Formale Modellierung
Vorlesung 10 vom 24.06.14: Formale Modellierung mit UML und OCL

Serge Autexier & Christoph Lüth

Universität Bremen

Sommersemester 2014

Fahrplan

- Teil I: Formale Logik
- Teil II: Spezifikation und Verifikation
  - Formale Modellierung mit der UML und OCL
  - Lineare Temporale Logik
  - Temporale Logik und Modellprüfung
  - Hybride Systeme
  - Zusammenfassung, Rückblick, Ausblick

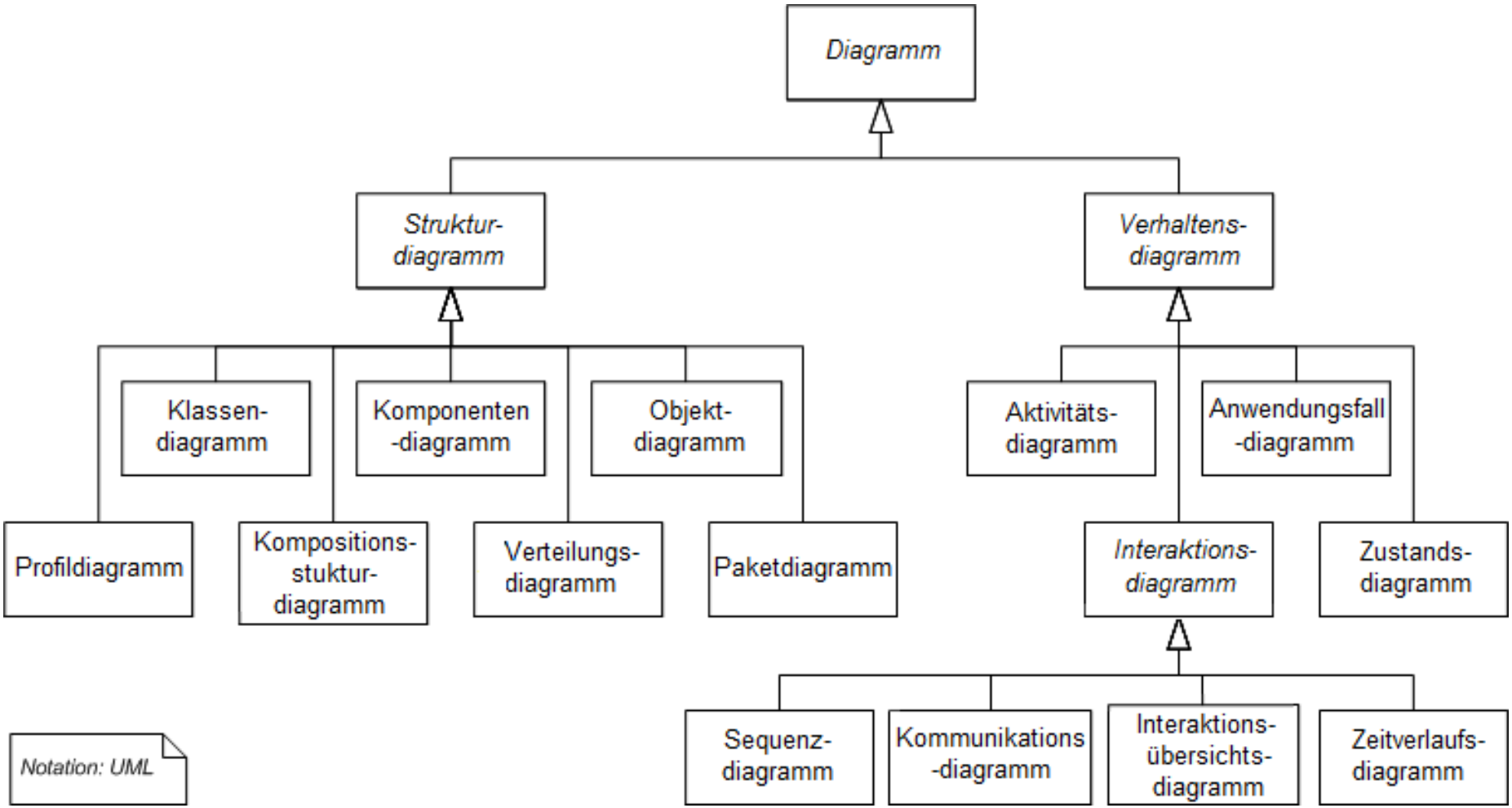
Unified Modeling Language (UML)

