

**Einführung in die Informatik II**  
**Entwurf durch Verträge:**  
**2. Erstellung von Verträgen mit OCL**

Prof. Bernd Brügge, Ph.D  
Institut für Informatik  
Technische Universität München

Sommersemester 2001

23. - 28. Mai 2001

**Themen der heutigen Vorlesung**

- ❖ Eine komplexere Anwendung von OCL
  - Spezifikation von UML-Modell-Eigenschaften in OCL
- ❖ Heuristiken für guten OCL-Stil
- ❖ **Javadoc**: Ein Werkzeug zur Dokumentation von Java-Programmen
- ❖ Formulierung von OCL-Ausdrücken in Javadoc

**Ziele**

- ❖ Sie können OCL-Ausdrücke - insbesondere Verträge - in Javadoc formulieren:
  - Invarianten in Javadoc
  - Vor- und Nachbedingungen in Javadoc
- ❖ Sie verstehen das Zusammenspiel von
  - Problembeschreibung
  - Modellierung des Problems (UML)
  - Spezifikation von Verträgen (OCL)
  - Dokumentation von Verträgen (javadoc)

**Punkt-Notation vs. Pfeil-Notation in OCL**

- ❖ Wann benutzt man die Punkt-Notation (.), wann die Pfeil-Notation (->)?
- ❖ Die Punkt-Notation wird verwendet für
  - Zugriff auf Attribute aus dem UML-Modell
  - Aufruf von (Seiteneffekt-freien) Methoden aus dem UML-Modell
  - Aufruf von OCL-Operationen auf einzelnen OCL-Objekten
  - Navigation entlang von Assoziationen
- ❖ Die Pfeil-Notation wird verwendet für
  - Aufruf von OCL-Sammlungsoperationen auf OCL-Sammlungen

## Invarianten auf UML-Modell-Eigenschaften

- ❖ **Beispiel:** Wir haben eine Klasse im Spezifikationsmodell als abstrakt definiert und wollen nun verhindern, dass eine Instanzierung dieser Klasse (z.B. aufgrund fehlerhafter Implementierung) möglich ist.
- ❖ Das können wir in OCL formulieren.  
Dazu nutzen wir zwei weitere OCL-Operationen:
  - **type.allInstances: Set (type)**  
Menge aller Instanzen des OCL-Typs **type** und seiner Untertypen
  - **object.oclType: OclType**  
Typ (aus dem OCL-Modell) des OCL-Objekts **object**
- ❖ **Beispiel (aus TUMBoS):** Wir formulieren folgende Invariante, um sicherzustellen, dass **Transaktion** nicht direkt instanzierbar ist:

### Transaktion

```
Transaktion.allInstances->select(oclType =
    Transaktion) ->isEmpty
```

Die Unterklassen von **Transaktion** sind instanzierbar ⇒ nur echte Instanzen auswählen

## Das erweiterte OCL-Modell für das "Treueprogramm Silber & Gold"

<pre> <u>Kunde</u> Alter &gt;= 18 <u>Kundenkarte</u> gültigAb.istVor(verfälltAm) <u>KundenKarte</u> gedruckterName =     kunde.titel.concat(kunde.name) <u>Dienst</u> (punkteVerdient &gt; 0) implies     (punkteBezahlt = 0) <u>BonusProgramm</u> partner-&gt;size = 4 <u>BonusProgramm</u> ebene-&gt;size = 2         </pre>	<pre> <u>BonusProgramm</u> ebene-&gt;first.name = "Silber" <u>BonusProgramm::einschreiben(k:Kunde)</u> pre: not kunde-&gt;includes(k) Post: kunde = kunde@pre-&gt;including(k) <u>ProgrammPartner</u> anzahlKunden =     bonusProgramm.kunde-&gt;asSet-&gt;size <u>Transaktion::programm():BonusProgramm</u> pre: true post: result =     karte.mitgliedschaft.programm <u>Transaction</u> Transaction.allInstances-&gt;select(oclType     = Transaction)-&gt;isEmpty         </pre>
--	--

