Kapitel 11

Refactoring

11.1 Die Demeter-Regel (Lieberherr 89)

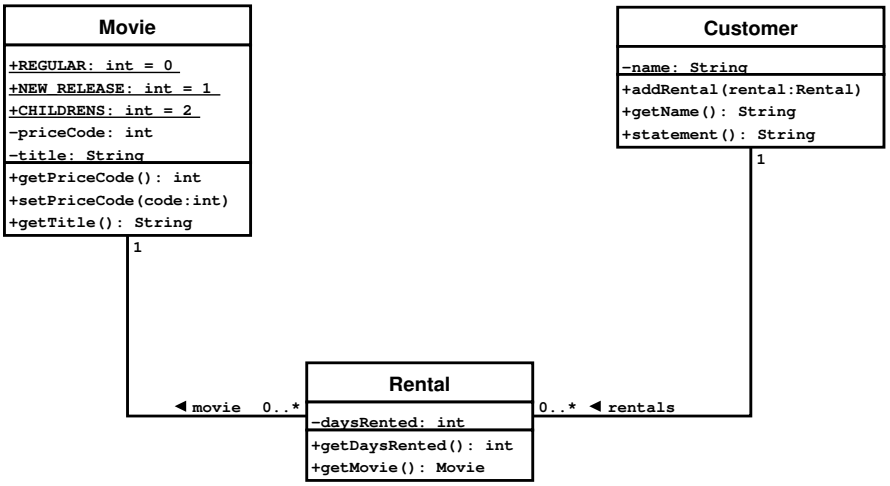
Klassen sollten nicht Wissen über die ganze Hierarchie, sondern nur über Nachbarklassen haben. Dies reduziert die Kopplung!
⇒ Demeter-Prinzip:

- Eine Klasse soll (außer *eigenen* Methoden) nur Methoden aufrufen, die zu Objekten gehören, die explizit als *Instanzvariablen* der Klasse vereinbart, selbst *erzeugt* oder als *Parameter* der aktuellen Methode übergeben sind.
- Verboten also: Aufruf von Methoden mit Bezugsobjekten, die erst durch einen Aufruf einer Methode einer anderen Klasse erzeugt wurden!

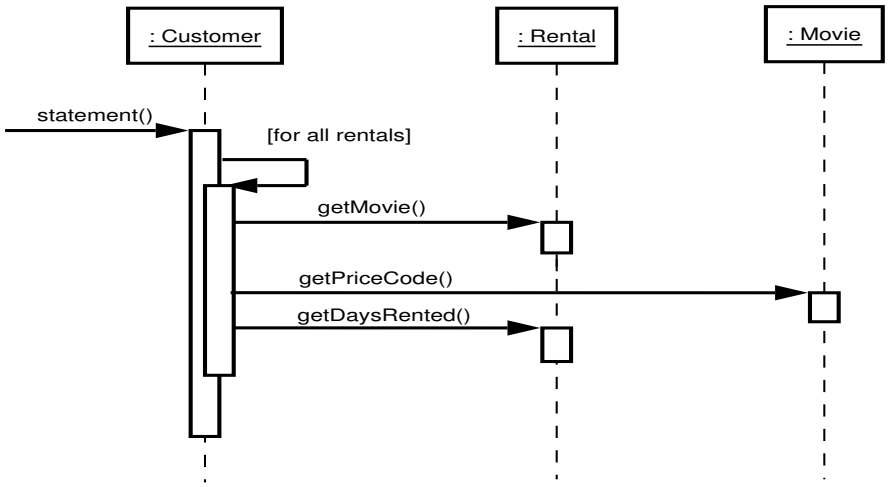
  Grund: dadurch entsteht Kopplung zwischen nicht benachbarten Klassen

Abhilfe bei Verstoß: Refaktorisierung! (s.u.)

Beispiel: Klassenhierarchie für Video-Verleih (Fowler 99)



Sequenzdiagramm für Ausdrucken der Monatsrechnung eines Kunden:



⇒ `getMovie` aus `Rental` liefert Objekt der Klasse `Movie` auf dem dann in Klasse `Customer` die Methode `getPriceCode` aufgerufen wird

⇒ Verstoß gegen Demeter-Regel erzeugt *Kopplung* zwischen `Movie` und `Customer`: Änderungen an `Movie` beeinflussen nicht nur `Rental`, sondern auch `Customer`

11.2 Refactoring im Überblick

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

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

Es gibt keine allgemeine Methode des Refactorings.

