

# Einführung in die Informatik II

## Entwurf durch Verträge:

### 1. Verträge und OCL (Einführung)

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

Sommersemester 2001

21. - 23. Mai 2001

## Überblick

- 21. Mai {
  - ❖ Das Konzept des **Entwurfs durch Verträge**
  - ❖ **OCL**: Sprache zur *Spezifikation von Verträgen*
    - Diskussion anhand eines größeren Beispiels (Bonus-System)
      - UML-Klassendiagramm des Bonus-Systems
    - Spezifikation von Invarianten in OCL
    - Spezifikation von Vorbedingungen und Nachbedingungen in OCL
- 23. Mai {
  - Heuristiken für den Einsatz und Gebrauch von OCL
  - Überprüfung von OCL-Spezifikationen
- 28. Mai {
  - ❖ **JavaDoc**: Sprache zur *Dokumentation von Verträgen*
- 30. Mai {
  - ❖ *Verträge für Lösungsklassen* am Beispiel von AVL-Bäumen
  - ❖ **Ausnahmen**: Technik zur Behandlung von *Vertragsbrüchen* zur Laufzeit.

Copyright 2001 Bernd Brügge

Einführung in die Informatik II TUM Sommersemester 2001

13:52:29 2

## Zur Wiederholung

- ❖ **Komponente**: Zentraler Begriff der Modellierung.
  - Eine Komponente ist eine Menge von Attributen und Operationen, die auf diesen Attributen definiert sind.
  - Die Menge aller Operationen einer Komponente bezeichnen wir auch als Dienst (während des System-Entwurfs) oder Schnittstelle (während des detaillierten Entwurfs)
  - Eine Komponente ist die Einheit für die Definition von *Zugriffsrechten* und die *Spezifikation der Schnittstelle*.
- ❖ **Zugriffsrechte**: "Welche Teile der Komponente sind wem zugänglich zu machen?"
  - abstract** → Spezifizierer, **private** → Implementierer
  - protected** → Erweiterer, **public** → Benutzer
- ❖ **Schnittstelle**: "Welche Merkmale werden von einer Klasse angeboten?"
- ❖ **Spezifikation**: Beschreibung der Schnittstelle in einer formalen Notation.

Copyright 2001 Bernd Brügge

Einführung in die Informatik II TUM Sommersemester 2001

13:52:29 3

## Ziele der heutigen Vorlesung

- ❖ Sie verstehen das Konzept *Entwurf durch Verträge*
- ❖ Sie verstehen den Unterschied zwischen graphischer Modellierung (in UML) und textueller Modellierung (in OCL)
- ❖ Sie verstehen die Grundbausteine von OCL (OCL-Datenstrukturen, OCL-Operationen)
- ❖ Anhand einer Problembeschreibung und eines gegebenen UML-Klassendiagramms können sie Invarianten, Vorbedingungen und Nachbedingungen identifizieren, und diese in OCL formulieren.
- ❖ Vertiefende Literatur:
  - Jos Warmer und Anneke Kleppe:  
*"The Object Constraint Language: Precise Modeling with UML"*,  
1999, Addison-Wesley  
⇒ Beispiel: Das "Royal and Loyal"-System

Copyright 2001 Bernd Brügge

Einführung in die Informatik II TUM Sommersemester 2001

13:52:29 4

## Was ist ein Vertrag?

- ❖ "Jeder Brief, der vor 18:00 Uhr in einen Briefkasten eingesteckt wird, wird bis zum nächsten Tag an jede beliebige Adresse geliefert."
- ❖ "Jeder Brief, der vor 18:00 Uhr in einen Briefkasten *auf dem Universitätsgelände der TU München* eingesteckt wird, wird bis zum nächsten Tag an jede Adresse in Deutschland geliefert."
- ❖ "Für 4 Euro wird jeder Brief mit höchstens 80 g Gewicht innerhalb von höchstens 4 Stunden an jede Adresse in München geliefert."
- ❖ "Alle Studentinnen und Studenten, die am Montag oder am Mittwoch um 10:15 im AudiMax sind, hören die Vorlesung 'Einführung in die Informatik II'".
- ❖ "Alle Studentinnen und Studenten, die *im Sommersemester 2001* am Montag oder am Mittwoch um 10:15 im AudiMax sind, hören die Vorlesung 'Einführung in die Informatik II', *wenn das Audio System funktioniert* ".

## Eigenschaften von Verträgen

- ❖ Ein **Vertrag (Kontrakt)** besteht aus mehreren Teilen, die wir *Rechte* und *Verpflichtungen* nennen.  
Ein Vertrag wird zwischen *Anbieter* und *Kunden* geschlossen.
- ❖ **Rechte** sind Spezifikationen für die Benutzung von Diensten einer Komponente (Klasse, Subsystem). Der *Kunde* des Vertrages muss sicherstellen, dass er keins dieser *Rechte* verletzt, wenn er einen Dienst nutzt.
- ❖ **Verpflichtungen** sind Spezifikationen für die korrekte Ausführung eines Dienstes. Der *Anbieter* des Vertrages muss sicherstellen, dass die *Verpflichtungen* erfüllt werden, sofern der Benutzer sich an die *Rechte* hält.
- ❖ Verträge sind nicht erfüllt, wenn entweder *Rechte* nicht eingehalten werden, oder *Verpflichtungen* nicht erfüllt sind.  
Wir sagen dann: "Der Vertrag ist **gebrochen!**"
- ❖ Bei einem Vertragsbruch ist immer klar, wer den Vertrag gebrochen hat:
  - wenn *Rechte* nicht eingehalten werden: Der *Kunde* des Vertrages
  - wenn *Verpflichtungen* nicht erfüllt sind: Der *Anbieter* des Vertrages

## Definition: Objekt-orientierter Vertrag

- ❖ Ein Vertrag gilt für alle von einer Klasse angebotenen Dienste.  
Für die einzelnen Dienste können *Bedingungen* spezifiziert werden:
  - *Vorbedingung*: Eine Bedingung, die gelten muss, bevor der Dienst ausgeführt werden kann.
  - *Nachbedingung*: Eine Bedingung, die (sofern alle Vorbedingungen erfüllt sind) nach der Ausführung des Dienstes gelten muss.
- ❖ Die Klasse, die die Dienste anbietet, ist der **Anbieter** (supplier).
- ❖ Die Klasse, die die Dienste eines Anbieters benutzt, ist der **Kunde** (client oder consumer).
- ❖ **Definition**: Ein **Objekt-orientierter Vertrag** (auch **OO-Vertrag** oder kurz **Vertrag** genannt) ist die Spezifikation der Schnittstelle eines Anbieters, bestehend aus Beschreibungen von Invarianten, sowie Vor- und Nachbedingungen für jeden angebotenen Dienst.
  - Ein OO-Vertrag existiert unabhängig davon, ob ein Kunde existiert (Unterschied zu *Verträgen* im rechtlichen Sinn).

## Definitionen: Vor- und Nachbedingung, Invariante

- ❖ **Vorbedingung**: Ein boolescher Ausdruck für eine Operation, der *vor* der Ausführung der Operation wahr sein muss.
- ❖ **Nachbedingung**: Ein boolescher Ausdruck für eine Operation, der *nach* der Ausführung der Operation wahr sein muss.
- ❖ **Invariante**: Ein boolescher Ausdruck, der für alle Instanzen einer Klasse immer wahr sein muss.
- ❖ **Vertragsbruch**: Eine Vorbedingung, eine Nachbedingung oder eine Invariante eines Vertrages ist verletzt, d.h. ihre Auswertung liefert "falsch" als Ergebnis.
  - Invarianten müssen immer wahr sein, über alle Zustände aller Instanzen der Klasse.
  - Vor- bzw. Nachbedingungen müssen nur zu bestimmten Zeiten wahr sein, nämlich vor bzw. nach der Ausführung der Operation, für die sie gelten.

