Die **O**bject **C**onstraint **L**anguage

Eine Einführung im Rahmen des SoPras

Frank Broemel

Gliederung

1. Einleitung
2. Ein Beispiel
3. Zusammenfassung der Erkenntnisse und Systematisierung
4. Literatur
5. Verwendung der OCL im SoPra

Die Integration in das SoPra-Vorgehensmodell

- Bisher: Anforderungsdefinition → Analyse → Entwurf
- Entwurf: Basis für die Verteilung der Implementierung

- Fragen dabei:
  a) Wie soll kommentiert werden ?
  b) Wie drückt man sich unmissverständlich aus ?
  (Gruppenmitglieder müssen Kommentare eindeutig interpretieren können)

  Antwort kann OCL sein

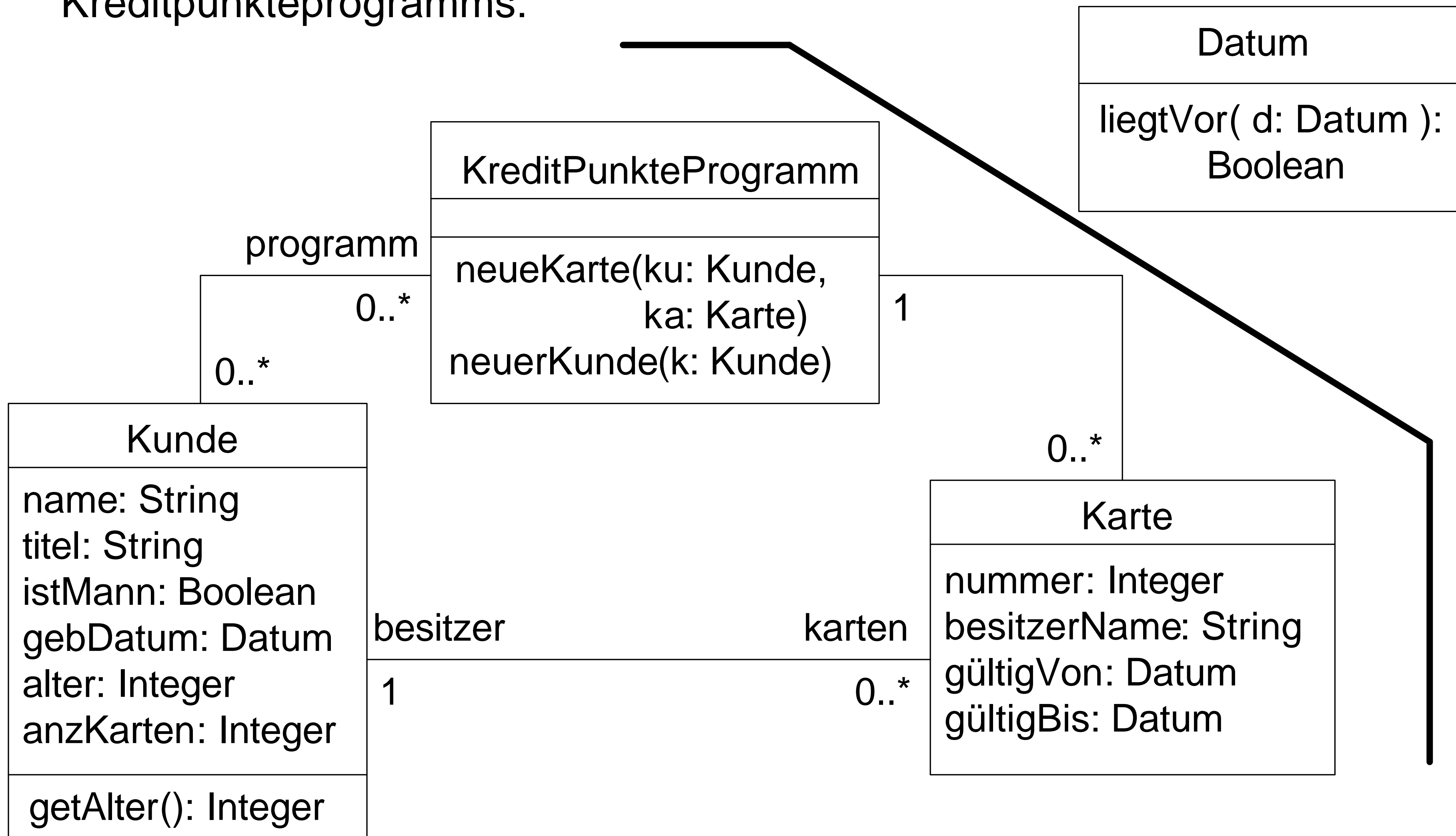
- Hinweise für das Testen

Was ist die OCL ?