## Wie schreibe ich gutes OCL?

- ❖ Ist unser OCL-Modell für TUMBoS gut?  
Was ist ein gutes OCL-Modell?
- ❖ **Frage 1:** In welchem Kontext sollte eine Invariante formuliert werden?
- ❖ Wir könnten alle Invarianten eines Klassendiagramms schreiben, indem wir immer denselben Kontext nehmen, z.B. **BonusProgramm**.  
Das hat drei Nachteile:
  1. Eine Navigation, die das ganze Klassendiagramm durchschreiten muss, erzeugt eine sehr enge Koppelung von Objekten. Ein Prinzip bei Objekt-Orientierung ist allerdings, Objekte zu entkoppeln, d.h. so wenig wie möglich von anderen Objekten abhängig zu machen.
  2. Wenn man nicht aufpasst, kann man schnell lange und komplexe Ausdrücke in OCL schreiben, die nicht sehr leserlich sind.
  3. Wenn sich das UML-Modell (oder sogar die Problemstellung) ändert, kann die Wartung von komplexen OCL-Ausdrücken ein Alptraum werden.

## Info II-Stilfibel für OCL

- ❖ Einschränkungen müssen einfach sein.
- ❖ Invarianten sollten im richtigen Kontext formuliert sein.

⇒ Stilregeln für "Gutes Programmieren in OCL"

### ***Info II-Stilfibel für OCL: Stilregeln zum Vermeiden komplexer OCL-Ausdrücke***

- ❖ Modellierer schreiben oft lange OCL-Ausdrücke, indem sie beispielsweise alle Invarianten einer Klasse in einer einzigen Invariante schreiben, um sich das wiederholte Aufschreiben des Kontextes zu ersparen.
  - Einschränkungen werden während der Modellierung gebraucht und sollten deshalb leicht zu lesen und zu schreiben sein.
- ❖ **Regel:** Ein komplizierter OCL-Ausdruck muss immer in mehrere Einschränkungen aufgeteilt werden.
- ❖ **Regel:** Jede Invariante, die mehrere boolesche **and** Verknüpfungen enthält, zerlegen wir in einzelne Invarianten, und zwar eine für jeden booleschen Ausdruck.
- ❖ **Regel:** Eine Invariante darf nur durch die kleinste mögliche Anzahl von Assoziationen navigieren
  - OCL-Ausdrücke, die über drei oder mehr Assoziationen hinweg navigieren, sind fast immer zu komplex!

### ***Info II-Stilfibel für OCL: Stilregeln zum Finden des richtigen Kontexts***

- ❖ **Dogma:** OCL-Invarianten sollte man nur im Kontext einer Klasse und ihrer unmittelbaren Nachbarschaft schreiben, nicht im Kontext des gesamten Modells.
- ❖ **Regeln** zum Finden des richtigen Kontexts:
  - Wenn die Invariante den Wert des Attributs einer Klasse beschränkt, dann ist die Klasse, die das Attribut enthält, der richtige Kontext.
  - Wenn die Invariante die Werte der Attribute von mehr als einer Klasse beschränkt, dann ist jede der Klassen, die eins der Attribute enthält, ein guter Kandidat für den Kontext.
  - Aufgrund unserer Kenntnis der Anwendungsdomäne können wir oft schon während der Modellierung in UML eine Klasse ermitteln, die verantwortlich für die Einhaltung einer bestimmten Einschränkung ist.  
Diese Klasse ist dann für diese Einschränkung der richtige Kontext.

### ***Vorgehensweise bei der Modellierung mit UML und OCL***

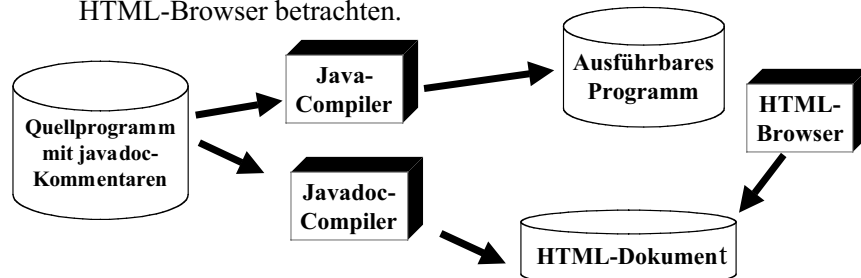
- ✓ **1. Schritt:** Umsetzung der Problembeschreibung in ein UML-Modell
- ✓ **2. Schritt:** Schrittweise Transformation des UML-Modells von Analysemodell bis zum Implementierungsmodell
- ✓ **3. Schritt:** Während der UML-Modellierung andauernde Identifikation von Einschränkungen (Invarianten, Vor- und Nachbedingungen).
- ✓ **4. Schritt:** Formulierung der Einschränkungen als OCL-Ausdrücke (visuell im UML-Modell oder textuell als OCL-Modell)
- ✓ **5. Schritt:** Umstrukturieren und Vereinfachen von OCL-Ausdrücken (mittels der Stilregeln).
- ❖ **6. Schritt:** Überprüfung des OCL-Modells auf Korrektheit und Widerspruchsfreiheit (mithilfe eines OCL-Übersetzers)
  - ⇒ Hauptstudium
- ❖ **7. Schritt:** Übersetzung der UML- und OCL-Modelle in Java und Javadoc

