Übung 12: Refactoring und Design Pattern

Refactoring ist eine Technik zur Verbesserung der Qualität von vorhandenem Code. Es funktioniert durch Anwenden einer Reihe von kleinen Schritten, von denen jeder die interne Struktur des Codes unter Beibehaltung seines äußeren Verhaltens ändert.

Sie beginnen mit einem Programm, das korrekt ausgeführt wird, aber nicht gut strukturiert ist. Refactoring verbessert die Struktur, wodurch es einfacher wird, das Programm zu pflegen und zu erweitern.

Der Startpunkt für die Übung

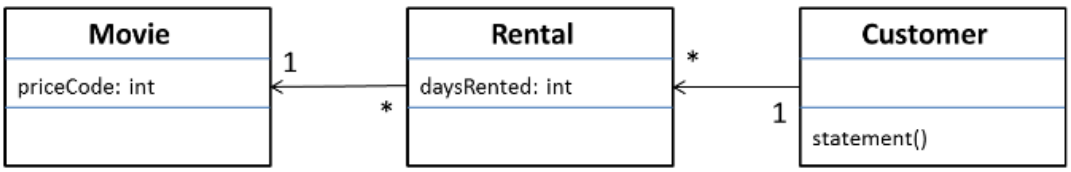
Das Beispielprogramm ist recht überschaubar. Es ist ein Programm zum Berechnen und Drucken des Beleges eines Kunden in einem Videogeschäft. Dem Programm wird mitgeteilt, welche Filme ein Kunde wie lange ausgeliehen hat. Es berechnet dann die Gebühren, die davon abhängen, wie lange der Film ausgeliehen wurde und identifiziert die Art des Films.

Es gibt 3 Arten von Filmen:

- normale Filme
- Kinderfilme
- Neuerscheinungen

Neben der Berechnung der Gebühren werden in der Abrechnung Bonuspunkte berechnet, die abhängig davon sind, ob der Film eine Neuerscheinung ist.

Hier ein dazu passendes Klassendiagramm:



Wir werden nun dieses Programm Schritt für Schritt überarbeiten. Glücklicherweise gibt es bereits einen TestCase (siehe Testfolder). Schauen Sie sich diesen an, um zu verstehen, was das Programm überhaupt macht.

Aufgabe 1: Extracting the Amount Calculation

Das offensichtliche erste Ziel ist die zu lange `statement()`-Methode in der Customer-Klasse. Aus der Methode sollte ein Teil des Codes herausgenommen werden, um daraus dann eine neue Methode zu extrahieren. Das Extrahieren einer Methode bedeutet, den Code zu entnehmen und daraus eine Methode zu machen. Ein "verdächtiges" Stück Code ist offensichtlich die switch-Anweisung:

```
//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;
}
```

Wenn wir eine Methode extrahieren, müssen wir im Code-Fragment nach Variablen suchen, deren Gültigkeitsbereich lokal für die Methode ist, die wir betrachten, also nach lokalen Variablen und Parametern.

Dieses Codesegment verwendet zwei: `each` und `thisAmount`. Von diesen wird `each` nicht durch den Code geändert, sondern nur `thisAmount`.

Die Extraktion sieht so aus:

- Wir führen eine neue Methode `amountFor` ein
- und ersetzen den Aufruf in der der `statement()`–Methode

```
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);
        ...
```

Achtung: Test laufen lassen und schauen, ob die Änderung etwas verändert hat!

Aufgabe 2: Umbenennen (Rename)

Es bietet sich an in der `amountFor`-Methode den Parameter umzubenennen; von `each` in `aRental` und das Attribut `thisAmount` in `result`.

Danach sieht der Code wie folgt aus:

```
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;
}
```

Tests laufen lassen!

Aufgabe 3: Moving amount calculation

Sieht man sich `amountFor` an, sieht man, dass Informationen aus der Rental-Klasse verwendet werden, aber keine Informationen der `Customer`-Klasse. In den meisten Fällen sollte sich eine Methode in dem Objekt befinden, dessen Daten es verwendet. Befindet sich diese Methode also im falschen Objekt? Sollte sie in `Rental` verschoben werden? - Ja.