- Allgemeine Modellierungssprache
- Spezifikation problemorientiert
- Übersetzung in verschiedene Programmiersprachen möglich
- Nur bestimme Aspekte sind formal

UML als formale Spezifikationssprache

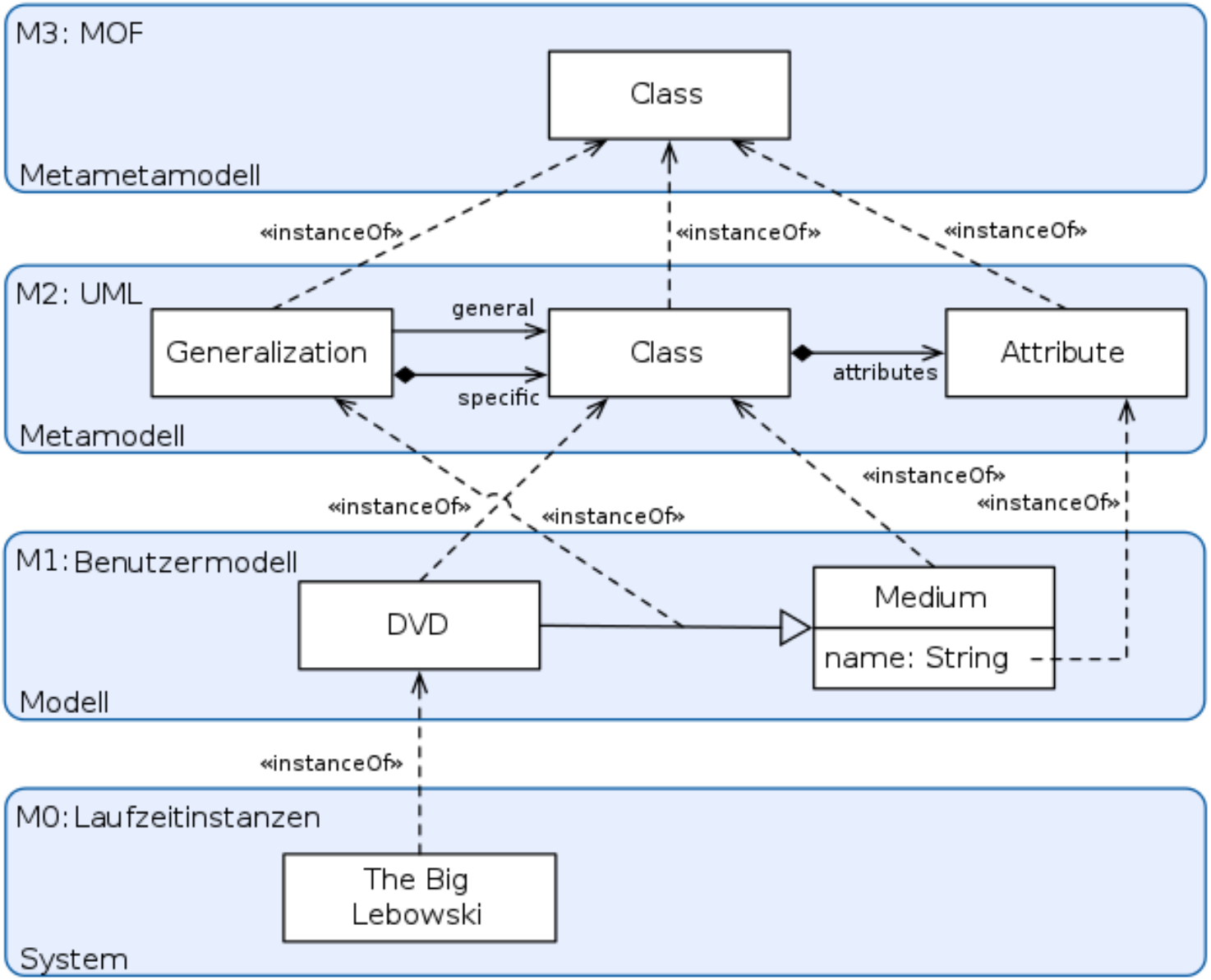
Diagrammtyp	Modellierte Aspekte	Formal
Klassendiagramm	Statische Systemstruktur	Ja
Paketdiagramm	Pakete, Namensräume	Nein
Objektdiagramm	Zustand von Objekten	(Ja)
Kompositionsstrukturdiagramm	Kollaborationen	Nein
Komponentendiagramm	Dynamische Systemstruktur	(Nein)
Verteilungsdiagramm	Implementierungsaspekte	Nein
Use-Case-Diagramm	Ablauf en gros	Nein
Aktivitätsdiagramm	Ablauf en detail	Nein
Zustandsdiagramm	Zustandsübergänge	Ja
Sequenzdiagramm	Kommunikation	Ja
Kommunikationsdiagramm	Struktur der Kommunikation	(Ja)
Zeitverlaufsdiagramm	Echtzeitaspekte	(Ja)