### ***Zusätzliche Informationen zu OCL***

- ❖ <ftp://ftp.omg.org/pub/docs/ad/97-08-08.pdf>  
Die OCL-Sprachreferenz (Version 1.1)
- ❖ <http://www.klasse.nl/ocl/index.htm>  
viele interessante Informationen zu OCL
- ❖ <http://www.software.ibm.com/ad/ocl>  
ein Übersetzer für die OCL-Grammatik.
- ❖ <http://dresden-ocl.sourceforge.net/index.html>  
ein anderes OCL-Compiler-Projekt

## Javadoc

- ❖ **Javadoc** ist ein Werkzeug, mit dem wir Kommentare aus einem Java-Programm in die Programmdokumentation übertragen können.
- ❖ Java-Programmtexte übersetzen wir normalerweise nur mit dem Java-Compiler, um ein ausführbares Programm zu bekommen.
- ❖ Wir können es auch mit dem Javadoc-Compiler übersetzen, um eine strukturierte HTML-Dokumentation des Programms zu erzeugen.
- ❖ Die HTML-Dokumentation können wir mit einem beliebigen HTML-Browser betrachten.



## Javadoc: Syntax

- ❖ Ein Javadoc-Kommentar besteht aus einer Zeichenkette zwischen den Zeichen `/**` und `*/`
- ❖ Beispiel: `/** Dies ist ein Javadoc-Kommentar */`
- ❖ Führende `*`-Zeichen in einem Javadoc-Kommentar werden ignoriert, ebenso wie Leerzeichen und Tabs vor führenden `*`-Zeichen.
- ❖ Javadoc-Kommentare werden nur dann erkannt, wenn sie direkt vor der Deklaration einer Klasse, einer Schnittstelle, eines Konstruktors, einer Methode oder einer Attributvariable platziert werden.
- ❖ Wir besprechen zunächst häufig benutzte Javadoc-"Schlüsselwörter" (document tags) für
  - die Dokumentation von Klassen und Schnittstellen
  - die Dokumentation von Konstruktoren und Methoden
- ❖ Dann zeigen wir, wie man Verträge als Javadoc-Kommentare schreibt.

## Javadoc-Schlüsselwörter zur Kommentierung von Klassen und Schnittstellen

- ❖ Javadoc-Tags werden mit einem vorangestellten `@`-Zeichen markiert.
- ❖ Die hier gezeigten Tags müssen am Anfang einer Kommentarzeile stehen, damit sie erkannt werden können.
  - `@author name-text`
    - Erzeugt einen "Author"-Eintrag für `name` in der Dokumentation.
  - `@version version-text`
    - Erzeugt einen "Version"-Eintrag.
  - `@see reference`
    - Erzeugt einen Querverweis (Hyperlink): "See also reference"
  - `@since since-text`
    - Erzeugt einen "Since"-Eintrag. Wird benutzt, um eine Erweiterung oder Änderung seit der letzten Version zu dokumentieren.
- ❖ In Javadoc-Kommentaren können HTML-Elemente verwendet werden.

## Beispiel: Ein Javadoc-Kommentar für eine Klasse

```
/**
 * Eine Klasse von Bildschirm-Fenstern.
 * Beispiel zur Benutzung:
 * <pre>
 *   Window win = new Window(parent);
 *   win.show();
 * </pre>
 *
 * @author Sami Shaio
 * @version %I%, %E%
 * @see java.awt.BaseWindow
 * @see java.awt.Button
 */
class Window extends BaseWindow {
    ...
}
```

