universitätbonn ROOTS

Kapitel 11
Refactoring

Stand: 25.1.2011

"Refactoring"

- Einstiegs-Beispiel

- Refactoring: Schritt für Schritt
  - Beispiel: „Extract Method“

- Indikationen für Refactoring ("Bad Smells in Code")

- Refactoring-Überblick

Was ist überhaupt “Refactoring“?

Refactoring (noun):
a change made to the internal structure of software
to make it easier to understand and cheaper to modify
without changing its observable behavior.

Refactor (verb):
to restructure software by applying a series of refactorings.

- Definition
  - Systematische Umstrukturierung des Codes ohne das Verhalten nach außen zu ändern
- Nutzen
  - bessere Lesbarkeit, Verständlichkeit
  - besseres Design
  - bessere Wartbarkeit und Wiederverwendbarkeit

Warum “Refactoring“ anwenden?

- “Refactoring“ verbessert das Design der Software
  - Oft geänderte Software verliert mit der Zeit ihre Struktur.
  - Redundanter Code erfordert vielfache Änderungen.
  - Extraktion gemeinsamer Teile ermöglicht jede Änderung an genau einer Stelle durchzuführen.
- “Refactoring“ macht die Software verständlicher
  - Verständlichkeit für den Programmierer, der vielleicht später am Programm Erweiterungen vornimmt
  - Wenn man sich in ein Programm einarbeitet, kann man es besser verstehen, wenn man es restrukturiert.
- “Refactoring“ hilft Bugs zu finden
  - Im Prozess des “Refactorings erwirbt man ein tiefes Verständnis für den Code. Man kann Bugs leichter finden.
- “Refactoring“ macht das Programmieren schneller
  - Wenn der Code ein gutes Design hat, gut verständlich ist und daher auch wenig Bugs hat, kann man schneller programmieren.

universitätbonn ROOTS

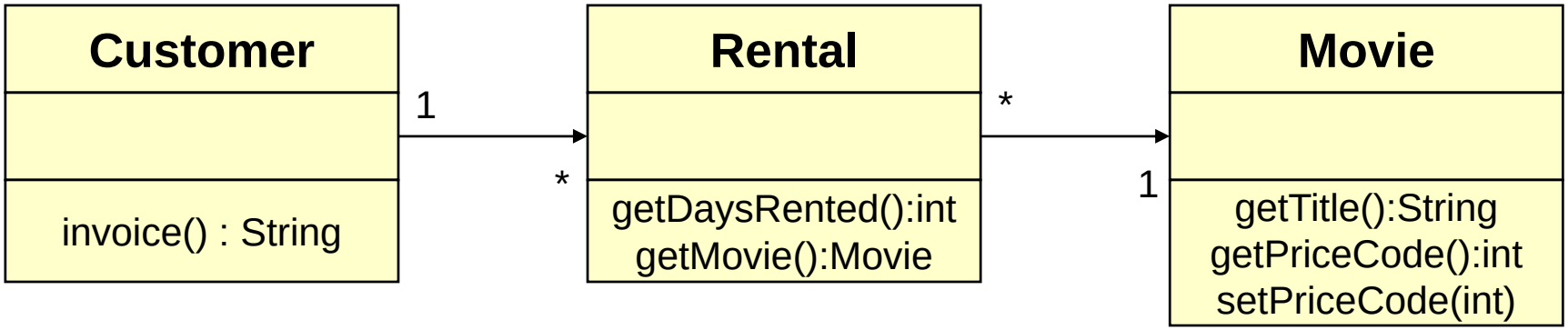
Refactoring: ein Anwendungsbeispiel

Anwendungsbeispiel: Videofilm-Verleih

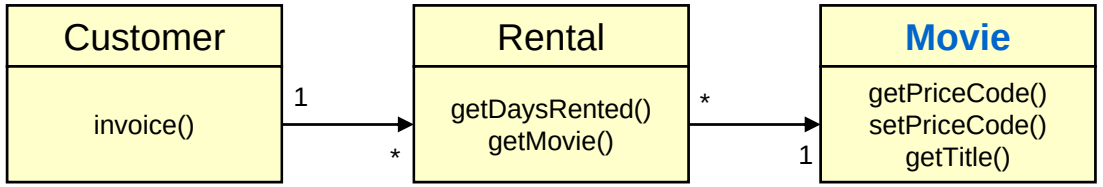
- **Videos** können im Laufe der Zeit zu verschiedenen **Preiskategorien** gehören (Normal, Jugend, Neuerscheinung).
- Jede Preiskategorie beinhaltet eine andere **Preisberechnung**.
- Jede Ausleihe führt zur **Gutschrift von Bonuspunkten**, die am Jahresende abgerechnet werden.
- Der Umfang der **Gutschrift** hängt ebenfalls von der **Preiskategorie** ab.
- Für jeden Kunden soll es möglich sein, eine **Rechnung** für die ausgeliehenen Videos auszudrucken
  - Titel und Preis eines jeden ausgeliehenen Videos
  - Summe der Ausleihgebühren
  - Summe der Bonuspunkte

Erster Versuch

- Jeder **Film (Movie)** kennt seinen Titel und seine Preiskategorie
- Jede **Ausleihe (Rental)** kennt den ausgeliehenen Film und die Leihdauer
- Jeder **Kunde (Customer)** kennt die Menge seiner aktuellen Ausleihen
- ... kann den Text der dafür fälligen Rechnung selbst ermitteln (Methode invoice())
- Instanz-Variablen sind privat, auf ihre Werte wird via Methoden zugegriffen

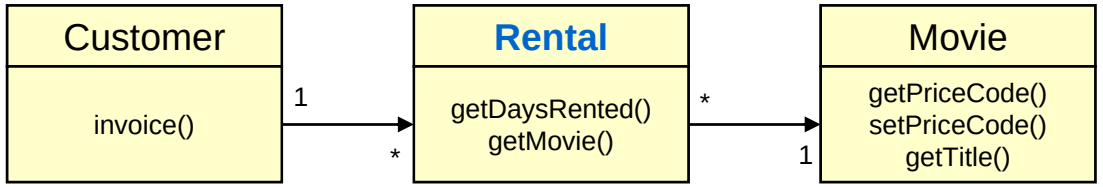


Movie



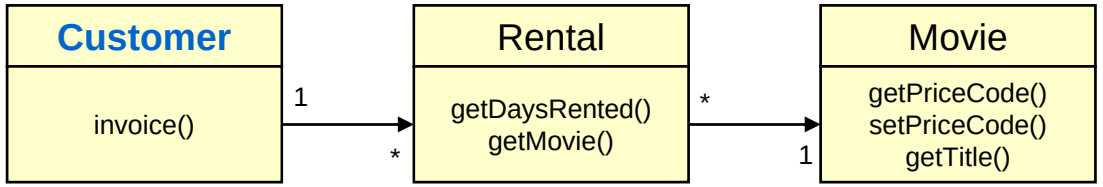

```
public class Movie {

   public static final int CHILDRENS = 2;
   public static final int REGULAR = 0;
   public static final int NEW_RELEASE =1;

   private String _title;
   private int _priceCode;

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

   public int getPriceCode() {
      return _priceCode;
   }

   public void setPriceCode(int arg) {
      _priceCode = arg;
   }

   public String getTitle() {
      return _title;
   }
}
```

Rental




```
class Rental {

   private Movie _movie;
   private int _daysRented;

   public Rental(Movie movie, int daysRented) {
      _movie = movie;
      _daysRented = daysRented;
   }

   public int getDaysRented() {
      return _daysRented;
   }

   public Movie getMovie() {
      return _movie;
   }
}
```

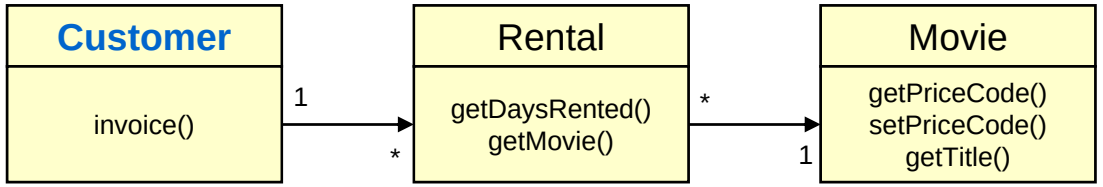
Customer




```
class Customer {

   private String _name;
   private Vector _rentals = new Vector();

   public Customer(String name) {
      _name = name;
   }

   public void addRental(Rental arg) {
      _rentals.addElement(arg);
   }

   public String getName() {
      return _name;
   }
```

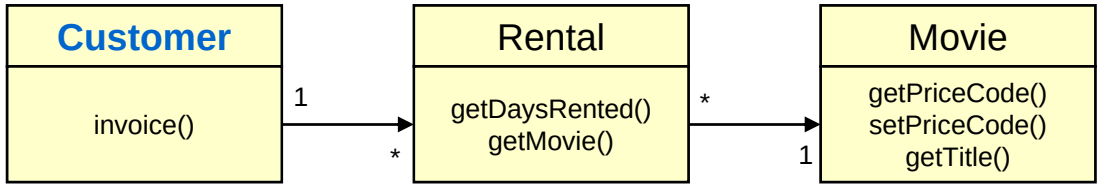
Auch wenn der Typ von _rentals sich mal ändert, bleibt für Clients von Customer das Hinzufügen eines neuen Ausleihobjekts gleich.

Die invoice()-Methode (Teil 1)



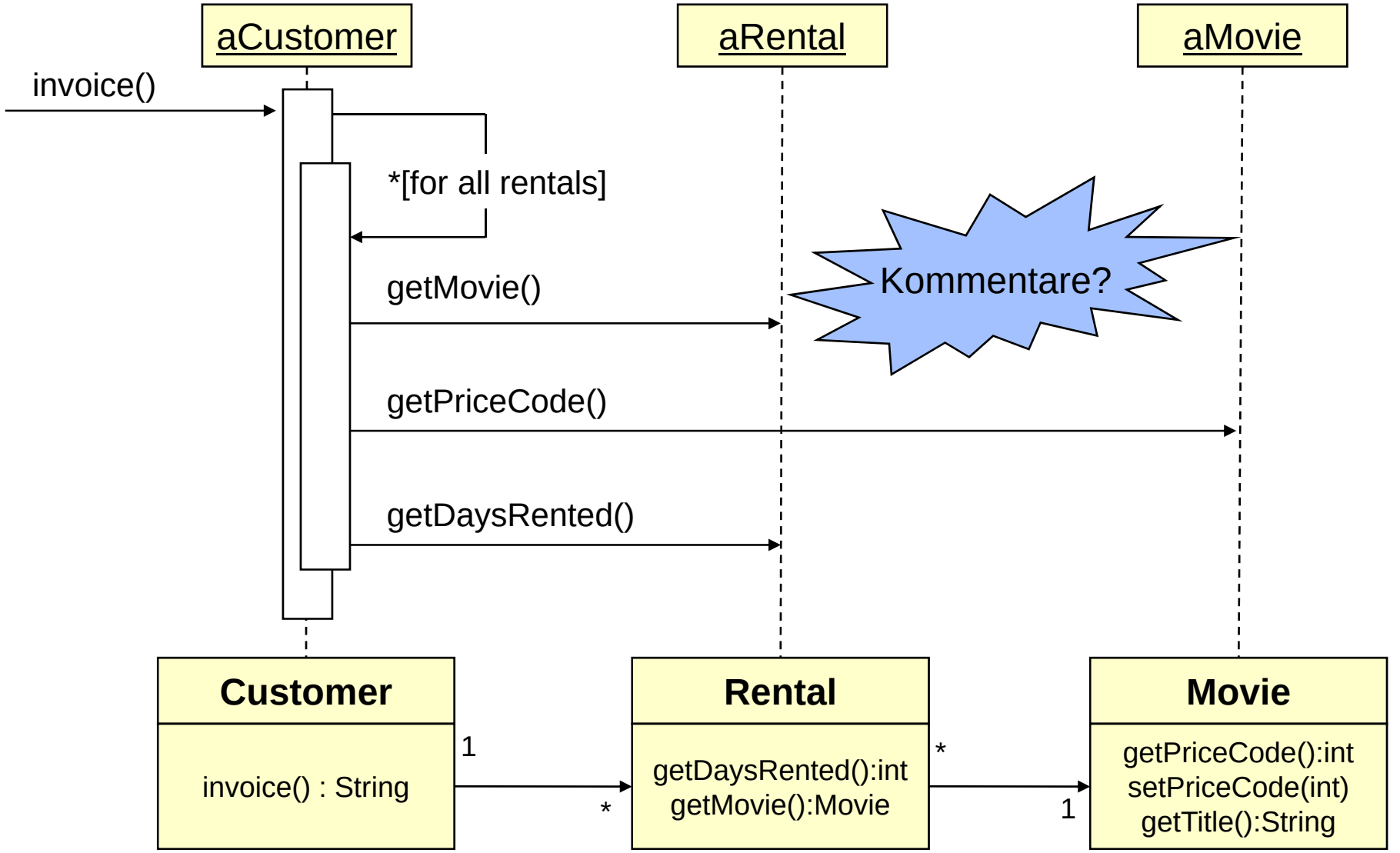

```
public String invoice() {
   double totalAmount = 0;
   int bonusPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = “Rental Record for “+getName()+ “\n“;

   while (rentals.hasMoreElements()) {
      double thisAmount = 0;
      Rental each = (Rental) rentals.nextElement();

      // 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.CHILDRENS:
            thisAmount += 1,5;
            if (each.getDaysRented() > 3)
               thisAmount += (each.getDaysRented()-3)*1.5;
            break;
      }
```

Die invoice()-Methode (Teil 2)




```
      // add frequent renter points
      bonusPoints ++;

      // add bonus for a two day new release rental
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
         each.getDaysRented()>1) bonusPoints ++;

      // 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) + "\n";
   result += "You earned "+ String.valueOf(bonusPoints) +
             " frequent renter points";
   return result;
}
```

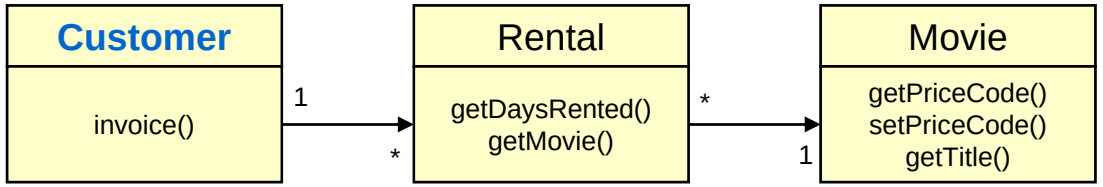
Interaktionen der invoice() Methode



universitätbonn ROOTS

Umverteilung der Verantwortlichkeiten

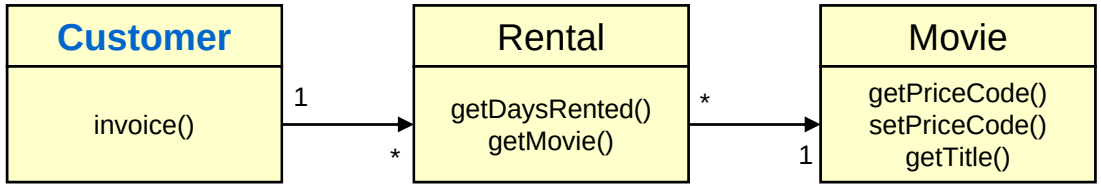
Schritte zur Besserung (1)