## Welche Notation verwenden wir für Verträge?

### ❖ Natürliche Sprache:

- *Vorteil*: Natürliche Sprache muss man nicht zusätzlich erlernen.
- *Nachteil*: Bei Verwendung der natürlichen Sprache macht man oft unerlaubterweise implizite Annahmen bzgl. der Rechte und Verpflichtungen zwischen den Vertragspartnern.

### ❖ Mathematische Notation:

- *Vorteil*: Alle, die die mathematische Notation beherrschen, können sich präzise und eindeutig darin ausdrücken.
- *Nachteil*: Wenige Leute, vor allem unter den Kunden von Informatik-Systemen, kennen sich in dieser Notation aus.

### ❖ Was wir wirklich brauchen:

- Eine Sprache für die Formulierung von Einschränkungen (constraints) mit der formalen Strenge der mathematischen Notation, aber auch der Leichtigkeit der natürlichen Sprache.  
⇒ OCL (Object Constraint Language)

## Eigenschaften von OCL

- ❖ OCL wird immer in Verbindung mit einem UML-Modell verwendet:
  - In OCL kann man zusätzliche Information für Modelle ausdrücken, die in UML nicht ausdrückbar sind.
- ❖ OCL ist präzise und eindeutig, kann aber auch gleichzeitig von Leuten gelesen werden können, die nicht Mathematiker sind.
- ❖ OCL ist eine deklarative Sprache, d.h. sie besteht nur aus Definitionen; es gibt insbesondere *keine Zuweisungen*.
- ❖ Ausdrücke in OCL haben keine Seiteneffekte, d.h. die Auswertung einer Vorbedingung, einer Nachbedingung oder einer Invariante verändert den Zustand des Systems nicht.
- ❖ OCL-Ausdrücke sind auswertbar, ohne dass das zugehörige UML-Modell ausgeführt werden muss.
- ❖ Eine vollständige Spezifikation der OCL-Grammatik (in Backus-Naur-Form) findet man im Anhang B in "*The Object Constraint Language*" von Warmer und Kleppe [Warmer].

## Definition: Einschränkung

### ❖ Definition Einschränkung (constraint):

- Eine Beschränkung des zulässigen Wertebereichs für einen Teil eines (objekt-orientierten) Modells oder Informatik-Systems.
- ❖ **Syntax**: Eine Einschränkung wird in UML als OCL-Ausdruck in geschweiften Klammern formuliert, der mit dem UML-Konstrukt verbunden ist, für das die Einschränkung spezifiziert wird.
- ❖ Anderes Wort für Einschränkung: **Zusicherung** (assertion)
- ❖ Einschränkungen erlauben uns, Sachverhalte zu modellieren, die wir nicht in UML ausdrücken können.

## Beispiel: TUMBoS (Problembeschreibung)

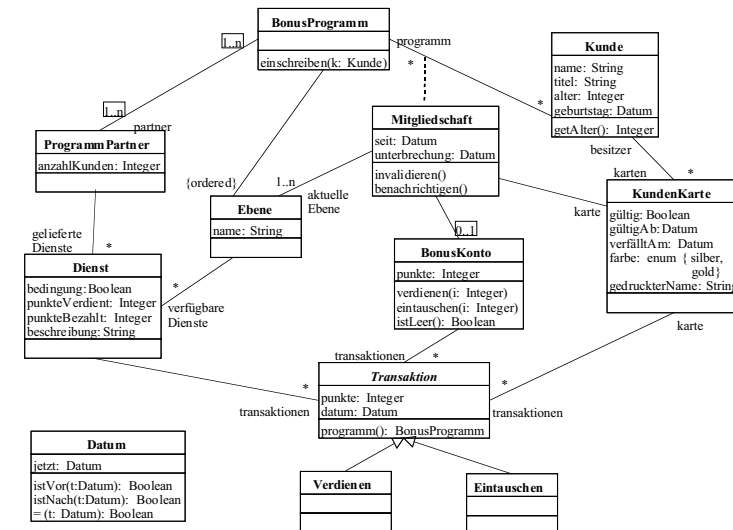
- ❖ Das System **TUMBoS** (TUM-Bonus-System) offeriert seinen Kunden verschiedene Arten von Geschenken, die von Firmen angeboten werden:
  - Bonuspunkte, Meilen für Flüge, Reduzierte Preise für Mietwagen
- ❖ Firmen, die im Rahmen von TUMBoS Geschenke für Kunden anbieten, heißen Programmpartner.
- ❖ Ein Kunde kann Mitglied des Programms werden, und bekommt eine Kundenkarte. Eine Kundenkarte kann nur einem Kunden gehören.
- ❖ Jeder Programmpartner bietet Dienste an, die für eine bestimmte Anzahl von Punkten in Anspruch genommen werden können.
- ❖ Kunden sammeln Punkte für bestimmte Einkäufe. Diese Punkte werden im Bonuskonto angesammelt. Punkte können benutzt werden, um Dienste von einem der Programmpartner in Anspruch zu nehmen.
- ❖ Es gibt zwei Arten von Transaktionen:
  - Verdienen: Man kann beim Benutzen von Diensten Punkte verdienen
  - Eintauschen: Man kann Punkte für Dienstleistungen eintauschen.

## Zusätzliche Anforderungen an TUMBoS

- ❖ Verwaltung aller Transaktionen - Verdienen und Eintauschen - auf allen Punktekontos.
- ❖ Senden von Karten an neue Kunden.
- ❖ Benachrichtigung von Kunden, die sich für eine höhere "Ebene" (Kundenklasse) qualifiziert haben:
  - Dem Kunden wird die passende Karte für die höhere Kundenklasse und einer Beschreibung aller damit zusätzlich nutzbaren Dienste zugeschickt.
- ❖ Abmeldung von Kunden, die nicht mehr am Bonusprogramm teilnehmen wollen:
  - Die angesammelten Bonuspunkte werden auf Null gesetzt, und die Karte wird ungültig gemacht.
- ❖ Invalidierung von Mitgliedschaften, wenn der Kunde seine Karte innerhalb eines Jahres nicht benutzt hat.

## Klassendiagramm für das TUMBoS-System

Nach [Warmer], pp. 12



## Erläuterungen zum TUMBoS-Klassendiagramm

- ❖ Zentrale Klasse des Klassendiagramms ist die Klasse **BonusProgramm**. Ein Informatik-System hat für jedes Bonus-System, das es anbietet, genau eine Instanz dieser Klasse.
- ❖ Eine Firma, die im Rahmen eines **BonusProgramms** den **Kunden Dienste** anbietet, heißt **ProgrammPartner**.
- ❖ Ein **BonusProgramm** kann viele **ProgrammPartner** haben.
- ❖ Ein **ProgrammPartner** kann viele **Dienste** anbieten
- ❖ **Dienste** können für eine bestimmte Anzahl von Punkten *eingetauscht* werden, oder man *verdient* für einen bestimmten **Dienst** eine Anzahl von Punkten. Die Anzahl der Punkte wird im **BonusKonto** verwaltet.

