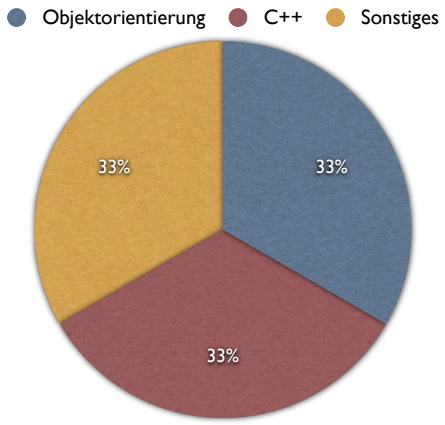
Modularisierung

Andreas Zeller

Klausur



Evaluation

http://frweb.cs.uni-sb.de/03.Studium/08.Eva/

Modularisierung

Andreas Zeller

4

Modularisierung

Änderungen lokal halten!

5

Begriffe

- Kopplung
- Kohäsion
- Code-Duplikate
- Einkapselung
- Lokalisierung
- Refactoring

6

Vgl. Kapitel 7 im BlueJ-Buch

Refactoring im Überblick

Refactoring (wörtl. „Refaktorisieren“) bedeutet das *Aufspalten* von Software in weitgehend unabhängige *Faktoren*

…oder anders ausgedrückt: Umstrukturieren von Software gemäß den Zerlegungsregeln zur Modularisierung.

Mit Refactoring kann man

- die Struktur eines *objektorientierten Entwurfs* verbessern
- nicht nur Entwürfe, sondern auch *bereits codierte Systeme* überarbeiten.

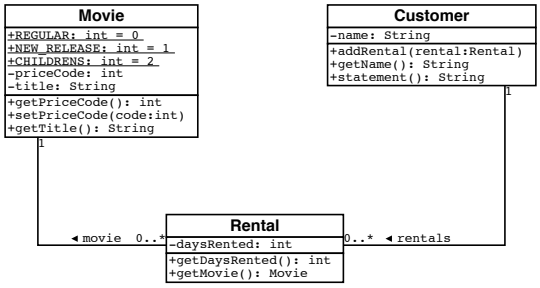
Hierzu gibt es *Kataloge* von Refactoring-Methoden, ähnlich wie bei → *Entwurfsmustern*

Beispiel: Der Videoverleih

Gegeben ist ein Programm zum Erstellen von Rechnungen in einem Videoverleih:

- *Welche* Videos hat der Kunde *wie lange* ausgeliehen?
- Es gibt drei *Arten* von Videos: Normal, Kinder und Neuerscheinungen.
- Es gibt *Rabatt* auf das verlängertes Ausleihen von normalen und Kinder-Videos (nicht jedoch für *Neuerscheinungen*)
- Es gibt *Bonuspunkte* für Stammkunden (wobei das Ausleihen von Neuerscheinungen Extra-Punkte bringt)

Ausgangssituation



Die Videoarten (`priceCode`) werden durch Klassen-Konstanten (unterstrichen) gekennzeichnet.

Die gesamte Funktionalität steckt im Erzeugen der Kundenrechnung – der Methode `Customer.statement()`.