Diagramme in UML 2.3



Quelle: Wikipedia

Semantik der UML: Metamodellierung



Quelle: Wikipedia

OCL

- Object Constraint Langauge
- Mathematisch präzise Sprache für UML
- Entwickelt in den 90ern
- Formale Constraints an UML-Diagrammen
  - Datentypen gegeben durch UML

OCL Basics

- Getypte Sprache
- Dreiwertige Logik
- Ausdrücke immer im Kontext:
  - Invarianten an Klassen, Interfaces, Typen
  - Vor/Nachbedingungen an Operationen oder Methoden

OCL Syntax

- Invarianten:

```
context class
  inv: expr
```

- Vor/Nachbedingungen:

```
context Type :: op(arg1 : Type) : ReturnType
  pre: expr
  post: expr
```

- `expr` ist ein OCL-Ausdruck vom Typ `Boolean`

Undefiniertheit in OCL

- Undefiniertheit propagiert (alle Operationen strikt)

  → OCL-Std. §7.5.11

- Ausnahmen:
  - Boolsche Operatoren (`and`, `or` beidseitig nicht-strikt)
  - Fallunterscheidung
  - Test auf Definiertheit: `oclIsUndefined` mit

$$\texttt{oclIsUndefined(e)} = \begin{cases} \texttt{true} & e = \bot \\ \texttt{false} & \text{otherwise} \end{cases}$$

- Resultierende Logik: dreiwertig

Dreiwertige Logik

- Wahrheitstabelle (starke Kleene-Logik, K_3):

	$\neg$
$\bot$	$\bot$
0	1
1	0

$\wedge$	$\bot$	0	1
$\bot$	$\bot$	0	$\bot$
0	0	0	0
1	$\bot$	0	1

$\vee$	$\bot$	0	1
$\bot$	$\bot$	$\bot$	1
0	$\bot$	0	1
1	1	1	1

$\longrightarrow$	$\bot$	0	1
$\bot$	$\bot$	$\bot$	1
0	1	1	1
1	$\bot$	0	1

$\longleftrightarrow$	$\bot$	0	1
$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	1	0
1	$\bot$	0	1

- Fun Fact: K_3 hat keine Tautologien oder Widersprüche
  - Aussagen, die unter allen Belegungen zu 1 bzw. 0 auswerten
- Es gilt bspw. $[\![\neg A \vee B]\!]_v = [\![A \longrightarrow B]\!]_v$,
  aber nicht $(\neg A \vee B) \longleftrightarrow (A \longrightarrow B)$

OCL Typen

- Basistypen:
  - `Boolean`, `Integer`, `Real`, `String`
  - `OclAny`, `OclType`, `OclVoid`
- Collection types: `Set`, `OrderedSet`, `Bag`, `Sequences`
- Modelltypen