## Instanziierung von TUMBoS: "Treueprogramm Gold & Silber"

- ❖ Das "Treueprogramm Gold & Silber" hat vier Programmpartner:
  - den Supermarkt **Schalldi**
  - die Tankstellenkette **Oktoni**
  - die Mietwagenfirma **Rostig**
  - die Fluggesellschaft **Luftig**
- ❖ Punkte kann man wie folgt verdienen und eintauschen:
  - Bei **Schalldi** bekommt man 5 Punkte für jeden Kauf über 25 Euro.
  - Bei **Oktoni** kriegt man 5 Punkte auf jeden Einkauf.
  - Bei **Rostig** bekommt man 20 Punkte pro ausgegebenen 100 Euro.
  - Bei **Luftig** bekommt man 1 Punkt pro 100 Kilometer auf einem bezahlten Flug.
  - Für 20000 Punkte gibt es einen Freiflug nach Timbuktu.

## "Treueprogramm Gold & Silber" (2)

- ❖ Es gibt zwei verschiedene Arten von Karten: Silber und Gold.
- ❖ Bedingungen, um eine Goldkarte zu kriegen:
  - drei aufeinanderfolgende Mitgliedsjahre mit mehr als 5000 Euro an jährlichen Einkäufen, oder
  - ein Jahr Mitgliedschaft, in dem der Kunde mehr als 15000 Euro ausgegeben hat.
- ❖ Zusätzliche Dienste für Inhaber einer Goldkarte:
  - Alle 2 Monate gibt es bei Schaldi ein freies Geschenk im Wert von ungefähr 12 Euro.
  - Oktoni bietet bei allen Einkäufe 10% Rabatt.
  - Bei Luftig kann man für den Preis eines Touristenklasse-Fluges in der Geschäftsklasse fliegen.

## Was wir noch modellieren wollen

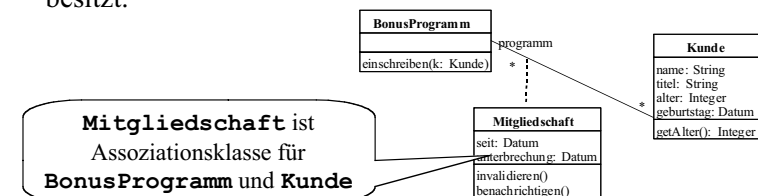
- ❖ Einschränkungen im "Treueprogramm Gold & Silber":
  - Kunden im Bonusprogramm müssen älter als 18 Jahre sein.
  - Der Name auf der Kundenkarte ist identisch mit dem Namen des Kunden.
  - Das Bonusprogramm darf nicht mehr als 4 Programmpartner zur gleichen Zeit haben.
- ❖ Diese Einschränkungen sind im UML-Klassendiagramm nicht vorhanden und sind auch nicht einfach in UML zu modellieren.
- ❖ Wie kann man sie gut modellieren?  
⇒ OCL

## Einschub: Zusätzliche UML-Konzepte

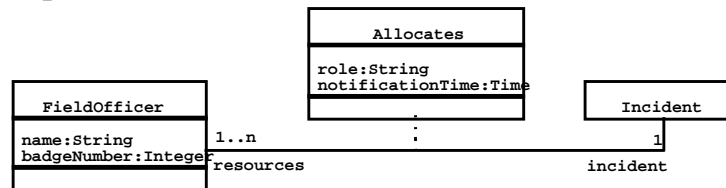
- ❖ Bevor wir jetzt ins Detail gehen: Im Klassendiagramm für das **BonusProgramm** benutzen wir zwei zusätzliche UML-Konzepte:
  - Assoziationsklassen
  - Enumerierte Typen
- ❖ Im weiteren brauchen wir außerdem noch ein anderes UML-Konzept:
  - Stereotypen

## Einschub: Assoziationsklassen in UML

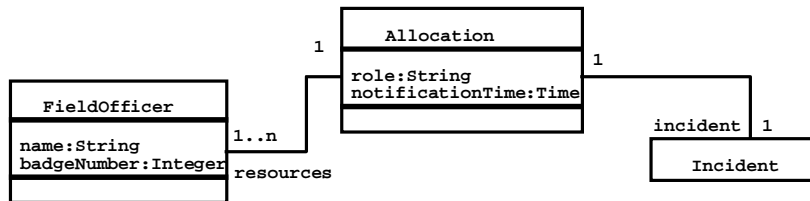
- ❖ Wir haben ja bereits gesehen, dass Assoziationen und Attribute sehr ähnlich sein können.
- ❖ Assoziation und Klassen können sich auch ähneln. Eine Assoziation kann nämlich auch Attribute und Operationen besitzen.
- ❖ Definition **Assoziationsklasse**: Eine Assoziation zwischen zwei Klassen, wobei die Assoziation selber Attribute und Operationen besitzt.



## Beispiel einer Assoziationsklasse



- ❖ Eine Assoziationsklasse kann man immer in eine normale UML-Klasse mit Assoziationen zu den beteiligten Klassen konvertieren:



- ❖ **Wichtig:** Nach der Umsetzung besteht keine direkte Beziehung mehr zwischen den beiden ursprünglich assoziierten Klassen!

## Verwendung von Assoziationsklassen

1. Eine Assoziationsklasse dient zur detaillierteren Modellierung einer *Beziehung* zwischen zwei Klassen. Beispiel: **Mitgliedschaft**

- **Verwendung als eigenständige Klasse (schlecht):**

Ein **Kunde** ist assoziiert mit vielen **Mitgliedschaften**.  
Ein **Bonusprogramm** ist assoziiert mit vielen **Mitgliedschaften**.

- **Verwendung als Beziehung (gut):**

Ein **Kunde** kann *Mitglied* von vielen **BonusProgrammen** *sein*.  
Ein **Bonusprogramm** kann viele **Kunden** *als Mitglieder haben*.

2. Eine Assoziationsklasse kann ohne die Klassen, die sie verbindet, nicht existieren, ist aber mehr als ein einfaches Attribut.

- Eine **Mitgliedschaft** verbindet **BonusProgramm** und **Kunde** und hat eigene Attribute (**seit**, **unterbrechung**) und Operationen (**invalidieren()**, **benachrichtigen()**), die weder zu **BonusProgramm** noch zu **Kunde** gehören.

## Einschub: Enumerationstypen in UML

- ❖ Eine **Enumerationstyp** (auch **Aufzählungstyp**) ist ein spezieller UML-Typ, der oft benutzt wird, wenn ein Attribut nur einige wenige Werte annehmen kann, die sich explizit aufzählen lassen.

- ❖ Beispiele:

- `enum {rot, orange, grün}`
- `enum {silber, gold, platin}`

- ❖ Werte von UML-Enumerationstypen können innerhalb eines OCL-Ausdrucks verwendet werden, wenn man die Werte mit dem Präfixzeichen **#** versieht. (bedeutet hier nicht "**protected**!")

### Kundenkarte

```

Farbe = #gold implies
mitgliedschaft.aktuelleEbene = "gold"
  
```

Beispiel für einen OCL-Ausdruck  
(Syntax wird gleich genauer erklärt)

## Einschub: UML-Stereotypen

- ❖ Ein Ziel beim Entwurf von UML war es, Notationen für die Modellierung einer möglichst breiten Klasse von Informatik-Systemen bereitzustellen.

- Eine begrenzte Anzahl von Notationen kann dieses Ziel nicht erreichen, denn man kann nicht alle Anwendungs- und Lösungsdomänen vorhersehen.

- ❖ Deshalb sind in UML zwei Erweiterungsmechanismen eingebaut. Den einen haben wir schon kennengelernt: **Einschränkungen**. Der zweite Erweiterungsmechanismus heißt **Stereotyp**.

- ❖ **Definition Stereotyp:** UML-Stereotypen sind in doppelte spitze Klammern **<< >>** eingeschlossene Zeichenketten.