1) Invoice()-Methode aufteilen
   - ◆ Berechnung von Beitrags- und Bonuspunkten pro Ausleihe extrahieren
   - ➢ leichter verständlich
   - ➢ Teile eventuell in anderen Kontexten wieder verwendbar

2) Extrahierte Methoden in passendere Klassen verlagern
   - ◆ Methoden näher an "ihre" Daten (Beitrags- und Bonuspunktberechnung hängen von Preis-Code ab)
   - ➢ weniger Abhängigkeiten zwischen Klassen

Schritte zur Besserung (2)



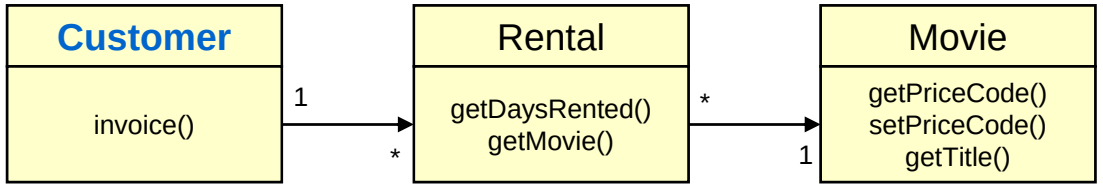
3) Invoice()-Methode weiter aufteilen
   - Berechnung von Gesamt-Beitrags- und Bonuspunkten für alle Ausleihen des Kunden aus invoice() extrahieren
   - Entkopplung / Trennen von Belangen („separation of concerns“)
     - Bessere Verständlichkeit, Änderbarkeit, Wiederverwendbarkeit

4) Voraussetzung für 3: Temporäre Variablen eliminieren
   - Auslagerung von Methoden erleichtern (z.B. Summierung der Bonuspunkte)
   - Vereinfachung

5) Ersetzung von Fallunterscheidungen (switch-statement) durch Nachrichten
   - kleinere, klarere Methoden
   - Erweiterbarkeit um zusätzliche Fälle ohne Änderung der clients einer Klasse
   - z.B. zusätzliche Preiskategorien einführen ohne Änderung von invoice()

Extrahieren der Betragsberechnung

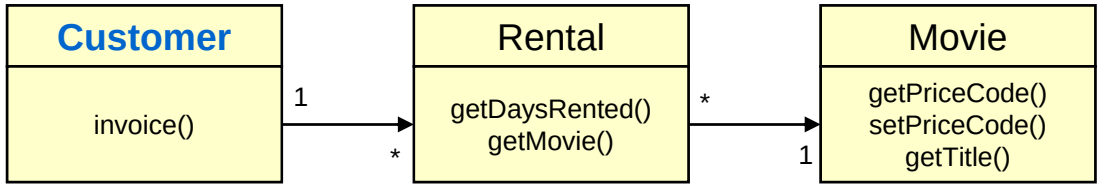
→ amountFor(Rental)




```
public String invoice() {
   double totalAmount = 0;
   int bonusPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = “Rental Record for “+getName()+ “\n“;

   while (rentals.hasMoreElements()) {
      double thisAmount = 0;
      Rental each = (Rental) rentals.nextElement();

      // determine amounts for each rental
      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;
      }
```

Extrahieren der Betragsberechnung

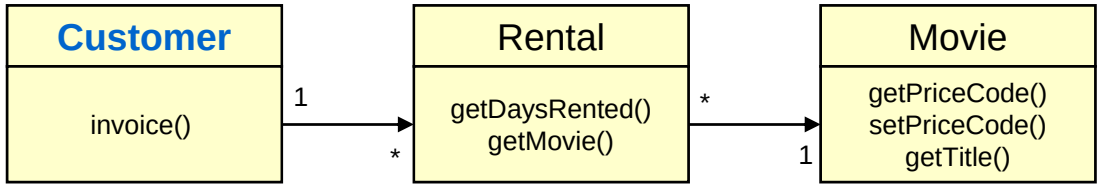
→ amountFor(Rental)




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

   while (rentals.hasMoreElements()) {

      Rental each = (Rental) rentals.nextElement();

      double thisAmount =
        amountFor(each);
```

```
private double amountFor(Rental each) {
   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;
}
```

Ändern lokaler Variablenamen: Vorher

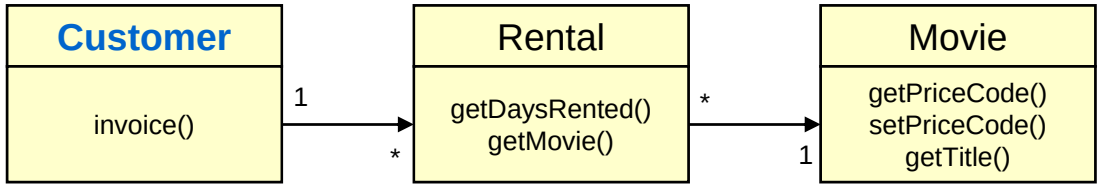


Im neuen Kontext sinnvolle Namen:

- `each` → `aRental`
- `thisAmount` → `result`

```
class Customer { ...
  private double amountFor(Rental each   ) {
    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 ;
  }
}
```

Ändern lokaler Variablennamen: Nachher

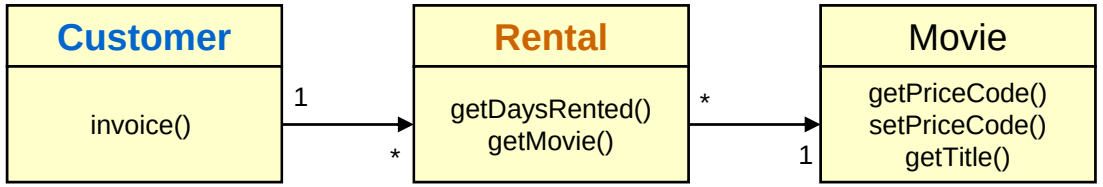


Im neuen Kontext sinnvolle Namen:

- `each` → `aRental`
- `thisAmount` → `result`

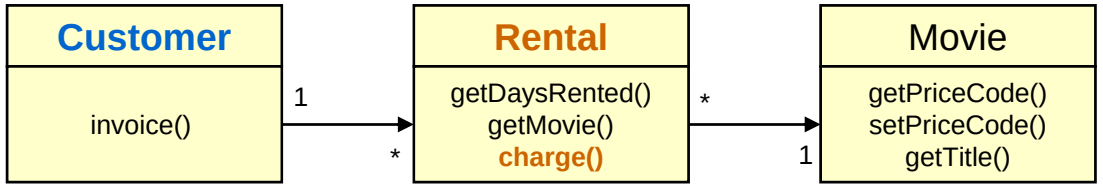
```
class Customer { ...
  private double amountFor(Rental aRental) {
    double result     = 0;
    switch (aRental.getMovie().getPriceCode()) {
       case Movie.REGULAR:
         result     += 2;
         if (aRental.getDaysRented() > 2)
             result    += (aRental.getDaysRented()-2)*1.5;
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

Beitragsberechnung nach "Rental" verlagern: Vorher




```
class Customer { ...
  private double amountFor(Rental aRental) {
    double 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.CHILDRENS:
         result     += 1.5;
         if (aRental.getDaysRented() > 3)
            result     += (aRental.getDaysRented()-3)*1.5;
         break;
    }
    return result;
  }
}
```

Beitragsberechnung nach "Rental" verlagern: Nachher



Private
Weiterleitungsmethode kann eliminiert werden

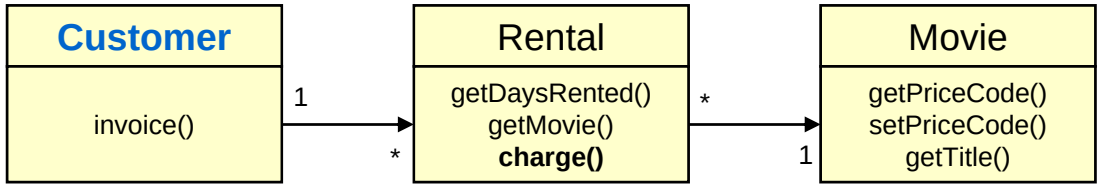
```
class Customer { ...
  private double amountFor(Rental aRental) {
    return aRental.charge();
  }
}
```

Umbenennungen:

- `amountFor(...)` → `charge()`
- `aRental` → `this` (weggelassen)

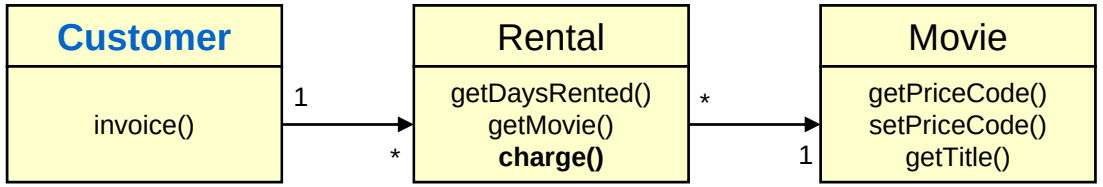
```
class Rental   { ...
  private double   charge(              ) {
    double result     = 0;
    switch (        getMovie().getPriceCode()) {
       case Movie.REGULAR:
         result     += 2;
         if (        getDaysRented() > 2)
             result    += (        getDaysRented()-2)*1.5;
 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

Aufrufstelle anpassen: Vorher



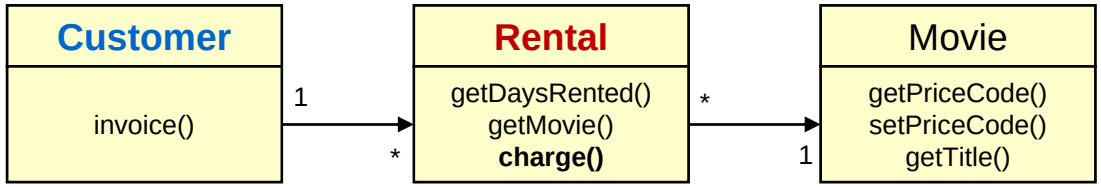

```
public String invoice() {
   double totalAmount = 0;
   int bonusPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = "Rental Record for "+getName()+ "\n";
   while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();

      double thisAmount= amountFor(each);

      // add frequent renter points
      bonusPoints ++;
      // add bonus for a two day new release rental
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
         each.getDaysRented()>1) bonusPoints ++;

      // 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) + "\n";
   result += "You earned "+ String.valueOf(bonusPoints) +
             " frequent renter points";
   return result;
}
```

Aufrufstelle anpassen: Nachher



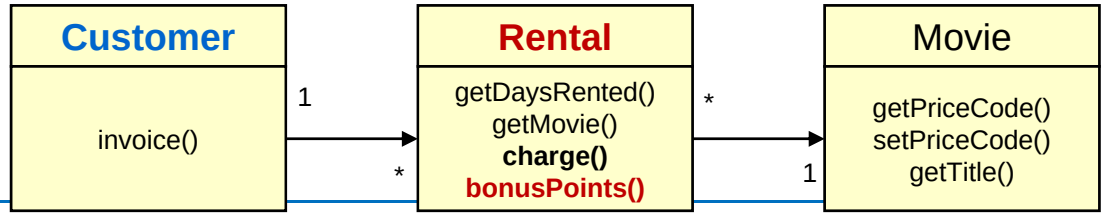

```
public String invoice() {
   double totalAmount = 0;
   int bonusPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = "Rental Record for "+getName()+ "\n";
   while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();

      double thisAmount= each.charge();

      // add frequent renter points
      bonusPoints ++;
      // add bonus for a two day new release rental
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
         each.getDaysRented()>1) bonusPoints ++;

      // 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) + "\n";
   result += "You earned "+ String.valueOf(bonusPoints) +
             " frequent renter points";
   return result;
}
```

Bonuspunkt-Berechnung extrahieren




```
public String invoice() {
   double totalAmount = 0;
   int bonusPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = "Rental Record for "+getName()+ "\n";
   while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();

      double thisAmount= each.charge();

      // add frequent renter points
      bonusPoints ++;
      // add bonus for a two day new release rental
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
         each.getDaysRented()>1) bonusPoints++;