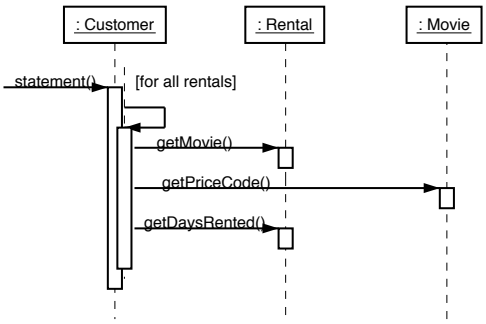
Customer.statement()

```
1   public String statement() {
2       double totalAmount = 0.00;
3       int frequentRenterPoints = 0;
4       Enumeration rentals = _rentals.elements();
5       String result = "Rental Record for " + getName() + "\n";
6
7       while (rentals.hasMoreElements()) {
8           double thisAmount = 0.00;
9           Rental each = (Rental) _rentals.nextElement();
10
11          // Kosten pro Video berechnen
12          switch (each.getMovie().getPriceCode()) {
13          case Movie.REGULAR:
14              thisAmount += 2.00;
15              if (each.getDaysRented() > 2)
16                  thisAmount += (each.getDaysRented() - 2) * 1.50;
17              break;
18
19          case Movie.NEW_RELEASE:
20              thisAmount += each.getDaysRented() * 3.00;
21              break;
22
23          case Movie.CHILDRENS:
24              thisAmount += 1.50;
```



```
25              if (each.getDaysRented() > 3)
26                  thisAmount += (each.getDaysRented() - 3) * 1.50;
27              break;
28          }
29
30          // Bonuspunkte berechnen
31          frequentRenterPoints++;
32
33          if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
34              each.getDaysRented() > 1)
35              frequentRenterPoints++;
36
37          // Zeile berechnen
38          result += "\t" + each.getMovie().getTitle() + "\t" +
39              String.valueOf(thisAmount) + "\n";
40          totalAmount += thisAmount;
41      }
42
43      // Summe
44      result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
45      result += "You earned " + String.valueOf(frequentRenterPoints) +
46          " frequent renter points";
47      return result;
48  }
```


Sequenzdiagramm



Probleme mit diesem Entwurf

- Nicht objektorientiert – Filmpreise sind z.B. Kunden zugeordnet
- Mangelnde Lokalisierung – Das Programm ist nicht robust gegenüber Änderungen:
  - Erweiterung des Ausgabeformats (z.B. HTML statt Text): Schreibt man eine neue Methode `htmlStatement()`?
  - Änderung der Preisberechnung: was passiert, wenn neue Regeln eingeführt werden? An wieviel Stellen muß das Programm geändert werden?

Ziel: Die einzelnen *Faktoren* (Preisberechnung, Bonuspunkte) voneinander trennen!

Methoden aufspalten („Extract Method“)

Als ersten Schritt müssen wir die viel zu lange `statement()`-Methode aufspalten. Hierzu führen wenden wir das Refactoring-Verfahren „Extract Method“ ein.

Extract Method

„Extract Method“ ist eine der verbreitesten Refactoring-Methoden. Sie hat die allgemeine Form:

> Es gibt ein Codestück, das zusammengefasst werden kann.
>
> *Wandle das Codestück in eine Methode, deren Name den Zweck der Methode erklärt!*

Extract Method (2)

```
void printOwing(double amount) {
    printBanner();
    // print details
    System.out.println("name: " + _name);
    System.out.println("amount: " + amount);
}
```

wird zu

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}
void printDetails(double amount) {
    System.out.println("name: " + _name);
    System.out.println("amount: " + amount);
}
```

Spezifisches Problem: Umgang mit lokalen Variablen.

Extract Method (3)

```
1   public String statement() {
2       double totalAmount = 0.00;
3       int frequentRenterPoints = 0;
4       Enumeration rentals = _rentals.elements();
5       String result = "Rental Record for " + getName() + "\n";
6
7       while (rentals.hasMoreElements()) {
8           Rental each = (Rental) _rentals.nextElement();
9           double thisAmount = amountFor(each);        // NEU
10
11          // Bonuspunkte berechnen
12          frequentRenterPoints++;
13
14          if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
15              each.getDaysRented() > 1)
16              frequentRenterPoints++;
17
18          // Zeile berechnen
19          result += "\t" + each.getMovie().getTitle() + "\t" +
20              String.valueOf(thisAmount) + "\n";
21          totalAmount += thisAmount;
22      }
23
24      // Summe
25      result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
26      result += "You earned " + String.valueOf(frequentRenterPoints) +
27          " frequent renter points";
```