- ❖ Einschränkungen und Stereotypen machen UML erweiterbar: Sie erlauben Modellierern, neue Arten von Modellierungskonzepten zu erzeugen.

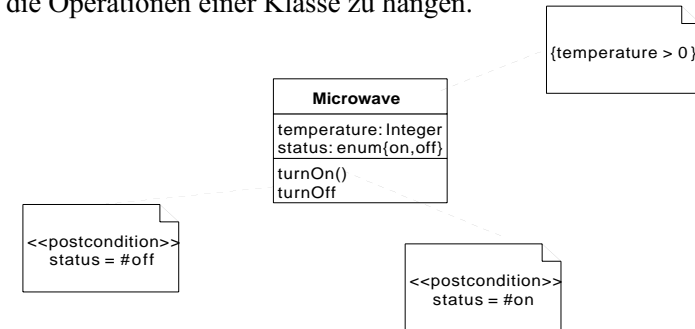
## Beispiele für UML-Stereotypen

### ❖ Beispiele für Stereotypen in UML

- **<<interface>>**:  
Wird in UML benutzt, um eine Schnittstelle zu bezeichnen.
- **<<invariant>>**:  
Wird in UML benutzt, um eine Invariante zu bezeichnen.
- **<<precondition>>**:  
Wird in UML benutzt, um eine Vorbedingung zu bezeichnen.
- **<<postcondition>>**:  
Wird in UML benutzt, um eine Nachbedingung zu bezeichnen.

## Modellierung von Vor- und Nachbedingungen in UML

- ❖ In UML gibt es keine einfache visuelle Möglichkeit, die Vor- und Nachbedingungen eines Vertrages zu spezifizieren.
- ❖ Es ist möglich, die Vor- und Nachbedingungen als UML-Notizen an die Operationen einer Klasse zu hängen.



## Modellierung in OCL

- ❖ Viele UML-Notizen machen Klassendiagramme sehr schnell unleserlich.
- ❖ Wir bevorzugen deshalb OCL für die Spezifikation von Verträgen.
- ❖ OCL unterstützt die Modellierung von Verträgen:
  - Invarianten
  - Vor- und Nachbedingungen
- ❖ Wir schauen uns zunächst die Modellierung von Invarianten an.

## Modellierung von Invarianten

- ❖ "Kunden im Bonusprogramm müssen älter als 18 Jahre sein".
  - 1. allgemeines Konzept:**  
**Invarianten für Attribute einer Klasse** (der Typ eines Attributs ist entweder ein UML-Basistyp oder eine Klasse)
- ❖ "Der Name auf der Kundenkarte ist identisch mit dem Namen des Kunden".
  - 2. allgemeines Konzept:**  
**Invarianten für Attribute von assoziierten Klassen**
- ❖ "Das Bonusprogramm darf nicht mehr als 4 Programmpartner zur gleichen Zeit haben".
  - 3. allgemeines Konzept:**  
**Invarianten für Assoziationen mit Multiplizität 1-n**
- ❖ Wir besprechen jedes dieser Konzepte einzeln, angefangen mit Invarianten für Attribute einer Klasse

## Invarianten für einfache Attribute einer Klasse

- ❖ **Beispiel:**  
"Kunden im Bonusprogramm müssen älter als 18 Jahre sein."
- ❖ Diese Invariante können wir als OCL-Ausdruck schreiben:  
**context Kunde inv: alter >= 18**
- ❖ **context** und **inv** sind reservierte Worte in OCL.
- ❖ **Kunde** und **alter** sind Bezeichner aus dem UML-Klassendiagramm
- ❖ Die Klasse **Kunde** ist hier der **Kontext** der Invariante, d.h. die Invariante wird für diese Klasse formuliert.
- ❖ Der Kontext stellt den Ausgangspunkt für die **Navigation** innerhalb des UML-Diagramms dar.
- ❖ Der boolesche Ausdruck **alter >= 18** ist ein OCL-Ausdruck, der die Invariante für die Klasse **Kunde** beschreibt.

## Andere Notation für OCL-Ausdrücke

- ❖ Eine andere Art, Invarianten in OCL auszudrücken:  
**Kunde**  
**alter >= 18**
- ❖ In dieser Repräsentation unterstreicht man den Klassennamen, der den Kontext der Invariante angibt.
- ❖ Diese Konvention ist nicht mehr Teil des aktuellen UML-OCL-Standards, wird aber noch häufig verwendet:
  - Warmer und Kleppe benutzen diese Notation
  - wir benutzen diese Notation in der Info II-Vorlesung

## Modellierung mit OCL

- ❖ Alle Attribute einer Kontextklasse dürfen als Operanden in einem OCL-Ausdruck verwendet werden. Der folgende OCL-Ausdruck ist also auch möglich (aber nicht unbedingt sinnvoll):

Kunde  
alter >= 18 **and** name = 'Meyer'

**and** ist ein OCL-Schlüsselwort

- ❖ OCL-Ausdrücke dienen zur Spezifikation wichtiger Eigenschaften eines Modells, die entweder graphisch überhaupt nicht ausdrückbar sind, oder textuell einfach leichter zu modellieren sind als graphisch.

## Modellierung von Invarianten

- ✓ "Kunden im Bonusprogramm müssen älter als 18 Jahre sein".
  - 1. allgemeines Konzept:**  
**Invarianten für Attribute einer Klasse** (der Typ eines Attributs ist entweder ein UML-Basistyp oder eine Klasse)
- ➔ "Der Name auf der Kundenkarte ist identisch mit dem Namen des Kunden".
  - 2. allgemeines Konzept:**  
**Invarianten für Attribute von assoziierten Klassen**
- ❖ "Das Bonusprogramm darf nicht mehr als 4 Programmpartner zur gleichen Zeit haben".
  - 3. allgemeines Konzept:**  
**Invarianten für Assoziationen mit Multiplizität 1-n**

## Invarianten für Objekt-Attribute

- ❖ Wenn der Typ eines Attributs keiner der UML-Standardtypen ist (**Boolean**, **Integer**, **String**), sondern selbst eine Klasse, z.B. **Datum**, dann kann man bei der Formulierung der Invariante auch Operationen benutzen, die auf dieser Klasse definiert sind.
- ❖ **Beispiel:** "In der Kundenkarte hat das Attribut **gültigAb** ein früheres Datum als das Attribut **verfälltAm**."
- ❖ **OCL-Ausdruck für diese Invariante:**

```
Kundenkarte  
gültigAb.istVor(verfälltAm)
```

**istVor()** ist eine Operation  
der Klasse **Datum**

## Invarianten für Attribute von assoziierten Klassen

- ❖ In OCL kann man auch Invarianten für Attribute von Klassen definieren, die durch eine Assoziation zu einer anderen Klasse modelliert sind.
  - Als Attribut-Bezeichner kann man dabei den Rollennamen nehmen, der am anderen Assoziationsende (bei der assoziierten Klasse) steht. Falls die assoziierte Klasse keinen Rollennamen hat, wird der Klassenname (mit kleingeschriebenem Anfangsbuchstaben) als Attribut-Bezeichner verwendet.
- ❖ **Beispiel (aus TUMBoS):** Wir erwarten für jede **Kundenkarte**, dass ihr Attribut **gedruckterName** den konkatenierten Attributen **titel** und **name** der assoziierten Instanz der Klasse **Kunde** entspricht.
- ❖ **Die zugehörige OCL-Invariante lautet:** **besitzer** ist der Rollename für die assoziierte Klasse **Kunde**  
**KundenKarte**  
**gedruckterName = besitzer.titel.concat(besitzer.name)**
  - Bei der Formulierung dieser Invariante gehen wir davon aus, dass die UML-Klasse **String** eine öffentliche Methode **concat()** hat.

