. Then type “Movie” as class name and “_movie” as field name. Check “Create getters and setters”. Then the eclipse do the extract class for you.

Rename

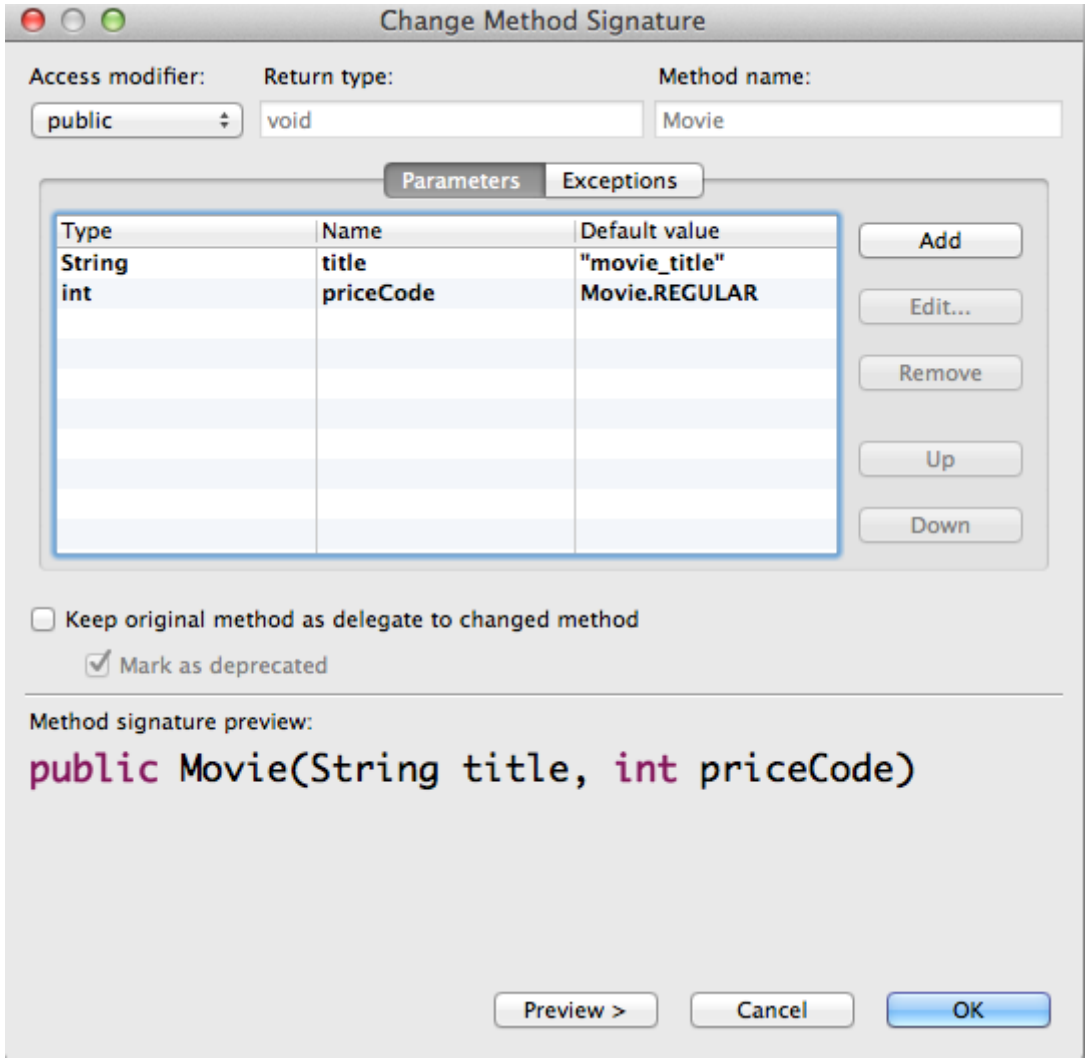
The names of getter and setter (e.g., get_title and set_title) generated by Eclipse do not follow our naming convention. Let use the renaming tool of Eclipse. Right click at the variable, then select “Refactor” -> “Rename”. Then we just need to enter the new name, e.g., getTitle and setTitle.

Move Field

Since the priceCode is in the Movie class, the 3 constant priceCode variables, CHILDRENS, REGULAR, and NEW_RELEASE should be moved to the Movie class. We use Move Field on each variable. Move the cursor to one variable each time, right click and select “Refactor” -> “Move”. Type “Movie” as destination. Unckeck “Keep original member as delegate to moved member”.

Change Method Signature

Now, let us move to the Movie class. We want to set the title and priceCode when a Movie object is created. Therefore, we use “Change Method Signature”. Move the cursor to the constructor of the Movie, right click and select “Refactor” -> “Change Method Signature”. Add title and priceCode as parameters.



In the constructor, add the following code (convention: red ~~strikethrough~~ for code we remove; blue **boldface** for code we added).

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

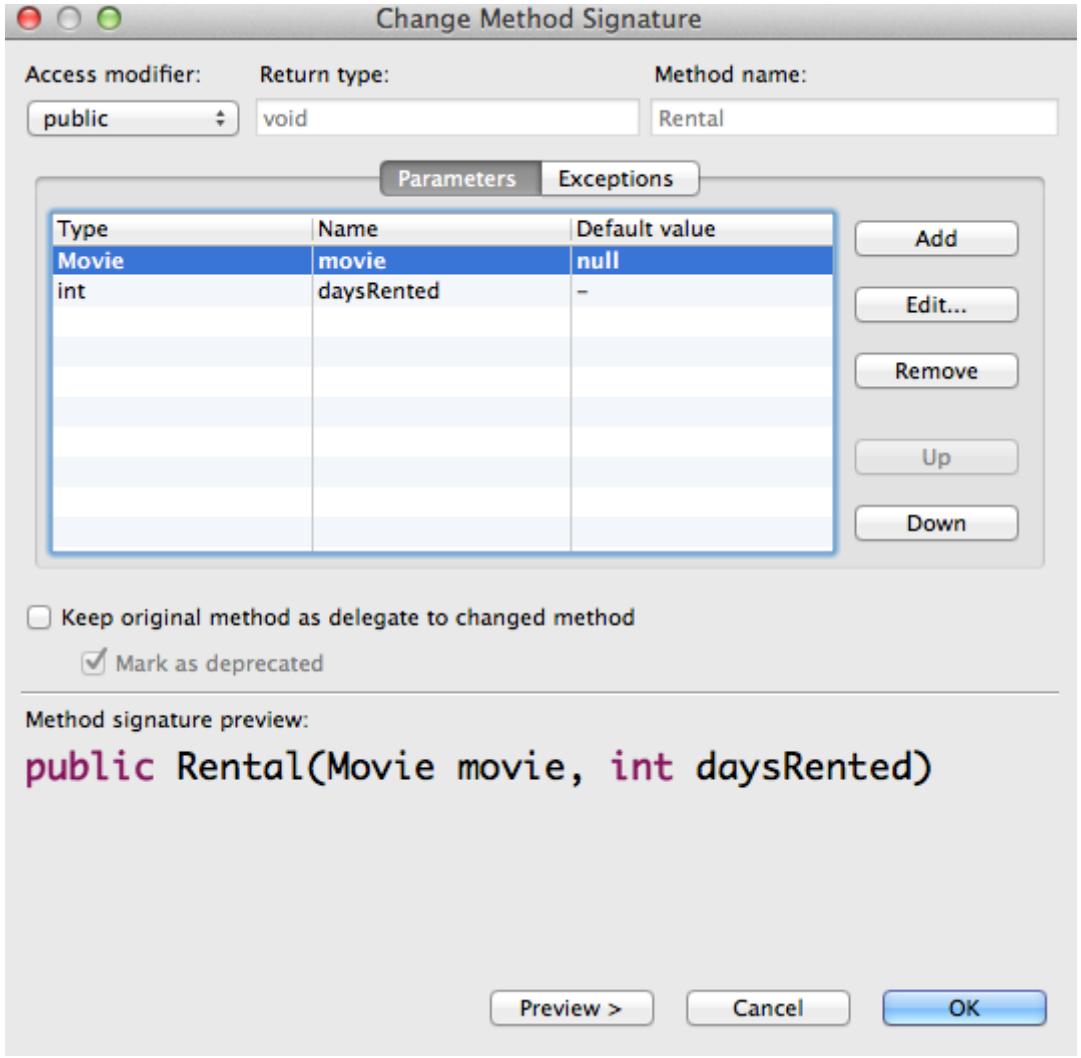
Next, let's move to the Rental class. We do not want to create a Movie object for each rental. Then we use Change Method Signature again. We first create three movies in the TestMovieRental and class as follows.

```
public class TestMovieRental {
  ...
  String M1_Title;
  int M1_PriceCode;
  String M2_Title;
  int M2_PriceCode;
  String M3_Title;
  int M3_PriceCode;
  Movie M1;
  Movie M2;
  Movie M3;

  @Before
  public void setUp() {
    ...
    M1_Title = "Oz The Great and Powerful";
    M1_PriceCode = Movie.NEW_RELEASE;

    M2_Title = "The Dark Knight";
    M2_PriceCode = Movie.REGULAR;

    M3_Title = "Wreck it Ralph";
    M3_PriceCode = Movie.CHILDRENS;

    M1 = new Movie("Oz The Great and Powerful", Movie.NEW_RELEASE);
```

```
  M2 = new Movie("The Dark Knight", Movie.REGULAR);
  M3 = new Movie("Wreck-it Ralph", Movie.CHILDRENS);
 }

 public void test1() {
  C1.addRental(new Rental(M1_Title, M1_PriceCode, start, end1));
  C1.addRental(new Rental(M2_Title, M2_PriceCode, start, end2));
  C1.addRental(new Rental(M3_Title, M3_PriceCode, start, end3));
  C1.addRental(new Rental(M1, start, end1));
  C1.addRental(new Rental(M2, start, end2));
  C1.addRental(new Rental(M3, start, end3));
  ...
 }
 public void test2() {
  C2.addRental(new Rental(M1_Title, M1_PriceCode, start, end1));
  C2.addRental(new Rental(M3_Title, M3_PriceCode, start, end2));
  C2.addRental(new Rental(M1, start, end1));
  C2.addRental(new Rental(M3, start, end2));
  ...
 }
 public void test3() {
  C3.addRental(new Rental(M2_Title, M2_PriceCode, start, end1));
  C3.addRental(new Rental(M3_Title, M3_PriceCode, start, end2));
  C3.addRental(new Rental(M2, start, end1));
  C3.addRental(new Rental(M3, start, end2));
  ...
 }
}
```