Vielmehr gibt es einen *Katalog* von Methoden, ähnlich wie bei *Entwurfsmustern*

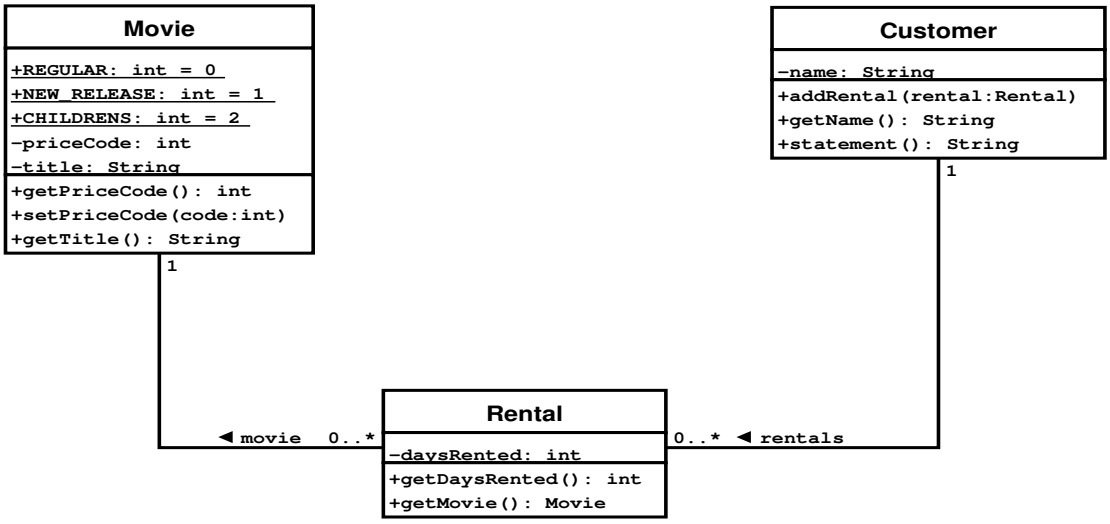
11.3 Beispiel: Der Videoverleih[1]

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 verlängerte Ausleihen von normalen und Kinder-Videos (nicht für *Neuerscheinungen*)
- Es gibt *Bonuspunkte* für Stammkunden (das Ausleihen von Neuerscheinungen bringt Extra-Punkte)

[1] Dieses Beispiel und die nachfolgenden Refaktorisierungen sind entnommen aus: Martin Fowler, *Refactoring Improving the design of existing code*, Addison-Wesley, 1999. Nehmen Sie die englische Originalfassung; die deutsche „Wort-für-Wort“-Übersetzung ist schlecht lesbar.

Ausgangssituation:



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

Die gesamte Funktionalität steckt im Erzeugen der Kundenrechnung der Methode `statement` der Klasse `Customer`.

Erzeugen der Kundenrechnung: `Customer.statement()`

```
public String statement() {
    double totalAmount = 0.00;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\
     n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0.00;
        Rental each = (Rental) rentals.nextElement();

        // Kosten pro Video berechnen
        switch (each.getMovie().getPriceCode()) {
```

```
        case Movie.REGULAR:
            thisAmount += 2.00;
            if (each.getDaysRented() > 2)
                thisAmount += (each.getDaysRented() - 2)
                 * 1.50;
            break;

        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3.00;
            break;

        case Movie.CHILDRENS:
            thisAmount += 1.50;
            if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented() - 3)
                 * 1.50;
            break;
        }

        // Bonuspunkte berechnen
        frequentRenterPoints++;

        if ((each.getMovie().getPriceCode() == Movie.
         NEW_RELEASE) &&
            each.getDaysRented() > 1)
            frequentRenterPoints++;

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

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

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 wir das Refactoring-Verfahren „Extract Method“ ein.
„Extract Method“ ist eine der verbreitetsten 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:*

```
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, deren Werte explizit in die neue Methode übertragen werden müssen (und ggf. wieder zurück).

Zusätzlich Umbenennung von `each` in `aRental`:

```
public String statement() {
    double totalAmount = 0.00;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " +
        getName() + "\n";

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        double thisAmount = amountFor(each); // NEU

        // Bonuspunkte berechnen
        frequentRenterPoints++;

        if ((each.getMovie().getPriceCode() ==
             Movie.NEW_RELEASE) &&
            each.getDaysRented() > 1)
            frequentRenterPoints++;

        // Zeile berechnen
        result += "\t" + each.getMovie().getTitle() +
            "\t" + String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }

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

```
public double amountFor(Rental aRental) { // NEU
    double thisAmount = 0.00;

    switch (aRental.getMovie().getPriceCode()) {
    case Movie.REGULAR:
        thisAmount += 2.00;
        if (aRental.getDaysRented() > 2)
            thisAmount += (aRental.getDaysRented() - 2)
                * 1.50;
        break;

    case Movie.NEW_RELEASE:
        thisAmount += aRental.getDaysRented() * 3.00;
        break;

    case Movie.CHILDRENS:
        thisAmount += 1.50;
        if (aRental.getDaysRented() > 3)
            thisAmount += (aRental.getDaysRented() - 3)
                * 1.50;
        break;
    }

    return thisAmount;
}
```

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” 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.*

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:

```
class Rental {
    // ...
    public double getCharge() {        // NEU
        double charge = 0.00;

        switch (getMovie().getPriceCode()) {
        case Movie.REGULAR:
            charge += 2.00;
            if (getDaysRented() > 2)
                charge += (getDaysRented() - 2) * 1.50;
            break;

        case Movie.NEW_RELEASE:
            charge += getDaysRented() * 3.00;
            break;

        case Movie.CHILDRENS:
            charge += 1.50;
            if (getDaysRented() > 3)
                charge += (getDaysRented() - 3) * 1.50;
            break;
        }

        return charge;
    }
}
```

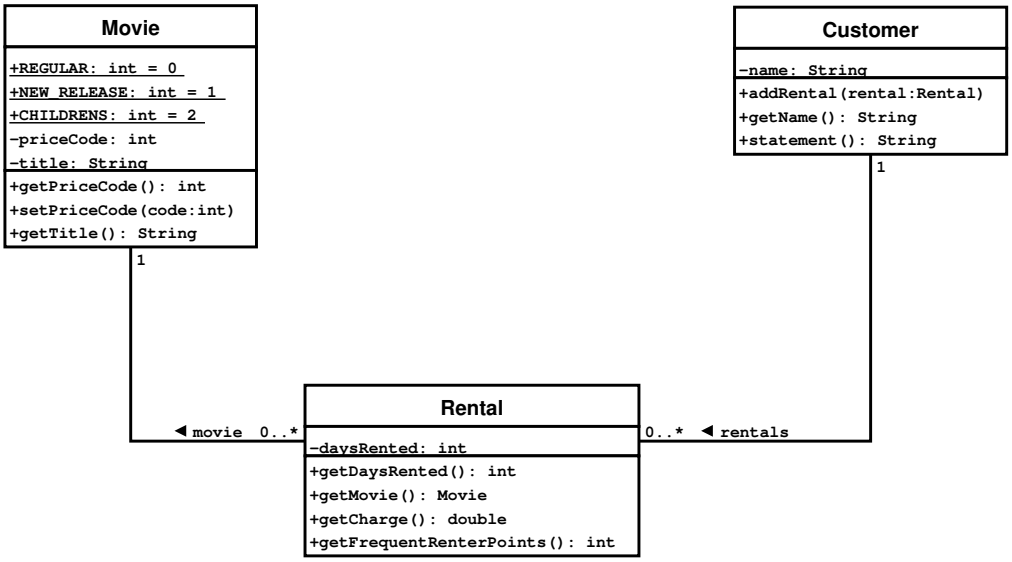
Die umgearbeitete `Customer`-Methode `amountFor()` delegiert nun die Berechnung an `getCharge()`:

```
class Customer {
    // ...
    public double amountFor(Rental aRental) { // NEU
        return aRental.getCharge();
    }
}
```

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()`.

Klassen nach dem Bewegen von Methoden:

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



Abfrage-Methoden einführen (“Replace Temp with Query”)

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“ hat die allgemeine Form:

> 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.*

Beispiel:

```
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 neue Methoden ein:

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

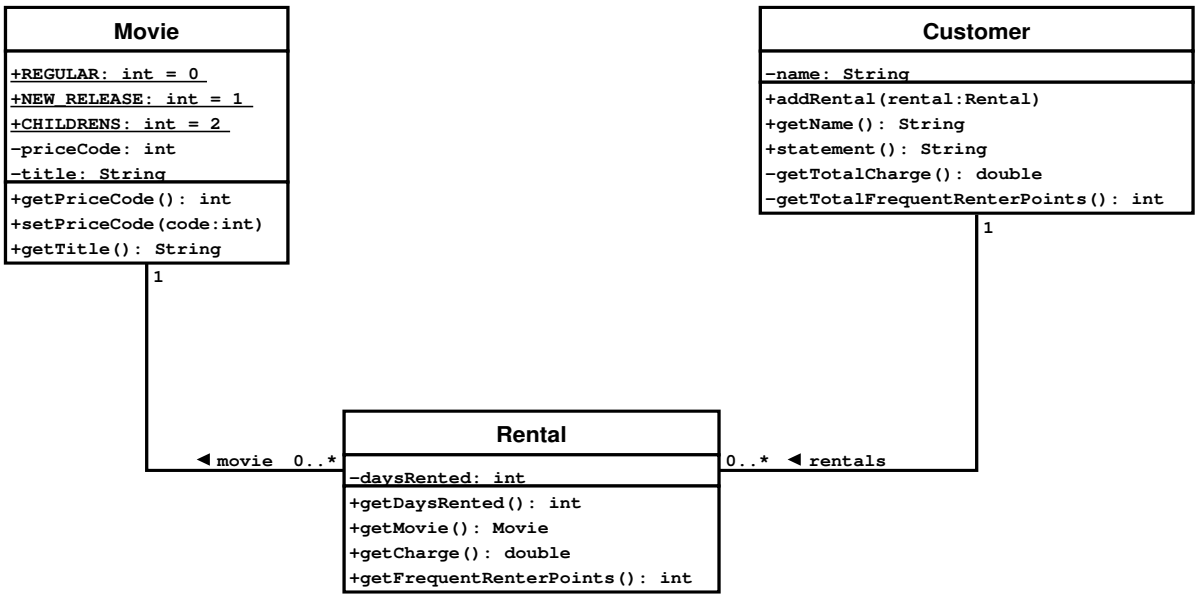
```
public String statement() {
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " +
        getName() + "\n";

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        result += "\t" + each.getMovie().getTitle() +
          "\t" + String.valueOf(each.getCharge()) + "\n";
    }

    result += "Amount owed is " +
```

```
            String.valueOf(getTotalCharge()) + "\n";
        result += "You earned " +
            String.valueOf(getTotalFrequentRenterPoints()) +
            " frequent renter points";
        return result;
    }
    private double getTotalCharge() { // NEU
        double charge = 0.00;
        Enumeration rentals = _rentals.getElements();
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            charge += each.getCharge();
        }
        return charge;
    }
    private int getTotalFrequentRenterPoints() { // NEU
        int points = 0;
        Enumeration rentals = _rentals.getElements();
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            points += each.getFrequentRenterPoints();
        }
        return points;
    }
```

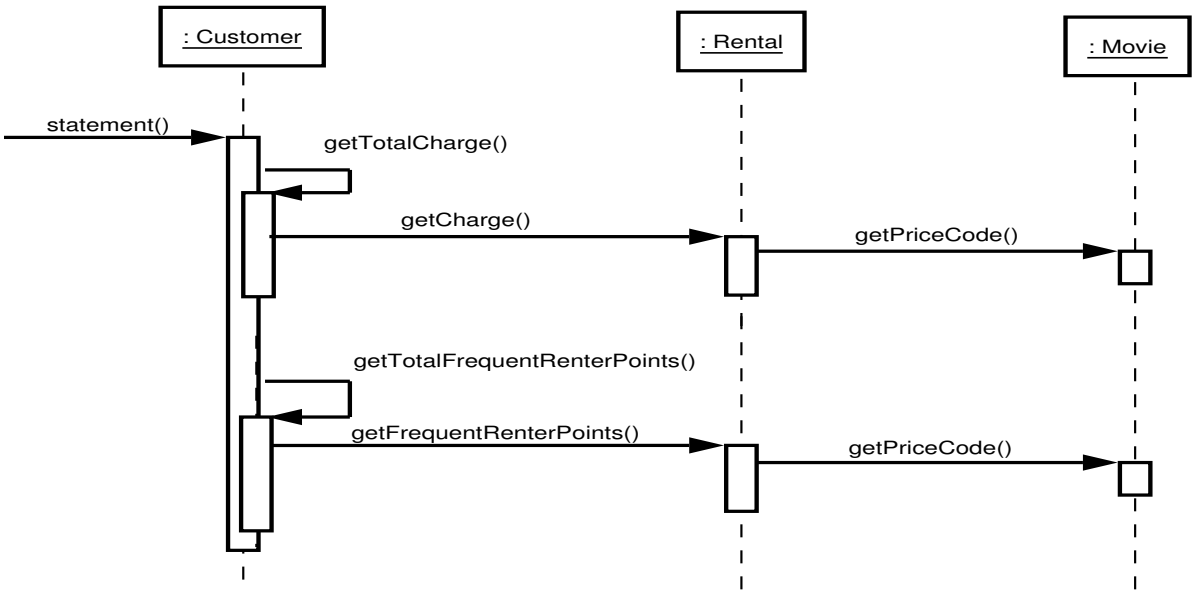
Die `statement()`-Methode ist schon deutlich kürzer geworden!

Klassen nach dem Einführen von Queries:

Neue private Methoden `getTotalCharge` und `getTotalFrequentRenterPoints`:



Sequenzdiagramm nach dem Einführen von Queries



Einführen einer HTML-Variante:

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

Kein Problem mehr, alternative Rechnungs-Formate auszugeben.