## Invarianten über Sammlungen von Objekten

- ❖ Eine **1:n-Assoziation** ( $n$  steht für eine Multiplizität  $> 1$  an einem Ende der Assoziation) verbindet ein einzelnes Objekt mit einer **Sammlung** (collection) von mehreren Objekten der assoziierten Klasse.
- ❖ Manchmal ist es wichtig, Invarianten über Sammlungen zu formulieren, z.B. "Ein Bonusprogramm darf nicht mehr als 4 Programmpartner zur gleichen Zeit haben".
- ❖ **Beispiel:**
  - **Im Klassendiagramm:** Ein **BonusProgramm** hat 1 oder viele **Programmpartner**.
  - **Einschränkung (in OCL):** Das "Treueprogramm Gold & Silber", eine Instanz von **BonusProgramm**, hat eine Verbindung zu 4 Instanzen von Typ **ProgrammPartner** (*Schalldi, Oktani, Rostig, Luftig*)
- ❖ OCL bietet deshalb einen vordefinierten Datentyp **collection** an und offeriert eine Anzahl von Operationen, die wir **Sammlungs-Operationen** (collection operations) nennen.

## OCL-Ausdrücke und Datentypen

- ❖ Einschränkungen werde als OCL-Ausdrücke (OCL expressions) geschrieben.
- ❖ Die Operanden von OCL-Ausdrücken sind Objekte und Merkmale.
- ❖ Jedes OCL-Objekt hat einen OCL-Typ, der die Operationen definiert, die auf diesem Objekt aufgerufen werden können.
- ❖ OCL unterscheidet folgende OCL-Typen (\*):
  - **Vordefinierte Typen:**
    - Basistypen: **Integer**, **Real**, **String** und **Boolean**
    - Sammlungstypen: **Collection**, **Set**, **Bag** und **Sequence**
  - **Benutzer-definierte OCL-Typen:**
    - *Alle* Typen in dem assoziierten UML-Diagramm (jede Klasse oder Schnittstelle) sind automatisch OCL-Typen.
- (\*) Typ und Klasse bedeuten auch in Info II dasselbe (genaue Unterscheidung im Hauptstudium)

## Der OCL-Typ Boolean

- ❖ Die Werte **L** und **O** werden in OCL als **true** und **false** geschrieben. Viele Standard-Operatoren aus der Booleschen Algebra sind in OCL definiert:

Boolescher Operator		OCL Operator
Nicht:	$\neg$	<b>not</b>
Und:	$\wedge$	<b>and</b>
Oder:	$\vee$	<b>or</b>
Implikation:	$\Rightarrow$	<b>implies</b>
Gleichheit:	$=$	<b>equals</b>
Ungleichheit:	$\neq$	<b>xor</b>

**xor** ist ein binärer boolescher Operator, der **true** ergibt, wenn seine beiden Operanden vom Typ **Boolean** nicht denselben Wert haben; sonst gibt er den Wert **false**.

## OCL-Ausdrücke vom Typ Boolean

- ❖ **Beispiel (aus TUMBoS):**
  - **Einschränkung:** "Ein Kunde kann an einem Dienst nur Punkte verdienen, wenn er diesen Dienst nicht durch das Eintauschen von Punkten bezahlt hat."
  - **andere Formulierung:** "Ein Kunde kann in einem Dienst keine Punkte verdienen, wenn er den Dienst durch das Eintauschen von Punkten bezahlt hat."

- ❖ **OCL-Ausdruck:**

### Dienst

**(punkteVerdient > 0) implies  
(punkteBezahlt = 0)**

## Die OCL Typen Integer und Real

- ❖ Der Typ **Integer** repräsentiert die ganzen Zahlen aus der Mathematik.
  - Weil OCL eine Modellierungssprache ist, gibt es keine obere Grenze für **Integer**-Werte.
- ❖ Der Typ **Real** repräsentiert die reellen Zahlen aus der Mathematik
- ❖ Für **Integer** und **Real** gelten die aus der Mathematik bekannten binären Operationen: Addition (+), Subtraktion (-), Multiplikation(\*), Division(/), Modulo (**mod**) und ganzzahlige Division (**div**).
  - Beispiel: **2654 \* 4.3 + 101**
- ❖ **Integer**- und **Real**-Zahlen können mit den bekannten arithmetischen Vergleichsoperationen direkt verglichen werden:
  - Beispiel: **(12 > 22.7) = false**

## Weitere Operationen auf OCL-Typen Integer und Real

- ❖ **Unäre Operatoren:**
  - **abs:** Ergibt den Betrag (ohne Vorzeichen) eines Wertes
    - Beispiel: **(-8.9).abs = 8.9**
  - **floor:** Rundet einen Wert vom Typ **Real** nach unten auf den nächstniedrigeren Wert vom Typ **Integer** ab
    - Beispiele: **(-2.4).floor = 3**, **(4.6).floor = 4**
  - **round:** Rundet den Wert einer **Real**-Zahl auf die nächste **Integer**-Zahl.
    - Beispiel: **(4.6).round = 5**
- ❖ **Binäre Operatoren:**
  - **max:** Ermittelt das Maximum von 2 Zahlen.
    - Beispiel: **33.max(12) = 33**
  - **min:** Ermittelt das Minimum von 2 Zahlen.
    - Beispiel: **(33.7).min(12) = 12**

## Werttypen und Objekttypen

- ❖ OCL unterscheidet zwischen Werte- und Objekttypen. Beides sind OCL-Typen, d.h., sie sind beide instanzierbar.
- ❖ **Werttyp:** Der Wert der Instanz eines Wertetyps kann sich niemals ändern.
  - Beispiel: **Integer** ist ein Werte-Typ. Die Ganzzahl 1 (Eine Instanz vom Typ **Integer**) kann niemals ihren Wert ändern und zur 2 werden. (Die 2 ist eine andere Instanz vom Typ **Integer**)
- ❖ **Objekttyp:** Der Wert der Instanz eines Objekttyps kann sich ändern.
  - **Beispiel (aus TUMBoS):** Die Klasse **Kunde** aus einem UML-Modell ist ein Objekttyp. Eine Instanz vom Typ **Kunde** kann die Werte ihrer Attribute ändern, und trotzdem bleibt sie dieselbe Instanz.
- ❖ Alle OCL-Basistypen und Sammlungstypen sind Wertetypen. Benutzer-definierte OCL-Typen (aus dem UML-Modell) sind entweder Werte- oder Objekttypen.