Also den Code in die `Rental`-Klasse umziehen:

```
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;
}
```

Und in der `Customer`-Klasse sieht der Aufruf danach recht unspektakulär aus:

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

}
```

Ach ja: Tests laufen lassen!

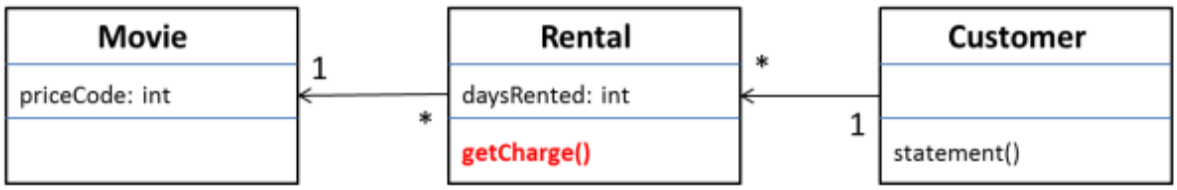
Und nun können wir sogar noch den Aufruf von `amountFor` komplett loswerden. Wir ersetzen in der `Customer`-Klasse:

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

Durch

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

Schon ganz gut aufgeräumt! Verantwortlichkeiten glattgezogen und weniger Code in der kritischen `statement`-Methode. Das Model sieht nun so aus:



Final können wir hier noch ein paar Zeilen löschen und ersetzen:

```
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;
```

Aufgabe 4: Extracting Frequent Renter Points

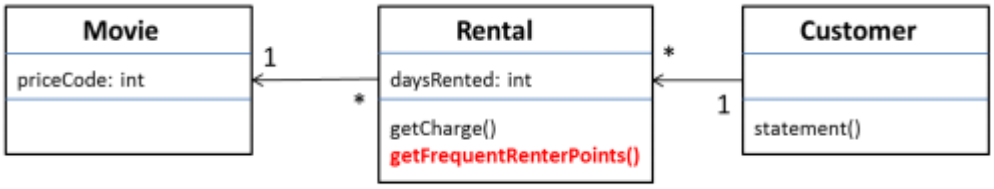
Im nächsten Schritt machen wir das Gleiche für die `frequentRenterPoints`. Zunächst wird eine Methode extrahiert und schieben dies gleich in `Rental`:

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

Und in der Customer Klasse in der `statement`-Methode ersetzen:

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

Modell sieht dann so aus:



und wieder: Tests laufen lassen!

Aufgabe 5: Removing Temps

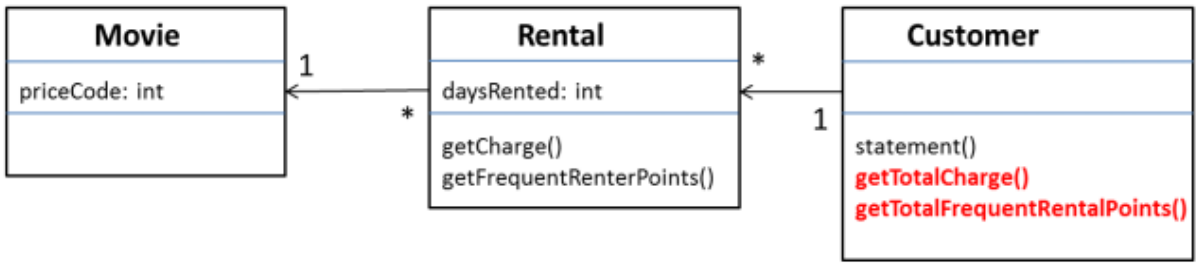
Als nächsten Schritt schmeißen wir ein paar temporäre Variablen in `Customer` raus. Dazu führen wir folgende Methode ein in der `Customer`-Klasse ein:

```
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;
}
```

und machen folgende Änderungen in der `statement`-Methode:

```
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;
}
```

Das Model dazu:



Aufgabe 6: Replacing the Conditional Logic on Price Code with Polymorphism

Nun kommt mein Lieblingsthema. Wir werden das switch-Statement durch Polymorphie ersetzen.

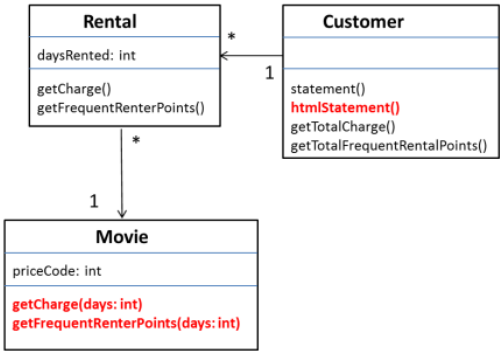
Zunächste verschieben wir die Methode `getCharge` in die `Movie`-Klasse, da können sämtliche `getMovie`()-Aufrufe gelöscht werden, denn wir befinden uns ja in der `Movie`-Klasse. Offensichtlich gehört dieser Codeteil hier hin:

```
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;
        }
```

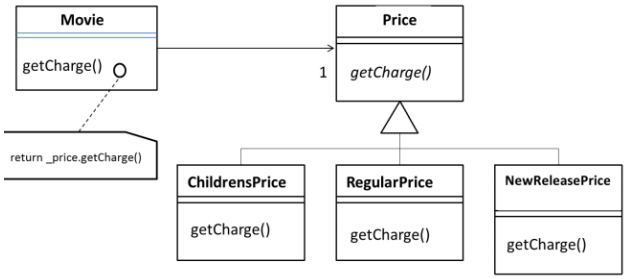
Damit ändert sich der Aufruf in der `Rental`-Klasse zu:

```
Class Rental…

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

Die gleiche Änderung gilt auch für die `getFrequentRenterPoints`-Methode. Auch diese wandert in die `Movie`-Klasse. Das Modell sieht dann so aus:



Als, nächstes führen wir eine `Price`-Klasse ein. Das soll im Klassendiagramm so aussehen:



Sprich, wir erstellen 4 Klassen:

- `Price` (das ist die Basisklasse. Kann/Soll die `abstract` sein?)
- `ChildrensPrice`
- `RegularPrice`
- `NewReleasePrice`

Die `Movie`-Klasse ändern wir, wie folgt:

```
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");
                }
        }
```

Für die `Price`-Klassen git:

```
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;
        }
}
```

un, spendieren wir der `Price`-Basisklasse eine abstrakte Methode `abstract double getCharge(int dasyRented)`. In den Unterklassen können wir nun den Code aus der `getCharge`-Methode von der `Movie`-Klasse in die jeweilige `Price`-Klassen umziehen:

```
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;
        }
```

Damit reduziert sich der Aufruf in der `Movie`-Klasse zu:

```
class Movie …

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

Den gleichen Spaß machen wir mit der `getFrequentRenterPoints`-Methode.

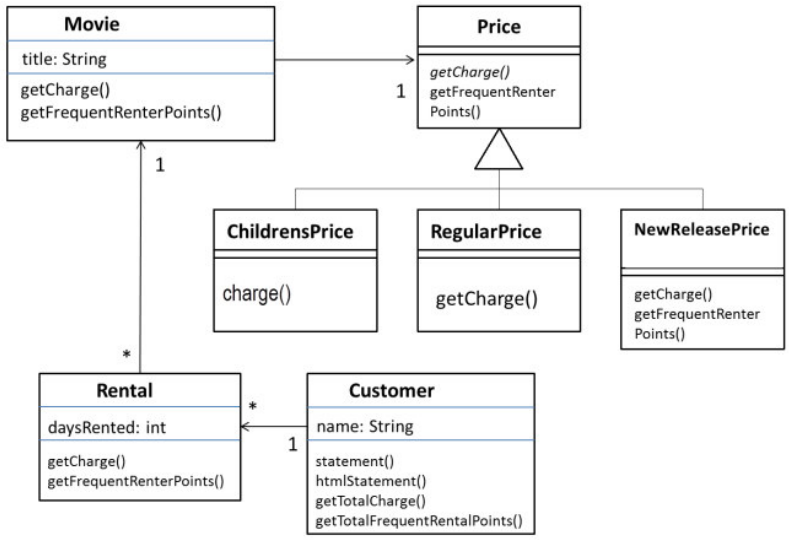
Das Schöne hier ist, dass nur `NewReleasePrice` eine spezielle Implementierung benötigt, während die Basisfunktionalität in der Basisklasse `Price` passieren kann.

Wie immer: Testen nicht vergessen!

This is the end!

Wenn wir fertig sind, sollte der Test immernoch duchlaufen. Das Model hat sich deutlcih geändert, zeigt aber eine viel objektorientiertere Struktur. Vor allem is wichtig, dass der Code klarer und strukturierter geworden ist.

Final sieht das Model so aus:



Done!