Rechnung in HTML-Format drucken:

```
public String htmlStatement() {
    Enumeration rentals = _rentals.elements();
    String result = "<H1>Rental Record for <EM>" +
        getName() + "</EM></H1>\n";

    result += "<UL>";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

        result += "<LI> " + each.getMovie().getTitle() +
          ": " + String.valueOf(each.getCharge()) + "\n";
    }
    result += "</UL>";

    result += "Amount owed is <EM>" +
        String.valueOf(getTotalCharge()) + "</EM><P>\n";
    result += "You earned <EM>" +
        String.valueOf(getTotalFrequentRenterPoints()) +
        "</EM> frequent renter points<P>";
    return result;
}
```

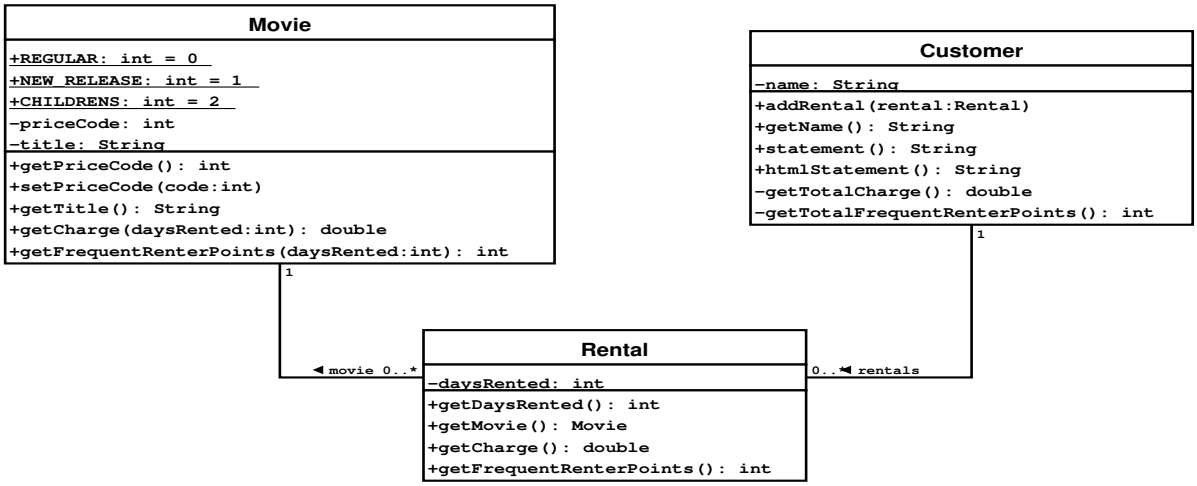
Weiteres Verschieben von Methoden

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

```
class Rental {
    // ...
    public double getCharge() {        // NEU
        double charge = 0.00;

        switch (getMovie().getPriceCode()) {
        case Movie.REGULAR:
            charge += 2.00;
            if (getDaysRented() > 2)
                charge += (getDaysRented() - 2) * 1.50;
            break;

        case Movie.NEW_RELEASE:
            charge += getDaysRented() * 3.00;
            break;

        case Movie.CHILDRENS:
            charge += 1.50;
            if (getDaysRented() > 3)
                charge += (getDaysRented() - 3) * 1.50;
            break;
        }

        return charge;
    }
}
```

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

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



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

```
class Movie {
    // ...
    public double getCharge(int daysRented) {  // NEU
        double charge = 0.00;

        switch (getPriceCode()) {
        case Movie.REGULAR:
            charge += 2.00;
            if (daysRented > 2)
                charge += (daysRented - 2) * 1.50;
            break;
```

```
        case Movie.NEW_RELEASE:
            charge += daysRented * 3.00;
            break;

        case Movie.CHILDRENS:
            charge += 1.50;
            if (daysRented > 3)
                charge += (daysRented - 3) * 1.50;
            break;
        }
        return charge;
    }

    public int getFrequentRenterPoints(int daysRented) {
            // NEU
        if ((getPriceCode() == Movie.NEW_RELEASE) &&
            daysRented > 1) return 2;
        else return 1;
    }
}
```

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

```
class Rental {
    // ...
    public double getCharge() {       // NEU
        return getMovie().getCharge(_daysRented);
    }
    public int getFrequentRenterPoints() {   // NEU
        return getMovie().getFrequentRenterPoints(
          _daysRented);
    }
}
```

Fallunterscheidungen durch Polymorphie ersetzen ("Replace Conditional Logic with Polymorphism")

Fallunterscheidungen innerhalb einer Klasse können fast immer durch Einführen von Unterklassen ersetzt werden.