Basistypen und Operationen

- `Integer` ($\mathbb{Z}$) → OCL-Std. §11.5.2
- `Real` ($\mathbb{R}$) → OCL-Std. §11.5.1
  - `Integer` Subklasse von `Real`
  - `round`, `floor` von `Real` nach `Integer`
- `String` (Zeichenketten) → OCL-Std. §11.5.3
  - `substring`, `toReal`, `toInteger`, `characters` etc.
- `Boolean` (Wahrheitswerte) → OCL-Std. §11.5.4
  - `or`, `xor`, `and`, `implies`
  - Sowie Relationen auf `Real`, `Integer`, `String`

Collection Types

- `Set`, `OrderedSet`, `Bag`, `Sequence`
- Operationen auf allen Kollektionen: → OCL-Std. §11.7.1
  - `size`, `includes`, `count`, `isEmpty`, `flatten`
  - Kollektionen werden immer flachgeklopft
- `Set` → OCL-Std. §11.7.2
  - `union`, `intersection`,
- `Bag` → OCL-Std. §11.7.3
  - `union`, `intersection`, `count`
- `Sequence` → OCL-Std. §11.7.4
  - `first`, `last`, `reverse`, `prepend`, `append`

Collection Types: Iteratoren

- Iteratoren: Funktionen höherer Ordnung
- Alle definiert über `iterate` → OCL-Std. §7.7.6:

```
coll-> iterate(elem: Type, acc: Type= expr | expr[elem, acc])
```

```
iterate(e: T, acc: T= v)
{
  acc= v;
  for (Enumeration e= c.elements(); e.hasMoreElements();){
    e= e.nextElement();
    acc.add(expr[e, acc]); // acc= expr[e, acc]
    }
  return acc;
}
```

- Iteratoren sind alle strikt

Modelltypen

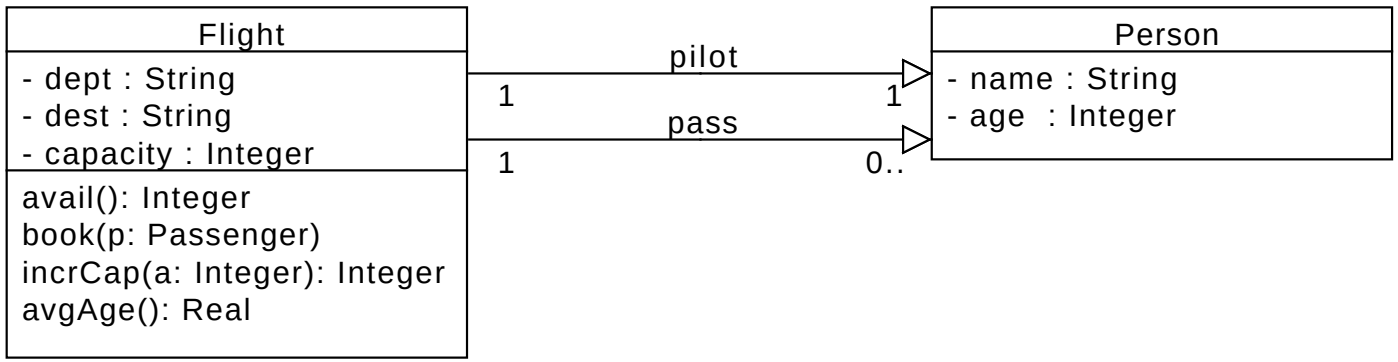
- Aus Attribute, Operationen, Assoziationen des Modells
- Navigation entlang der Assoziationen
- Für Kardinalität 1 Typ `T`, sonst `Set(T)`
- Benutzerdefinierte Operationen in Ausdrücken müssen zustandsfrei sein (Stereotyp `<<query>>`)

Beispiel

Ein Flugbuchungssystem

Jeder Flug hat ein Start, ein Ziel, eine Kapazität (Anzahl verfügbarer Sitze), einen Piloten sowie eine Menge von zugeordneten Passagieren; Piloten und Passagiere sind Personen.
Jede Person hat einen Namen und ein Alter.