You should also follow similar approach in the Main class.

Then we move to the Rental class. Remove the title and priceCode parameters but add the movie.



Eclipse will popup a window showing that there are some problems when you apply this refactoring. We click "continue" then manually edit theRental class as follows.

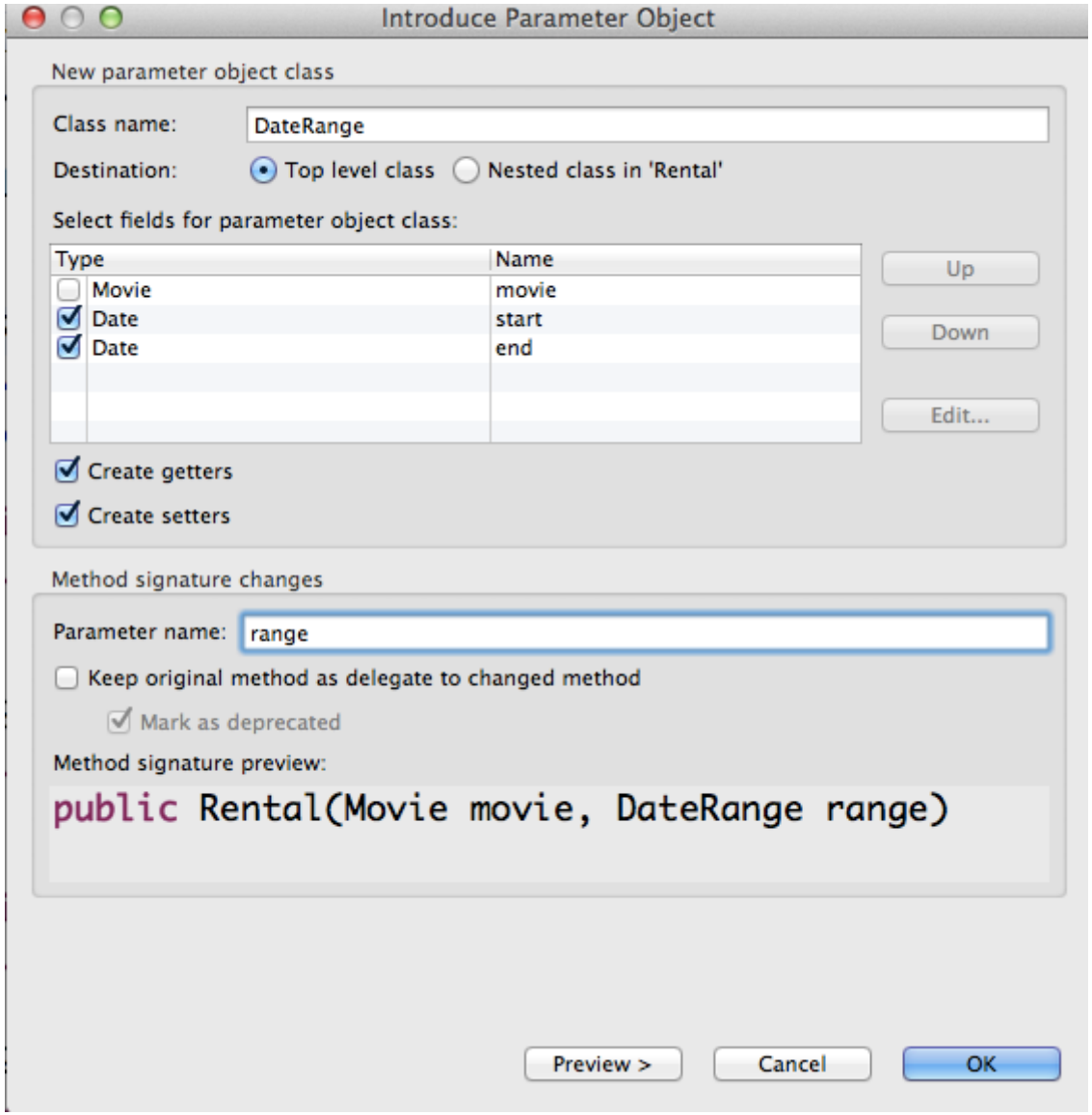
```
class Rental {
    private Movie _movie = new Movie("movie_title", Movie.REGULAR);
    private int _daysRented;
```

```
    public Rental(Movie movie, Date start, Date end) {
        _movie.set_title(title);
        _movie.setPriceCode(priceCode);
        _movie = movie;
        _daysRented = (int)((end.getTime() - start.getTime()) / (1000 * 60 * 60 * 24));
    }
    ...
}
```

Introduce Parameter Object

I don't know how many times I come across pairs of values that show a range, such as start and end dates and upper and lower numbers. I can understand why this happens, after all I did it all the time myself. I always try to use ranges instead. Now, we would like to use Introduce Parameter Object for the date range of the Rental constructor.