      // 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) + "\n";
   result += "You earned "+ String.valueOf(bonusPoints) +
             " frequent renter points";
   return result;
}
```

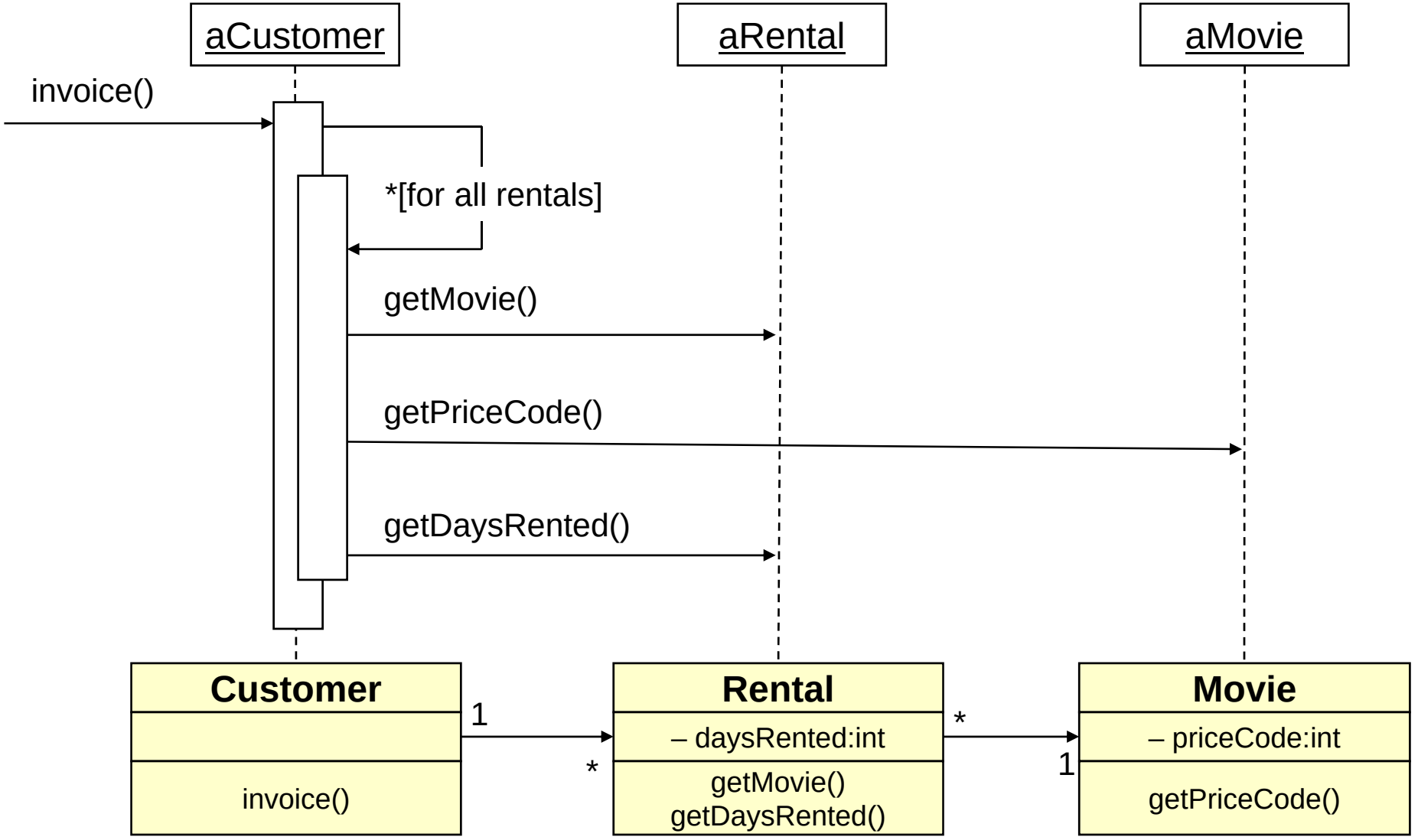
Bonuspunkt-Berechnung extrahieren



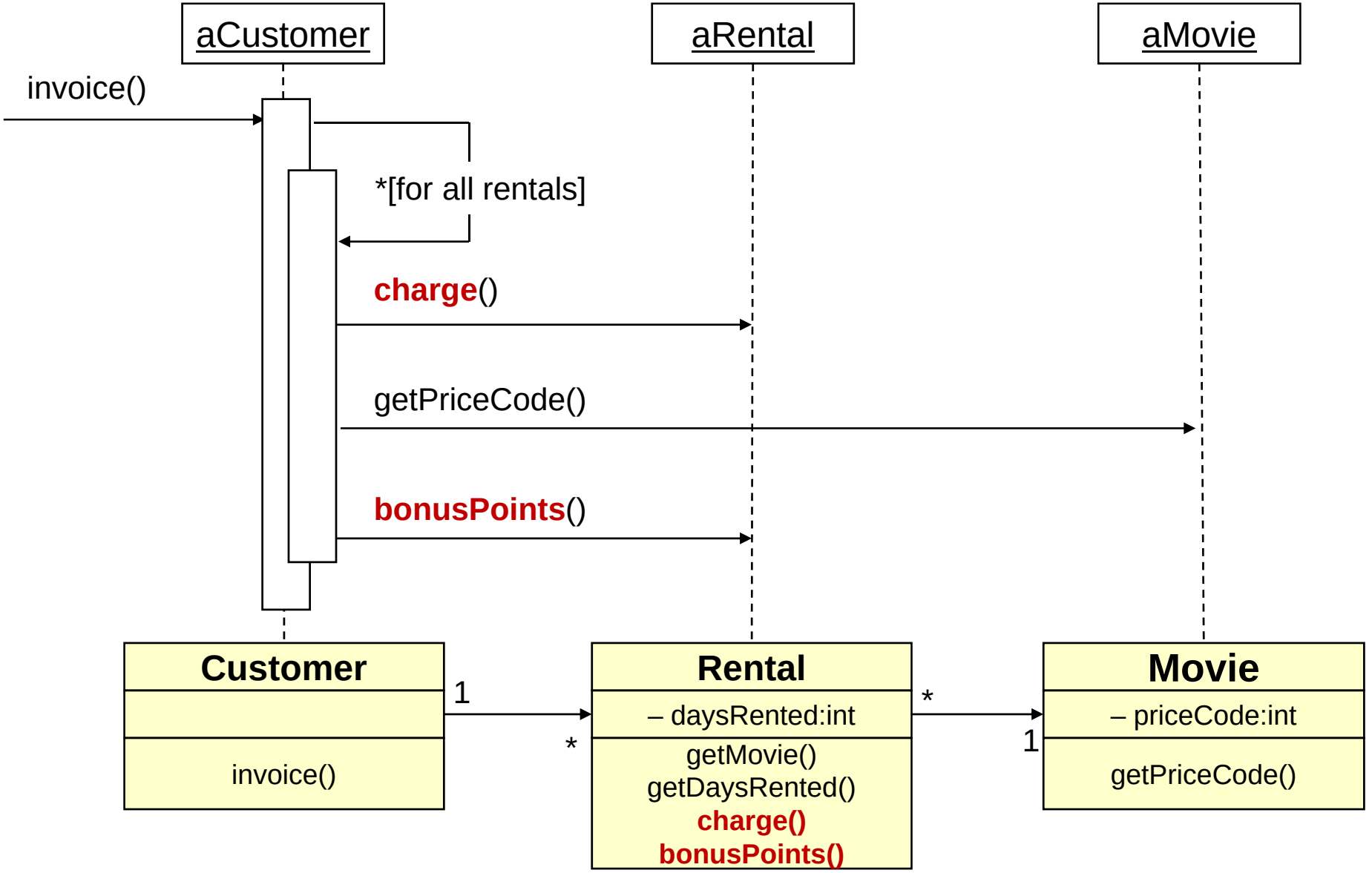

```
public String invoice() {
   double totalAmount = 0;
   int bonusPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = "Rental Record for "+getName()+ "\n";
   while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();

      double thisAmount= each.charge();


      bonusPoints += each.bonusPoints();

      // show figure
      result += "\t"
                Str
      totalAmount +=
   }
   // add footer line
   result += "Amount owed is "+ String.valueOf(totalAmount) + "\n";
   result += "You earned "+ String.valueOf(bonusPoints) +
             " frequent renter points";
   return result;
}
```

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

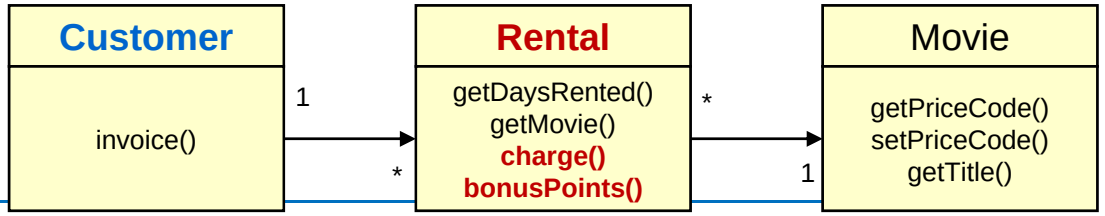
Kosten- und Bonuspunkt-Berechnung extrahieren: Vorher



Kosten- und Bonuspunkt-Berechnung extrahieren: Nachher



Schritte zur Besserung

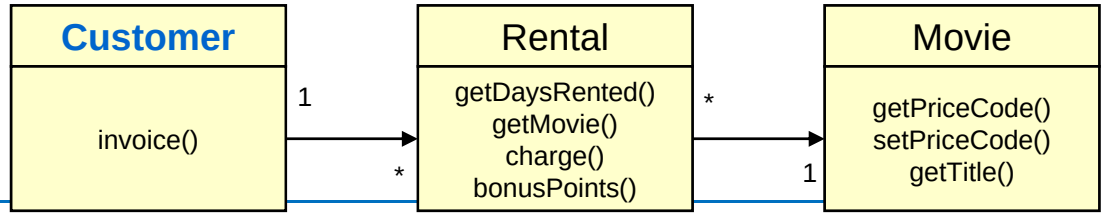


- ✓ invoice()-Methode aufteilen
- ✓ Teilmethoden in passendere Klassen verlagern

- Temporäre Variablen eliminieren
  - Vereinfachung
  - Auslagerung von Methoden erleichtern (z.B. Summierung der Bonuspunkte)

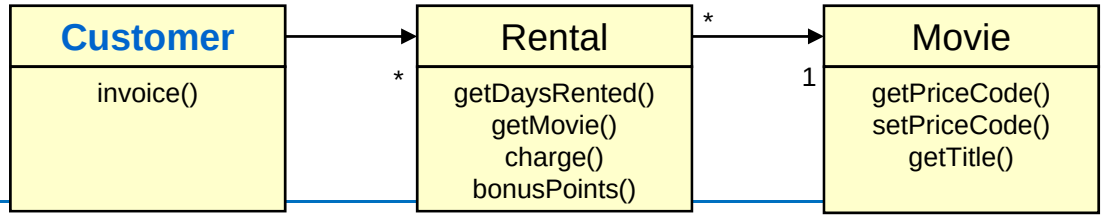
- Ersetzung von Fallunterscheidungen durch Nachrichten

Temporäre Variablen eliminieren: thisAmount




```
public String invoice() {
   double totalAmount = 0;
   int bonusPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = "Rental Record for "+getName()+ "\n";
   while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();

      double thisAmount = each.charge();

      bonusPoints += each.bonusPoints();

      // 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) + "\n";
   result += "You earned "+ String.valueOf(bonusPoints) +
             " frequent renter points";
   return result;
}
```

Temporäre Variablen eliminieren: thisAmount




```
public String invoice() {
   double totalAmount = 0;
   int bonusPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = "Rental Record for "+getName()+ "\n";
   while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();



      bonusPoints += each.bonusPoints();

      // show figures for this rental
      result += "\t" + each.getMovie().getTitle()+ "\t" +
                String.valueOf(each.charge())+ "\n";
      totalAmount += each.charge();
   }
   // add footer lines
   result += "Amount owed is "+ String.valueOf(totalAmount) + "\n";
   result += "You earned "+ String.valueOf(bonusPoints) +
             " frequent renter points";
   return result;
}
```

Eine lokale Variable wird eliminiert indem alle Verwendungen der Variablen durch den ihr zugewiesenen Ausdruck ersetzt werden.

Diesen Vorgang nennt man „inlining“.

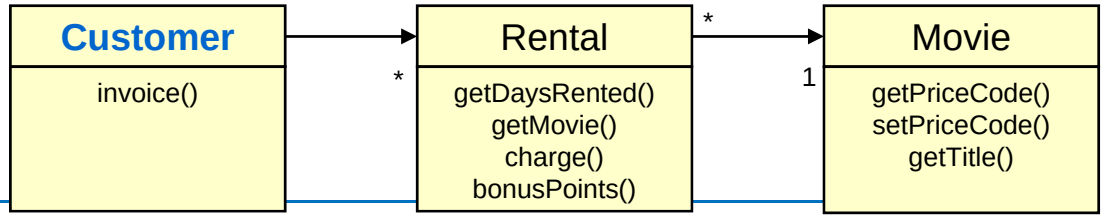
Achtung: „Inlining“ ist nur dann verhaltenserhaltend, falls

- der Variablen nur ein mal etwas zugewiesen wird und
- der zugewiesene Ausdruck keine Seiteneffekte hat!

Konsequenzen des Inlining

- Bessere Lesbarkeit (Kein suchen: „Wo kommt dieser Wert her?“)
- Evtl. redundante Berechnungen

Schleife splitten und temporäre Variablen eliminieren



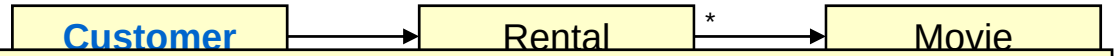

```
public String invoice() {
   double totalAmount = 0;
   int bonusPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = "Rental Record for "+getName()+ "\n";
   while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();




      bonusPoints += each.bonusPoints();

      // show figures for this rental
      result += "\t" + each.getMovie().getTitle()+ "\t" +
                String.valueOf(each.charge())+ "\n";
      totalAmount += each.charge();
   }
   // add footer lines
   result += "Amount owed is "+ String.valueOf(totalAmount) + "\n";
   result += "You earned "+ String.valueOf(bonusPoints) +
             " frequent renter points";
   return result;
}
```

Wir wollen aus dieser Schleife zwei getrennte machen, die sich jeweils nur um ein Anliegen kümmern: Bonuspunkte ODER Rechnungsbetrag.

→ Klarere Verantwortlichkeiten

→ Bessere Wiederverwendbarkeit (jede Funktionalität einzeln)

Schleife splitten und temporäre Variablen eliminieren




```
public String invoice() {


   Enumeration rentals = _rentals.
   String result = "Rental Record
   while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();




      // show figures for this rental
      result += "\t" + each.getMovie().getTitle()+ "\t" +
                String.valueOf(each.charge())+ "\n";

   }
   // add footer lines
   result += "Amount owed is "+ String.valueOf(totalCharge()) + "\n";
   result += "You earned "+ String.valueOf(totalBonusPoints()) +
             " frequent renter poi
   return result;
}
```

```
private double totalCharge() {
    double result = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()){
        Rental each = (Rental) rentals.nextElement();
        result += each.charge();
    }
    return result;
}
```

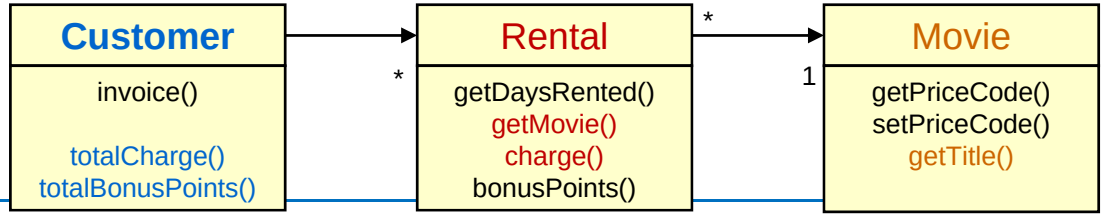
Ersatz für totalAmount

Ersatz für bonusPoints