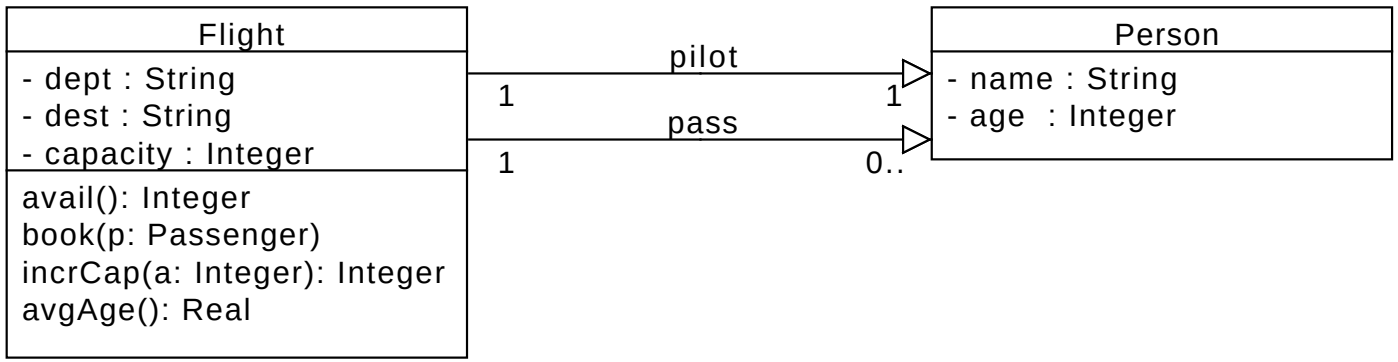
OCL im Beispiel



Start und Ziel sind immer unterschiedlich.

context Flight
inv: self.dept <> self.dest

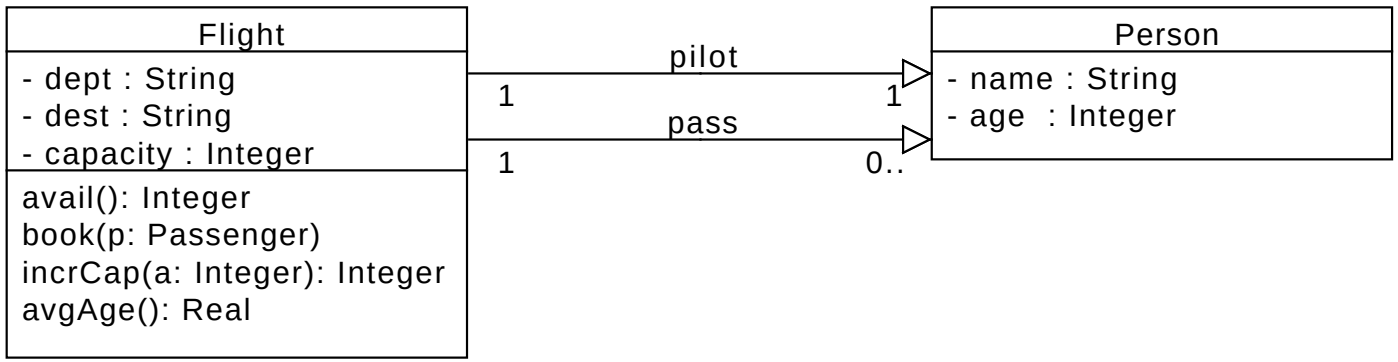
OCL im Beispiel



Der Pilot muss über 25 sein.

```
context Flight
inv:    self.pilot.age >= 25
```