```
28      return result;
29  }
30
31  public double amountFor(Rental aRental) {       // NEU
32      double thisAmount = 0.00;
33
34      switch (aRental.getMovie().getPriceCode()) {
35      case Movie.REGULAR:
36          thisAmount += 2.00;
37          if (aRental.getDaysRented() > 2)
38              thisAmount += (aRental.getDaysRented() - 2) * 1.50;
39          break;
40
41      case Movie.NEW_RELEASE:
42          thisAmount += aRental.getDaysRented() * 3.00;
43          break;
44
45      case Movie.CHILDRENS:
46          thisAmount += 1.50;
47          if (aRental.getDaysRented() > 3)
48              thisAmount += (aRental.getDaysRented() - 3) * 1.50;
49          break;
50      }
51
52      return thisAmount;
53  }
```

Bewegen von Methoden („Move Method“)

Die Methode `amountFor()` hat eigentlich nichts beim Kunden zu suchen; vielmehr gehört sie zum Ausleihvorgang selbst.

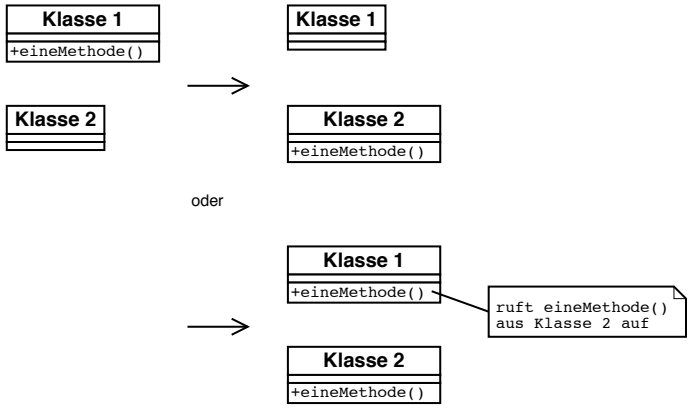
Hierfür setzen wir das Refactoring-Verfahren „Move Method“ ein.

Move Method

„Move Method“ hat die allgemeine Form:

> Eine Methode benutzt weniger Dienste der Klasse, der sie zugehört, als Dienste einer anderen Klasse.
>
> *Erzeuge eine neue Methode mit gleicher Funktion in der anderen Klasse. Wandle die alte Methode in eine einfache Delegation ab, oder lösche sie ganz.*

Move Method (2)



Spezifische Probleme: *Informationsfluss*, *Umgang mit ererbten Methoden*

Anwendung

Wir führen in der `Rental`-Klasse eine neue Methode `getCharge()` ein, die die Berechnung aus `amountFor()` übernimmt.

Rental.getCharge()

```
1   class Rental {
2       // ...
3       public double getCharge() {       // NEU
4           double charge = 0.00;
5
6           switch (getMovie().getPriceCode()) {
7           case Movie.REGULAR:
8               charge += 2.00;
9               if (getDaysRented() > 2)
10                  charge += (getDaysRented() - 2) * 1.50;
11              break;
12
13          case Movie.NEW_RELEASE:
14              charge += getDaysRented() * 3.00;
15              break;
16
17          case Movie.CHILDRENS:
18              charge += 1.50;
19              if (getDaysRented() > 3)
20                  charge += (getDaysRented() - 3) * 1.50;
21              break;
22          }
23
24          return charge;
25      }
26  }
```


Customer.amountFor()

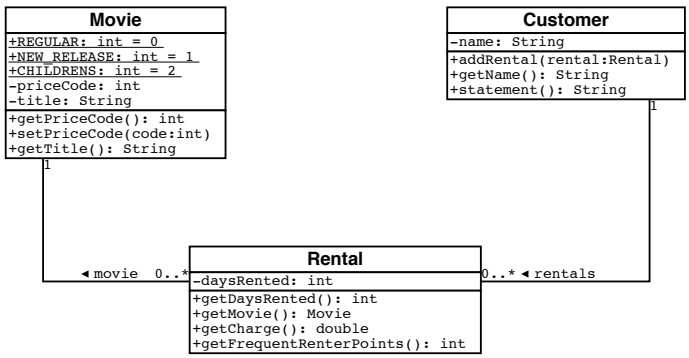
Die umgearbeitete `Customer`-Methode `amountFor()` delegiert nun die Berechnung an `getCharge()`:

```
1   class Customer {
2       // ...
3       public double amountFor(Rental aRental) { // NEU
4           return aRental.getCharge();
5       }
6   }
```

Genau wie das Berechnen der Kosten können wir auch das Berechnen der Bonuspunkte in eine neue Methode der `Rental`-Klasse verschieben – etwa in eine Methode `getFrequentRenterPoints()`.

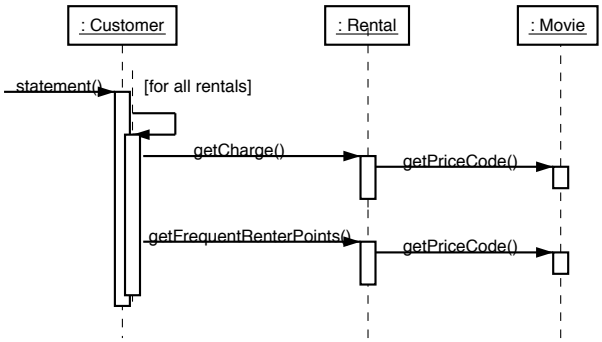
Neue Klassen

Die Klasse `Rental` hat die neuen Methode `getCharge()` und `getFrequentRenterPoints()`:



Neues Sequenzdiagramm

Die Klasse `Customer` muß sich nicht mehr um Preis-Codes kümmern; diese Verantwortung liegt nun bei `Rental`.



Abfrage-Methoden einführen

Die `while`-Schleife in `statement` erfüllt drei Zwecke gleichzeitig:

- Sie berechnet die einzelnen Zeilen
- Sie summiert die Kosten
- Sie summiert die Bonuspunkte

Auch hier sollte man die Funktionalität in separate Elemente aufspalten, wobei uns das Refactoring-Verfahren „Replace Temp with Query“ hilft.

Replace Temp with Query

Eine temporäre Variable speichert das Ergebnis eines Ausdrucks.

Stelle den Ausdruck in eine Abfrage-Methode; ersetze die temporäre Variable durch Aufrufe der Methode. Die neue Methode kann in anderen Methoden benutzt werden.

Replace Temp with Query (2)

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000.00) {
    return basePrice * 0.95;
else
    return basePrice * 0.98;
}
```

wird zu

```
if (basePrice() > 1000.00) {
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;
}

double basePrice() {
    return _quantity * _itemPrice;
}
```


Anwendung

Wir führen in der Customer-Klasse zwei private Methoden ein:

- `getTotalCharge()` summiert die Kosten
- `getTotalFrequentRenterPoints()` summiert die Bonuspunkte

```
1   public String statement() {
2       Enumeration rentals = _rentals.elements();
3       String result = "Rental Record for " + getName() + "\n";
4
5       while (rentals.hasMoreElements()) {
6           Rental each = (Rental) _rentals.nextElement();
7
8           result += "\t" + each.getMovie().getTitle() + "\t" +
9               String.valueOf(each.getCharge()) + "\n";
10      }
11
12      result += "Amount owed is " + String.valueOf(getTotalCharge()) + "\n";
13      result += "You earned " + String.valueOf(getTotalFrequentRenterPoints())
14          " frequent renter points";
15      return result;
```

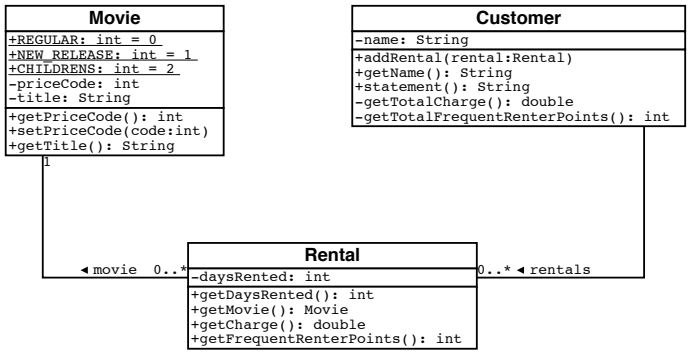
```
16  }
17
18  private double getTotalCharge() { // NEU
19      double charge = 0.00;
20      Enumeration rentals = _rentals.getElements();
21      while (rentals.hasMoreElements()) {
22          Rental each = (Rental) rentals.nextElement();
23          charge += each.getCharge();
24      }
25      return charge;
26  }
27
28  private int getTotalFrequentRenterPoints() { // NEU
29      int points = 0;
30      Enumeration rentals = _rentals.getElements();
31      while (rentals.hasMoreElements()) {
32          Rental each = (Rental) rentals.nextElement();
33          points += each.getFrequentRenterPoints();
34      }
35      return points;
36  }
```

`statement()` ist schon deutlich kürzer geworden!