- **Constraints** sind Bedingungen
- Formen von Constraints:
  - **Invarianten** : Bedingung, die immer erfüllt sein muss
  - **Vorbedingungen** : Bedingung, die vor Ausführung einer Methode erfüllt sein muss
  - **Nachbedingungen** : Bedingung, die nach Ausführung einer Methode erfüllt sein muss
- Intuitive, an Mathematik orientierte, Sprache
- Beschreibt nur, nicht zur Implementierung gedacht
- Textuelle Ergänzung zu Klassendiagrammen

Ein Beispiel

Kunden erhalten Karten von verschiedenen Firmen im Rahmen eines Kreditpunkteprogramms:



Beispiel: Invarianten für Attribute

Aussage: *„Kunden müssen mindestens 18 sein“*:

```
Context: Kunde
  alter ³ 18
```

← Klasse Kunde aus dem Modell

← Attribut von Kunde

- **Context** gibt die bezogene Klasse aus dem Modell an
- Danach stehen Attribute und Methoden der Klasse zur Verfügung

  ⇒ bilde booleschen Ausdruck (Constraint)

Beispiel: Invarianten für Attribute, 2

Auch Attribute, deren Typen Klassen sind, sind zugreifbar:

```
Context: Karte
   gültigVon.liegtVor(gültigBis)
```

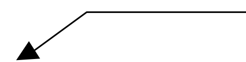
Datum

Methode von Datum
Über **Punktnotation** ist Zugriff auf
Methoden und Attribute von Datum möglich

☹ Achtung bei Methoden: In Constraints dürfen nur reine Abfragen verwendet werde, d.h. Methoden, die Klasse nicht modifizieren

Beispiel: Invarianten für assoziierte Klassen

- Aussage: *„Name auf der Karte soll der gleiche sein, wie der Name des zugehörigen Kunden“*



```
Context: Karte
besitzerName = kunde.name
```

Attribut von Kunde

kleingeschriebener Klassenname

alternativ: `= besitzer.name`

- Auf assoziierte Klassen mittels kleingeschriebenem Klassen- oder Rollennamen zugreifen
- Dann Methoden und Attribute der assoziierten Klasse wieder über Punktnotation zugreifbar

Beispiel: assoziierte Klassen und Mengen

- Beispiel oben: Karte gehört zu genau einem Kunden ⇒ Bezug ist klar
- Was, wenn Multiplizität größer 1 ist ?

```
Context: Kunde
    karten  ....
```

⇒ Keine Aussagen über einzelne Objekte möglich, aber über Menge

- Auf Mengen sind spezielle Operationen definiert, auf die mittels **Pfeilnotation** zugegriffen wird
- Aussage: *„Ein Kunde darf nicht mehr als 5 Karten besitzen“*

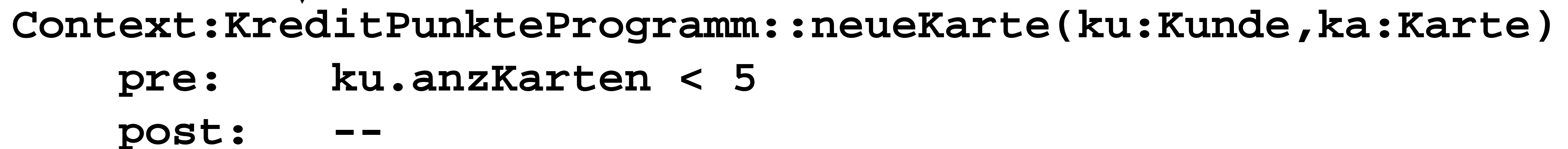
```
Context: Kunde
   karten®size £ 5
```

- Damit auch Aussagen über die Elemente möglich (s.u.)