Move the cursor to the constructor of Rental, right click and select “Refactor” -> “Introduce Parameter Object”



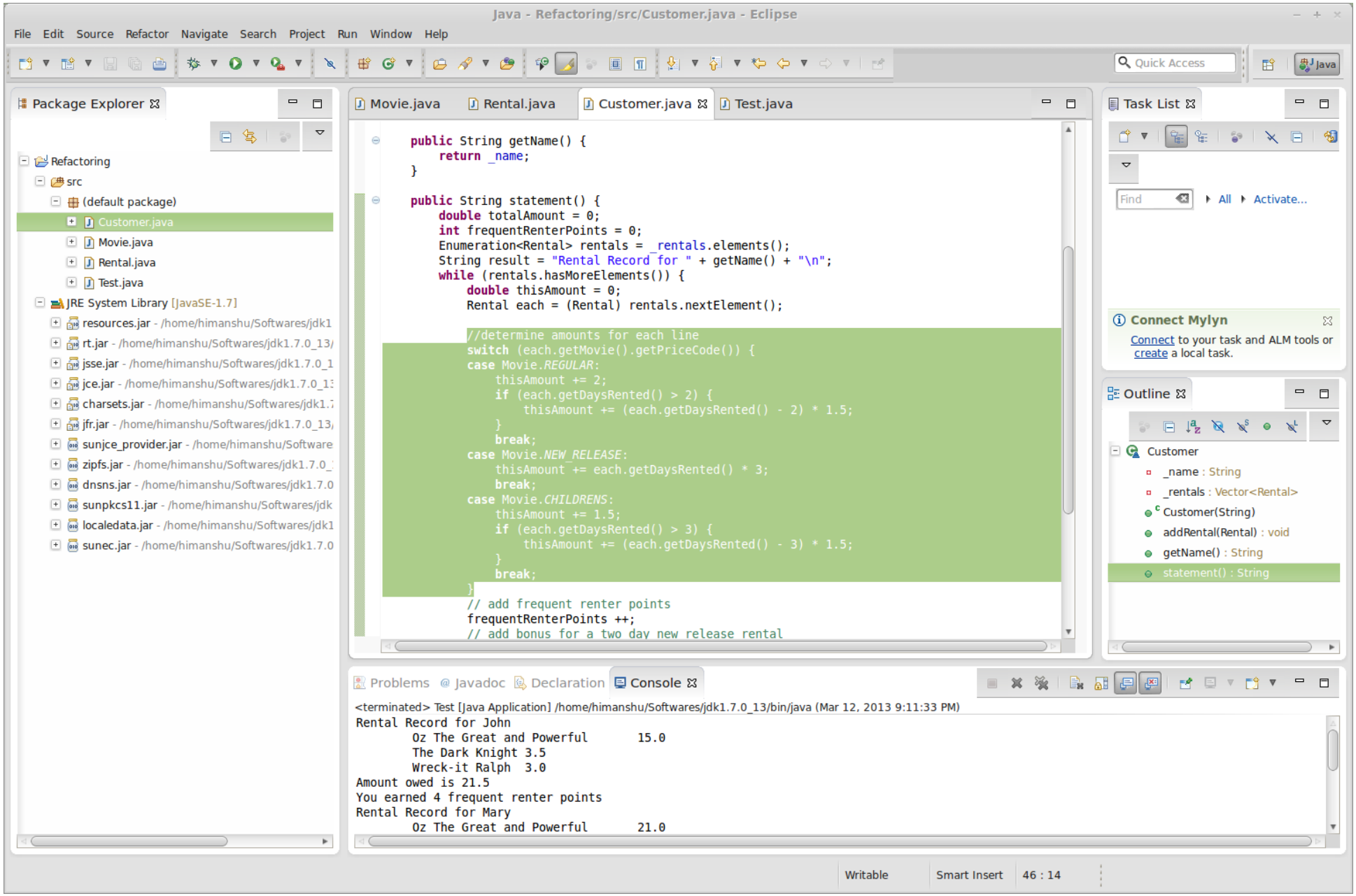
Type DateRange as class name, and check start and end as fields for parameter object class. The eclipse then create the DateRange class and update the function signature for us.

Extract Method

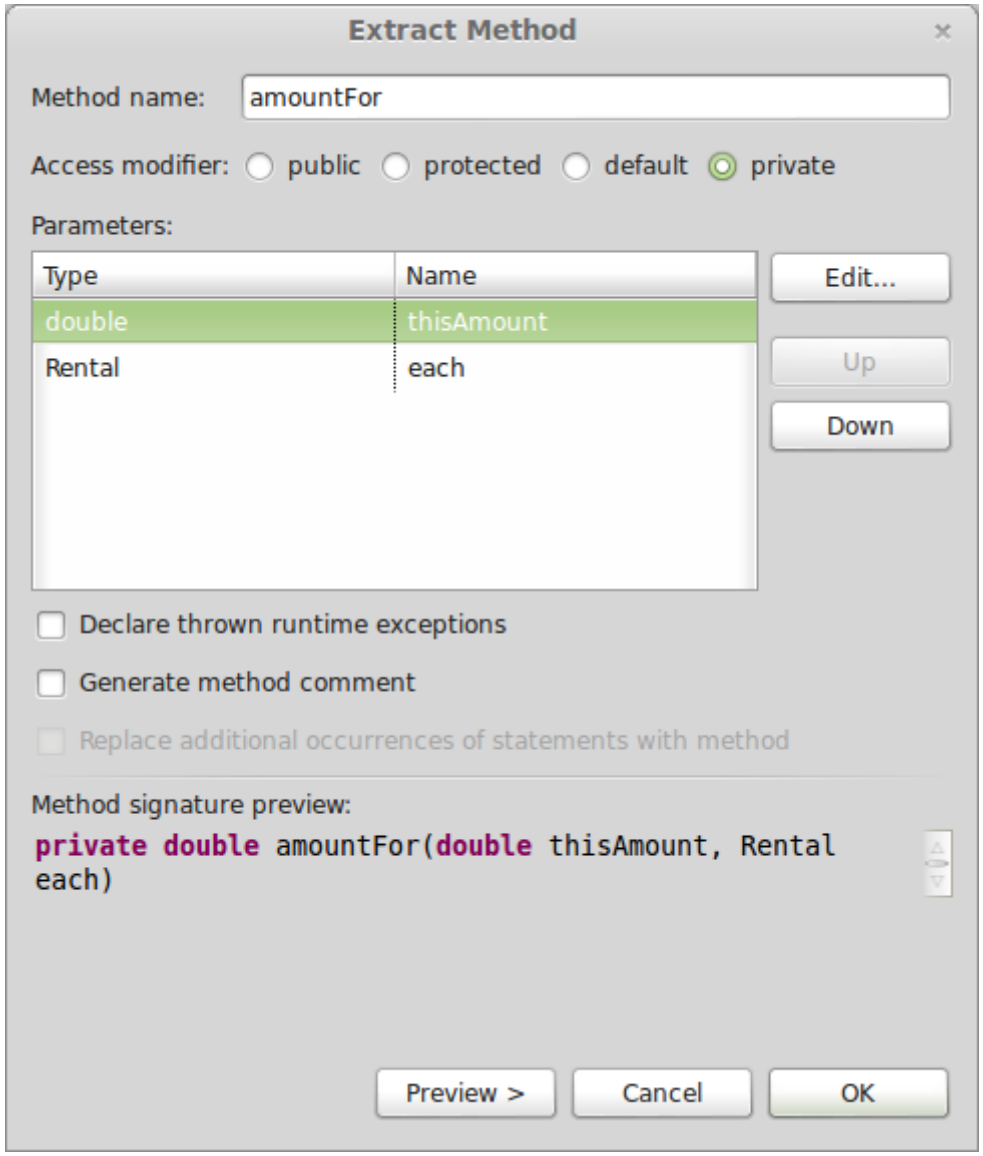
Our next step is to find a logical clump of code and use Extract Method. An obvious piece here is the switch statement. This looks like it would make a good chunk to extract into its own method. When we extract a method, as in any refactoring, we need to know what can go wrong. If we do the extraction badly, we could introduce a bug into the program. So before we do the refactoring we need to figure out how to do it safely. Here is a safe recipe from the "Refactoring" catalog for extract method.

First we 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 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 around the loop and is not altered until the switch gets to it. So we can just assign the new method's return value to thisAmount.

We use the Eclipse refactoring engine for this example. Select the code that you want to extract, right click and select "Refactor" -> "Extract Method".



Then type the method name. In this example, you can type amountFor. Click “OK,” and then Eclipse will perform the extract method automatically.



Note that, since Eclipse cannot tell the modified variables and non-modified variables, it passes both each and thisAmount as parameters. You can modify the method it extracted manually as follows.