31

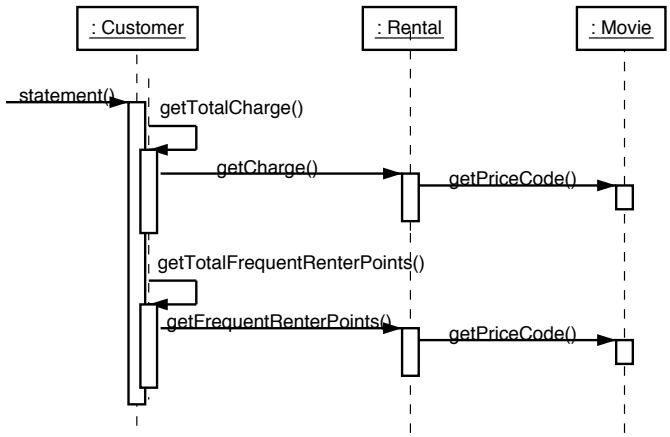
Neue Klassen

Neue private Methoden `getTotalCharge` und `getTotalFrequentRenterPoints`:



32

Neues Sequenzdiagramm



33

Einführen einer HTML-Variante

Da die Berechnungen von Kosten und Bonuspunkten nun komplett herausfaktorisiert sind, konzentriert sich `statement()` ausschließlich auf die korrekte Formatierung.

Nun ist es kein Problem mehr, alternative Rechnungs-Formate auszugeben.

Einführen einer HTML-Variante (2)

Die Methode `htmlStatement()` etwa könnte die Rechnung in HTML-Format drucken:

```
1   public String htmlStatement() {
2       Enumeration rentals = _rentals.elements();
3       String result = "<H1>Rental Record for <EM>" + getName() + "</EM></H1>\n"
4
5       result += "<UL>";
6       while (rentals.hasMoreElements()) {
7           Rental each = (Rental) _rentals.nextElement();
8
9           result += "<LI> " + each.getMovie().getTitle() + ": " +
10              String.valueOf(each.getCharge()) + "\n";
11      }
12      result += "</UL>";
13
14      result += "Amount owed is <EM>" + String.valueOf(getTotalCharge()) +
15          "</EM><P>\n";
16      result += "You earned <EM>" +
17          String.valueOf(getTotalFrequentRenterPoints()) +
18          "</EM> frequent renter points<P>";
19      return result;
20  }
```

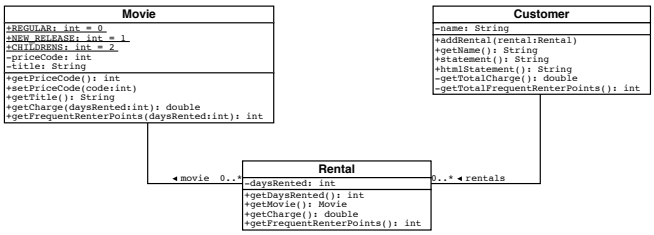

Weiteres Verschieben von Methoden

Wir betrachten noch einmal die Methode `getCharge()` aus der Klasse `Rental`.

Grundsätzlich ist es eine schlechte Idee, Fallunterscheidungen aufgrund der Attribute anderer Objekte vorzunehmen. Wenn schon Fallunterscheidungen, dann auf den eigenen Daten.