## OCL-Sammlungstypen

- ❖ Der OCL-Typ **Collection** ist die generische Oberklasse von Sammlungen von Objekten mit einem Element-Typ T.
- ❖ Unterklassen von **Collection** sind
  - **Set:** Eine Menge im mathematischen Sinne. Jedes Element kann nur einmal auftreten.
  - **Bag:** Eine Sammlung, in der Elemente mehr als einmal auftreten können (auch Multimenge genannt).
  - **Sequence:** Eine Multimenge, in der die Elemente geordnet sind.
- ❖ Der Elementtyp T kann entweder ein Werttyp (**Integer**, **String**, **Boolean**) oder ein Objekttyp (z.B. **Person**, **Mitgliedschaft**, **Kunde**) sein.
- ❖ Beispiele für Sammlungstypen:
  - **Set (Integer):** eine Menge von Ganzzahlen
  - **Bag (Person):** eine Multimenge von Personen
  - **Sequence (Kunde):** eine Sequenz von Kunden

## OCL-Operationen für OCL-Sammlungstypen (1)

- ❖ Auf vordefinierten OCL-Typen sind **OCL-Operationen** anwendbar.
- ❖ OCL-Operationen zum Ermitteln von Eigenschaften von Sammlungen:
  - size: Integer**  
Anzahl der Elemente in der Sammlung
  - includes(o:OclAny): Boolean**  
Wahr, wenn ein Element **o** in der Sammlung ist
  - count(o:OclAny): Integer**  
Anzahl, wie oft das Element **o** in der Sammlung enthalten ist
  - isEmpty: Boolean**  
Wahr, wenn die Sammlung leer ist.
  - notEmpty: Boolean**  
Wahr, wenn die Sammlung nicht leer ist.
- ❖ **Bemerkung:** Der OCL-Typ **OclAny** ist der allgemeinste OCL-Typ. Er umfasst alle anderen OCL-Typen.

## OCL-Operationen für OCL-Sammlungstypen (2)

- ❖ Operationen zur Erzeugung neuer Sammlungen (Ergebnis wieder eine Sammlung):
  - union(c1:Collection)**  
Vereinigung der Sammlungen mit der Sammlung **c2**
  - intersection(c2:Collection)**  
Durchschnitt der Sammlung mit Sammlung **c2** (enthält nur die Elemente, die sowohl in der Sammlung als auch in **c2** auftreten).
  - including(o:OclAny)**  
Sammlung mit allen Elementen der ursprünglichen Sammlung und dem Element **o**
  - select(expr:OclExpression)**  
Teilmenge mit allen Elementen der Sammlung, für die der OCL-Ausdruck **expr** wahr ist
- ❖ **Bemerkung:** Alle OCL-Ausdrücke sind Werte des Typs **OclExpression**.

## Aufruf von OCL-Operationen auf OCL-Sammlungstypen

- ❖ **Pfeil-Notation:** Der Aufruf einer OCL-Sammlungsoperation besteht aus der Konkatenation des Sammlungsbezeichners (z.B. **partner**) mit einem Pfeil **->** und dem Operationsbezeichner (z.B. **size**).
- ❖ **Beispiel (aus TUMBoS):** "Ein Bonusprogramm hat immer genau 4 Programmpartner."
- ❖ Mithilfe der Sammlungsoperation **size** (Kardinalität einer Sammlung) können wir diese Anforderung folgendermaßen ausdrücken:
- ❖ **OCL-Invariante (Rollenname als Sammlungsbezeichner):**  
BonusProgramm  
**partner->size = 4**
- ❖ **anders formuliert (Klassenname als Sammlungsbezeichner):**  
BonusProgramm  
**programmPartner->size = 4**

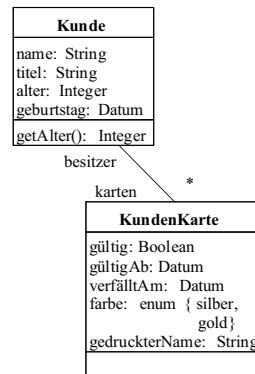
## Wie bekommt man eine OCL-Sammlung?

- ❖ Erzeugung einer Sammlung durch explizite Aufzählung der Elemente
- ❖ Durch Navigation entlang einer oder mehrerer 1:n-Assoziationen:
  - Bei Navigation entlang einer einzelnen 1:n-Assoziation erhält man eine **Menge**.
  - Bei Navigation entlang einer Folge von mehreren 1:n-Assoziationen erhält man eine **Multimenge**.
  - Bei Navigation entlang einer 1:n-Assoziation mit der Einschränkung **{ordered}** erhält man eine **Sequenz**.
- ❖ Durch das Aufrufen einer OCL-Sammlungsoperation, die eine andere Sammlung als Ergebnis liefert

## Navigation durch eine 1:n-Assoziation

- ❖ Die einmalige Navigation durch eine 1:n-Assoziation ergibt eine **Menge**.
- ❖ Beispiel:  
Kunde  
**karten->size <= 4**

karten bezeichnet eine Menge von KundenKarten

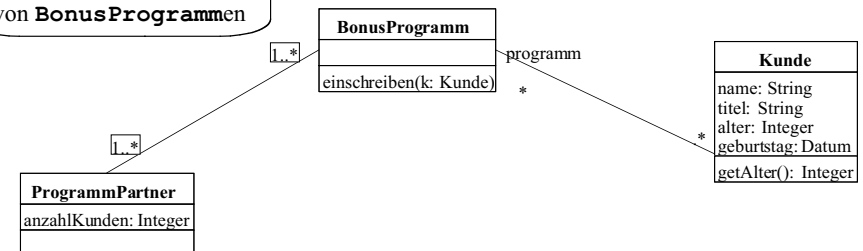


## Navigation durch mehrere 1:n-Assoziationen

- ❖ Wenn man durch mehr als eine 1:n-Assoziation nacheinander hindurchnavigiert, bekommt man eine **Multimenge**.
- ❖ Beispiel:  
ProgrammPartner  
**anzahlKunden = bonusProgramm.kunde->size**

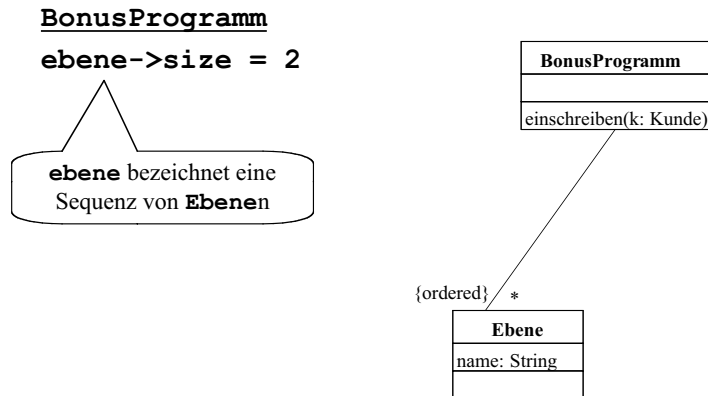
bonusProgramm bezeichnet eine Menge von BonusProgrammen

kunde bezeichnet eine Multimenge von Kunden



## Navigation durch eingeschränkte Assoziation

- ❖ Wenn man durch eine Assoziation mit der Einschränkung **{ordered}** navigiert, bekommt man eine *Sequenz*.
- ❖ Beispiel:



## Unterschied zwischen Set (Menge) und Bag (Multimenge)

- ❖ Schauen wir uns einmal das Attribut **anzahlKunden** in der Klasse **ProgrammPartner** an.
  - Dieses Attribut soll die Menge der Kunden angeben, die in mehr als einem **BonusProgramm** teilnehmen.
- ❖ OCL:  
ProgrammPartner  
**anzahlKunden = bonusProgramm.kunde->size**

### Problem:

- Ein Kunde kann an mehr als einem Bonusprogramm teilnehmen. Ein Kunde kann also mehr als einmal in der Sammlung **bonusProgramm.kunde** auftauchen.
  - **bonusProgramm.kunde** ist also eine Multimenge und keine Menge.
- In unserem obigen OCL-Ausdruck können Kunden also doppelt (oder noch öfter) gezählt werden, was wir natürlich nicht wollen.