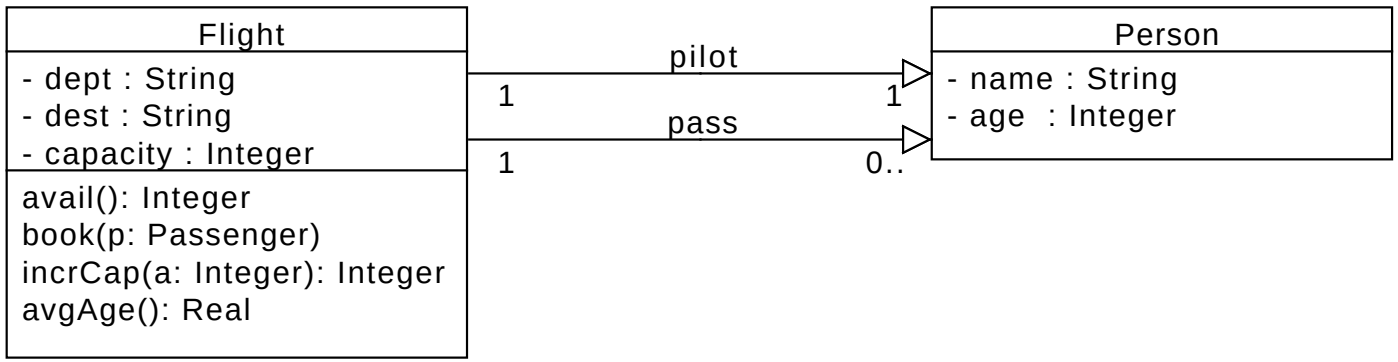
OCL im Beispiel



Es gibt nie mehr Passagiere als Kapazität.

```
context Flight
inv:    self.pass->size() <= self.capacity
```

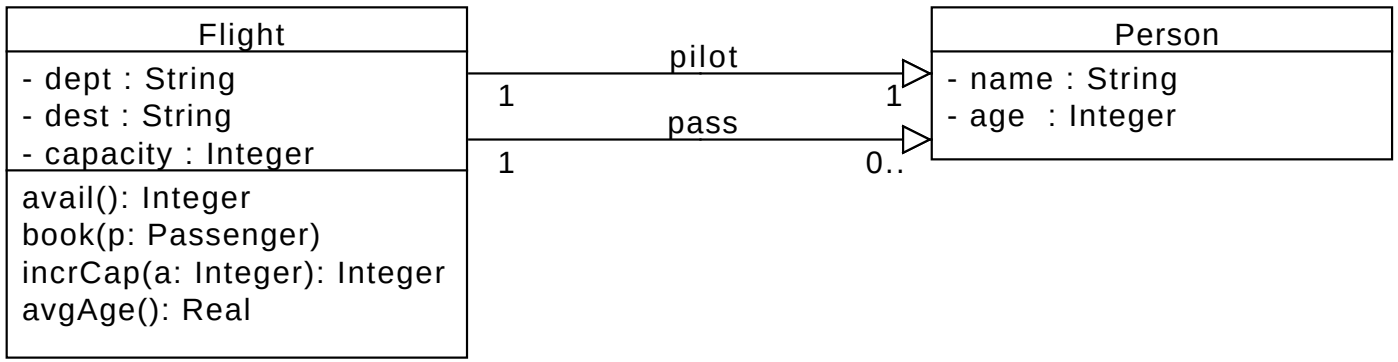
OCL im Beispiel



Jeder Flug hat mindestens einen Passagier über 18.

```
context Flight
inv:    self.pass->exists(p|p.age >= 18)
```

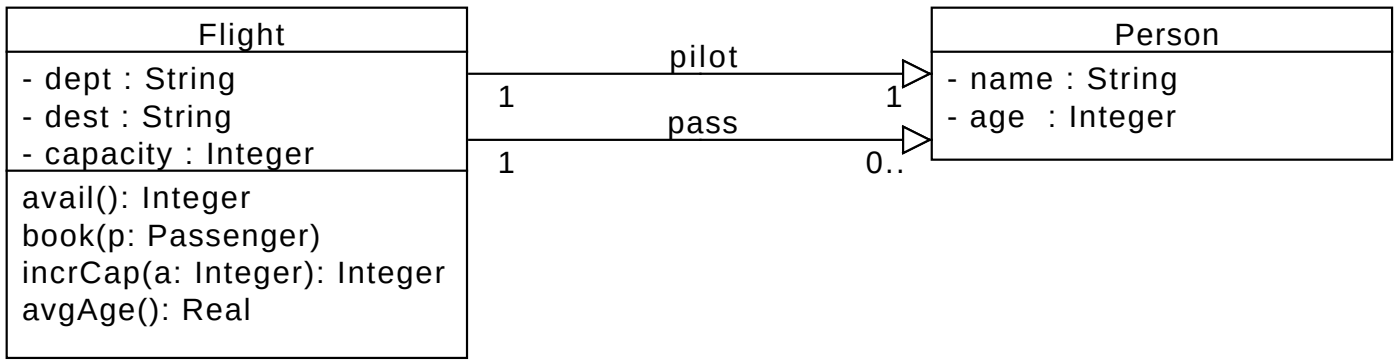
OCL im Beispiel



Der Pilot ist kein Passagier.

```
context Flight
inv:    not (self.pass->contains(self.pilot))
```