Weiteres Verschieben von Methoden (2)

Folge – `getCharge()` sollte in die Klasse `Movie` bewegt werden, und wenn wir schon dabei sind, auch `getFrequentRenterPoints()`:



Weiteres Verschieben von Methoden (3)

Klasse `Movie` mit eigenen Methoden zur Berechnung der Kosten und Bonuspunkte:

```
1   class Movie {
2       // ...
3       public double getCharge(int daysRented) {       // NEU
4           double charge = 0.00;
5
6           switch (getPriceCode()) {
7           case Movie.REGULAR:
8               charge += 2.00;
9               if (daysRented > 2)
10                  charge += (daysRented - 2) * 1.50;
11              break;
12
13          case Movie.NEW_RELEASE:
14              charge += daysRented * 3.00;
15              break;
16
17          case Movie.CHILDRENS:
18              charge += 1.50;
19              if (daysRented > 3)
20                  charge += (daysRented - 3) * 1.50;
```



```
21                  break;
22          }
23
24          return charge;
25      }
26
27      public int getFrequentRenterPoints(int daysRented) {       // NEU
28          if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
29              return 2;
30          else
31              return 1;
32      }
33  }
```

Weiteres Verschieben von Methoden (4)

In der `Rental`-Klasse delegieren wir die Berechnung an das jeweilige `Movie`-Element:

```
1   class Rental {
2       // ...
3       public double getCharge() {       // NEU
4           return getMovie().getCharge(_daysRented);
5       }
6
7       public int getFrequentRenterPoints() {       // NEU
8           return getMovie().getFrequentRenterPoints(_daysRented);
9       }
10  }
```


Polymorphie statt Fallentscheidungen

Fallunterscheidungen innerhalb einer Klasse können fast immer durch Einführen von Unterklassen ersetzt werden („Replace Conditional Logic with Polymorphism“).

Das ermöglicht weitere *Lokalisierung* – jede Klasse enthält genau die für sie nötigen Berechnungsverfahren.

Replace Cond. Logic with Polymorphism

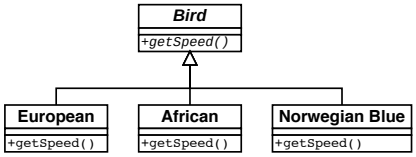
„Replace Conditional Logic with Polymorphism“ hat die allgemeine Form:

> Eine Fallunterscheidung bestimmt verschiedenes Verhalten, abhängig vom Typ des Objekts.
>
> *Bewege jeden Ast der Fallunterscheidung in eine überladene Methode einer Unterklasse. Mache die ursprüngliche Methode abstrakt.*

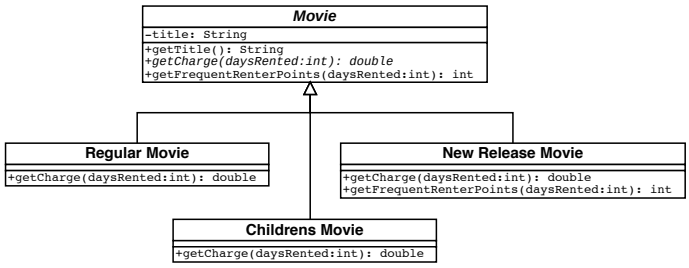
Die afrikanische Schwalbe

```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() -
                        getLoadFactor() * numberOfCoconuts();
        case NORWEGIAN_BLUE:
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
}
```

wird zu



Neue Klassenhierarchie – Erster Versuch



Erster Versuch (2)

Neue Eigenschaften:

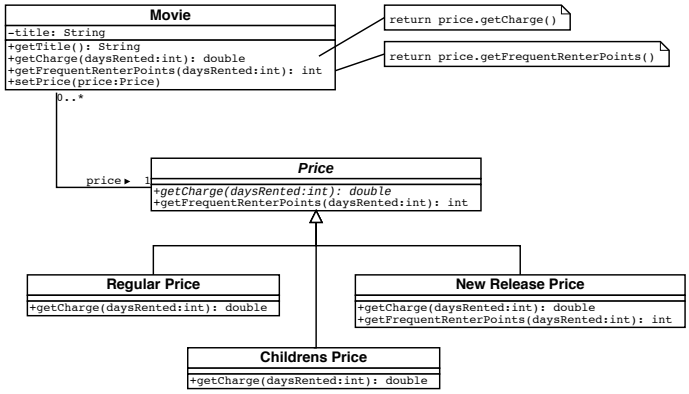
- Die Berechnung der Kosten wird an die Unterklassen abgegeben (abstrakte Methode `getCharge`)
- Die Berechnung der Bonuspunkte steckt in der Oberklasse, kann aber von Unterklassen überladen werden (Methode `getFrequentRenterPoints()`)

Problem dieser Hierarchie: Beim Erzeugen eines `Movie`-Objekts muss die Klasse bekannt sein; während ihrer Lebensdauer können Objekte keiner anderen Klasse zugeordnet werden.