Beispiel: Vor- und Nachbedingungen

- Aussagen über das Verhalten einer Methode
- Aussage: *„Ein Kunde soll nicht mehr als 5 Karten bekommen“*

Klasse, die Methode enthält　　　　Methodenname mit Parametern



```
Context:KreditPunkteProgramm::neueKarte(ku:Kunde,ka:Karte)
    pre:    ku.anzKarten < 5
    post:   --
```

- Parameter der Methode sind direkt zugreifbar
- Attribute und Methoden der Klasse sind mittels

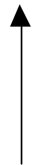
  `self.`<Attribut- / Methodenname>

  zugreifbar

Beispiel: Nachbedingungen

- Besonderheit bei Nachbedingungen: Zugriff auf den Zustand eines Teilausdrucks vor dem Methodenaufruf mit Suffix **@pre**:

```
Context:KreditPunkteProgramm::neueKarte(ku:Kunde,ka:Karte)
   pre:  ku.anzKarten < 5
   post: ku.anzKarten = ku.anzKarten@pre + 1
```



Wert von ku.anzKarten <u>vor</u> Aufruf von neueKarte(...)

Wert von ku.anzKarten <u>nach</u> Aufruf von neueKarte(...)

- Wichtig bei Methoden, die Klasse modifizieren
- @pre bezieht sich nur auf den direkt vorangehenden Teilausdruck. Der Rest des Ausdrucks bezieht sich auf die Objekte nach dem Methodenaufruf.

Zusammenfassung der Erkenntnisse und Systematisierung

Gesehen:

- Constraints beziehen sich als Invarianten oder Vor- / Nachbedingungen auf ein Modell
- Verbindung zum Modell wird über den Kontext hergestellt

Im folgenden:

1. Kontext
2. Aufbau der Constraints
    - Typen
    - Operationen

Kontext für Invarianten

Verbindung zum Modell wird über den Kontext hergestellt:

a) Kontext für Invarianten: eine Klasse

Syntax: **Context:** <Klassenname>
<Constraints>

Danach sind die sog. **Eigenschaften** der Klasse

- Attribute
- Methoden, welche keine Seiteneffekte haben,
- Assoziierte Klassen

in den Constraints verwendbar

- Schlüsselwort **self** zum expliziten Bezug auf Klasse (ist optional, Beispiel folgt)

→ <u>Invariante muss für jede Instanz der Klasse gelten</u>

Kontext für Vor- / Nachbedingungen

b) Kontext für Vor- / Nachbedingungen ist bezogene Methode:

Syntax: **Context:**<Klassenname>**::**<Methodenname>
(<Parm1>**:**<Klasse1>...**):<**Ergebnistyp>
pre: <Constraints für Vorbedingung>
post: <Constraints für Nachbedingung>

Danach sind

- übergebene Parameter
- Ergebnis der Methodenauswertung mittels Schlüsselwort **result**
- Eigenschaften der umgebende Klasse mittels Schlüsselwort **self.**

zugreifbar.

Kontext für Vor- / Nachbedingungen, 2

Beispiel:

```
Context: Kunde::getAlter():Integer
    pre:   --
    post:  result = self.alter
```

