Exercise 12: Refactoring and design patterns

Refactoring is a technique for improving 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 behaviour.

You start with a programme that runs correctly but is not structured well. Refactoring improves the structure, making it easier to maintain and expand the programme.

The starting point for the exercise

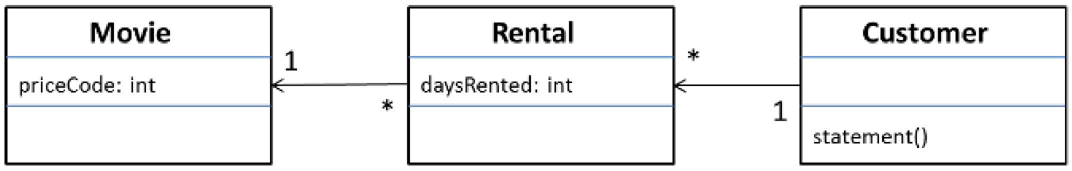
The sample programme is quite straightforward. It is a programme for calculating and printing a customer's statement in a video store. The programme is told which movies a customer has rented and for how long. It then calculates the fees, which depend on how long the movie is rented for, and identifies the type of movie.

There are 3 types of movie:

- normal movies
- children's movies
- new releases

In addition to calculating the fees, the invoicing process calculates bonus points, which depend on whether the movie is a new release.

Here is a corresponding class diagram:



We will now review this programme step by step. Fortunately, there is already a TestCase (see test folder). Take a look at this to understand what the programme does.

Task 1: Extracting the amount calculation

The obvious first goal is the too long `statement()` method in the Customer class. Some of the code should be removed from the method in order to extract a new method from it. Extracting a method means taking out code and making this into a method. A "suspicious" piece of code is obviously the switch statement:

```
//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.CHILDREN:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
        thisAmount += (each.getDaysRented() - 3) * 1.5;
        break;
}
```

When we extract a method, we must search the code fragment for variables whose validity range (scope) is local to the method we are looking at, i.e. for local variables and parameters.

This code segment uses two: `each` and `thisAmount`. Of the two, each is not changed by the code, but only `thisAmount`.

The extraction looks like this:

- we introduce a new method `amountFor`
- and replace the call in the `statement()` method

```
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.CHILDREN:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                        thisAmount += (each.getDaysRented() - 3) * 1.5;
                break;
        }
 return thisAmount;
}
```

```
while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();
        //determine amounts for each line
->      thisAmount = amountFor(each);
        ...
```

Note: Run the test and see if the modification has changed anything!

Task 2: Rename

A suitable approach is to rename the parameters in the `amountFor` method; from `each` to `aRental` and the attribute thisAmount to result.

After that, the code looks like this:

```
private int amountFor(Rental aRental) {

 int 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.CHILDREN:
                result += 1.5;
                if (aRental.getDaysRented() > 3)
                        result += (aRental.getDaysRented() - 3) * 1.5;
                break;
        }
 return result;
}
```

Run tests!

Task 3: Moving amount calculation

If we look at `amountFor`, we can see that information is used from the Rental class, but no information from the `Customer` class. In most cases, there should be a method in the object whose data it uses. So is this method in the wrong object? Should it be moved to `Rental`? - Yes

So move the code to the `Rental` class:

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

And in the `Customer` class, the call then looks rather unspectacular:

```
class Customer
        ...
        private double amountFor(Rental aRental) {
                return aRental.getCharge();

}
```

Oh yes.... Run tests!

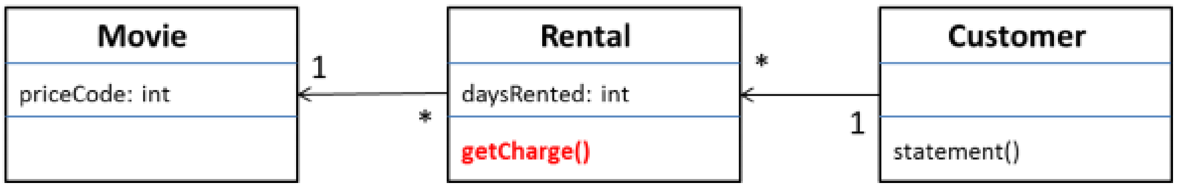
And now we can even get rid of the call of `amountFor` completely. In the `Customer` class, we replace:

```
//determine amounts for each line
thisAmount = amountFor(each);
```

with

```
//determine amounts for each line
thisAmount = each.getCharge();
```

It's already tidied up quite well! Responsibilities have been smoothed and there is less code in the critical `statement` method. The model now looks like this:



Finally, we can delete and replace a few more lines here:

```
public String statement() {

                        double totalAmount = 0;
                        int frequentRenterPoints = 0;
                        Enumeration rentals = _rentals.elements();
                        String result = "Rental Record for " + name() + "\n";
                        while (rentals.hasMoreElements()) {
löschen -->             double thisAmount = 0;
                                Rental each = (Rental) rentals.nextElement();
löschen -->             //determine amounts for each line
löschen -->             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.getDaysRe
                                //show figures for this rental
change -->                      result += "\t" + each.getMovie().getTitle()+ "\t" + String.valueOf(each.get
change -->              totalAmount += each.getCharge();
                        }
                        //add footer lines
                        result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
                        result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter
                        return result;
```

Task 4: Extracting frequent renter points

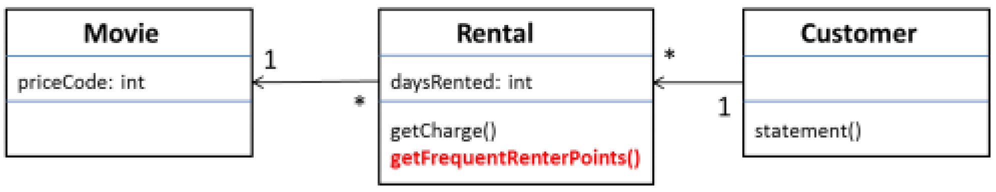
In the next step, we do the same for the frequentRenterPoints. First of all, a method is extracted and immediately put into Rental:

```
class Rental...
        int getFrequentRenterPoints() {
                if ((getMovie().getPriceCode() == Movie.NEW_RELEASE) && getDaysRented() > 1)
                        return 2;
                else
                        return 1;
        }
```

And in the Customer class in the `statement` method, replace:

```
while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        frequentRenterPoints += each.getFrequentRenterPoints();
```

The model then looks like this:



And again… Run tests!

Task 5: Removing temps

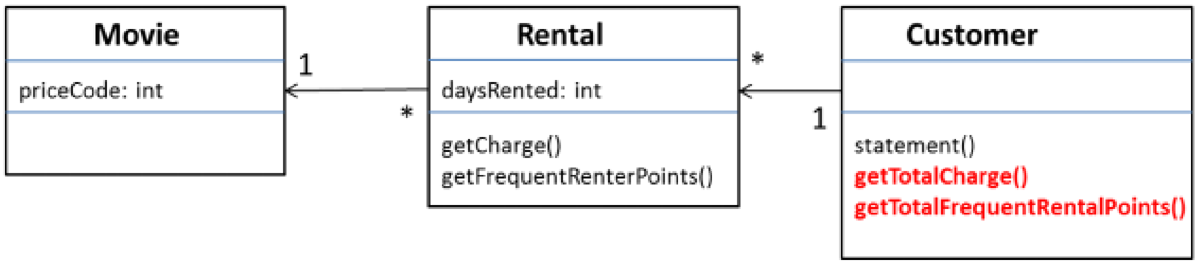
The next step is to throw out some temporary variables in `Customer`. To do this, we introduce the following method in the `Customer` class:

```
private double getTotalCharge(){

        double result = 0;
        Enumeration rentals = _rentals.elements();

        while (rentals.hasMoreElements()) {
                Rental each = (Rental) rentals.nextElement();
                result += each.getCharge();
        }
        return result;
}

private double getTotalFrequentRenterPoints(){

        double result = 0;
        Enumeration rentals = _rentals.elements();

        while (rentals.hasMoreElements()) {
                Rental each = (Rental) rentals.nextElement();
                result += each.getFrequentRenterPoints();
        }
        return result;
}
```

and make the following changes to the `statement` method:

```
public String statement() {

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

                                //show figures for this rental
                                result += "\t" + each.getMovie().getTitle()+ "\t" + String.value
löschen -->             totalAmount += each.getCharge();
                        }
                        //add footer lines
change -->          result += "Amount owed is " + String.valueOf(getTotalCharge()) + "\n";
change -->          result += "You earned " + String.valueOf(getTotalFrequentRenterPoints()) + " fre
                        return result;
}
```

The corresponding model:



Task 6: Replacing the conditional logic on price code with polymorphism

Now we come to my favourite topic. We will replace the switch statement with polymorphism.