Im Videoverleih kommt dies aber durchaus vor (z.B. Übergang von „Neuerscheinung“ zu „normalem Video“ oder „Kindervideo“ zu „normalem Video“ und zurück).

Neue Klassenhierarchie – Zweiter Versuch

Einführung einer Klassenhierarchie für Preiskategorien:



`setPrice()` ändert die Kategorie – jederzeit!

Neue Klassen-Hierarchie `Price`

Die Berechnungen sind für jede Preiskategorie ausfaktorisiert:

```
1   abstract class Price {
2       public abstract double getCharge(int daysRented);
3
4       public int getFrequentRenterPoints(int daysRented) {
5           return 1;
6       }
7   }
8
9   class RegularPrice extends Price {
10      public double getCharge(int daysRented) {
11          double charge = 2.00;
12          if (daysRented > 2)
13              charge += (daysRented - 2) * 1.50;
14          return charge;
15      }
16  }
17
18  class NewReleasePrice extends Price {
19      public double getCharge(int daysRented) {
20          return daysRented * 3.00;
21      }
22
```



```
23      public int getFrequentRenterPoints(int daysRented) {
24          if (daysRented > 1)
25              return 2;
26          else
27              return super.getFrequentRenterPoints(daysRented);
28      }
29  }
30
31  class ChildrensPrice extends Price {
32      public double getCharge(int daysRented) {
33          double charge = 1.50;
34          if (daysRented > 3)
35              charge += (daysRented - 3) * 1.50;
36          return charge;
37      }
38  }
```

Neue Klasse Movies

Die Movie-Klasse delegiert die Berechnungen jetzt an den jeweiligen Preis (_price):

```
1   class Movie { // ...
2
3       private Price _price;
4
5       double getCharge(int daysRented)
6       {
7           return _price.getCharge(daysRented);
8       }
9
10      int getFrequentRenterPoints(int daysRented)
11      {
12          return _price.getFrequentRenterPoints(daysRented);
13      }
14
15      void setPrice(Price price)
16      {
17          _price = price;
18      }
19  };
20
```


Neue Klasse Movies (2)

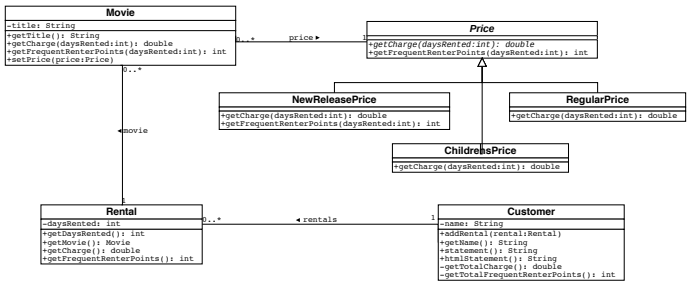
Die alte Schnittstelle getPriceCode wird hier nicht mehr unterstützt; neue Preismodelle sollten durch neue Price-Unterklassen realisiert werden.

Um getPriceCode dennoch weiter zu unterstützen, würde man

- die Preis-Codes wieder in die Klasse Movie einführen
- die Klasse Movie wieder mit einer Methode getPriceCode ausstatten, die – analog zu getCharge() – an die jeweilige Price-Subklasse delegiert würde
- die Klasse Movie mit einer Methode setPriceCode ausstatten, die anhand des Preiscodes einen passenden Preis erzeugt und setzt.

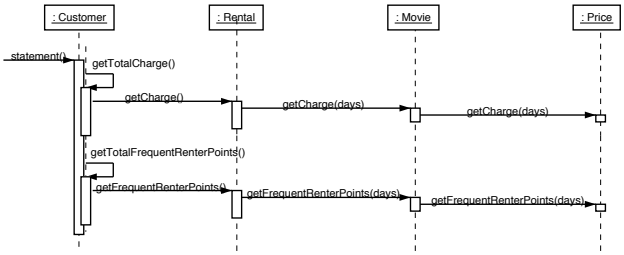
Alle Klassen im Überblick

So sieht die „ausfaktorisierte“ Klassenhierarchie aus:



Sequenzdiagramm

Dies ist der Aufruf der `statement()`-Methode:



Fazit

Der neue Entwurf

- hat besser verteilte Zuständigkeiten
- ist leichter zu warten
- kann einfacher in neuen Kontexten wiederverwendet werden.