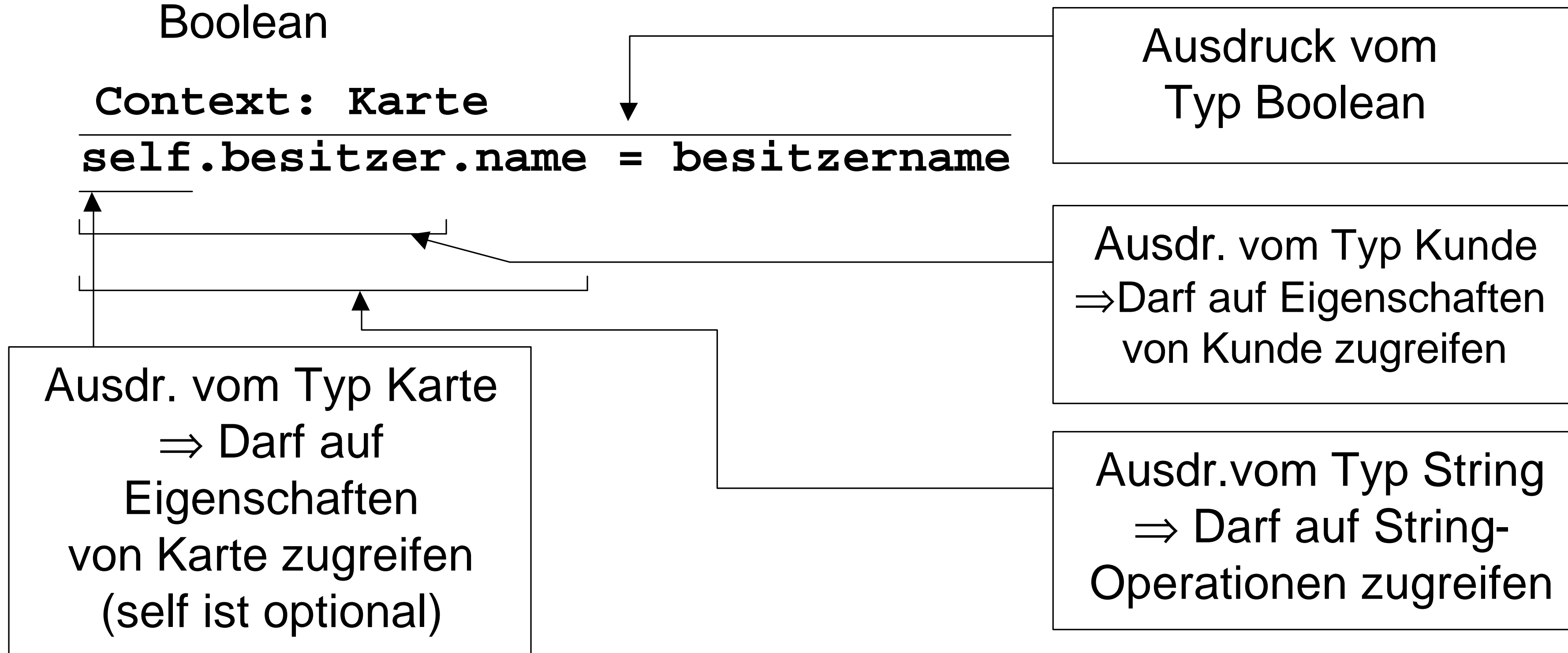
Dieses Beispiel dient nur zur Veranschaulichung der Syntax.

Da **`getAlter()`** nur einen Wert abfragt (also eine der zuvor definierten sog. Eigenschaften ist) und keine Vorbedingung hat, wäre eine Invariante hier erlaubt und auch besser geeignet:

```
Context: Kunde
    getAlter() = alter
```

Wie wird ein OCL-Constraint formuliert ?

- Ein OCL-Constraint wird sukzessive aus OCL-Ausdrücken aufgebaut
- Starten im Kontext. Implizit ist self mit umgebender Klasse als Typ vorhanden
- Jeder OCL-Ausdruck hat einen Typ, der bestimmt, wie der Ausdruck weiter ausgebaut werden kann
- Ein OCL- Ausdruck ist ein OCL-Constraint $\Leftrightarrow$ er ist vom Typ Boolean



Es gibt 3 ‚Arten' von Typen

Es bleibt zu sagen,

1. Welche Typen gibt es ?
2. Welche Operationen erlauben sie ?

Ad 1.:

- Basistypen
- Benutzerdefinierte (oder Modell-) Typen
- Collections
  - Set
  - Bag
  - Sequence

Rest des Vortrags: Hauptsächlich Vorstellung der erlaubten Operationen mit Angabe des Ergebnistyps nach Anwendung

Basistypen

- Ähnlich wie in Java:
  - Real - Integer
  - Boolean - String
- Besonderheit: Integer ist Subtyp von Real
- Erlaubte Operationen: analog zu Java, Details finden sich in UML-Spezifikation [OMG 99, Kap.7]

- Exemplarisch auf Boolean eingehen:
  1. not a a or b a and b a xor b a implies b
  2. a = b a <> b

  a, b sind dabei boolesche Ausdrücke

  Ergebnistyp: Boolean

Basistypen, 2

3. if <boolescher Ausdruck >
   then < Ausdruck1>
   else < Ausdruck2>
   endif

Ergebnistyp: Typ von Ausdruck1 oder Typ von Ausdruck2

Beispiel:

```
Context: Kunde
     titel = ( if   istMann
                      then "Herr"
                      else "Frau"
                 endif )
```

Benutzerdefinierte Typen

- Klassen aus dem Klassendiagramm
- Angesprochen werden Objekte:
  - als Kontext über Klassennamen
  - als assoziierte Objekte mittels kleingeschriebenem Klassen- bzw. Rollennamen
- Zugriff auf sog. **Eigenschaften** über Punktnotation

  Eigenschaften sind:
  - Attribute

    Ergebnistyp = Typ des Attributs
  - Methoden, sofern sie die Klasse nicht modifizieren (Queries)

    Bei Zugriff müssen Klammern immer mitangegeben werden.
    Ergebnistyp = Ergebnistyp der Methode
  - Assoziierte Klassen

    Ergebnistyp bei Multiplizität 1: assoziierte Klasse

    Bei anderer Multiplizität: sog. **Collection** (s.u.)