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

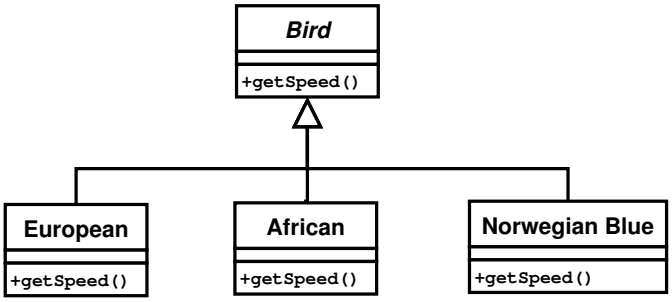
„Replace Conditional Logic with Polymorphism" hat die allgemeine Form:

> 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.*

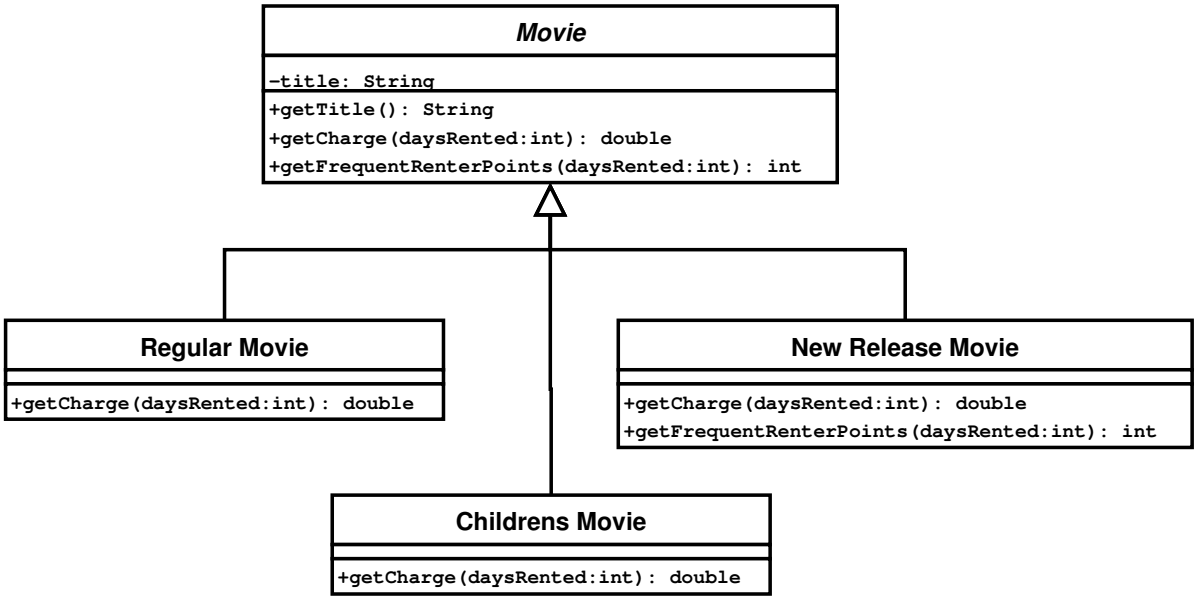
Beispiel:

```
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:



Neue Eigenschaften:

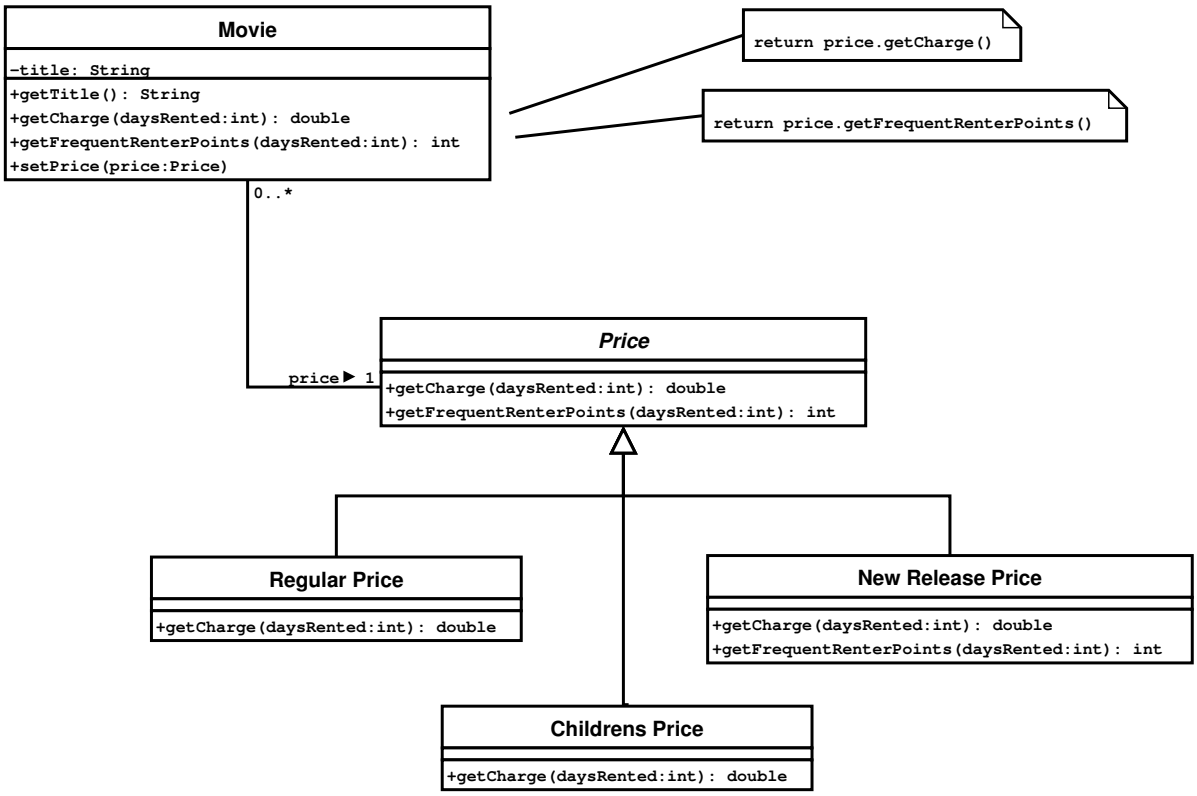
- Die Berechnung der Kosten wird an die Unterklassen abgegeben (abstrakte Methode `getCharge`)
- Die Berechnung der Bonuspunkte steckt in der Oberklasse, kann aber von Unterklassen überladen werden (Methode `getFrequentRenterPoints()`)

Problem dieser Hierarchie: Beim Erzeugen eines `Movie`-Objekts muß die Klasse bekannt sein; während ihrer Lebensdauer können Objekte nicht mehr einer 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:

Lösung: Einführung einer separaten Klassenhierarchie für den Preis:



Vorteil: Mit `setPrice()` kann die Kategorie jederzeit geändert werden!

Neue Klassen-Hierarchie `Price`

Die Berechnungen sind für jede Preiskategorie ausfaktorisiert:

```
abstract class Price {
    public abstract double getCharge(int daysRented);

    public int getFrequentRenterPoints(int daysRented) {
        return 1;
    }
}

class RegularPrice extends Price {
    public double getCharge(int daysRented) {
        double charge = 2.00;
        if (daysRented > 2)
            charge += (daysRented - 2) * 1.50;
        return charge;
    }
}

class NewReleasePrice extends Price {
    public double getCharge(int daysRented) {
        return daysRented * 3.00;
    }

    public int getFrequentRenterPoints(int daysRented) {
        if (daysRented > 1)
            return 2;
        else
            return super.getFrequentRenterPoints(
              daysRented);
    }
}
```

```
class ChildrensPrice extends Price {
    public double getCharge(int daysRented) {
        double charge = 1.50;
        if (daysRented > 3)
            charge += (daysRented - 3) * 1.50;
        return charge;
    }
}
```

Neue Klasse `Movies`

`Movie`-Klasse delegiert Berechnungen jetzt an jeweiligen Preis (`_price`):

```
class Movie { // ...

    private Price price;

    double getCharge(int daysRented) {
        return price.getCharge(daysRented);
    }

    int getFrequentRenterPoints(int daysRented) {
        return price.getFrequentRenterPoints(daysRented);
    }

    void setPrice(Price newPrice) {
        price = newPrice;
    }
}
```

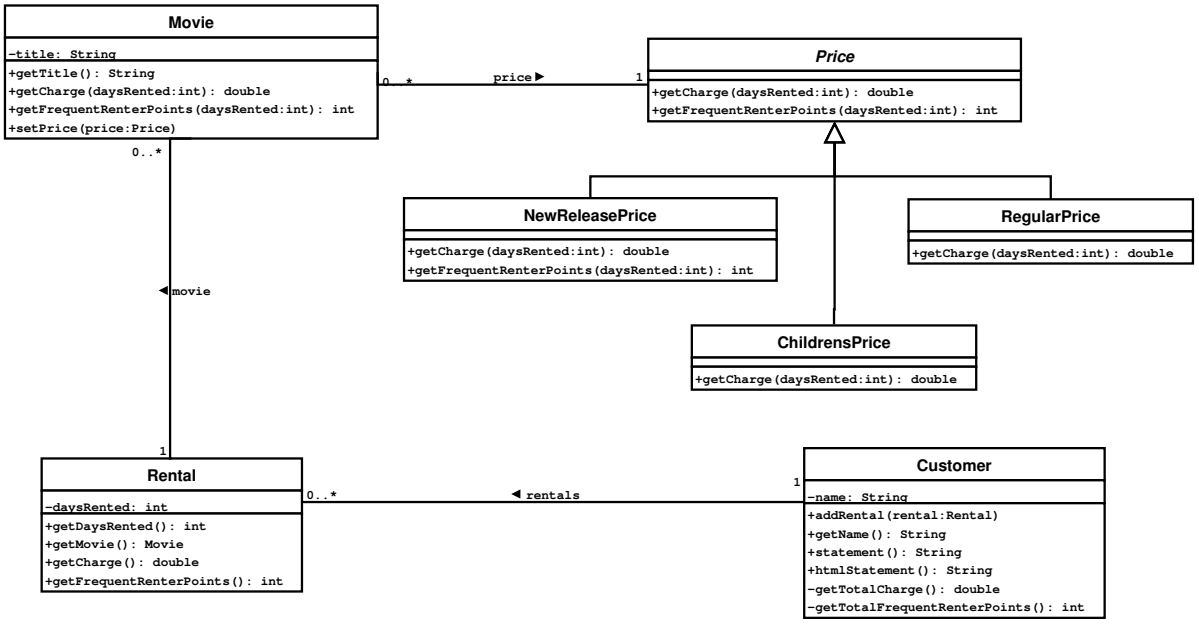
Die alte Schnittstelle `getPriceCode` wird hier nicht mehr unterstützt; neue Preismodelle sollten durch neue `Price`-Unterklassen realisiert werden.

Um `getPriceCode` dennoch weiter zu unterstützen, würde man

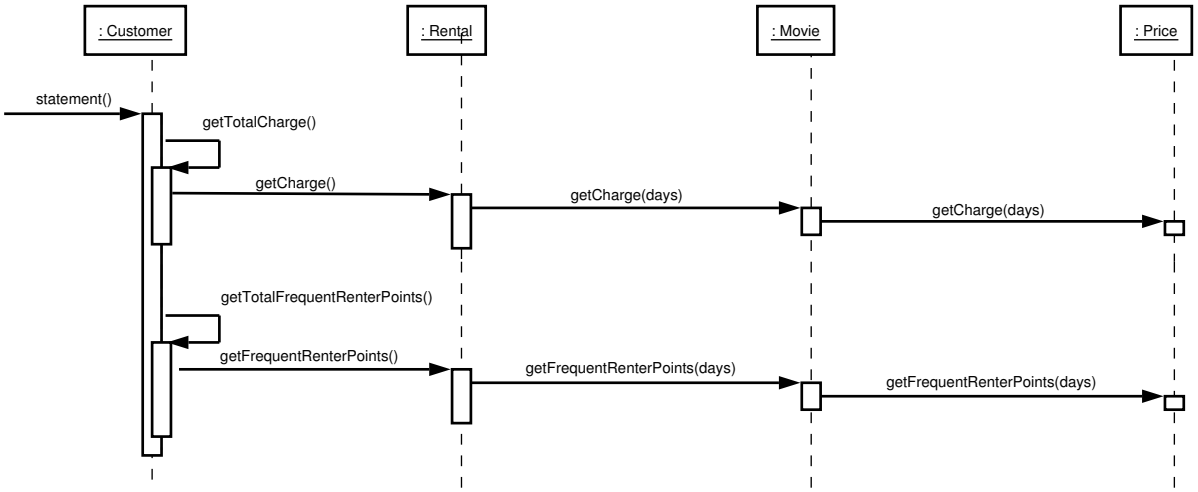
- die Preis-Codes wieder in die Klasse `Movie` einführen
- die Klasse `Movie` wieder mit einer Methode `getPriceCode` ausstatten, die analog zu `getCharge()` an die jeweilige `Price`-Subklasse delegiert würde
- die Klasse `Movie` mit einer Methode `setPriceCode` ausstatten, die anhand des Preiscodes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

Übung: Erstellen Sie entsprechenden Java-Code!

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.

11.4 Ein Refactoring-Katalog

Das Buch *Refactoring* von Fowler enthält einen Katalog von Refactoring-Verfahren. Hier ein Auszug:

- 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

- Organisieren von Daten:

  Replace Magic Number with Symbolic Constant Konstanten statt festen Werten

  Encapsulate Field öffentliches Attribut einkapseln

  Replace Data Value with Object Datum durch Objekt ersetzen

- 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

- 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 ...

11.5 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“).

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

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.