Wann Refaktorisieren?

Extreme Programming ist eine Kultur des ständigen Refaktorisierens; sie hat den Begriff *code smell* für Code eingeführt, der refaktorisiert werden muss. Merkmale:

- Duplizierter Code
- Zu grosse Methoden
- Parallele Klassen-Hierarchien
- Zyklische Abhängigkeiten
- Prozeduraler Code als Objekt getarnt
- `switch`-Anweisungen
- Viele `private` Methoden
- Viele Instanz-Variablen
- Variable, die nur machmal gesetzt wird
- Unbenutzter Code

Siehe auch: `http://www.c2.com/cgi/wiki?CodeSmell`

Ein Refactoring-Katalog

Das Buch *Refactoring* von Fowler enthält einen Katalog von Refactoring-Verfahren – so etwa:

Bewegen von Eigenschaften zwischen Objekten

Move Method – wie beschrieben

Move Field – analog zu „Move Method“ wird ein Attribut verschoben

Extract Class – Einführen neuer Klasse aus bestehender

Ein Refactoring-Katalog (2)

Organisieren von Daten

Replace Magic Number with Symbolic Constant – wie beschrieben

Encapsulate Field – öffentliches Attribut einkapseln

Replace Data Value with Object – Datum durch Objekt ersetzen

Ein Refactoring-Katalog (3)

Vereinfachen von Methoden-Aufrufen

Add/Remove Parameter – Parameter einführen/entfernen

Introduce Parameter Object – Gruppe von Parametern durch Objekt ersetzen

Separate Query from Modifier – zustandserhaltende Methoden von zustandsverändernden Methoden trennen

Replace Error Code with Exception – Ausnahmebehandlung statt Fehlercode

Ein Refactoring-Katalog (4)

Umgang mit Vererbung

Replace Conditional with Polymorphism – wie beschrieben

Pull Up Method – Zusammenfassen von dupliziertem Code in Oberklasse

Pull Up Field – Zusammenfassen von dupliziertem Attribut in Oberklasse

…und viele weitere …

Refactoring bestehenden Codes

Refactoring kann nicht nur während des Entwurfs benutzt werden, sondern auch in der Implementierungs- und Wartungsphase, um bestehenden Code zu überarbeiten.

Damit wirkt Refactoring der sog. *Software-Entropie* entgegen – dem Verfall von Software-Strukturen aufgrund zuvieler Änderungen.

Änderungen während der Programmierung sind jedoch *gefährlich,* da bestehende Funktionalität gefährdet sein könnte („Never change a running system“).

Refactoring bestehenden Codes (2)

Voraussetzungen für das Refactoring bestehenden Codes sind:

Automatisierte Tests, die nach jeder Änderung ausgeführt werden

Entwurfswerkzeuge, die einfache Änderungen am Entwurf ermöglichen

Dokumentationswerkzeuge, mit denen die Dokumentation stets auf dem neuesten Stand gehalten werden kann

Versionsverwaltung, damit frühere Versionen erhalten bleiben

Refactoring bestehenden Codes (3)

Gute Kommunikation innerhalb des Teams, damit Mitglieder über Änderungen informiert werden

Systematisches Vorgehen – etwa indem existierende und bekannte Refaktorisierungen eingesetzt werden, statt unsystematisch „alles einmal zu überarbeiten".

Vorgehen in kleinen Schritten mit Tests nach jeder Überarbeitung.

Zur Sicherheit tragen auch spezielle *Refaktorisierungs-Werkzeuge* bei, die auf Knopfdruck bestimmte Refaktorisierungen durchführen – wobei sie (hoffentlich!) die Semantik des Programms erhalten (Beispiel: Eclipse)