Collectionarten

- **Set**
  - bekannte mathematische Menge: jedes Element nur einmal
  - Erhält man immer bei Navigation über eine Assoziation mit Multiplizität größer 1

```
Context:Kunde
        karten®size < 5
```

Set von Objekten der Klasse Karte

- **Bag**
  - Multimenge, d.h. Elemente können mehrfach vorkommen
  - Erhält man bei Navigation über mehr als eine Assoziation mit Multiplizität größer 1

```
Context:KreditPunkteProgramm
kunde.karten®size
```

Bag von Karten, die von Kunden besessen werden

Collections

- Collection ist abstrakter Oberbegriff und wird selbst wird nicht verwendet
- Auf Collection-Operationen wird mit Pfeilnotation zugegriffen
- Sets und Bags können auch explizit angegeben werden:
  - Set{1, 4, 2}
  - Bag{1, 1, 4, 4, 4, 2}

- Collections können keine anderen Collections enthalten:

  ~~Set { Set{1, 2}, Set{3, 4} }~~ : gibt es nicht

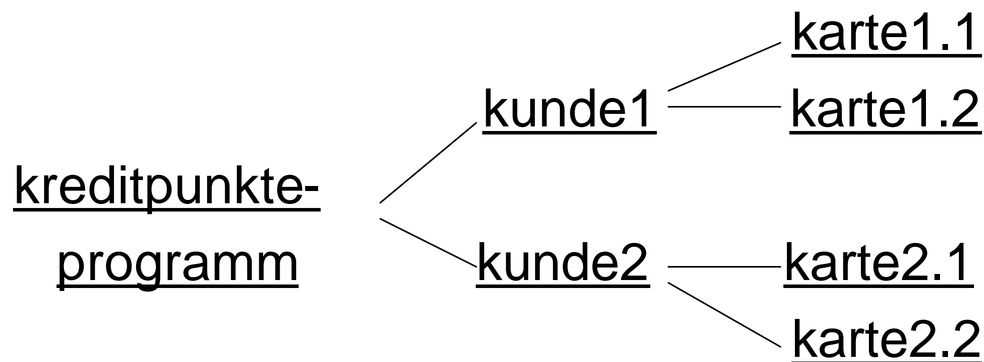
  statt dessen: Set{1, 2, 3, 4}

- Typ einer Collection
  - Konkreter Typ hängt vom Typ der Elemente ab:

    Set{1,2} hat Typ Set(Integer) $\neq$ Set(Real), wie z.B. Set{1.0, 2.0}
  - T Subtyp von T‘ $\Rightarrow$ Set(T) ist Subtyp von Set(T‘)

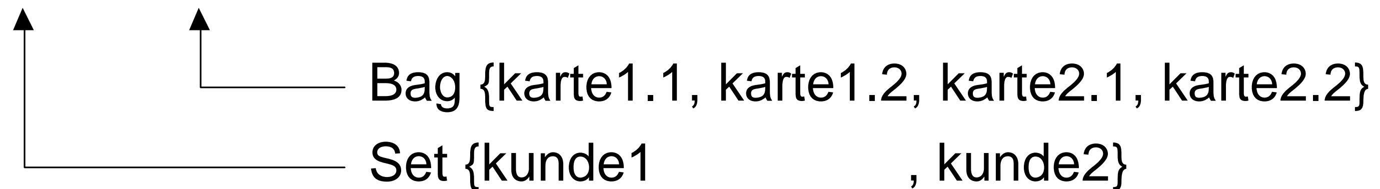
Collection: Bag

Beispiel-Objektdiagramm:



(Assoziationen von Karte zu KreditPunktePrgm. wurden ausgelassen)

```
Context:KreditPunkteProgramm
      kunde.karten ....
```



Collections: Anfragen an eine Collection

Liste der Operationen mit Ergebnistyp und kurzer Erläuterung

- **size** : Integer

  ⇒ Anzahl der Elemente

  Bag{1, 1}®size = 2

- **count(<element>):** Integer

  ⇒ Wie oft kommt das Element vor ?

  Bag{1, 1}®count(1) = 2

- **includes(<element>)** : Boolean

  includesAll(<collection>) : Boolean

  ⇒ Kommt Element bzw. kommen alle Elemente der Collection vor ?