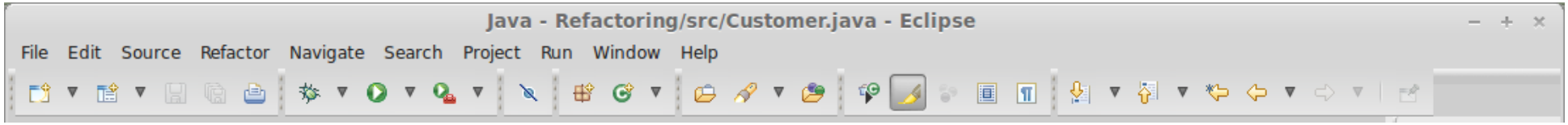
```
class Customer ...
  public String statement() {
      double totalAmount = 0;
      int frequentRenterPoints = 0;
```

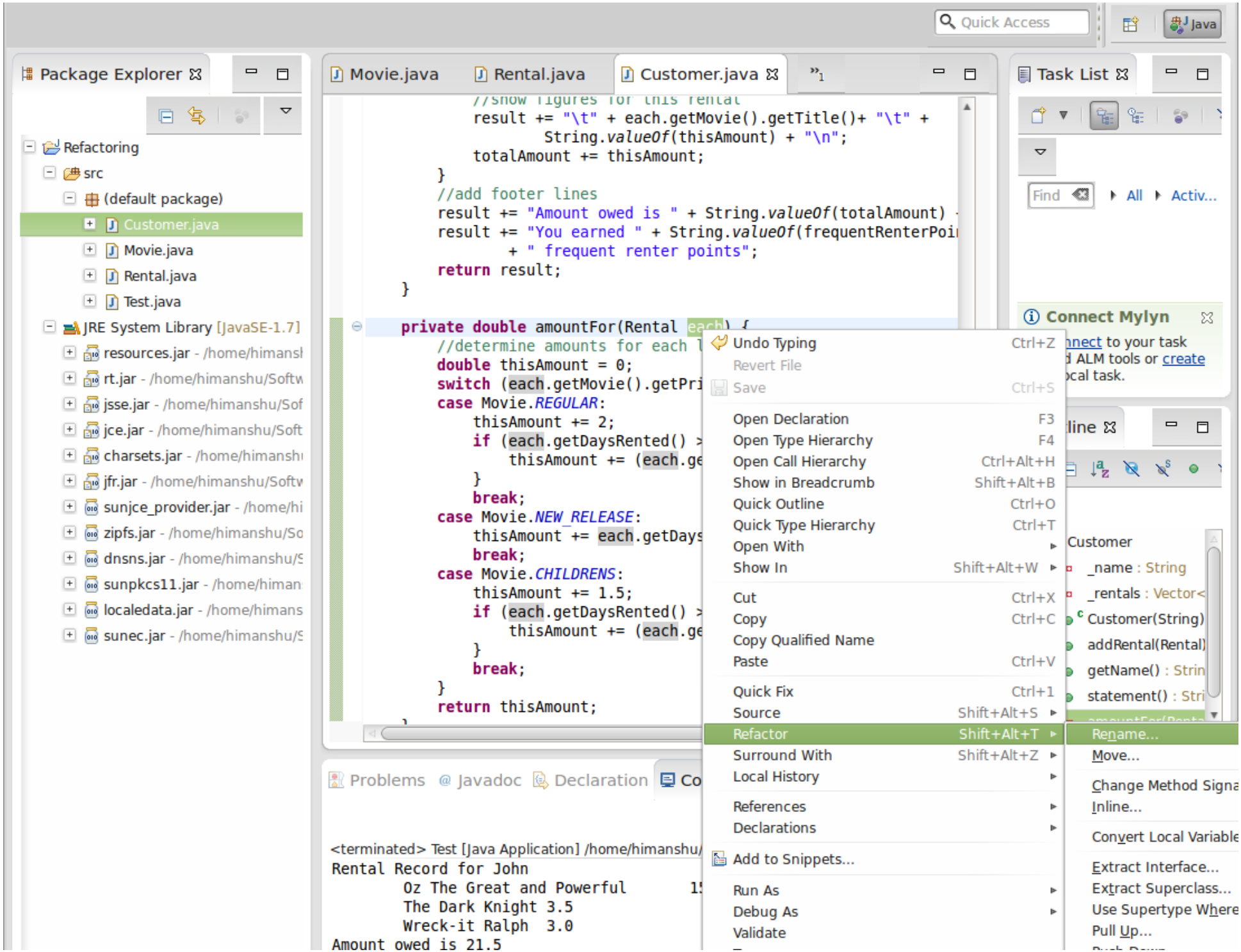
```
    StringBuilder result = new StringBuilder("Rental Record for " + getName() + "\n");
    for (Rental each : _rental) {
        double thisAmount = 0;
        double thisAmount = amountFor(thisAmount, each);

        // 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.append("\t").append(each.getMovie().getTitle());
        result.append("\t").append(String.valueOf(thisAmount));
        result.append("\n");

        totalAmount += thisAmount;
    }

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

private double amountFor(double thisAmount, Rental each) {
    //determine amounts for each line
```