```
private double totalBonusPoints() {
    double result = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()){
        Rental each = (Rental) rentals.nextElement();
        result += each.bonusPoints();
    }
    return result;
}
```

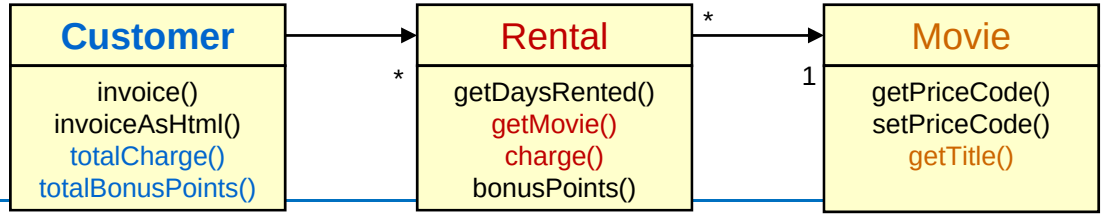
© 2000-2010 Dr. G. Kniesel

Endzustand der invoice()-Methode




```
public String invoice() {
   // add header line
   String result = "Rental Record for " +
                   getName() + "\n";

   // show figures for each rental
   Enumeration rentals = _rentals.elements();
   while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += "\t" +
                each.getMovie().getTitle() +
                "\t" +
                String.valueOf(each.charge()) +
                "\n";
   }

   // add footer lines
   result += "Amount owed is " +
             String.valueOf(totalCharge()) +
             "\n";
   result += "You earned " +
             String.valueOf(totalBonusPoints()) +
             " frequent renter points";

   return result;
}
```

Einfügen von invoiceAsHtml()

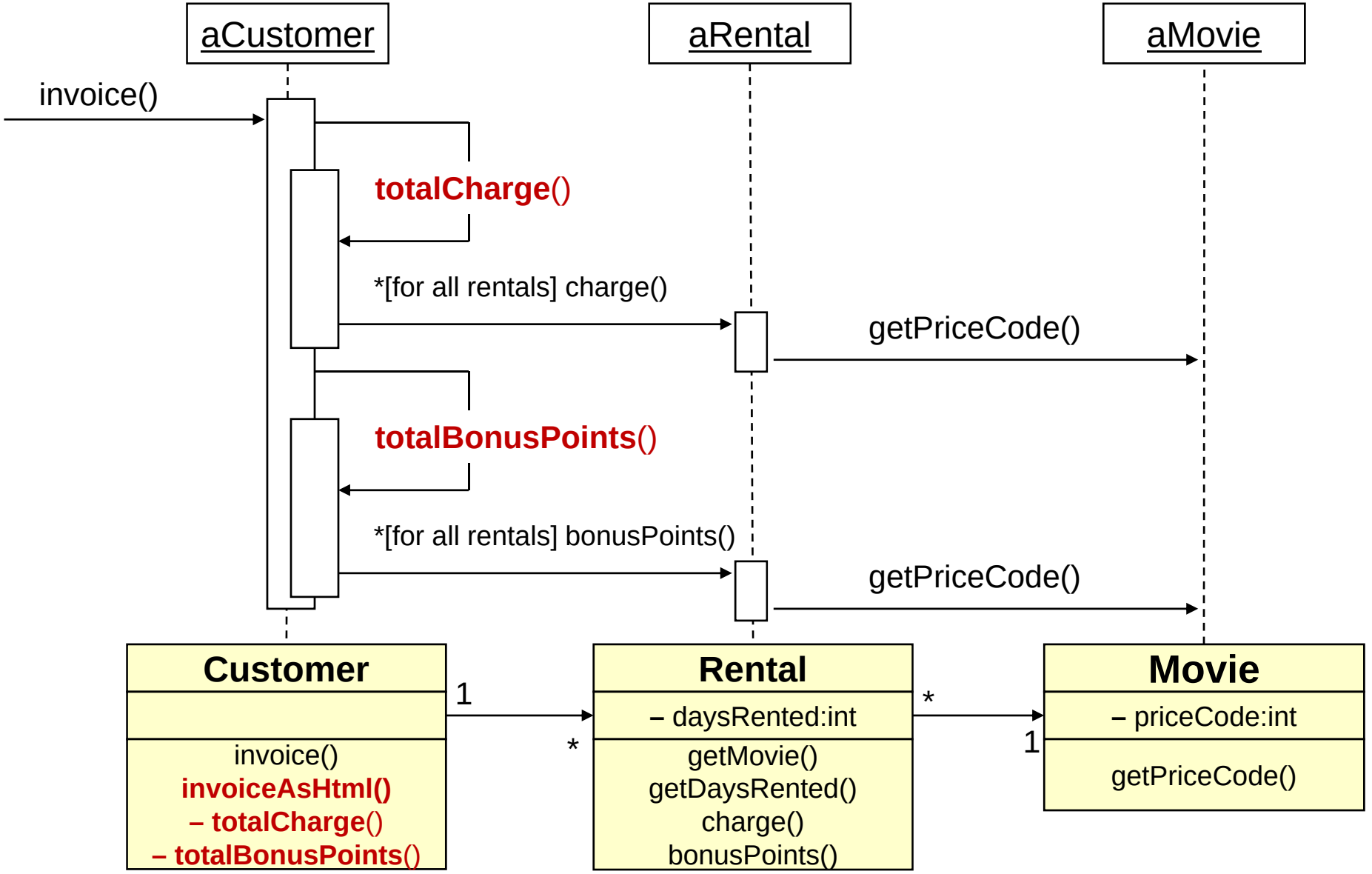



```
public String invoiceAsHtml() {
   // add header lines
   String result = "<H1>Rentals for <EM>" +
                   getName() + "</EM></H1><P>\n";

   // show figures for each rental
   Enumeration rentals = _rentals.elements();
   while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result +=
                each.getMovie().getTitle() +
                ": " +
                String.valueOf(each.charge()) +
                "<BR>\n";
   }

   // add footer lines
   result += "<P> You owe <EM>" +
             String.valueOf(totalCharge()) +
             "</EM></P>\n";
   result += "On this rental you earned <EM>" +
             String.valueOf(totalBonusPoints()) +
             "</EM> frequent renter points<P>";

   return result;
}
```

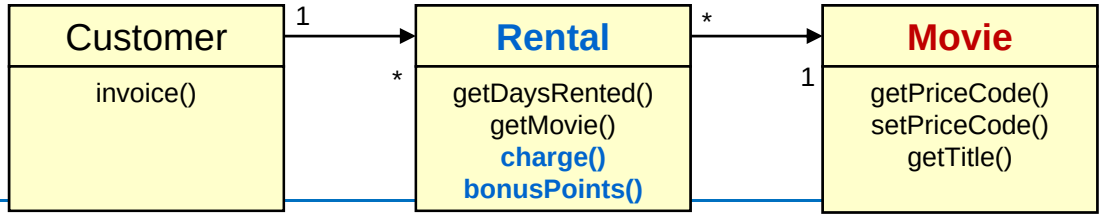
Die neue Methode reduziert sich auf Details der HTML-Formatierung.

Die eigentlichen Berechnungen werden wiederverwendet.

Temporäre Variablen eliminieren: Nachher



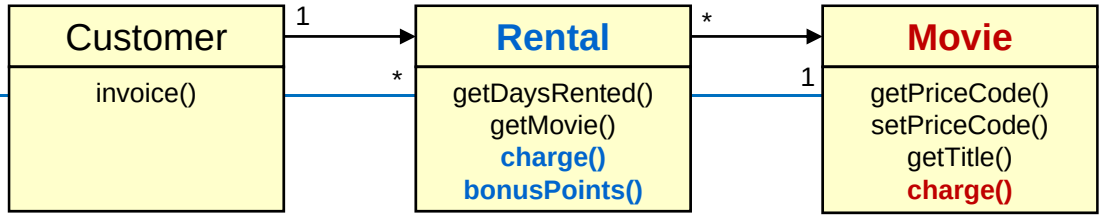
Schritte zur Besserung



- ✓ invoice()-Methode aufteilen
- ✓ Teilmethoden in passendere Klassen verlagern
- ✓ Temporäre Variablen eliminieren

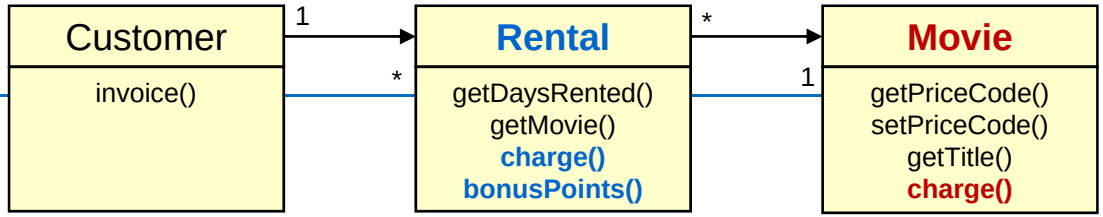
- Vorbereitung für Einführung neuer Preiskategorien: Ersetzung der preiscodeabhängigen Fallunterscheidung durch Nachrichten
  - ➔ Verlagern der preiscodeabhängigen Methoden nach Movie (zum Preiscode)
  - ➔ Anwenden des State Patterns (jeder Zustand implementiert Methoden die preiscodeabhängige Berechnungen durchführen auf seine eigene Art)

charge()-Methode aus Rental nach Movie verlagern: Vorher




```
class Rental  ...
 public double charge() {
   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.CHILDRENS:
         result += 1.5;
         if (getDaysRented() > 3)
            result += (getDaysRented()-3)*1.5;
         break;
   }
}
```

charge()-Methode aus Rental nach Movie verlagern: Nachher

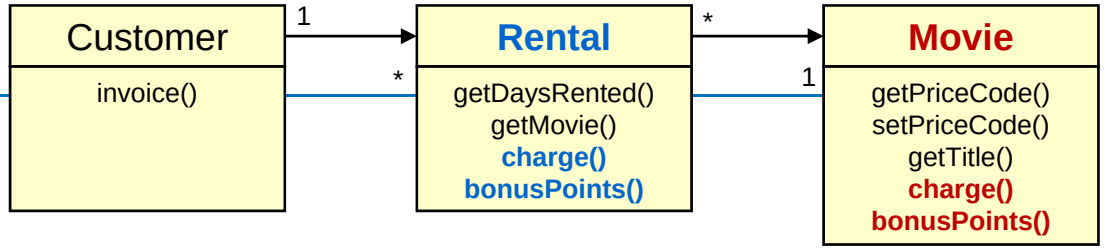



```
class Movie   ...
 public double charge(int daysRented) {
   double result = 0;
   switch (           getPriceCode()) {
      case Movie.REGULAR:
          result += 2;
         if (   daysRented   > 2)
             result += (   daysRented  -2)*1.5;
         break;
      case Movie.NEW_RELEASE: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

```
class Rental ...
    public double charge() {
        return _movie.charge(_daysRented);
    }
```

Die in Rental-Instanzen lokal verfügbare Information (daysRented) wird nun als Parameter übergeben

bonusPoints()-Methode aus Rental nach Movie verlagern

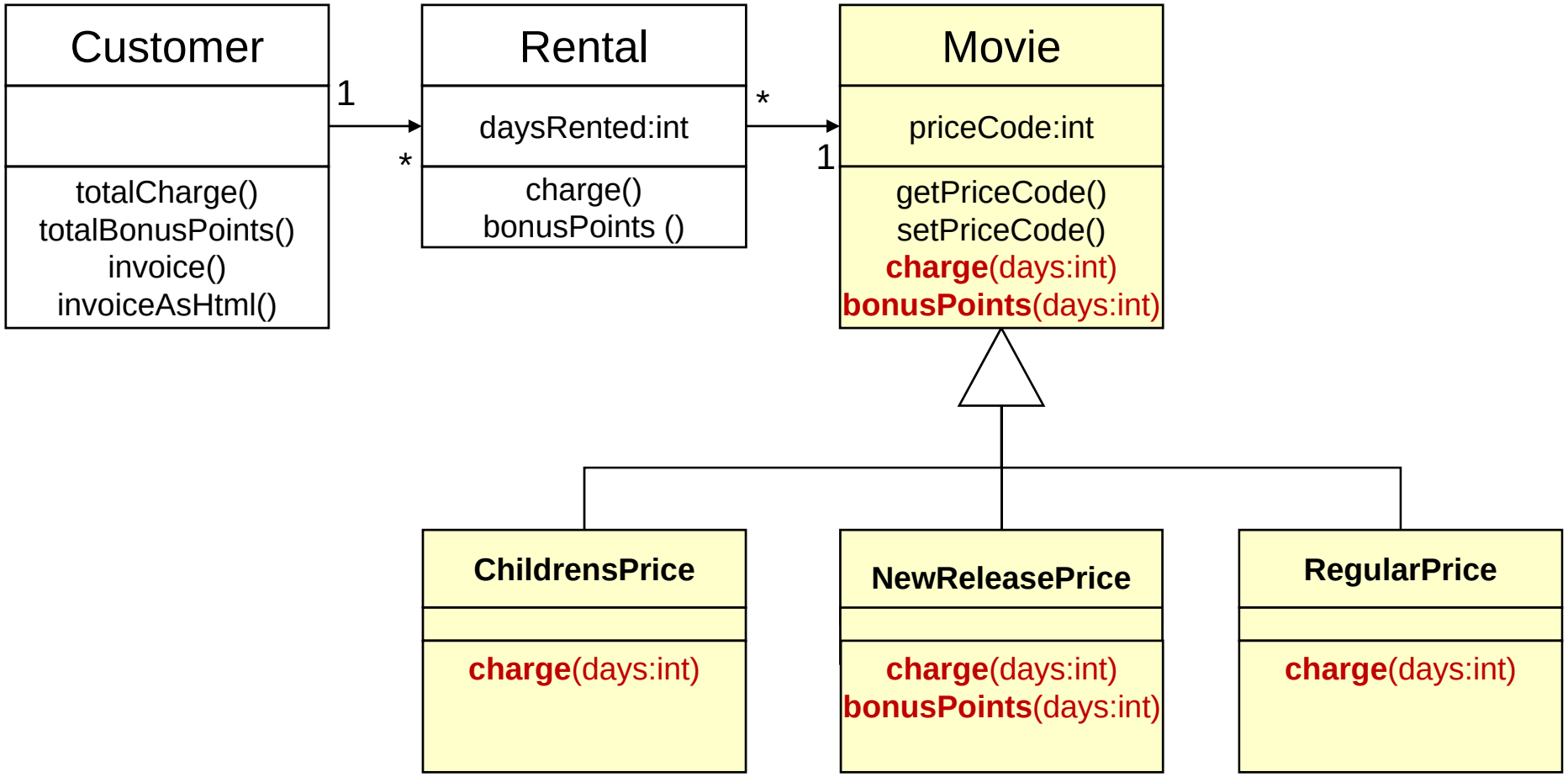



```
class Movie ...
   public int bonusPoints(int daysRented) {
      if ( (this.getPriceCode()==NEW_RELEASE)
           && daysRented>1)
         return 2;
      else
         return 1;
   }
```