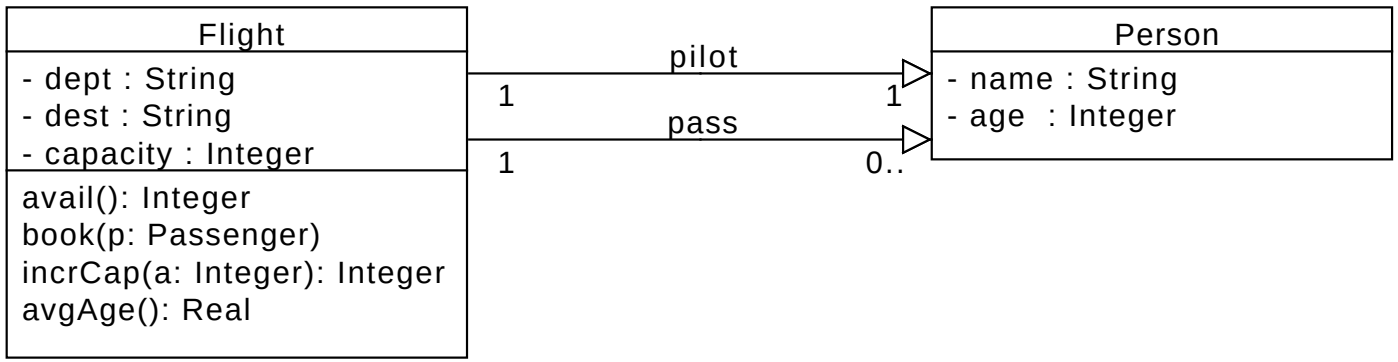
OCL im Beispiel



Jeder Passagier unter 18 wird von einem über 18 mit dem gleichen Namen (Elternteil, Geschwister) begleitet.

```
context Flight
inv: self.pass->all(p|p.age < 18 implies
                    self.pass->exists(q|q.name= p.name
                                      and q.age >= 18))
```

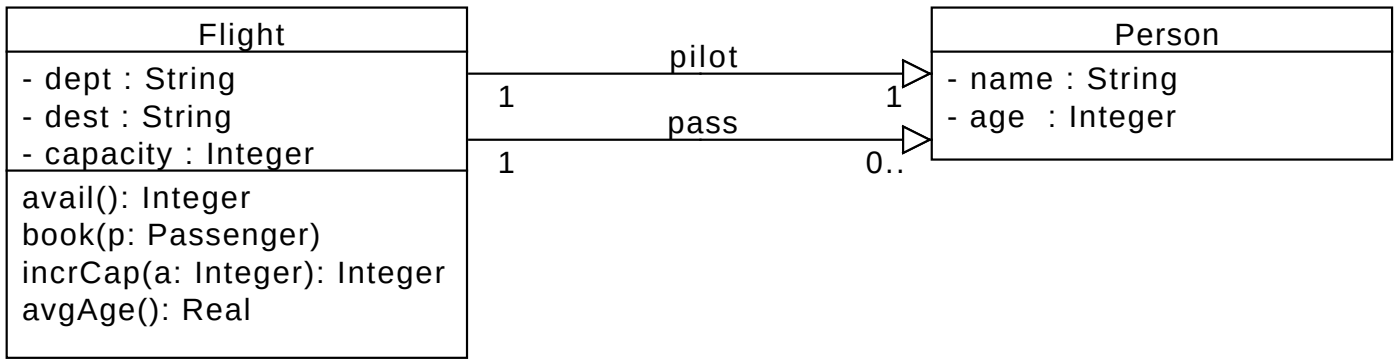
OCL im Beispiel



Die Operation avail gibt die Anzahl der noch freien Plätze zurück.

```
context Flight::avail() : Integer
post:   result = self.capacity - self.pass->size()
```