## Javadoc-Schlüsselworte zur Kommentierung von Methoden

- ❖ Methoden-Kommentare können die selben Tags wie Klassen-Kommentare enthalten, sowie zusätzlich:
  - **@param *parameter-name* *description***
    - Beschreibt den Parameter ***parameter-name*** der Methode, wobei die Beschreibung mehrzeilig sein kann. Der erste **@param** erzeugt eine "Parameters"-Sektion in der HTML-Seite.
  - **@return *description***
    - Für Methoden mit Resultat: Erzeugt eine "Returns"-Sektion, die eine Beschreibung des Rückgabewerts enthält, in der HTML-Seite.
  - **@deprecated *deprecated-text***
    - Erzeugt einen Kommentar, der aussagt, dass die Methode veraltet ist und nicht mehr benutzt werden sollte. Üblicherweise gibt man hier eine Methode an, die als Ersatz benutzt werden sollte.
    - Beispiel : **@deprecated Ersetzt durch foo()**

## Beispiel: Ein Javadoc-Kommentar für eine Methode

```
/**
 * Liefert das Zeichen an der spezifizierten Position.
 *
 * @param index Die Indexposition des gesuchten Zeichens.
 *             Güeltige Index-Werte sind
 *             <code>0</code> bis <code>length()-1</code>.
 * @return Das gewünschte Zeichen
 * @see java.lang.Character#charValue()
 */
public char charAt(int index) {
    ...
}
```

## Javadoc-Schlüsselworte zur Kommentierung von Attributvariablen

- ❖ Ein Attributvariablen-Kommentar kann nur die Schlüsselworte **@see**, **@since** und **@deprecated** enthalten

### ❖ Beispiel:

```
/**
 * Die X-Koordinate des Fensters.
 *
 * @see Window
 */
int x = 1263732;
```

## Javadoc-Kommentare können auch HTML enthalten

- ❖ Javadoc-Kommentar mit integriertem HTML-Code:

```
/**
 * Dies ist ein <b>Javadoc</b>-Kommentar.
 */
```

- ❖ Resultat:

Dies ist ein **Javadoc**-Kommentar.

- ❖ Wenn Sie HTML innerhalb eines Javadoc-Kommentars schreiben, verwenden Sie keine HTML-Elemente für Überschriften (heading tags) wie **<h1>**, **<h2>**, usw.
  - Da Javadoc ein HTML-Dokument mit standardisierter Struktur aus Ihren Kommentaren erzeugt, kann die Verwendung von HTML-Elementen, die die Dokumentstruktur ändern (wie z.B. zusätzliche Überschriften), zu Problemen bei der Formatierung des erzeugten Dokuments führen.

## Javadoc-Schlüsselworte zur Kommentierung von Verträgen

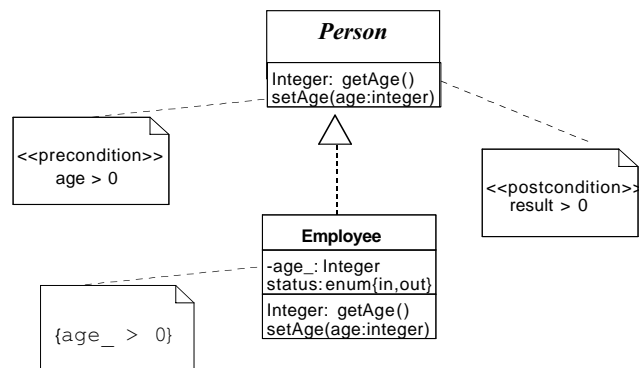
- ❖ Javadoc-Kommentare können auch Einschränkungen enthalten.
  - Invarianten als Teil des Javadoc-Kommentars zu einer Klassen- oder Schnittstellendefinition. Als Kontext dient die kommentierte Klasse bzw. Schnittstelle.
  - Vor- und Nachbedingungen als Teil des Javadoc-Kommentars zu einer Methoden- oder Konstruktordeklaration. Als Kontext dient die kommentierte Methode bzw. der kommentierte Konstruktor.
- ❖ **Schlüsselworte:**
  - @invariant *E***  
Beschreibt eine Invariante, wobei *E* ein OCL-Ausdruck ist.
  - @pre *E***  
Beschreibt eine Vorbedingung, wobei *E* ein OCL-Ausdruck ist.
  - @post *E***  
Beschreibt eine Nachbedingung, wobei *E* ein OCL-Ausdruck ist.
- ❖ **Hinweis:** Bei Verwendung von Javadoc ohne eine passende Erweiterung (**iDoclet**) werden diese Tags ignoriert.