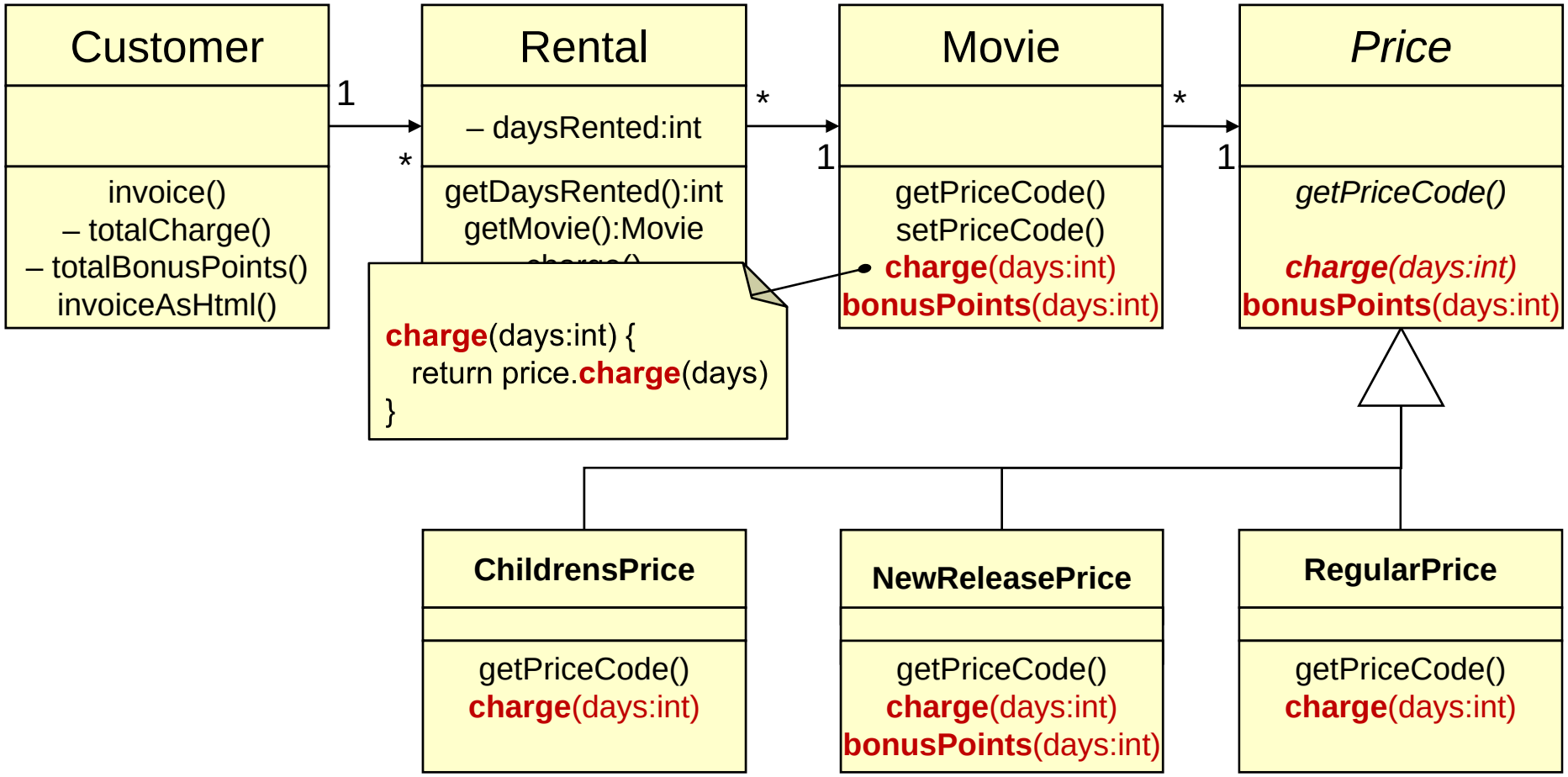
```
class Rental ...
    public int bonusPoints() {
        return _movie.bonusPoints(_daysRented);
    }
```

Polymorphismus via Vererbung

Hier nicht anwendbar:
ein Film hätte immer eine fixe Preiskategorie



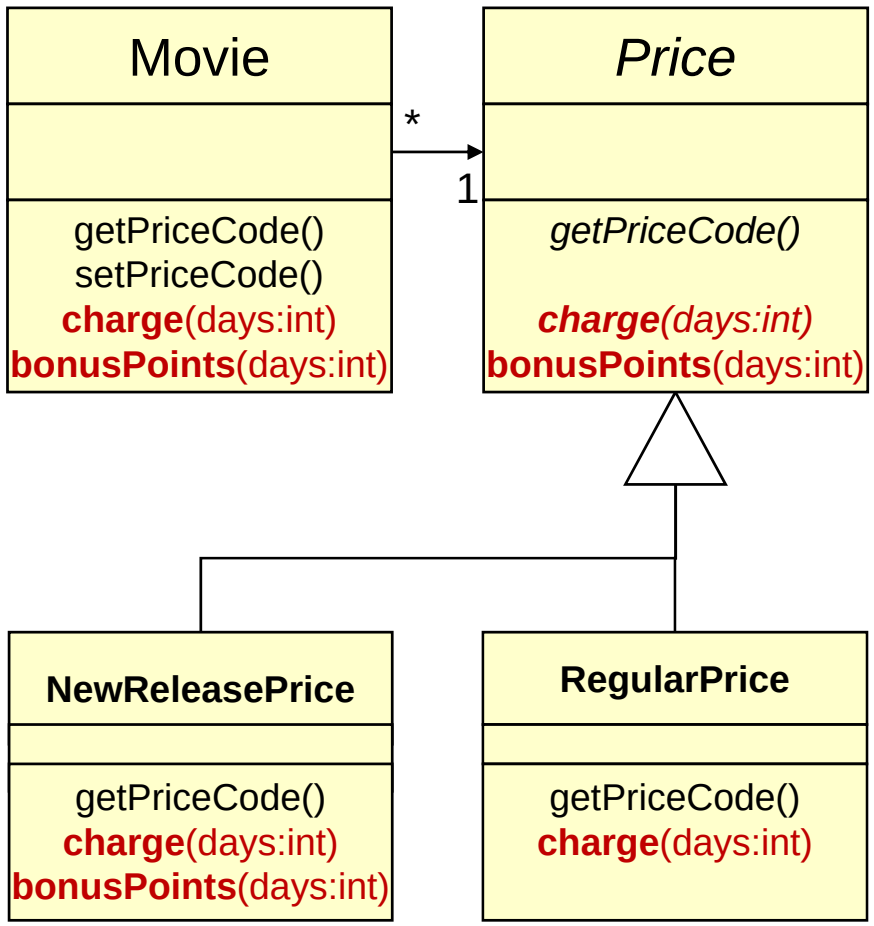
Polymorphismus via State Pattern



Polymorphismus via State Pattern

Schritte

1. Klassen Price, ..., RegularPrice erzeugen
2. Darin getPriceCode()-Methoden implementieren
3. Ersetzung von Preis-Code durch Preis-Objekt (in Movie)
   - setPriceCode(int)
   - getPriceCode
   - Konstruktor
4. charge() und bonusPoints() von Movie nach Price verlagern
5. Fallunterscheidungen durch Polymorphimus ersetzen
   - jeden Fall der charge() Methode aus Price in die charge()-Methode einer Unterklasse auslagern
   - analog für bonusPoints()



Schritt 1-3: Ersetzung von Preis-Code durch Preis-Objekt

```
class Movie { ...
   private int _priceCode;

   public Movie(String name, int priceCode) {
      _name = name;
      _priceCode = priceCode;
   }
   public int getPriceCode() {
      return _priceCode;
   }
   public void setPriceCode(int arg) {
      _priceCode = arg;
   }
}
```

3

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

   public Movie(String name, int priceCode) {
      _name = name;
      setPriceCode(priceCode);
   }
   public int getPriceCode() {
      return _price.getPriceCode();
   }
   public void setPriceCode(int 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");
      }
   }
}
```

1+2

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

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

```
class ChildrensPrice extends Price {
   public int getPriceCode() {
     return Movie.CHILDRENS;
   }
}
```

```
class NewReleasePrice extends Price {
   public int getPriceCode() {
     return Movie.NEW_RELEASE;
   }
}
```

Schritt 4-5: Fallunterscheidung durch Polymorphimus ersetzen

```
class Movie { ...
  public double charge(int daysRented) {
    return _price.charge(daysRented);
  }
```

4

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

5

```
abstract class Price ...
   abstract public double charge(int days);
```

```
class RegularPrice extends Price { ...
   public double charge(int daysRented){
      double result =2;
      if (daysRented > 2)
         result += (daysRented -2)*1.5;
      return result;
   }
```

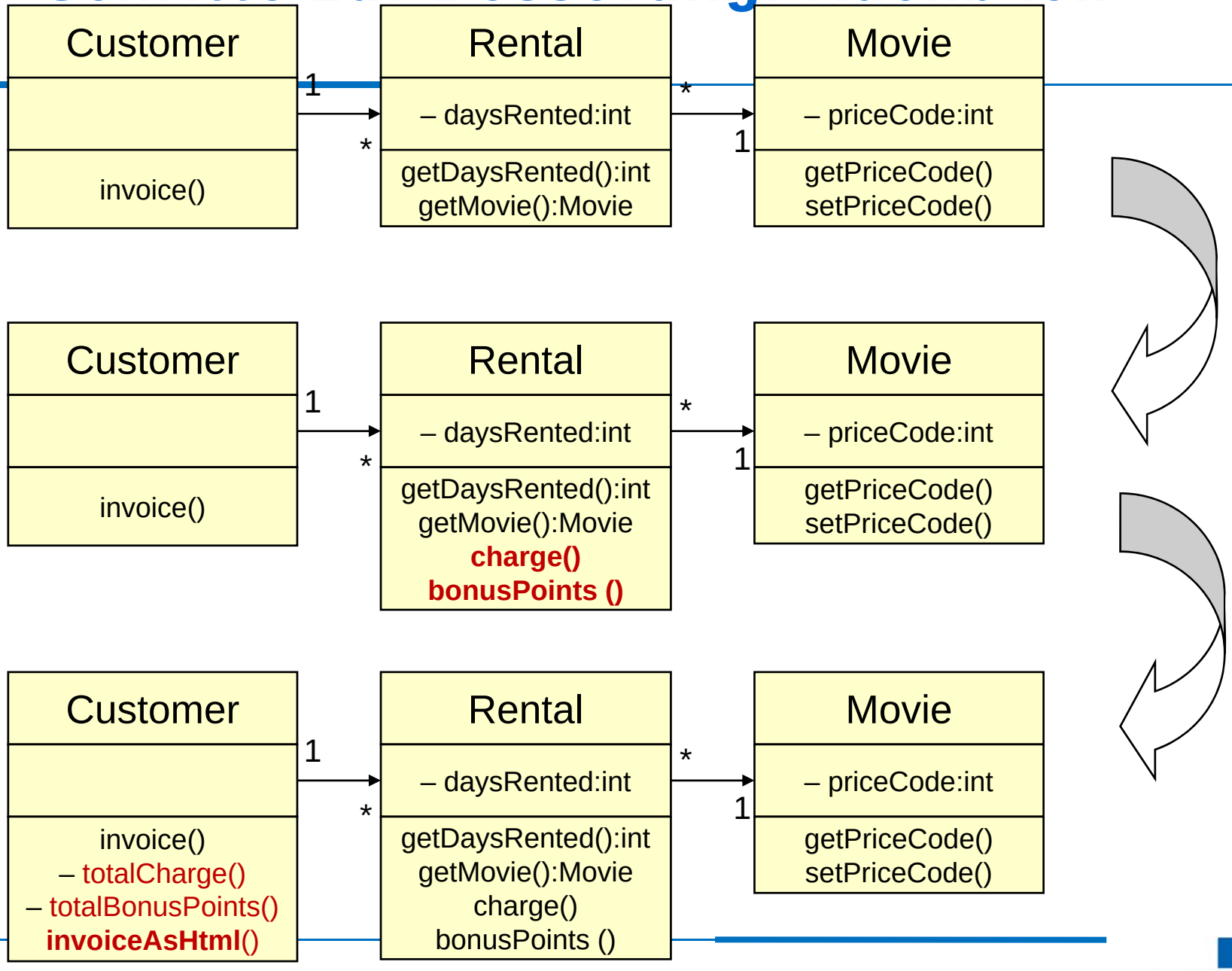
```
class ChildrensPrice extends Price { ...
   public double charge(int daysRented){
      double result = 1.5;
      if (daysRented > 3)
         result += (daysRented -3) * 1.5;
      return result;
   }
```