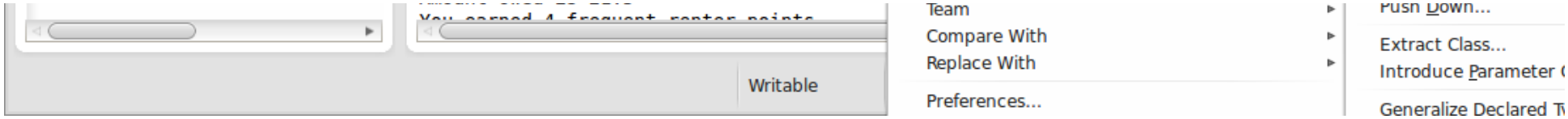
```
    double thisAmount = 0;
    switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
        thisAmount += 2;
        if (each.getDaysRented() > 2) {
            thisAmount += (each.getDaysRented() - 2) * 1.5;
        }
        break;
    case Movie.NEW_RELEASE:
        thisAmount +=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

Rename

Now that we have broken the original method down into chunks, we can work on them separately. We don't like some of the variable names inamountFor, and this is a good place to change them. We want to rename each with each with aRental. We can do it with Eclipse. Right click at the variable, then select “Refactor” -> “Rename”. Then we just need to enter the new name; Eclipse then will rename the variable automatically.




Quick Access
Java
Package Explorer
Refactoring
src
(default package)
Customer.java
Movie.java
Rental.java
Test.java
JRE System Library [JavaSE-1.7]
resources.jar - /home/himansh
rt.jar - /home/himanshu/Softw
jsse.jar - /home/himanshu/Sof
jce.jar - /home/himanshu/Soft
charsets.jar - /home/himanshu
jfr.jar - /home/himanshu/Softw
sunjce_provider.jar - /home/hi
zipfs.jar - /home/himanshu/So
dnsns.jar - /home/himanshu/S
sunpkcs11.jar - /home/himan
localedata.jar - /home/himans
sunec.jar - /home/himanshu/S
Movie.java
Rental.java
Customer.java
//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)
result += "You earned " + String.valueOf(frequentRenterPoi
+ " frequent renter points";
return result;
}
private double amountFor(Rental each) {
//determine amounts for each
double thisAmount = 0;
switch (each.getMovie().getPri
case Movie.REGULAR:
thisAmount += 2;
if (each.getDaysRented() >
thisAmount += (each.ge
}
break;
case Movie.NEW_RELEASE:
thisAmount += each.getDays
break;
case Movie.CHILDRENS:
thisAmount += 1.5;
if (each.getDaysRented() >
thisAmount += (each.ge
}
break;
}
return thisAmount;
Task List
Find
All
Activ...
Connect Mylyn
nect to your task
d ALM tools or create
ocal task.
line
Customer
_name : String
_rentals : Vector<
Customer(String)
addRental(Rental)
getName() : Strin
statement() : Stri
Undo Typing Ctrl+Z
Revert File
Save Ctrl+S
Open Declaration F3
Open Type Hierarchy F4
Open Call Hierarchy Ctrl+Alt+H
Show in Breadcrumb Shift+Alt+B
Quick Outline Ctrl+O
Quick Type Hierarchy Ctrl+T
Open With
Show In Shift+Alt+W
Cut Ctrl+X
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Quick Fix Ctrl+1
Source Shift+Alt+S
Refactor Shift+Alt+T
Surround With Shift+Alt+Z
Local History
References
Declarations
Add to Snippets...
Run As
Debug As
Validate
Rename...
Move...
Change Method Signa
Inline...
Convert Local Variable
Extract Interface...
Extract Superclass...
Use Supertype Where
Pull Up...
Problems
Javadoc
Declaration
<terminated> Test [Java Application] /home/himanshu/
Rental Record for John
Oz The Great and Powerful 15
The Dark Knight 3.5
Wreck-it Ralph 3.0
Amount owed is 21.5



Now, compile and test to ensure we haven't broken anything.

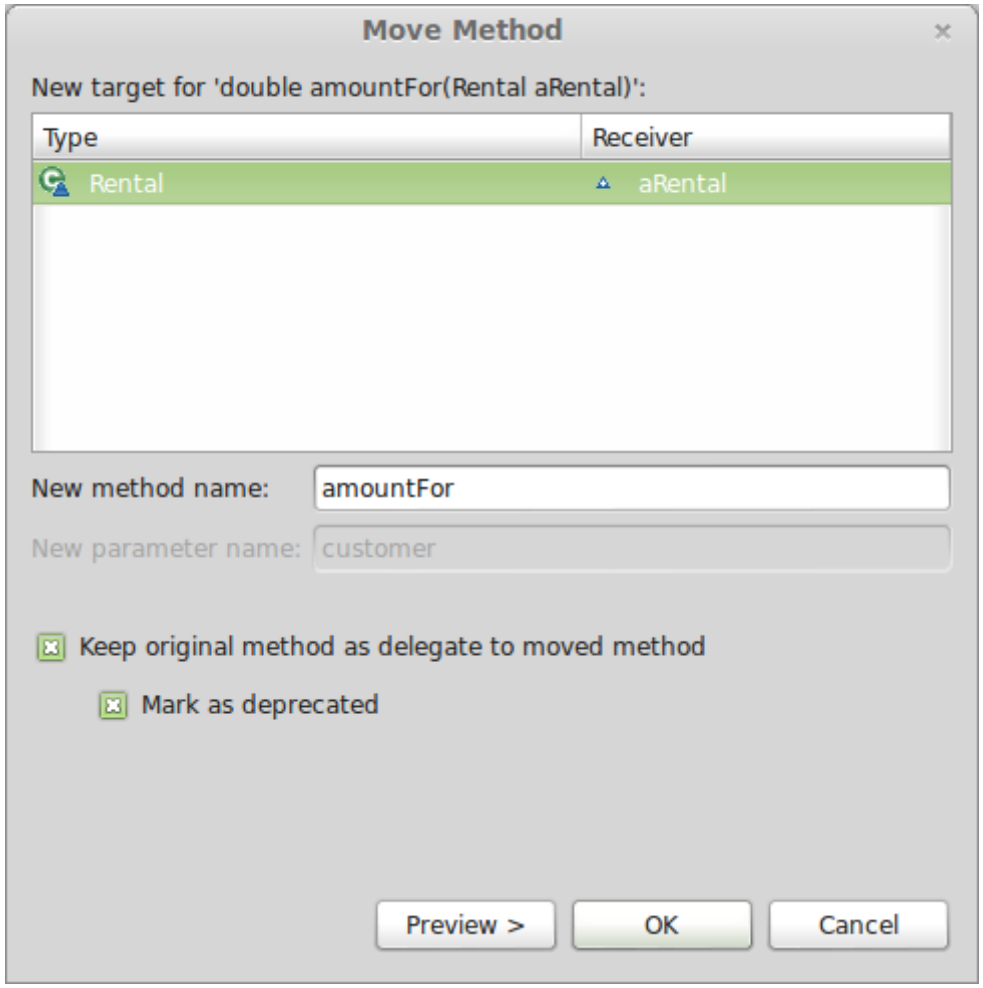
Is renaming worth the effort? Absolutely. Good code should communicate what it is doing clearly, and variable names are a key to clear code. Never be afraid to change the names of things to improve clarity. With good find and replace tools, it is usually not difficult. Strong typing and testing will highlight anything you miss.

Moving the Amount Calculation

Move Method

As we look at amountFor, we can see that it uses information from the rental, but does not use information from the customer. This immediately raises our suspicions that the method is on the wrong object. In most cases a method should be on the object whose data it uses, thus the method should be moved to the rental. To do this we use Move Method. With this you first copy the code over to rental, adjust it to fit in its new home.

We still can use Eclipse to do Move Method automatically. Right click within the function, then select “Refactor” -> “Move”. We can type “getCharge” as the new method name. Check “Keep original original method as delegate to moved method”. Then click “OK”. Note that Eclipse will show a warning message indicating the visibility of the method will be changed. The reason is that the amountFor is private; the getCharge cannot be private since it will be invoked in another class. In this case, we just click “Continue”. It then removes the parameter and move the amountFor to Rental class with name getCharge. In addition, the body of Customer amountFor is delegated to the new method.



Now, compile and test to see whether we have broken anything.

The next step is to find every reference to the old method and adjust the reference to use the new method as follows. Note that, if you do not check "Keep original original method as delegate to moved method" when you use Eclipse Move Method tool, Eclipse will automatically adjust the reference to use the new method. In addition, the original method amountFor will be removed.

```
class Customer …
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        StringBuilder result = new StringBuilder("Rental Record for " + getName() + "\n");
```

```
    for (Rental each : _rentals) {
        double thisAmount = ~~amountFor(each)~~ each.getCharge();

        // add frequent renter points
        frequentRenterPoints ++;
        // add bonus for a two day new release rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
                && each.getDaysRented() > 1) frequentRenterPoints++;
        //show figures for this rental
        result.append("\t").append(each.getMovie().getTitle());
        result.append("\t").append(String.valueOf(thisAmount));
        result.append("\n");

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

In this case this step is easy because 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.

The next thing that strikes us is that thisAmount is now redundant. It is set to the result of each.charge and not changed afterward. Thus we can eliminate thisAmount by using Fowler's Replace Temp with Query (p.g. 120) as follows.

```
```

```
class Customer …
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;

        StringBuilder result = new StringBuilder("Rental Record for " + getName() + "\n");
        for (Rental each : _rentals) {

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

            result.append("\t").append(each.getMovie().getTitle());
            result.append("\t").append(String.valueOf(thisAmount each.getCharge()));
            result.append("\n");

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

        result.toString();
    }
```

Now, compile and test to make sure we haven't broken anything.

We like to get rid of temporary variables such as this as much as possible. Temps are often a problem in that they cause a lot of parameters to be passed around when they don't have to be. 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.

Exercise - Extracting Renter Points

The next step is to do a similar thing for the frequent renter points. It seems reasonable to put the responsibility on the rental. First we need to use Extract Method on the frequent renter points part of the code. Note that, you need to remove frequentRenterPoints from the parameter list of the new method manually. Then we move the method to the Rental class with Move Method. You can give the method with a new name getFrequentRenterPoints. Compile and test after refactoring.

Removing Temps

As we suggested before, temporary variables can be a problem. They are useful only 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 require these totals. We like to use Replace Temp with Query 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. We began by replacing totalAmount with a charge method on customer:

```
class Customer ...

    public String statement() {

        double totalAmount = 0;

        int frequentRenterPoints = 0;


        StringBuilder result = new StringBuilder("Rental Record for " + getName() + "\n");


        for (Rental each : _rentals) {


            frequentRenterPoints += each.getFrequentRenterPoints();
```

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

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

    return result.toString();
}
public double getTotalCharge() {
    double result = 0;
    for (Rental each : _rentals) {
        result += each.getCharge();
    }
    return result;
}
```

This isn't the simplest case of Replace Temp with Query, totalAmount was assigned to within the loop, so we have to copy the loop into the query method.

Exercise - Removing frequentRenterPoints

Do the same to remove frequentRenterPoints by creating a getTotalFrequentRenterPoints method in Customer class.

The concern with this refactoring lies in performance. The old code executed the "while" loop once, the new code executes it three times. A while loop that takes a long time might impair performance. Many programmers would not do this refactoring simply for this reason. Until we profile we cannot tell how much time is needed for the loop to calculate or whether the loop is called often enough for it to affect the overall performance of the system. Don't worry about this while refactoring. 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.

These queries are now available for any code written in the customer class. 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 have 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. We can write a method htmlStatement as follows:

```
Customer ...
    public String htmlStatement() {
        StringBuilder result = new StringBuilder("<H1>Rentals for <EM>").append(getName());
        result.append("</EM></H1><P>\n");
        for (Rental each : _rentals) {
            //show figures for each rental
            result.append(each.getMovie().getTitle()).append(": ");
            result.append(String.valueOf(each.getCharge())).append("<BR>\n");
        }
        //add footer lines
        result.append("<P>You owe <EM>" + String.valueOf(getTotalCharge()));
        result.append("</EM><P>\n");
        result.append("On this rental you earned <EM>");
        result.append(String.valueOf(getTotalFrequentRenterPoints()));
        result.append("</EM> frequent renter points<P>");

        return result.toString();
    }
```

By extracting the calculations we can create the htmlStatement method and reuse all of the calculation code that was in the original statement method. We didn't copy and paste, so if the calculation rules change there is only one place in the code to go to. Any other kind of statement will be really quick and easy to prepare. The refactoring did not take long. We spent most of the time figuring out what the code did, and we would have had to do that anyway.

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. This implies that getCharge should move onto movie as follows:

```
class Movie …
    double getCharge(int daysRented) {
        //determine amounts for each line
        double result = 0;
        switch (getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (daysRented > 2) {
                result += (daysRented - 2) * 1.5;
            }
            break;
        case Movie.NEW_RELEASE:
            result += daysRented * 3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (daysRented > 3) {
                result += (daysRented - 3) * 1.5;
            }
            break;
        }
```

```
    return result;
  }
```

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

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

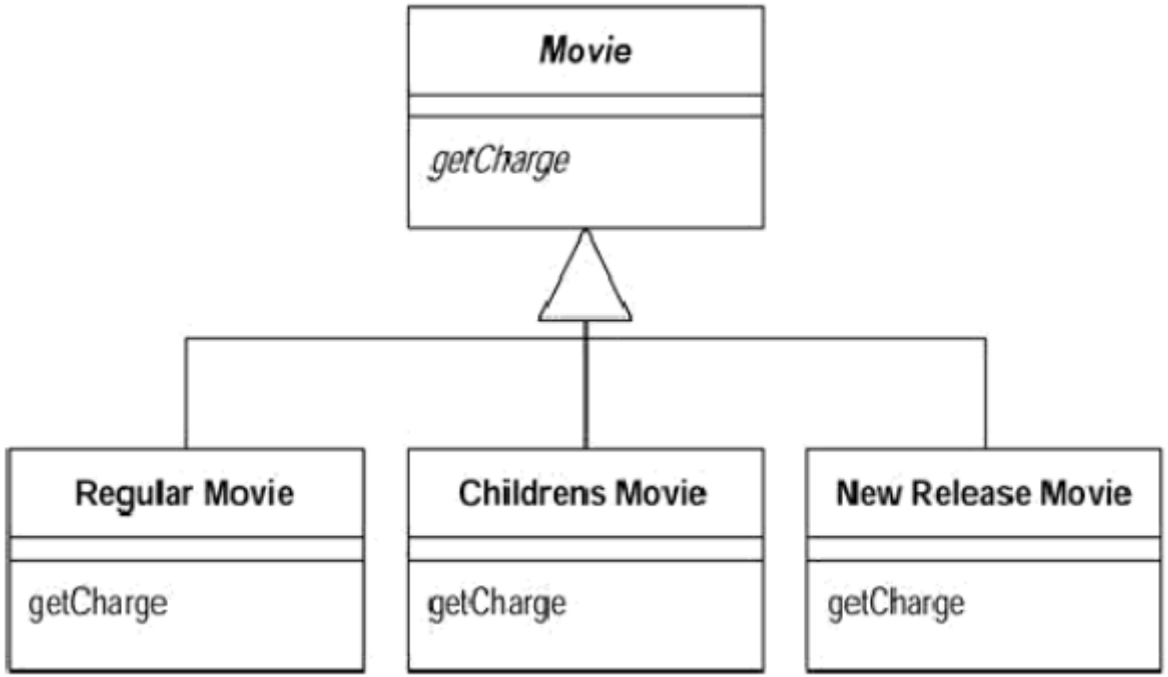
For this to work we had to pass in the length of the rental, which of course is data from the rental. The method effectively uses two pieces of data, the length of the rental and the type of the movie. Why do we prefer to pass the length of rental to the movie rather than the movie type to the rental?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

Exercise - Move getFrequentRenterPoints form Rental to Movie

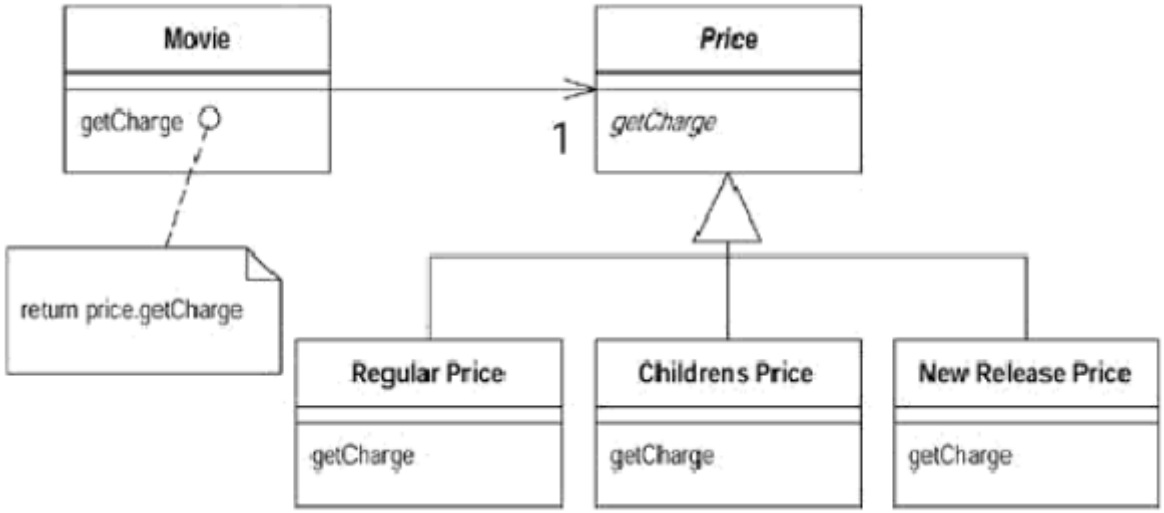
We've moved the getCharge method, do the same with the frequent renter point calculation. That keeps both things that vary with the type together on the class that has the type.

At last ... Inheritance

We have several types of movie that have different ways of answering the same question. This sounds like a job for subclasses. We can have three subclasses of movie, each of which can have its own version of charge:



This allows us 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. With the State pattern the classes look like:



By adding the indirection we can do the subclassing from the price code object and change the (type of) price whenever we need to.

To introduce the state pattern, we will use three refactorings. First we'll move the type code behavior into the state pattern with Replace Type Code with State/Strategy. Then we can use Move Method to move the switch statement into the price class. Finally we'll use Replace Conditional with Polymorphism to eliminate the switch statement.

Replace Type Code With State/Strategy

This first step is to use Self Encapsulate Field on the type code to ensure that all uses of the type code go through getting and setting methods. Because most of the code came from other classes, most methods already use the getting method. However, the constructors do access the price code:

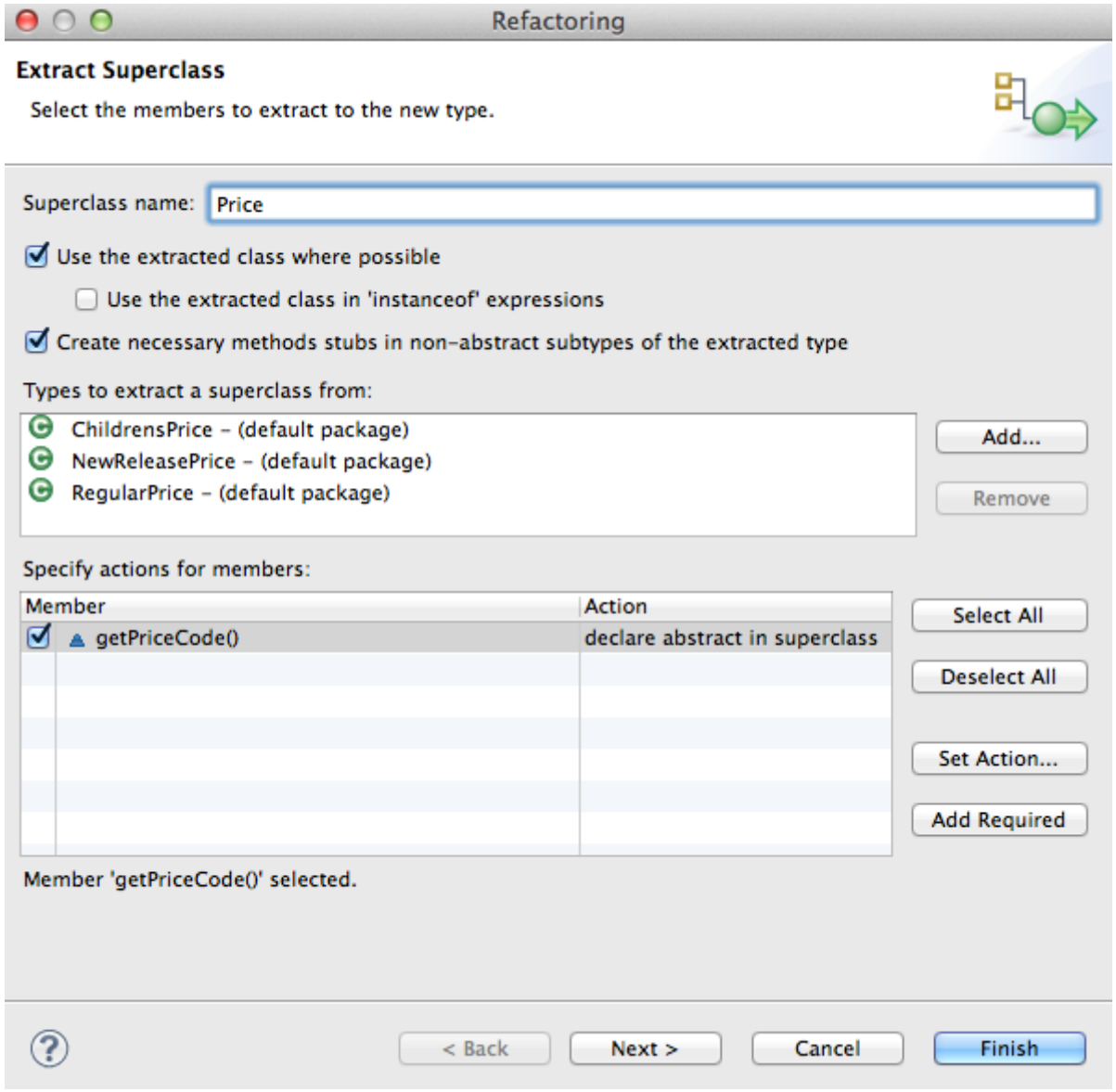
```
class Movie ...
    public Movie(String title, int priceCode) {
        _title = title;
        _priceCode = priceCode;
        setPriceCode(priceCode);
    }
```

We can use the setting method instead.

Now, compile and test to make sure we didn't break anything. Now we first add the RegularPrice, ChildrensPrice, and NewReleasePrice.