- **isEmpty, notEmpty** : Boolean

Collections: Anfragen an eine Collection, 2

Es stehen auch Quantoren zur Verfügung:

- **exists(<boolescher Ausdruck>)** : Boolean

  ⇒ Existiert ein Element, für das die boolesche Expression zu true ausgewertet werden kann ?

- **forAll(<boolesche Ausdruck>)** : Boolean

  ⇒ Kann die boolesche Expression für alle Elemente zu true ausgewertet werden ?

```
Context Kunde
   karten®forAll(gültigVon.liegtVor(<heute>))
```

Collections: Erzeugen neuer Collections

- Elementtypen der Sets bzw. Bags identisch oder Subtypen
- Ergebnis ist neue Collection

Set und Bag sind ineinander überführbar:

- **`asSet`**: erzeugt eine Set, die die Elemente der ursprünglichen Bag nur einmal enthält. (d.h. Duplikate werden entfernt)
  Beispiel:
  `Bag{1,1,2}®asSet = Set{1,2}`

- **`asBag`**: erzeugt eine Bag, die die gleichen Elemente wie die ursprüngliche Set enthält.
  Beispiel:
  `Set{1,2}®asBag = Bag{1,2}`

Collections: Erzeugen neuer Collections, 2

Liste weiterer Operationen mit kurzen Erläuterungen

- **`union`**: Vereinigung von Set und/oder Bag

  Ergebnistyp: a) Set ⇔ beide Collection sind Sets

  b) Bag ⇔ eine Collection ist Bag

  Beispiel:

  ```
  Set{1,2}®union(Set{1,3}) = Set{1,2,3}
  Bag{1,2}®union(Set{1,3}) = Bag{1,1,2,3}
  ```

- **`intersection`**: Schnitt von Set und/oder Bag

  Ergebnistyp: a) Bag ⇔ beide Collection sind Bags

  b) Set ⇔ eine Collection ist Set

  Beispiel:

  ```
  Bag{1,1}®intersection(Bag{1,1,1,1}) = Bag{1,1}
  Set{1,2}®intersection(Bag{1,1,1}) = Set{1}
  ```

Collections: Erzeugen neuer Collections, 2

- **`including(element)`:** Hinzufügen eines Elements
  Entspricht der Vereinigung (im obigen Sinne) mit einelementiger Set

- **`excluding(element)`:** Entfernen eines Elements
  Ergebnistyp: unverändert
  Im Fall einer Bag, werden alle Vorkommen des Elements entfernt
  Beispiel:

```
Bag{1, 1, 2} ® excluding(1) = Bag{2}
```

Kommentare

- Kommentare in OCL:

  -- <Kommentartext, bis maximal Ende der Zeile>

Literatur

- Literatur zur OCL:
  - [WK 99] Jos Warmer, Anneke Kleppe. The Object Constraint Language. Addison Wesley, Reading, Massachusetts, 1999.

    Vortrag basiert im Wesentlichen auf diesem Buch.
  - [OMG 99] OMG. UML 1.3 Specification. 1999. http://www.omg.org/technology/documents/formal/unified_modeling_language.htm

    Kapitel 7 widmet sich auf 20 Seiten der OCL. Knappe und präzise Beschreibung der Sprache.
  - [HK 99] Martin Hitz, Gerti Kappel. UML @ Work. dpunkt.verlag, 1999.

    Das gesamte Buch ist als Einstieg in die UML und in das Modellieren zu empfehlen. Das Kapitel zur OCL ist sehr kompakt und geht schnell über das hier gezeigte hinaus.

Verwendung der OCL im Rahmen des SoPras

Wie soll OCL eingesetzt werden ?

- Möglichst viele Kommentare durch OCL-Constraints ersetzen
- Auch offensichtliche Dinge angeben
- Nicht zu kompliziert
  - Keine ‚langen Strecken' durch das Klassendiagramm laufen
  - Es muss nicht alles mittels OCL beschrieben werden

  → Einsatz von OCL soll Verständlichkeit <u>erhöhen</u>

Abgabe

- OCL-Constraint auf separatem Dokument, das Bezug auf ein Klassendiagramm nimmt, abgeben
- Alternativ: Constraints in Together angeben und Html-Dokumentation abgeben
- Keine Werkzeugunterstützung ⇒ Korrektheitsüberprüfung durch Betreuer