```
class NewReleasePrice extends Price { ...
   public double charge(int daysRented) {
      return daysRented * 3;
   }
```

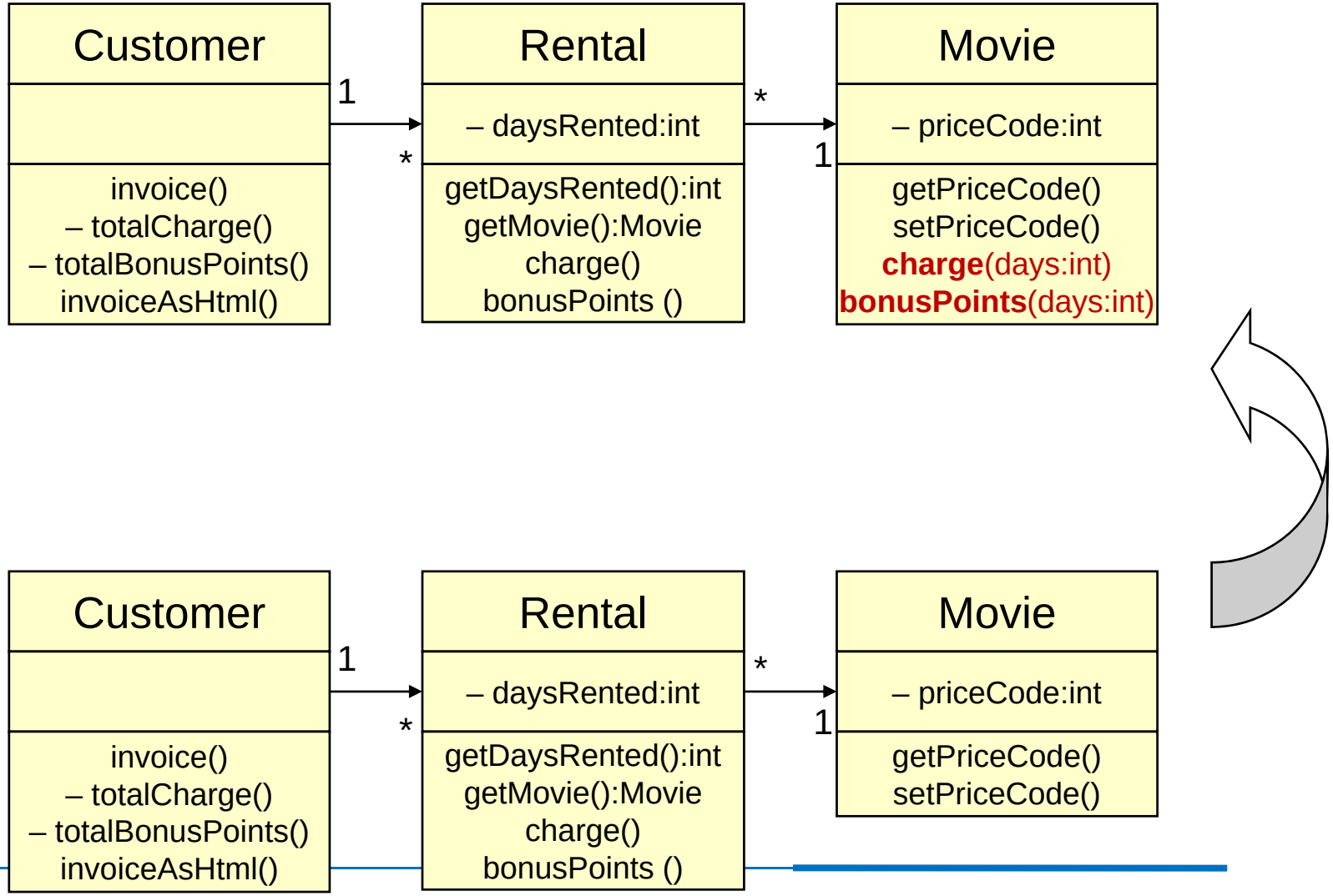
universitätbonn ROOTS

Rückblick auf das Beispiel: Zusammenfassung der Refactoring-Schritte

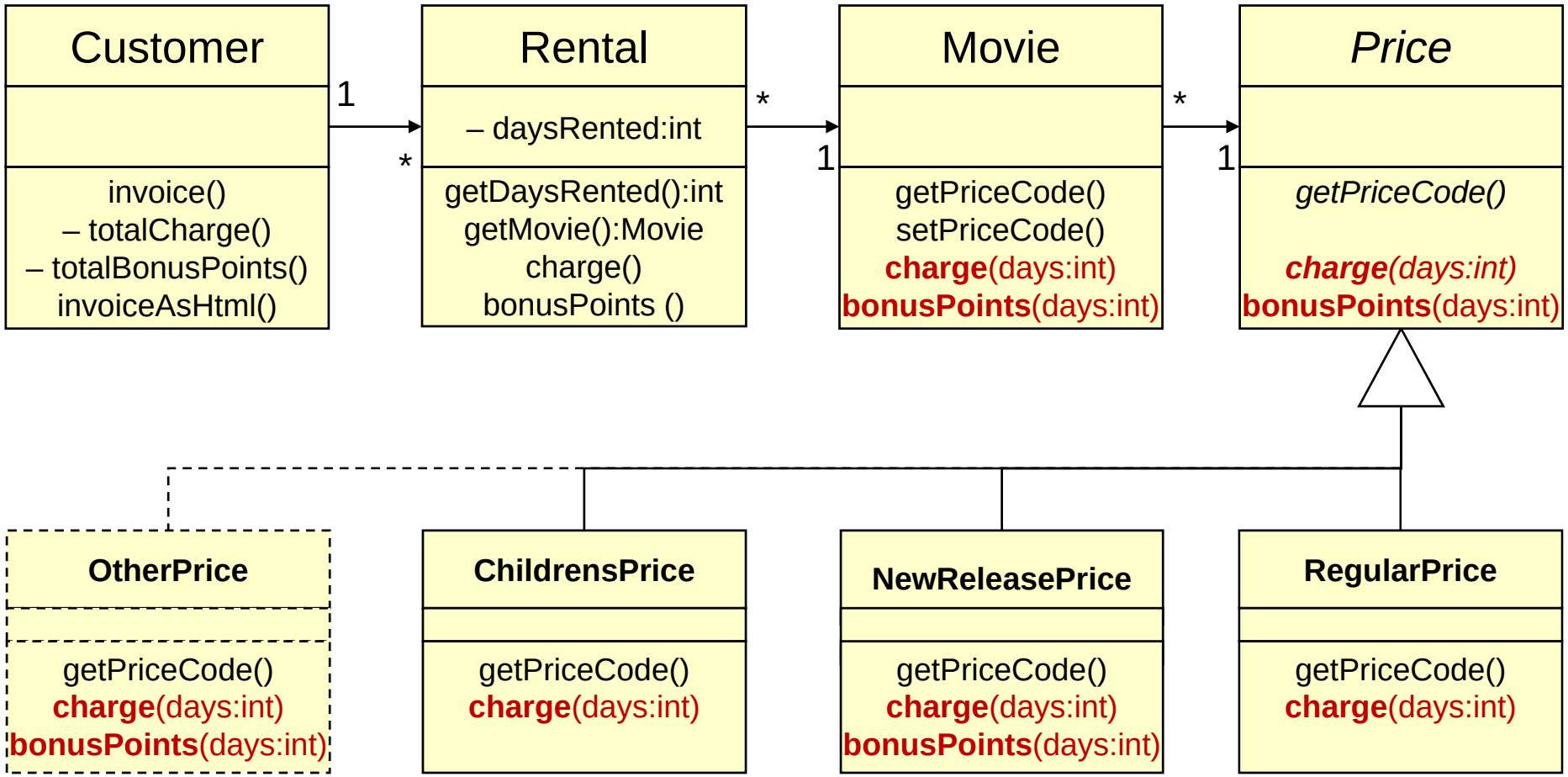
Schritte zur Besserung: Rückblick



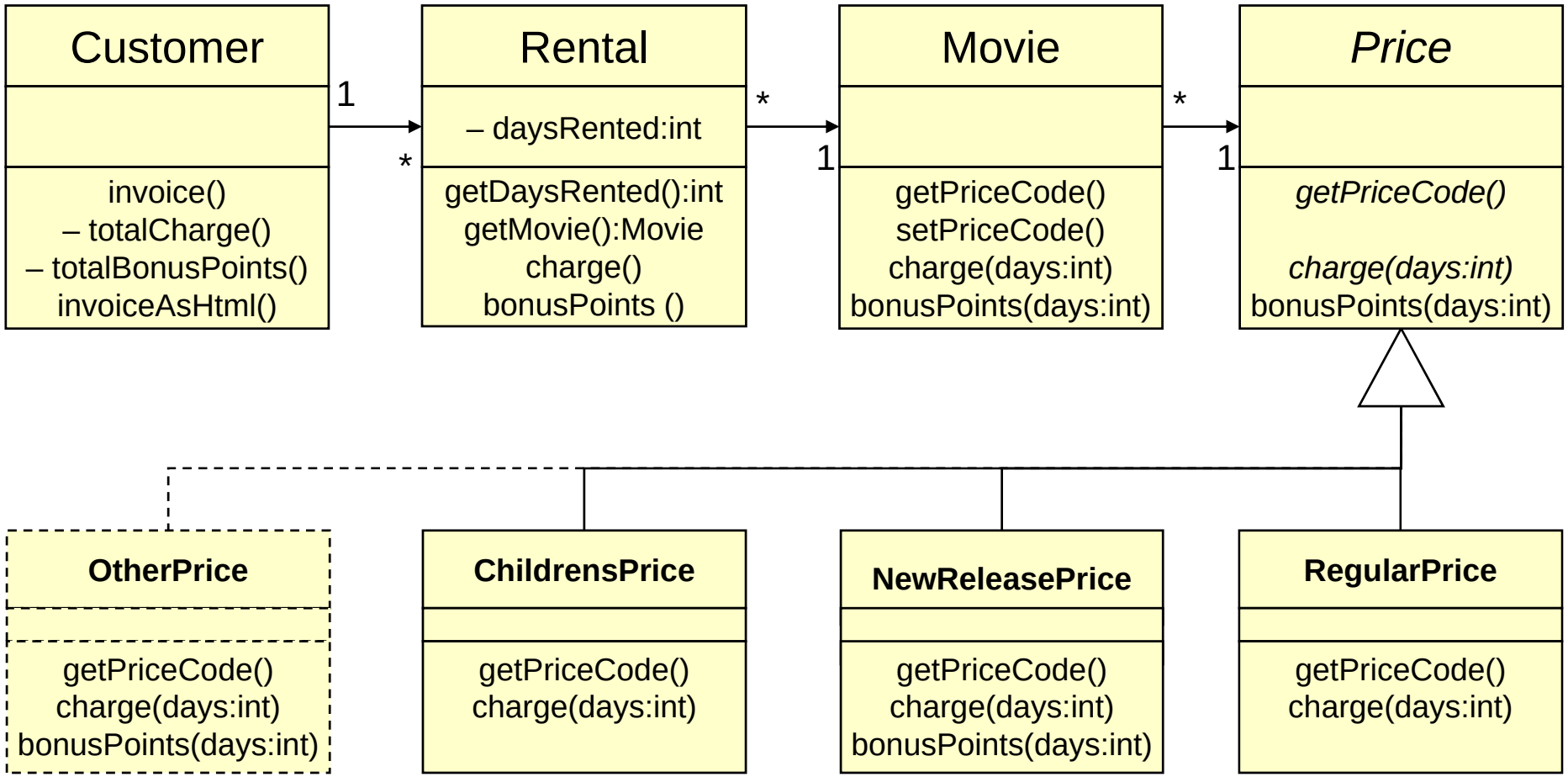
Schritte zur Besserung: Rückblick (2)



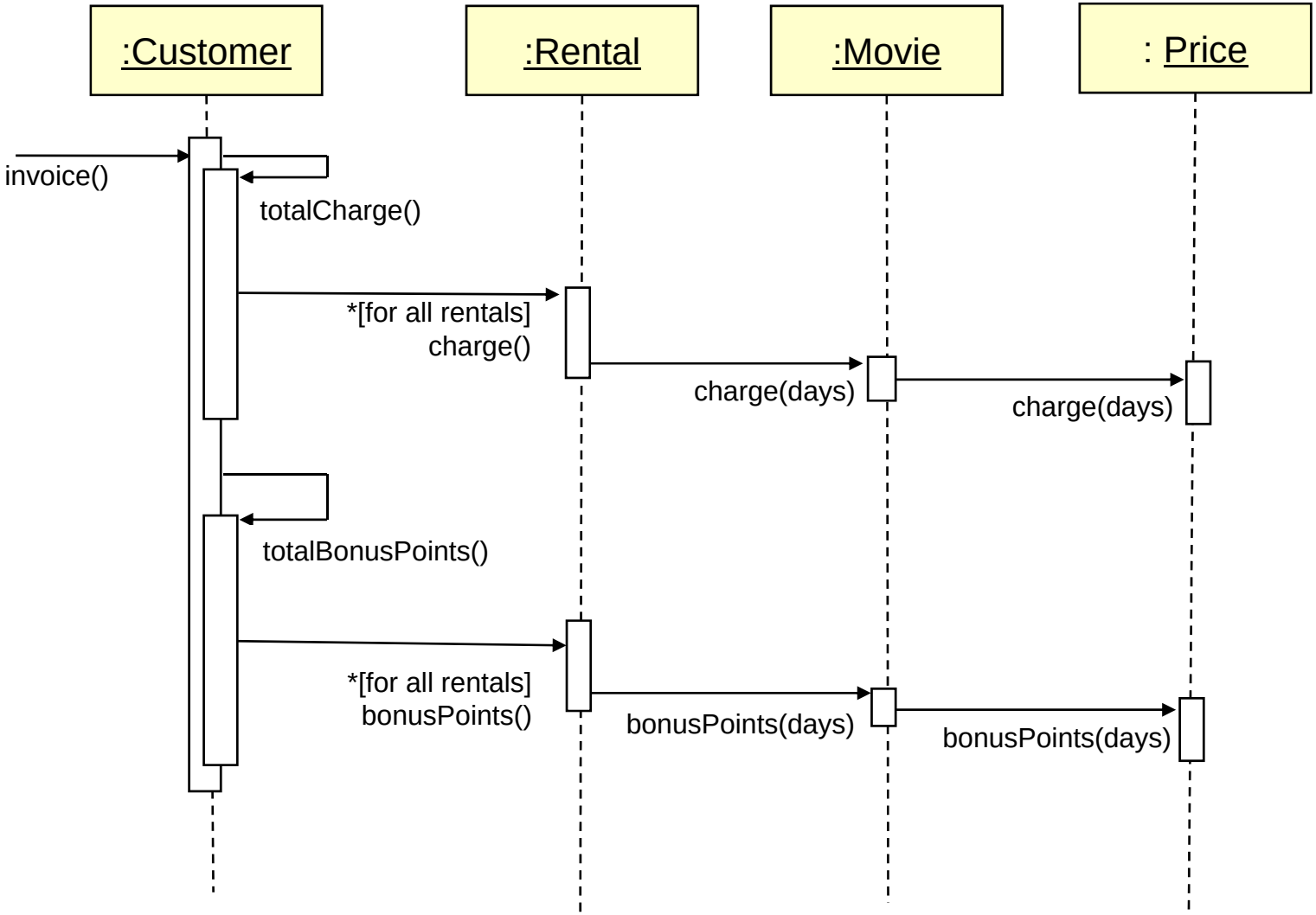
Schritte zur Besserung: Rückblick (3)



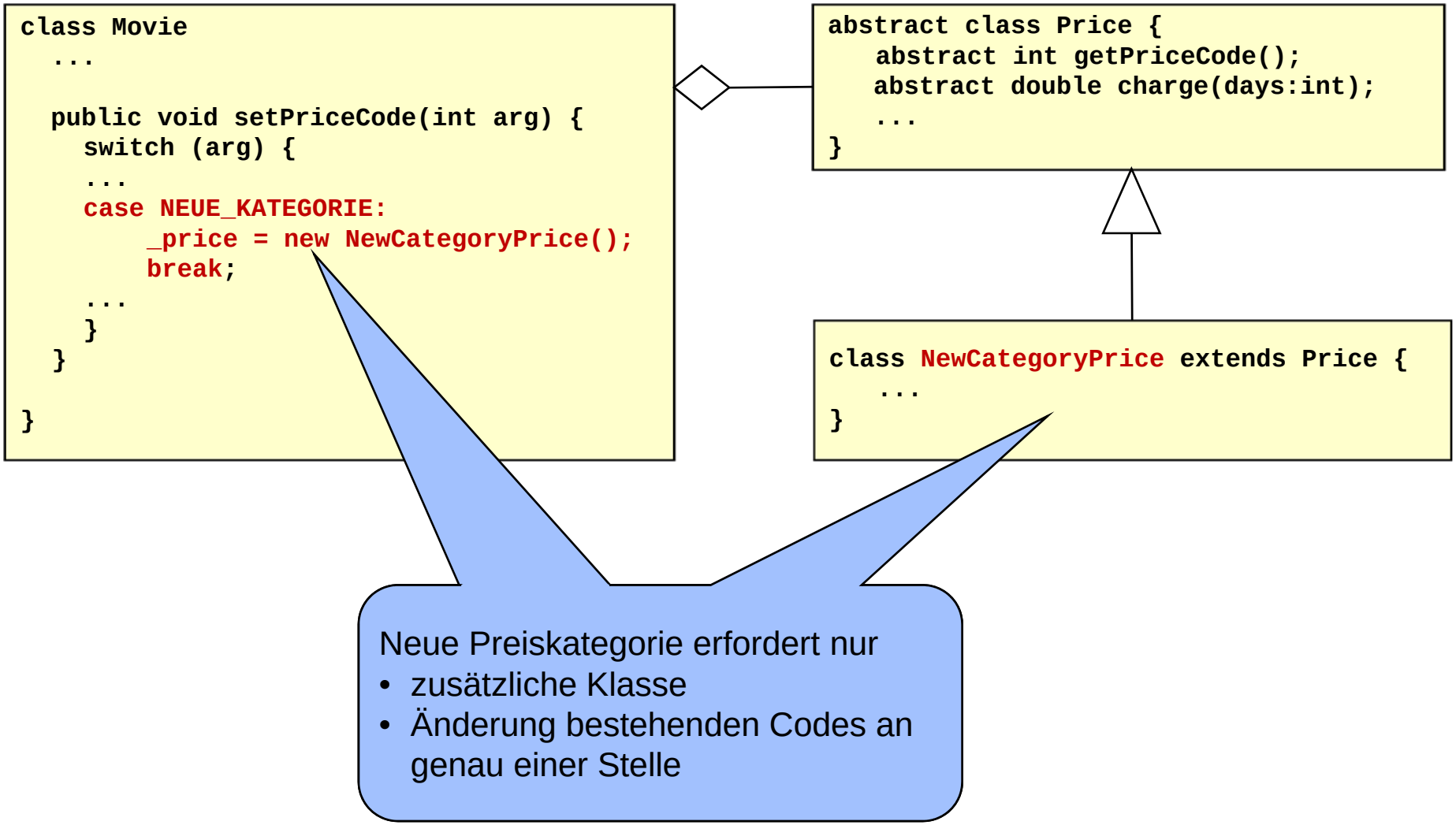
Endzustand: Klassen-Diagramm



Endzustand: Sequenz-Diagramm für invoice()-Aufruf



Nutzen: Einfache Erweiterbarkeit (2)



universitätbonn ROOTS

Was also ist Refactoring?

Was ist “Refactoring“?

Refactoring (noun):
a change made to the internal structure of software
to make it easier to understand and cheaper to modify
without changing its observable behavior.

Refactor (verb):
to restructure software by applying a series of refactorings.

- Definition
  - Systematische Umstrukturierung des Codes ohne das Verhalten nach außen zu ändern
- Nutzen
  - bessere Lesbarkeit, Verständlichkeit
  - besseres Design
  - bessere Wartbarkeit und Wiederverwendbarkeit

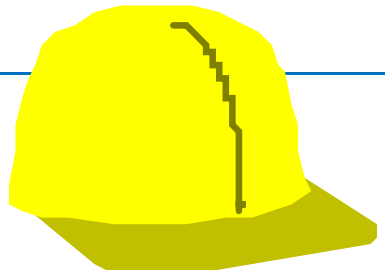
Was heißt "Systematische Umstrukturierung"?

- Klare Anweisungen
  - was
  - wann
  - wie
- Festgelegter Ablauf
  - kleine Schritte
  - Tests nach jedem Schritt

➔ Disziplin!

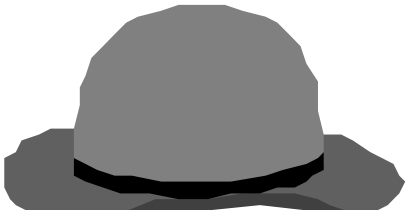
Ablauf: Kent Beck's "Hüte-Metapher"

- Was will ich **umstrukturieren**?
- Gibt es einen Test? → Test schreiben!
- Refactoring durchführen
- Testen → Fehler beheben!



Refactoring Hat

Programmer



Function Adding Hat

- Was will ich **hinzufügen**?
- Test schreiben
- Funktionserweiterung durchführen
- Testen → Fehler beheben!

universitätbonn ROOTS

Refactoring – Schritt für Schritt

Refactoring „Extract Method“ als Beispiel was mit systematischem Vorgehen beim Refactoring gemeint ist

Refactoring-Katalog (siehe Buch von Martin Fowler)

- ➔ Komposition von Methoden
  - Extract Method
  - Inline Method
  - Replace Temp with Query
  - Inline Temp
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - ...
- Verlagerung von Methoden
  - ...
- Strukturierung von Daten
  - ...
- Vereinfachung von Fallunterscheidungen
  - ...
- Vereinfachung von Methodenaufrufen
  - ...
- Vererbung
  - ...

Extract Method

- Indikation
  - Code-Fragment das logisch zusammengehört
- Behandlung
  - durch aussagekräftig benannte Methode ersetzen