## Benutzung von Javadoc

- ❖ Aufruf:
 

```
javac <Namen der Dateien mit Java-Programmtext>
```
- ❖ **javac** erzeugt aus den im Programmtext enthaltenen Deklarationen und den entsprechenden Kommentaren eine Anzahl von HTML-Seiten, die alle öffentlichen und geschützten Klassen, Schnittstellen, Konstruktoren, Methoden und Attributvariablen dokumentieren.
- ❖ **javac** kann auch eine Dokumentation erstellen, wenn das Java-Programm keine Javadoc-Kommentare enthält. In diesem Fall werden die deklarierten Programmteile unkommentiert aufgelistet.
- ❖ **javac** generiert die folgenden Dateien:
  - Eine **.html**-Datei für jede **.java**-Datei
  - Darstellung der Klassenhierarchie (**overview-tree.html**)
  - Index aller dokumentierten Programmteile (**index-all.html**)

## Beispiel für die Dokumentation von Verträgen: Ausgangspunkt: UML-Modell und OCL-Modell



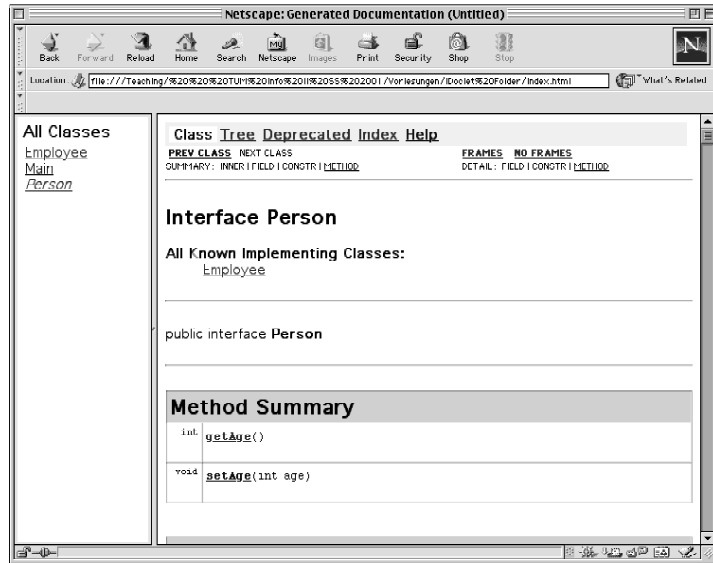
## Java-Code für Person (Vertragsbeschreibung in Javadoc)

```
public interface Person
{
    /**
     * @pre age > 0 // age always positive
     */
    public void setAge(int age);

    /**
     * @post result > 0 // age always positive
     */
    public int getAge();
}
```

Person.html

## Dokumentation der Schnittstelle von Person



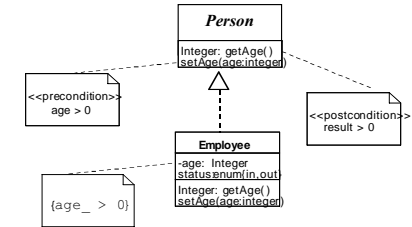
## Java-Code für Employee (Vertragsbeschreibung in Javadoc)

```

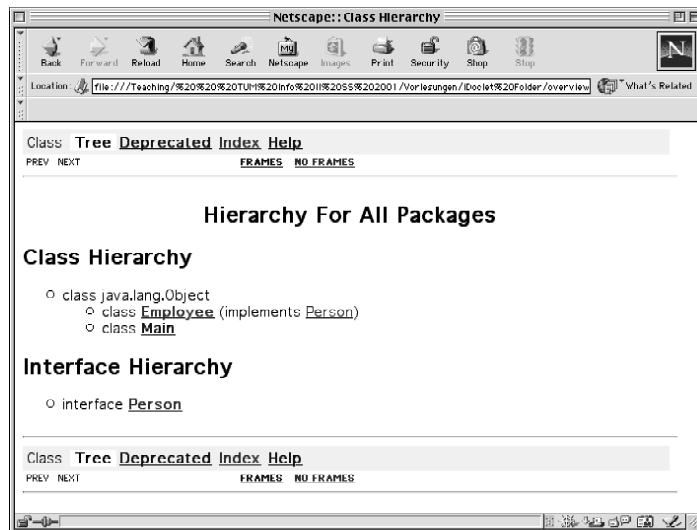
/**
 * @invariant age_ > 0
 */
public class Employee implements Person {
    private int age_;
    /**
     * @pre age > 0
     */
    public Employee( int age ) {
        age_ = age;
    }
    public int getAge() {
        return age_;
    }
    public void setAge( int age ) {
        age_ = age;
    }
}