## Konvertierungen zwischen OCL-Sammlungen

- ❖ OCL bietet Operationen zur Konvertierung von OCL-Sammlungen:
  - **asSet**  
Transformiert eine Multimenge bzw. Sequenz in eine Menge
  - **asBag**  
Transformiert eine Menge bzw. Sequenz in eine Multimenge
  - **asSequence**  
Transformiert eine Menge bzw. Multimenge in eine Sequenz.
- ❖ Mit **asSet** können wir den inkorrekten Ausdruck

### ProgrammPartner

**anzahlKunden = bonusProgramm.kunde->size**

verbessern, indem wir alle mehrfach enthaltenen Kunden nur einmal zählen:

### ProgrammPartner

**anzahlKunden = bonusProgramm.kunde->asSet->size**

## Operationen auf dem OCL-Typ Sequence

**first: T**

Das erste Element einer Sequenz

**last: T**

Das letzte Element einer Sequenz

**at(index: Integer): T**

Das Element mit dem Index **index** in einer Sequenz

**Beispiel (aus TUMBoS):** "Die erste Ebene, die man in einem Bonusprogramm erreichen kann, hat den Namen 'Silber'."

**Zugehörige OCL-Invariante:**

BonusProgramm:

**ebene->first.name = "Silber"**

## Noch ein Beispiel für die Benutzung von Set

### select(expr: OCLExpression)

Die Teilmenge aller Elemente der Sammlung, für die der OCL-Ausdruck **expr** wahr ist

- ❖ **Beispiel (aus TUMBoS):** "Die Anzahl der gültigen Karten, die ein Kunde besitzt, muss der Anzahl der Bonusprogramme entsprechen, in denen der Kunde Mitglied ist."

### ❖ OCL Invariante:

#### Kunde

```
programm->size =
karten->select(gültig = true)->size
```

## OCL-Modelle gehören immer zu UML-Modellen

- ❖ **Definition OCL-Modell:** Die Menge aller OCL-Ausdrücke, die zu einem UML-Modell gehört.
- ❖ Es gibt zwei Möglichkeiten, die Zugehörigkeit von OCL-Modell und UML-Modell auszudrücken.

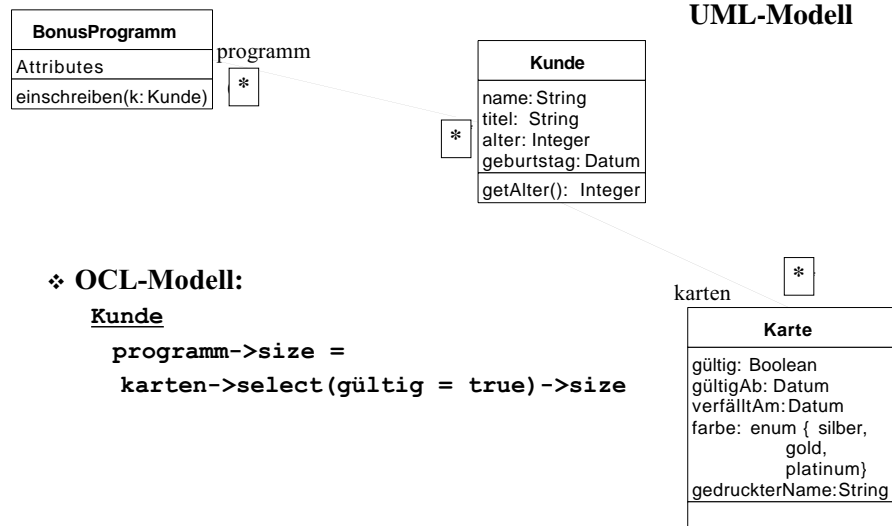
### 1. Alle OCL-Ausdrücke sind UML-Einschränkungen

- *Vorteil:* Alles in einer Datei
- *Nachteil:* Das UML-Modell wird schnell unleserlich

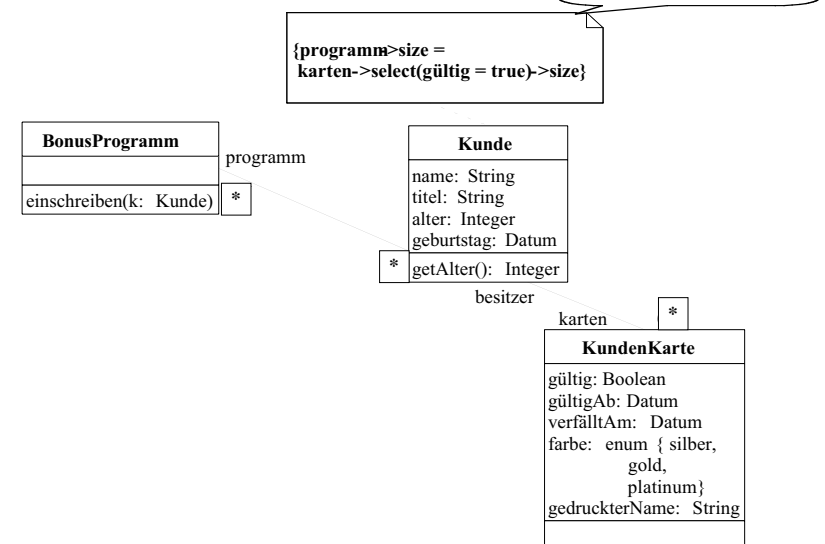
### 2. Alle OCL-Ausdrücke werden als separater Text gespeichert

- *Vorteil:* Das UML-Modell bleibt leserlich
- *Nachteil:* UML-Modell und OCL-Modell sind in zwei verschiedenen Dateien gespeichert, was zu Konsistenz-Problemen führen kann (Namensänderungen im UML-Modell müssen explizit auch im OCL-Modell nachgeführt werden, und umgekehrt).

## OCL-Modell als separater Text



## OCL-Modell als Teil vom UML-Modell



## Modellierung in OCL

- ❖ OCL unterstützt die Modellierung von
  - √ Invarianten
  - » Vor- und Nachbedingungen

## Kontext bei Vor- und Nachbedingungen

- ❖ Der Kontext einer Vor- oder Nachbedingung ist eine Operation, die wir die Kontext-Operation nennen.
- ❖ Allgemeine Form einer Vor- und Nachbedingung:  
**Komplette Signatur der Operation**  
**pre: OCL-Ausdruck**  
**post: OCL-Ausdruck**
- ❖ **pre** und **post** sind Schlüsselworte in OCL
- ❖ Die Signatur wird aus dem UML-Modell übernommen
- ❖ Bezeichner in den OCL-Ausdrücken sind reservierte OCL-Worte oder Namen von Klassen, Attributen, Operationen aus dem UML-Modell.
- ❖ Die formalen Parameter der Signatur können bei der Formulierung von Vor- und Nachbedingungen auch benutzt werden.

## Nachbedingungen können zeitliche Aspekte beschreiben

- ❖ Bei der Spezifikation von Nachbedingungen spielen auch zeitliche Aspekte eine Rolle, wenn wir uns auf den Zustand des Objekts unmittelbar vor oder nach der Operationsausführung beziehen.
- ❖ Wir benutzen zwei spezielle OCL-Schlüsselworte, um temporale Aspekte in Nachbedingungen zu modellieren.

### @pre

Repräsentiert, an einen beliebigen Attribut-Bezeichner angefügt, den Wert, den das Attribut vor Ausführung der Operation hatte.

Beispiel: **kunde@pre** bezeichnet die Menge aller Kunden vor der Ausführung einer Operation.