```
void printOwing(double amount) {
   printBanner();

   // print details
   System.out.println(“name“+_name);
   System.out.println(“amount“+ amount);
}
```

```
void printOwing (double amount) {
   printBanner();
   printDetails(amount);
}

void printDetails (double amount) {
   System.out.println (“name“+_name);
   System.out.println (“amount“+ amount);
}
```

Schritte

- Neue Methode erzeugen und sinnvoll benennen
  - immer „private“
- Code kopieren
- Lokale Variablen der Ursprungsmethode im extrahierten Code suchen
  - Variablen, die nur noch in der neuen Methode benutzt werden
    - ⇨ lokale Variablen der neuen Methode
  - Variablen, die in der neuen Methode verwendet werden
    - ⇨ Parameter der neuen Methode
  - Variablen, die in neuer Methode verändert und in der alten weiter benutzt werden
    - ⇨ falls nur eine: als Ergebnis der neuen Methode zurückgeben
    - ⇨ mehr als eine: Teilmethode nicht extrahierbar!
      (evtl. vorbereitend andere Refactorings versuchen, die Variablen eliminieren oder Gruppen von Variablen zu einem Objekt zusammenfassen)
- Kompilieren
- In Ursprungsmethode
  - extrahierten Code ersetzen durch Aufruf der neuen Methode
  - Deklaration nicht mehr benötigter lokaler Variablen löschen
- Kompilieren
- Testen

Beispiel: keine lokalen Variablen im extrahierten Block

```
void printOwing(double amount) {

   Enumeration e = :orders.elements();
   double outstanding = 0.0;

   // print banner
   System.out.println("**********************");
   System.out.println("*** Customer owes ****");
   System.out.println("**********************");

   // calculate outstanding
   while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      outstanding += each.getAmount();

   // print details
   System.out.println("name"+ _name);
   System.out.println("amount"+ outstanding);
}
```

➔ Extraktion des Codes für Banner-Druck

Beispiel: keine lokalen Variablen im extrahierten Block

```
void printOwing(double amount) {

   Enumeration e = :orders.elements();
   double outstanding = 0.0;

   printBanner();

   // calculate outstanding
   while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      outstanding += each.getAmount();

   // print details
   System.out.println(“name“+ _name);
   System.out.println(“amount“+ outstanding);
}
```

```
private void printBanner() {
   System.out.println("**************************");
   System.out.println("*** Customer owes ****");
   System.out.println("**************************");
}
```

➔ Extraktion des Codes für "print details"

- lokale Variable, die nicht verändert wird („outstanding“)

Beispiel: lokale Variable, die nicht verändert wird

```
void printOwing(double amount) {

   Enumeration e = :orders.elements();
   double outstanding = 0.0;

   printBanner();




   // calculate outstanding
   while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      outstanding += each.getAmount();

   printDetails(outstanding);


}
```

```
private void printDetails(double outstanding) {
   System.out.println("name"+ _name);
   System.out.println("amount"+ outstanding);
}
```

- ➔ Extraktion des Codes für die Berechnung
  - ◆ lokale Variable, die verändert und anschließend benutzt wird („outstanding“)
  - ◆ lokale Variable, die verändert und anschließend nicht mehr benutzt wird („e“)

Beispiel: lokale Variable, die verändert wird

```
void printOwing(double amount) {

   Enumeration e = :orders.elements();
   double outstanding = 0.0;

   printBanner();




   outstanding = getOutstanding();




   printDetails(outstanding);


}
```

```
private double getOutstanding() {
   Enumeration e = orders.elements();
   double outstanding = 0.0;
   while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
       outstanding += each.getAmount();
   }
   return outstanding ;
}
```

➔ Extraktion des Codes für die Berechnung

- falls lokale Variable, vorher in der Ursprungsmethode zugewiesen wird

Beispiel: lokale Variable, die verändert wird (auch vorher)

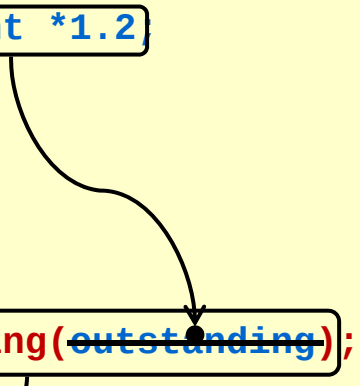
```
void printOwing(double amount) {


   double outstanding = amount *1.2;

   printBanner();




   outstanding = getOutstanding(outstanding);



   printDetails(outstanding);


}
```

```
private double getOutstanding(double startValue) {
   Enumeration e = orders.elements();
   double result      = startValue;
   while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      result       += each.getAmount();
   }
   return result      ;
}
```

- ➔ Nun sind weitere Refactorings möglich
  - Extraktion des Codes für die Berechnung
  - zweimaliges „inline temp“ für Zuweisungen an lokale Variable „outstanding“
  - falls lokale Variable, vorher in der Ursprungsmethode zugewiesen wird
  - ➔ so kann „outstanding“ aus „printOwing“-Methode eliminiert werden

Beispiel: Elimination von „outstanding“

```
void printOwing(double amount) {

   double outstanding = amount *1.2;

   printBanner();



   outstanding = getOutstanding(outstanding);



   printDetails(outstanding);


}
```



- ➔ Nun sind weitere Refactorings möglich
  - ◆ zweimaliges „inline temp“ für Zuweisungen an lokale Variable „outstanding“
  - ➔ so kann „outstanding“ aus „printOwing“-Methode eliminiert werden

Beispiel: Endzustand von printOwing() und extrahierte Methoden

```
void printOwing(double amount) {


   printBanner();


   printDetails(getOutstanding(amount *1.2));


}
```

```
private void printBanner() {
   System.out.println("**********************");
   System.out.println("*** Customer owes ****");
   System.out.println("**********************");
}
```

```
private void printDetails(double outstanding) {
   System.out.println("name"+ _name);
   System.out.println("amount"+ outstanding);
}
```

```
private double getOutstanding(double startValue) {
   Enumeration e = orders.elements();
   double result = startValue;
   while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      result += each.getAmount();
   }
   return result;
}
```

universitätbonn ROOTS

„Bad Smells“ – Indikationen für Refactoring

Lange Parameterliste

- Problem
  - Verständlichkeit
  - Fehleranfälligkeit
  - dauernde Änderungen
- Idee
  - Parameter-Werte aus bereits bekannten Objekten besorgen

➔ Parameter ersetzen durch Methodenaufruf

- an anderen Parameter oder Instanz-Variable

➔ Parametergruppe ersetzen durch Objekt aus dem die Werte stammen

- anschließend Mehodenaufrufe an diesen einen Parameter

➔ Parametergruppe durch Objekt einer neuen Klasse ersetzen

- für ansonsten nicht zusammengehörige Parameter

- Ausnahme
  - wenn man bewusst keine Abhängigkeit zu einer bestimmten Klasse erzeugen will

Lange Parameterliste: Beispiel

Vorher

```
obj.method(w, w.get2(), x.get3(), x.get4(), y, z);
```

```
void method(a1, a2, a3, a4, a5, a6){
   ...
}
```

- Abhängigkeiten an jeder Aufrufstelle
  - Typ von w
  - Typ von x
  - Typ von y
  - Typ von z

Nachher