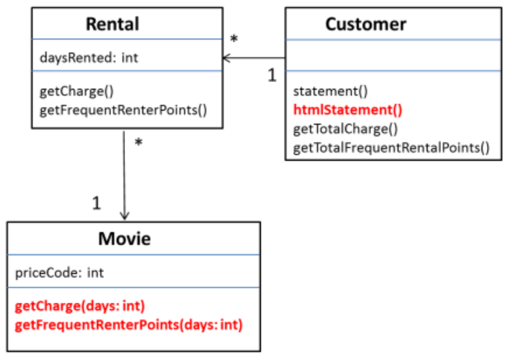
First, we move the getCharge method to the `Movie` class, then all getMovie() calls can be deleted since we are already in the Movie class. Obviously, this code snippet belongs here:

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

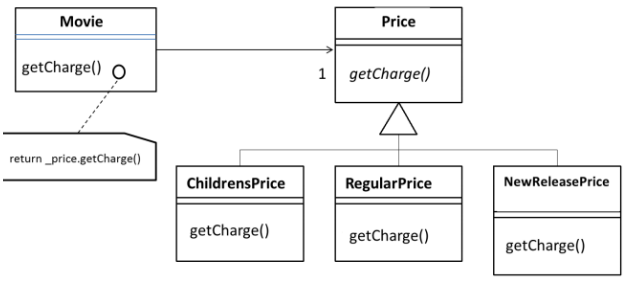
This changes the call in the `Rental` class to:

```
Class Rental…

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

The same change applies to the `getFrequentRenterPoints` method. This also moves into the `Movie` class. The model then looks like this:



Next, we introduce a `Price` class. It should look like this in the class diagram:



In other words, we create 4 classes:

- `Price` (is the base class. Can/should it be `abstract`?)
- `ChildrensPrice`
- `RegularPrice`
- `NewReleasePrice`

We change the `Movie` class as follows:

```
public class Movie {

        private Price _price;

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

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

        public void setPriceCode(int arg) {
                switch (arg) {
                        case Movie.REGULAR:
                                _price = new RegularPrice();
                                break;
                        case Movie.CHILDREN:
                                _price = new ChildrensPrice();
                                break;
                        case Movie.NEW_RELEASE:
                                _price = new NewReleasePrice();
                                break;
                        default:
                                throw new IllegalArgumentException(“Incorrect Price Code”);
                }
        }
```

For the `Price` classes, the following applies:

```
abstract class Price {
        abstract int getPriceCode() {
}

class ChildrensPrice extends Price {
        int getPriceCode() {
                Return Movie.CHILDREN;
        }
}

class NewReleasePrice extends Price {
        int getPriceCode() {
                Return Movie.NEW_RELEASE;
        }
}

class RegularPrice extends Price {
        int getPriceCode() {
                Return Movie.REGULAR;
        }
}
```

And now let's give the `Price` base class an abstract method `abstract double getCharge(int daysRented)`. In the subclasses, we can now move the code taken from the `getCharge` method out of the `Movie` class and into the respective `Price` classes:

```
class RegularPrice…

        double getCharge(int daysRented) {
                double result = 2;
                if (daysRented > 2)
                        result += (daysRented - 2) * 1.5;
                return result;
        }


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;
        }
```

This reduces the call in the `Movie` class to:

```
class Movie …

        double getCharge(int daysRented) {
                return _price.getCharge(daysRented);
        }
```

We have the same fun with the `getFrequentRenterPoints` method.

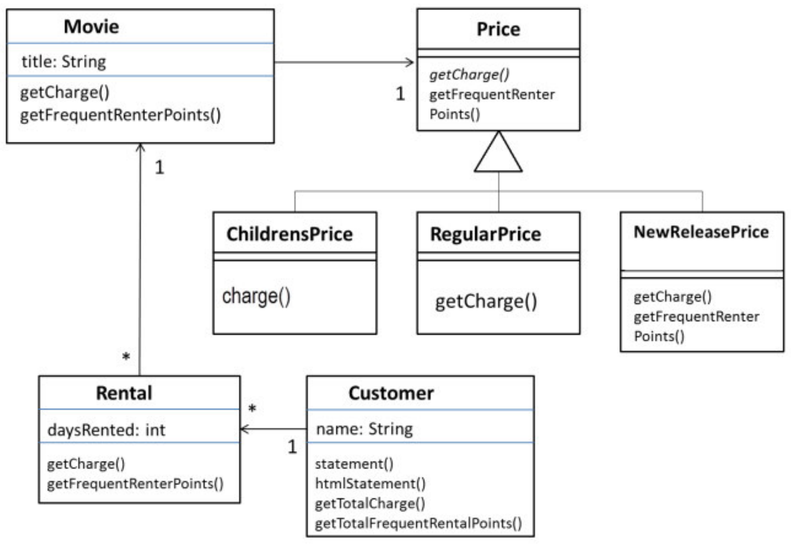
The nice thing here is that only `NewReleasePrice` needs a special implementation, while the basic functionality can happen in the base class `Price`.

As always: don't forget to test!

This is the end!

When we're done, the test should still run properly. The model has changed significantly, but it shows a much more object-oriented structure. Above all, it is important that the code has become clearer and better structured.

The final model looks like this:



Done!