```
public class ChildrensPrice {
    public int getPriceCode() {
        return Movie.CHILDRENS;
    }
}
public class NewReleasePrice {
    public int getPriceCode() {
        return Movie.NEW_RELEASE;
    }
}
public class RegularPrice {
    public int getPriceCode() {
        return Movie.REGULAR;
    }
}
```

We provide the type code behavior in the price object. We can use Extract Superclass here. Right click one of the three classes, then select “Refactor” -> “Extract Superclass”. Set “Price” as the Superclass name. Make sure that you add all the sub price classes. Check the getPriceCode member and select “declare abstract in superclass”.



We do this with an abstract method on price and concrete methods in the subclasses. We can compile the new classes at this point.

Now we need to change movie's accessors for the price code to use the new class:

```
class Movie ...
    private int _priceCode;
    private Price _price;
```

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

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

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

Now we apply Move Method to getCharge by moving it from Movie into Price class:

```
class Price …
    double getCharge(int daysRented) {
        //determine amounts for each line
```

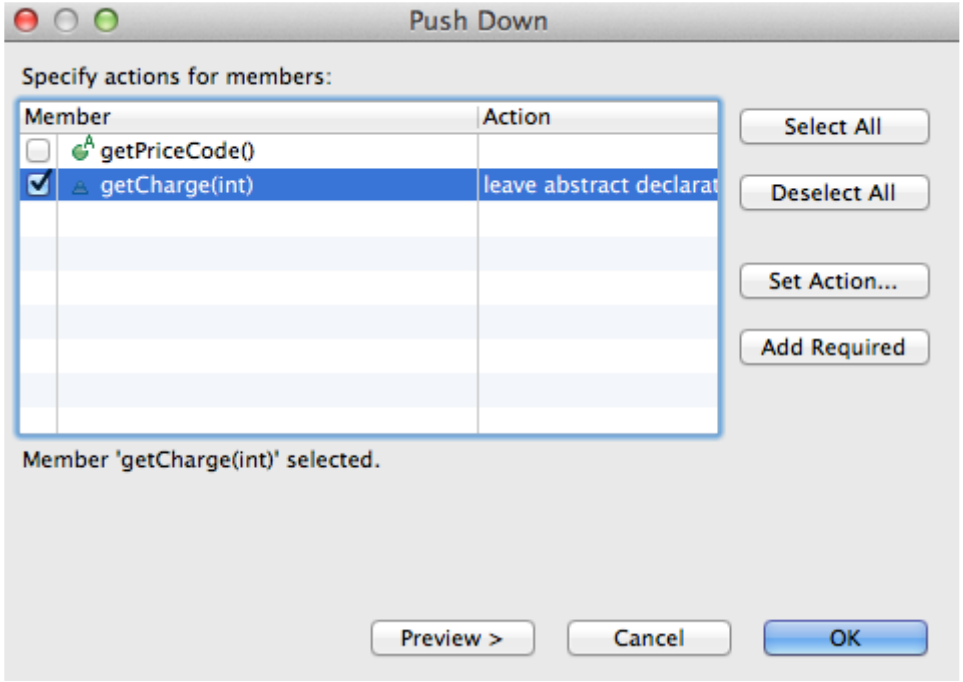
```
    double result = 0;
    switch (getPriceCode()) {
    case Movie.REGULAR:
        result += 2;
        if (daysRented > 2) {
            result += (daysRented - 2) * 1.5;
        }
        break;
    case Movie.NEW_RELEASE:
        result += daysRented * 3;
        break;
    case Movie.CHILDRENS: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

```
class Movie …
    double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (daysRented > 2) {
```

```
            result += (daysRented - 2) * 1.5;
        }
        break;
    case Movie.NEW_RELEASE:
        result += daysRented * 3;
        break;
    case Movie.CHILDRENS:
        result += 1.5;
        if (daysRented > 3) {
            result += (daysRented - 3) * 1.5;
        }
        break;
    }
    return result;
    return _price.getCharge(daysRented);
}
```

Replace Conditional with Polymorphism

Since there is the switch statement in the getCharge function. We do the Replace Conditional with Polymorphism by taking one leg of the case statement at a time and creating an overriding method. We first Push Down the getCharge. Right click one of the Price class, then select “Refactor” -> “Push Down”. Check the getCharge function. Then select “leave abstract declaration”.



For example, to modify ChildrenPrice:

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

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

This overrides the parent case statement, which we just leave as it is. We compile and test for this case then take the next leg, compile and test. **You need to repeat the above for RegularPrice and NewReleasePrice**.

Putting in the state pattern was quite an effort. Was it worth it? The gain is that if I change any of price's behavior, add new prices, or add extra price-dependent behavior, the change will be much easier to make. The rest of the application does not know about the use of the state pattern. For the tiny amount of behavior we currently have, it is not a big deal. In a more complex system with a dozen or so price-dependent methods, this would make a big difference.