```
obj.method(w, x, newParam);
```

```
void method(a1, a34, a56) {
   a2 = w.get2();
   a3 = a34.get3();
   a4 = a34.get4();
   a5 = a56.getY();
   a6 = a56.getZ();
   ...
}
```

- Abhängigkeiten in Methode
  - Typ von w
  - Typ von x
  - Typ von newParam

Änderungsanfälligkeit („Divergente Änderungen“)

- Symptom
  - verschiedene Änderungsarten betreffen gleiche Klasse
  - Beispiel
    - ⇨ neue Datenbank: Methode 1 bis 3 in Klasse C ändern
    - ⇨ neue Kontoart: Methode 6 bis 8 in Klasse C ändern

- Behandlung
  - Klasse aufteilen
    - ⇨ C_DB Methode 1 bis 3
    - ⇨ C_Konto Methode 6 bis 8
    - ⇨ C Restliche Methoden

- Effekt
  - Lokalisierung von Änderungen

Verteilte Änderungen

- Symptom
  - eine Änderung betriff viele Klassen
- Problem
  - schlechte Modularisierung
  - Fehleranfälligkeit
- Behandlung
  - Methoden verlagern
  - Felder verlagern
  - ... so dass Änderungen in nur einer Klasse erforderlich sind
  - Evtl. geeignete Klasse erzeugen
  - Evtl. Klassen zusammenfassen
- Effekt
  - Lokalisierung von Änderungen

Neid: „Begehre nicht deines Nächsten Hab und Gut!“

- Symptom
  - Methode die sich vorwiegend um eine bestimmte anderen Klasse „kümmert“
  - Typisch: viele „get...()“-Aufrufe an andere Klasse
- Behandlung allgemein
  - „Neidische Methode“ in andere Klasse verlagern
  - evtl. „neidischen Teil“ der Methode extrahieren und verlagern
- Behandlung nicht-eindeutiger Fälle
  - Methode in Klasse verlagern die „am stärksten beneidet wird“

  oder
  - verschiedene Teilmethoden in verschiedene Klassen verlagern
- Ausnahmen
  - Strategy und Visitor Pattern
  - allgemein: Bewusste Dekomposition um divergente Änderungen zu bekämpfen

Daten-Klumpen

- Symptom
  - Daten die immer gemeinsam vorkommen/ benutzt werden
  - Instanzvariablen oder Parameter
- Behandlung
  - Extraktion der Daten in eigene Klasse
    - ⇨ „Extract Class“
    - ⇨ „Introduce Parameter Object“
    - ⇨ „Preserve Whole Object“
  - anschließend anhand „Neid“-Kriterium Methoden verlagern
- Effekt
  - bessere Modularisierung
  - kürzere Parameterlisten

Fixierung auf primitive Datentypen

- Symptom
  - viele Variablen von primitiven Datentypen
  - Beispiel: „String adresse“
- Behandlung
  - Extraktion der Variablen in eigene Klasse
  - Allgemein
    - ⇨ „Replace Data Value with Object“: „Adresse adresse“
  - Typ-Codierung
    - ⇨ „Replace Type Code with Class“
    - ⇨ „Replace Type Code with Subclasses“
    - ⇨ „Replace Type Code with State / Strategy“
  - mehrere zusammengehörige Variablen
    - ⇨ s. „Daten-Klumpen“
- Effekt
  - bessere Erweiterbarkeit

Fallunterscheidungen (Switch-Statements)

- Symptom
  - Fallunterscheidungen selektiert Methodenaufrufe
  - oft in Verbindung mit „Typ-Code“
- Problem
  - Redundanz: oft gleiche Fallunterscheidungen an vielen Stellen
  - schlechte Erweiterbarkeit
- Behandlung
  - Fallunterscheidungen als Teilmethode extrahieren
  - ... in Klasse verlagern zu der der Typ-Code logisch gehört
  - ... Typ-Code durch Unterklassen ersetzen
    - ⇨ „Replace Type Code with Subclasses“
  - ... jeden Fall in entsprechende Methode einer Unterklassen verlagern
    - ⇨ „Replace Conditional with Polymorphism
  - Wenn dabei eine neue Klassenhierarchie für Typ-Code erzeugt wird
    - ⇨ „Replace Type Code with State / Strategy“

Fallunterscheidung (Fortsetzung)

- Behandlung bei wenigen, festgelegten Alternativen
  - ... wenn also keine Erweiterbarkeit erforderlich ist
  - „Replace Parameter with explicit Methods“
    - ⇨ Eigene Methode für jeden Fall
    - ⇨ Fallunterscheidung eliminieren
    - ⇨ Aufrufer ruft spezifische Methode auf, statt spezifischen Typ-Code zu setzen
- Behandlung von Tests auf „null“
  - „Introduce Null Object“
    - ⇨ Erwarteten Objekttyp um eine Unterklasse erweitern
    - ⇨ ... deren Methoden das tun, was im „null“ Fall getan werden soll
    - ⇨ Statt „null“ solche „Null-Objekte“ übergeben
    - ⇨ Fallunterscheidung eliminieren
- Effekte
  - einfacherer Code
  - keine Redundanzen
  - bessere Erweiterbarkeit

Faule Klasse

- Symptom
  - Klassen, die fast nichts mehr tun
    - ⇨ Überbleibsel des Refactoring-Prozesses
    - ⇨ auf Verdacht angelegt und dann doch nie benötigt
- Problem
  - jede „faule Klasse “bringt unnötige Kosten
    - ⇨ Wartungs-
    - ⇨ Verständnis-
    - ⇨ Laufzeit-
  - das gleiche gilt für „faule Hierarchien“ („faule Unterklassen“)
- Behandlung
  - Eliminierung der faulen Klasse
    - ⇨ faule Unterklasse in Oberklasse integrieren („Collapse Hierarchy“)
    - ⇨ faule Klasse in Client integrieren („Inline Class“)

Spekulative Allgemeinheit

- Symptom
  - Jemand sagt: „Ich glaub man wird auch mal ... brauchen.“
- Problem
  - komplexeres Design
  - schwerer zu verstehen
  - schwerer zu benutzen
  - schwerer zu warten
  - ... und all das, nur auf Verdacht, ohne das es wirklich gebraucht wird!
- Behandlung
  - „faule“ abstrakte Klassen: eliminieren („Collapse Hierarchy“)
  - überflüssiges Forwarding: Aggregat in Client integrieren („Inline Class“)
  - überflüssige Parameter: eliminieren („Remove Parameter“)
  - übermäßig abstrakte Methodennamen: umbenennen

Temporäre Felder

- Symptom
  - Instanz-Variablen, die manchmal nicht initialisiert / benutzt werden
  - Beispiel
    - ⇨ eine Methode implementiert komplexen Algorithmus der viele Parameter hat
    - ⇨ Parameter werden nicht übergeben, sondern als Instanzvariablen angelegt
- Problem
  - Verständnis („Was soll das denn?“)
- Behandlung
  - Variable und darauf zugreifende Methoden ein eigene Klasse extrahieren
- Effekt
  - bessere Verständlichkeit
  - bessere Modularisierung

Verweigertes Vermächtnis

- Symptom
  - Unterklassen nutzen geerbte Variablen nicht
  - Unterklassen implementieren geerbte Methoden so dass sie nichts tun oder Exceptions werfen
- Problem
  - Vererbung falsch angewendet
  - Subtypbeziehung nicht angebracht
- Behandlung
  - Vererbungs-Hierarchie verändern
    - ⇨ „verweigerte Anteile“ aus Oberklasse in neue Unterklasse auslagern
    - ⇨ „verweigernde Methoden“ aus anderen Unterklassen eliminieren

  oder
  - Vererbung durch Aggregation und Forwarding ersetzen

Kommentare im Methoden-Rumpf

- Symptom
  - Kommentar erklärt was als nächstes geschieht
- Problem
  - Code ist offensichtlich nicht verständlich genug
- Behandlung
  - Teilmethode extrahieren (mit aussagekräftigem Namen)
  - Teilmethoden umbenennen (aussagekräftigere Namen)
  - „Assertions“ benutzen um Randbedingungen explizit zu machen
- Effekt
  - selbstdokumentierender Code
  - selbstcheckender Code (Assertions)

universitätbonn ROOTS

Refactoring Zusammenfassung

Warum soll ich es anwenden?

Wann soll ich es anwenden?

Refactoring und Design

Refactoring und Effizienz

Wie sag ich's dem Chef?

Wo sind die Grenzen?

Warum soll man “Refactoring“ anwenden?

- Programme sind schwer zu warten wenn sie
  - ... unverständlich sind
  - ... Redundanzen enthalten
  - ... komplexe Fallunterscheidungen enthalten
  - ... Änderungen bestehenden Codes erfordern, um Erweiterungen zu implementieren

- Probleme
  - Oft geänderte Software verliert ihre Struktur.
  - Je mehr Code um so unverständlicher
  - Redundanter Code / Inkonsistenzen

Warum soll man “Refactoring“ anwenden?

- “Refactoring“ macht Software leichter wartbar
  - Extraktion gemeinsamer Teile: jede Änderung an genau einer Stelle durchführen
  - "Rule of three": Wenn man das zweite Mal das gleiche tut ist Code-Duplizierung noch OK. Beim dritten Mal sollte spätestens Umstrukturiert werden.
- “Refactoring“ macht Software leichter verständlich
  - Einarbeitung ist leichter, wenn man sofort restrukturiert
  - hilft einem selbst und denen die später kommen
- “Refactoring“ hilft Bugs zu finden
  - besseres Verständnis für den Code erleichtert Fehlersuche
- “Refactoring“ macht das Programmieren schneller
  - in verständlichen ist und fehlerarmen Code kann Neues schneller eingebaut werden

Wann soll man „Refactoring“ anwenden?

- „Refactoring“ ist ständiger Teil des Entwicklungsprozesses
- „Refactoring“ findet nach Bedarf statt ...
  - keine geplante Aktivität
- ... wenn man neue Funktionen hinzufügt
  - Wenn man neue Funktionen hinzufügt und denkt, es wäre einfacher, wenn diese Funktion da und jene dort wäre, dann soll man erst umstrukturieren
- ... wenn man einen Fehler sucht
  - Durch “Refactoring“ wird der Code einfacher. Man findet leichter Bugs.

Wann soll man „Refactoring“ anwenden?

- Code-Review
  - fremden Programmierern den eigenen Code erklären
- Sinn von Code-Reviews
  - Know-how-Transfer
  - Qualitätssicherung (Fehlersuche, besseres Design, ...)
  - Verbesserungs-Vorschläge
- Refactoring während eines Code-Reviews
  - Der Code wird einfacher verständlich.
  - Verbesserungs-Vorschläge werden sofort umgesetzt.
  - Man erkennt neue Möglichkeiten
- Extreme Programming
  - permanenter Code-Review („programming in pairs“)
  - permanentes Refactoring

Refactoring und Design

- Bisher
  - Refactoring zur Wartung existierender Software
- Nun
  - Refactoring als alternative SW-Entwicklungsmethode
- Traditionelle Phaseneinteilung
  - erst Design
  - während der Implementierung das Design strikt einhalten
  - iterativ vorgehen
- „Extreme Programming“
  - minimales Design
  - während der Implementierung bei Bedarf Refactoring anwenden
  - Design und Implementierung jederzeit gemischt in kleinen Schritten
  - „extrem iterative“ Vorgehensweise

Refactoring und Indirektion

- Refactoring bedeutet meistens Indirektion
  - Zerlegung großer Methoden in kleine Methoden
  - Zerlegung großer Objekte in kleine Objekte
- Vorteile von Indirektion
  - Gemeinsame Nutzung („sharing“) von Programmlogik
    - ⇨ Methoden aus Oberklassen
    - ⇨ Teilmethoden
  - Trennung von Intention und Implementation
    - ⇨ Intention = aussagekräftige Namen
    - ⇨ Implementation nutzt Namen von Teilmethoden
    - ⇨ größtenteils selbstdokumentierender Code
  - Lokalisation von Änderungen
    - ⇨ Teilmethode
    - ⇨ Unterklasse
  - Ersatz für Fallunterscheidungen
    - ⇨ Nachrichten
- Nachteile von Indirektion
  - erhöhte Komplexität

Refactoring und Performance

- Indirektion geht auf Kosten der Laufzeit
  - Also doch kein Refactoring?

Empfehlung

- Zuerst auf gutes Design konzentrieren
  - Auch “Refactoring“ gehört dazu
- Grundregeln der Optimierung (nach Dijkstra)
  - 1. Don‘t do it.
  - 2. Don‘t do it yet.
- Optimierung so spät wie möglich!
  - Voreilige Optimierungen müssen oft mit viel Aufwand rückgängig gemacht werden
- Nur häufig durchlaufene Programmteile („hot spots“) optimieren
  - Statistik: für ca. 80% der Laufzeit sind 10-20% des Codes verantwortlich
  - den Rest zu optimieren ist Zeitverschwendung
  - Profiling-Tools nutzen

Umgang mit API-Änderungen

Viele Refactorings ändern Schnittstellen: Umbenennung, Signatur-Änderung, Verschiebung:

- Ist die Menge der Clients bekannt?
  - Ja: Prüfen ob sie von der Änderung betroffen sind
  - Nein: Reparatur oder Vorbeugung
- Reparatur
  - Methoden: Forwarding-Methode mit alter Signatur einfügen („Bridge Methode“)
  - Felder: ???
- Vorbeugung
  - „published Interface“ schlank halten
    - ⇨ Variablen und Methoden so lange wie möglich „privat“ belassen
  - „code ownership“ flexibler gestalten
    - ⇨ „privates“ soll jeder ändern dürfen

Wie sag ich's dem Chef?

- Technisch kompetenter Chef
  - meist kein Problem
- Qualitäts-orientierter Chef
  - Qualitätsgewinn betonen
- Termin-orientierter Chef (Achtung: gibt evtl. vor qualitäts-orientiert zu sein)
  - Chef will termingerechte Fertigstellung
  - Wie ich das hinkriege ist meine Sache
  - ➔ Nichts sagen, einfach tun!

Grenzen des Refactoring

- Komplexe Design-Änderungen
  - Erst wohldurchdachtes Redesign versuchen!
  - Kriterium: „Was sind die Folgen und wie leicht kann die Änderung rückgängig gemacht werden?“
- „Völlig vermurkste Software“
  - Komplett neu schreiben!
- Abgabetermin steht kurz bevor
  - Nutzen des Refactoring würde zu spät zum tragen kommen

Refactoring – Gesamtzusammenfassung

- Bedeutung von Strukturverbesserung für Software-Qualität
  - Lesbarkeit, Verständlichkeit, Änderbarkeit
- Bedeutung von automatisierten Tests
  - Schnelles Feedback ob Strukturänderung auch das Verhalten geändert hat
- Bedeutung von verhaltenserhaltender Strukturverbesserung
  - Fehlerfreiheit
  - Maximale Nutzung vorhandener Tests
- Bedeutung von feingranularer Strukturverbesserung in kleinen Schritten
  - Fehler zeigen sich schon nach wenigen Änderungen
  - Leicht Fehlerursache zu identifizieren

- Refactoring = verhaltenserhaltende Strukturverbesserung
  - Manuell ➔ systematisches Vorgehen in kleinen Schritten und mit Tests
  - Automatisiert ➔ Werkzeug überprüft vor dem Refactoring die Vorbedingungen die gelten müssen, damit die Transformation auf dem gegebenen Programm verhaltenserhaltend ist.

Weiterführende Informationen zum Thema Refactoring

- Martin Fowler: „Refactoring – Improving the Design of Existing Code“, Addison-Wesley, 1999.
  - Grundlage dieses Kapitels der Vorlesung
  - Das einführende Beispiel und alle anderen Inhalte stammen daraus.
  - Umfangreicher Katalog von Refactorings und Bad Smells

- Martin Fowler: www.refactoring.com
  - Trägt viel nützliche Informationen zum Thema zusammen, u.a.
    - ⇨ Auflistung von Refactoring Tools
    - ⇨ Links zu verwandten Seiten

- Newsgroup zum Thema
  - groups.yahoo.com/group/refactoring
  - Hier schreibt auch Martin Fowler ab und zu etwas...

Selbsttest

- Was ist Refactoring?
- Warum ist es erforderlich?
- Was ist der Nutzen von Refactoring?
- Was ist der Zusammenhang von Refactoring und Tests?
- Warum sollte man Umstrukturierung und Funktionserweiterung nicht mischen?
- Was sind „Bad smells“?
- Warum sollte man „Bad smells“ kennen?
- Beispiel von Bad Smells?
- Beispiele von Refactorings (aus Videoverleih Fallstudie)
- Extract Method Refactoring im Detail (incl. Problemfälle)
- Refactoring von APIs versus Refactoring von „geschlossenen“ Systemen