```

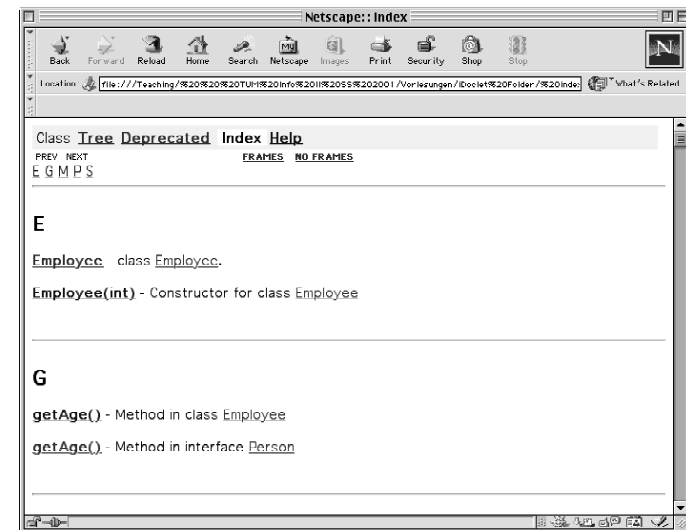
Employee.html



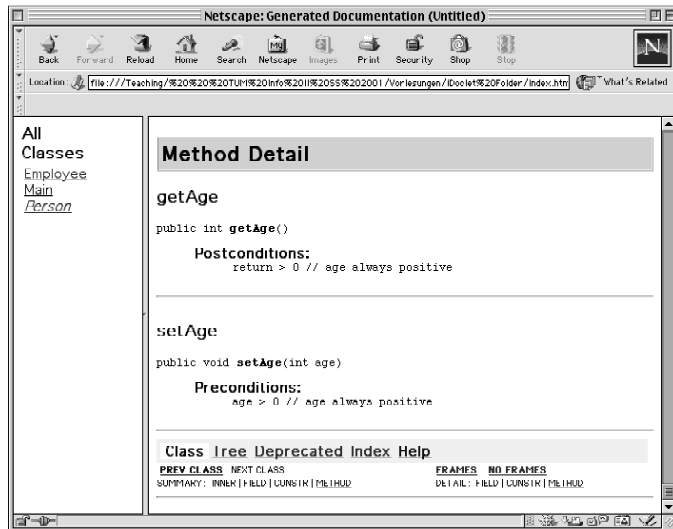
## Dokumentation der gesamten Klassenhierarchie (overview-tree.html)



## Index aller Programmteile (index-all.html)



## Dokumentation des Vertrages von Person (erzeugt mit javadoc und iDoclet)



## Zusätzliche Informationen zu Javadoc

### ❖ Informationen zu Javadoc

- <http://java.sun.com/j2se/javadoc/index.html>  
ausführliche Dokumentation zu javadoc

### ❖ Informationen über Werkzeuge für Verträge

- <http://www.reliable-systems.com/tools/>
  - **iContract**, ein Werkzeug zur automatischen Überprüfung von Verträgen während der Programmausführung. Die Verträge werden mit den Javadoc-Tags **@invariant**, **@pre** und **@post** formuliert.
- <http://icplus.sourceforge.net/>
  - **iDoclet**, eine Javadoc-Erweiterung zur Einbettung von Verträgen in HTML-Dokumentationen.
  - **iControl**, ein interaktives Werkzeug zur Auswahl, welche (mit **iContract** umgesetzten) Einschränkungen überprüft werden sollen.

## Zusammenfassung

- ❖ Bei der Formulierung von OCL-Ausdrücken benutzen wir einige Heuristiken, um die Ausdrücke einfach und übersichtlich zu halten.
- ❖ Javadoc ist eine Sprache und ein Werkzeug zur Erstellung einer strukturierten HTML-Dokumentation für Java-Programme.
- ❖ Wir können Javadoc (mit iDoclet) benutzen, um Verträge im Java-Programm zu dokumentieren.
  - Es gibt Werkzeuge, die die automatische Dokumentation bzw. Überprüfung von Verträgen unterstützen.
- ❖ **UML**, **OCL** und **Javadoc** sind drei Sprachen:
  - Ausgehend von der Problembeschreibung *modellieren* wir das Problem während der Analyse- und Entwurfsphasen in **UML**.
  - Während der detaillierten Entwurfsphase *spezifizieren* wir Einschränkungen und schreiben das Vertragsmodell in **OCL**.
  - Während der Implementationsphase *dokumentieren* wir die Verträge mit **Javadoc**.