### result

Das Resultat der Operation unmittelbar nach der Ausführung

Beispiel für Spezifikation von Vor- und Nachbedingungen:

**Typ1::operation(arg:Typ2): Resultatstyp**

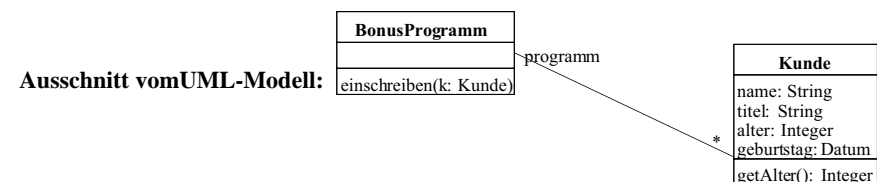
**pre: arg.attr1 = true**

**post: result = (arg.attr2@pre + 1)**

## Modellierung von Vor- und Nachbedingungen in OCL

**Beispiel (aus TUMBoS):** "Ein Kunde, der neu in ein Bonusprogramm eingeschrieben wird, darf vorher nicht Mitglied in diesem Programm sein. Nach der Einschreibung ist er Mitglied des Programms."

- ❖ Diese Einschränkung können wir als Vor- und Nachbedingung in OCL formulieren, und zwar für die Operation **einschreiben()** in der Klasse **BonusProgramm**



**BonusProgramm::einschreiben(k: Kunde)**

**pre: not kunde->includes(k)**

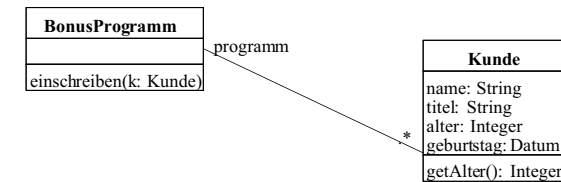
**post: kunde = kunde@pre->including(k)**

## Noch einmal in aller Ruhe

❖ Eine Operationsspezifikation erfolgt in drei Schritten:

1. Schritt: Spezifikation der Signatur
2. Schritt: Spezifikation der Vorbedingungen
3. Schritt: Spezifikation der Nachbedingungen

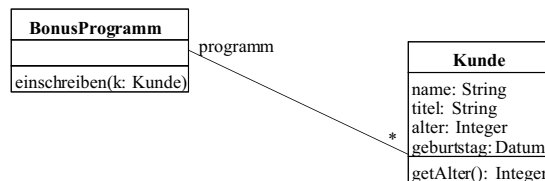
## 1. Schritt: Spezifikation der Signatur



Dafür brauchen wir uns nur die Klasse **BonusProgramm** anzuschauen:

**BonusProgramm::einschreiben(k:Kunde)**

## 2. Schritt: Spezifikation der Vorbedingungen



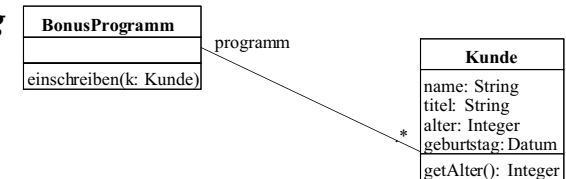
Jetzt prüfen wir, ob der Kunde nicht bereits ein Mitglied im Bonusprogramm ist. Dazu navigieren wir von **BonusProgramm** zur Klasse **Kunde**. Mit Hilfe der Sammlungsoperation **includes()** prüfen wir, ob der gegebene Kunde **k** in der Sammlung ist:

**pre: not kunde->includes(k)**

**Bemerkungen:**

- **includes(o:OclAny): Boolean**  
Wahr, wenn das Objekt **o** in der Sammlung enthalten ist
- **kunde->includes(k)** ist also wahr, wenn **k** in der OCL-Sammlung **kunde** enthalten ist.

## 3. Schritt: Spezifikation der Nachbedingung



❖ Jetzt müssen wir noch sicherstellen, dass die Menge aller Kunden nach der Ausführung der Operationen **einschreiben()** auch wirklich den neuen Kunden enthält:

**post: kunde = kunde@pre->including(k)**

**Bemerkungen:**

- **kunde@pre** ist im Kontext die Menge der Kunden **vor** der Ausführung der Operation **einschreiben()**.
- Der OCL-Ausdruck ist nur wahr, wenn **kunde** - die Menge der Kunden **nach** der Ausführung von **einschreiben()** - den Kunden **k** enthält.

## Ein weiteres Beispiel für eine Vor- und Nachbedingung

**Beispiel (aus TUMBoS):** "Eine Programmanfrage an eine Transaktion liefert das Bonusprogramm, zu dem die Transaktion gehört."

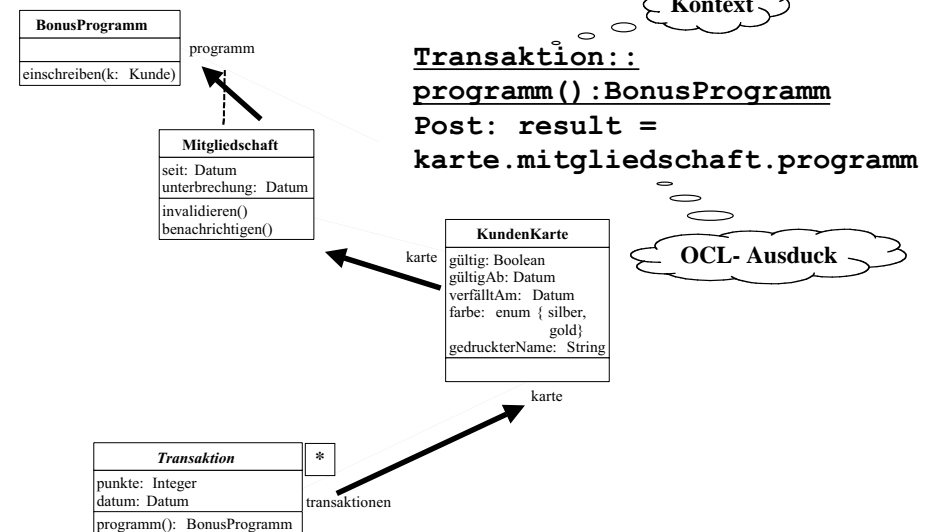
**Transaktion::programm(): BonusProgramm**

**pre: true**  
**post: result = karte.mitgliedschaft.programm**

### Bemerkungen:

- Die Operation hat keine spezielle Vorbedingung: Dies kann man durch **true** ausdrücken (oder die Vorbedingung ganz weglassen)
- **karte** ist die Kundenkarte, mit der die Transaktion initiiert wird.
- **karte.mitgliedschaft** ist die Mitgliedschafts-Beziehung, zur der die Kundenkarte gehört.
- **karte.mitgliedschaft.programm** bezeichnet das Bonusprogramm, dem die die Kundenkarte durch die Mitgliedschafts-Beziehung zugeordnet ist.

## Navigation im UML-Modell



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

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

## Zusammenfassung wichtiger Konzepte

- ❖ **UML** hat zwei **Erweiterungsmechanismen**: Einschränkungen und Stereotypen.
  - Verträge kann man mit Stereotypen und Einschränkungen modellieren
- ❖ **OCL ist eine formale Sprache** zur textuellen Formulierung von Einschränkungen.
- ❖ Mit OCL kann man Verträge formulieren, insbesondere Invarianten sowie Vor- und Nachbedingungen
- ❖ Ein OCL-Modell kann man als Komplement eines UML-Modells ansehen:
  - OCL-Modelle adressieren Aspekte, die sich in graphischen UML-Modellen nicht einfach darstellen lassen, vor allem Einschränkungen, die sich auf mehr als eine Klasse beziehen, und Verträge.
- ❖ Das OCL-Modell sollte man separat vom zugehörigen UML-Modell speichern, sonst wird das UML-Modell schnell unleserlich.