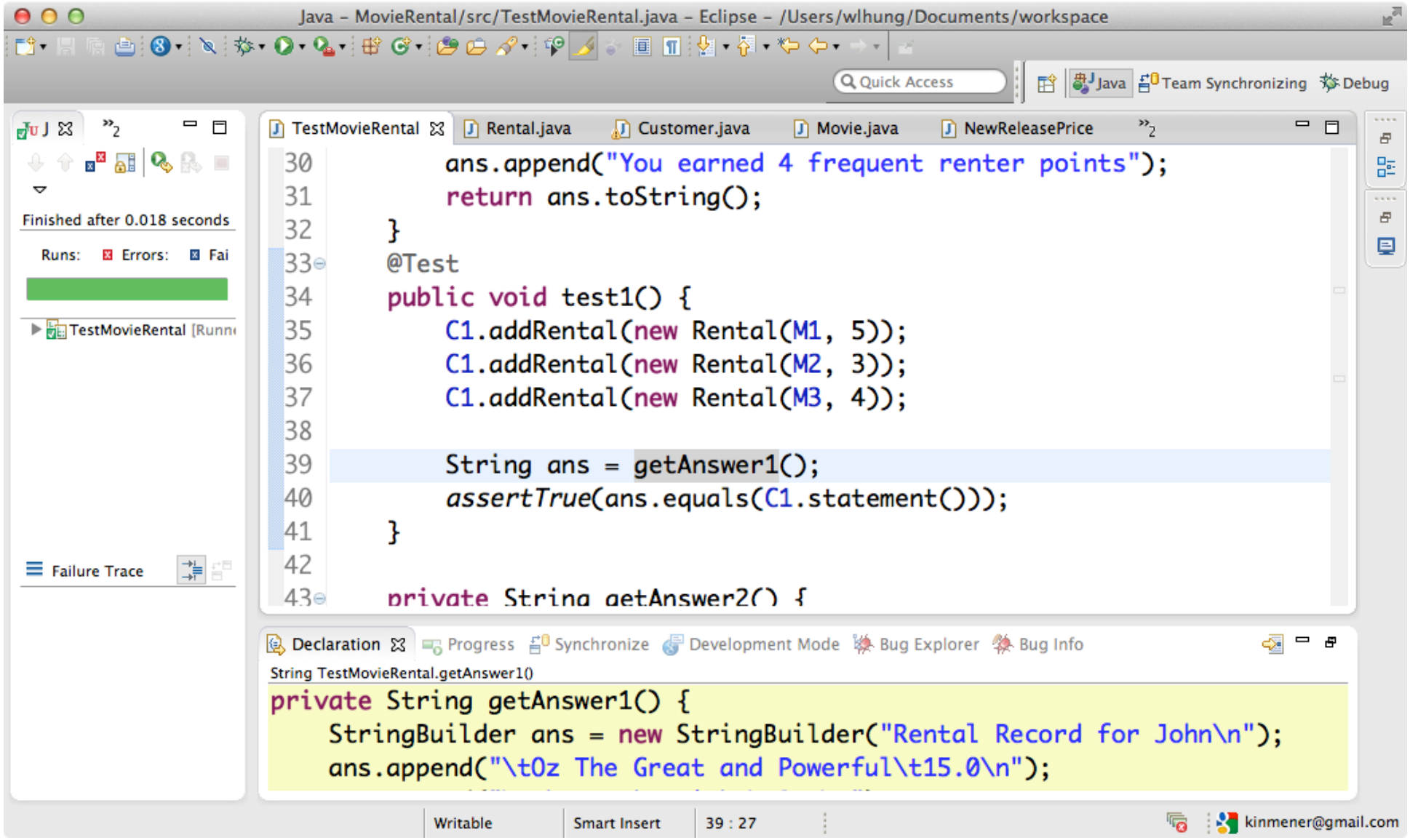
Exercise - Use Pull Up to refactor getFrequentRenterPoints

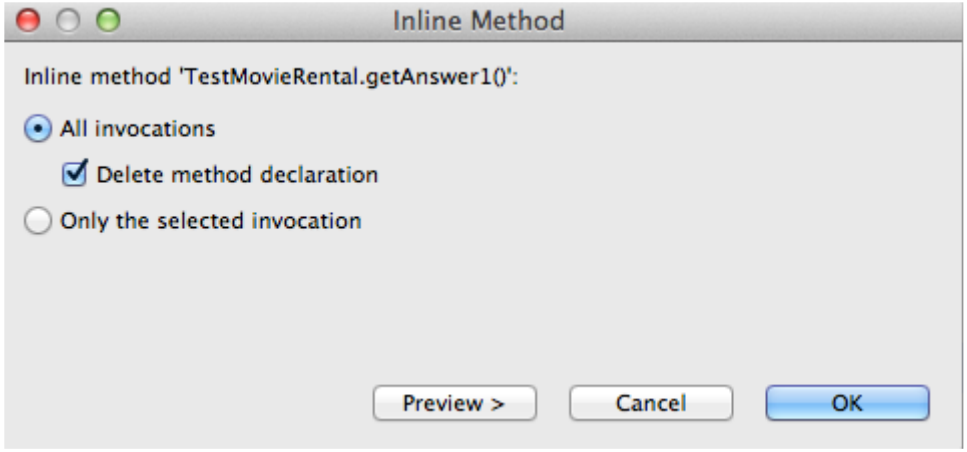
First, you should create the getFrequentRenterPoints function of RegularPrice. In this case, you always return 1. Since the logic of the ChildrensPrice is the same. You use the refactoring tool of Eclipse to Pull Up the getFrequentRenterPoints function toPrice class. However, the logic of NewReleasePrice is different. You need to override its getFrequentRenterPoints. Next, delegate the getFrequentRenterPoints of Movie to _price.getFrequentRenterPoints().

Inline Method

Sometimes you do come across a method in which the body is as clear as the name. Or you refactor the body of the code into something that is just as clear as the name. When this happens, you should then get rid of the method. We use Inline Method when someone is using too much indirection and it seems that every method does simple delegation to another method, we get lost in all the delegation.

Let’s move to the TestMovieRental, for each test, the answer is get from a function, e.g., getAnswer1(). Let us perform Inline Method on the function getAnswer1(). Move the cursor to the function, right click and select “Refactor” -> “Inline”.


Java – MovieRental/src/TestMovieRental.java – Eclipse – /Users/wlhung/Documents/workspace
Quick Access
Java
Team Synchronizing
Debug
Finished after 0.018 seconds
Runs:
Errors:
Fai
TestMovieRental [Runn
Failure Trace
TestMovieRental
Rental.java
Customer.java
Movie.java
NewReleasePrice
30 ans.append("You earned 4 frequent renter points");
31 return ans.toString();
32 }
33 @Test
34 public void test1() {
35 C1.addRental(new Rental(M1, 5));
36 C1.addRental(new Rental(M2, 3));
37 C1.addRental(new Rental(M3, 4));
38
39 String ans = getAnswer1();
40 assertTrue(ans.equals(C1.statement()));
41 }
42
43 private String getAnswer2() {
Declaration
Progress
Synchronize
Development Mode
Bug Explorer
Bug Info
String TestMovieRental.getAnswer1()
private String getAnswer1() {
StringBuilder ans = new StringBuilder("Rental Record for John\n");
ans.append("\tOz The Great and Powerful\t15.0\n");
Writable
Smart Insert
39 : 27
kinmener@gmail.com



Select "All invocations" and check "Delete method declaration". **Repeat this procedure to inline getAnswer2() and getAnswer3().**

We've now completed the second major refactoring. It is going to be much easier to change the classification structure of movies, and to alter the rules for charging and the frequent renter point system.

What to Submit

There were several exercises explicitly called out in this tutorial that you were to do on your own. In addition, you should have followed along in all of the steps described in the body of the tutorial, including a couple of places where you were to effectively repeat an example that we showed explicitly on one piece for two additional pieces that were extremely similar.

Submit your code in a zip file, where the code is at the top level of the zip file. Were I to unzip it, I would have all of your source files in the same top-level directory. No packages please.