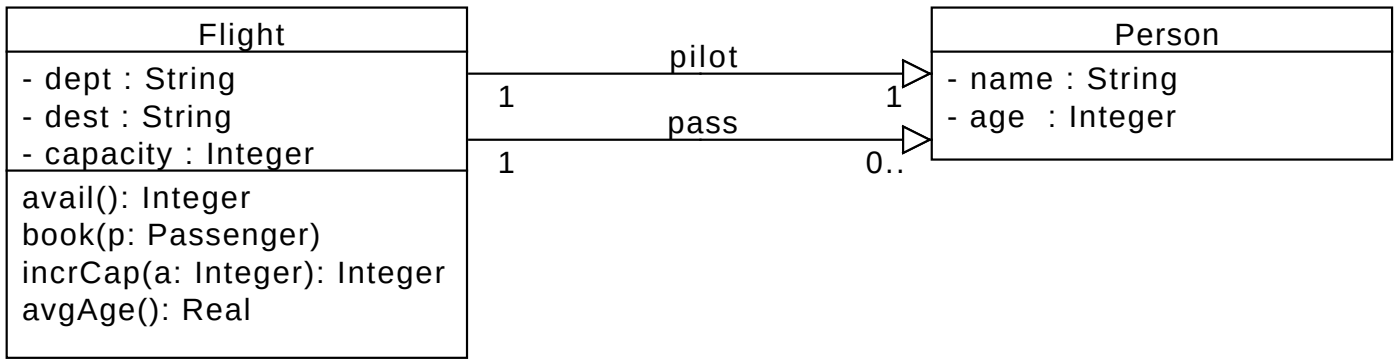
OCL im Beispiel



Wenn noch ein Platz frei ist, soll book den Passagier zu diesem Flug hinzufügen.

```
context Flight::book(p: Person)
pre:    self.capacity - self.pass->size() > 0
post:   self.pass->contains(p)
```

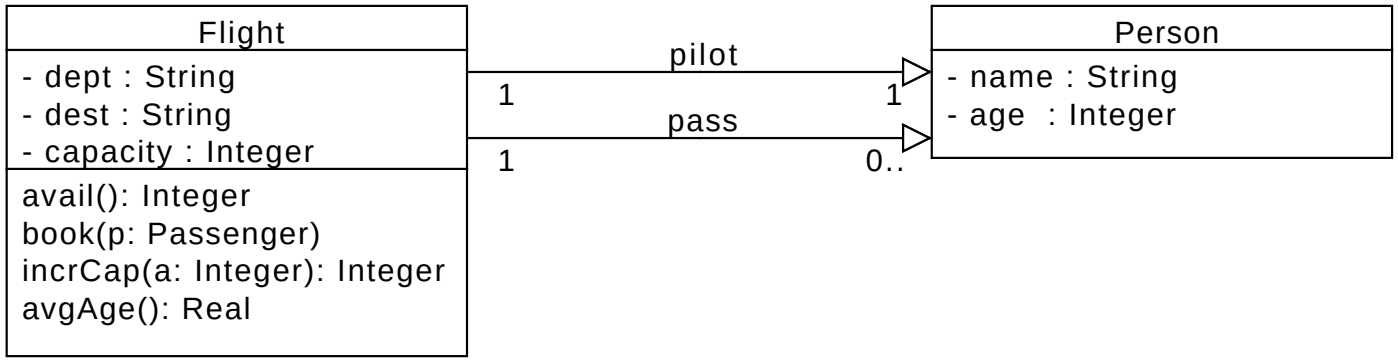
OCL im Beispiel



Die Operation incrCap erhöht die Kapazität des Fluges.

```
context Flight::incrCap(a: Integer) : Integer
pre:  self.capacity+ a- self.pass->size() >= 0
post: self.capacity = @pre(self.capacity)+ a
      result = self.capacity
```

OCL im Beispiel



Die Operation avgAge soll das Durchschnittsalter der Fluggäste berechnen.

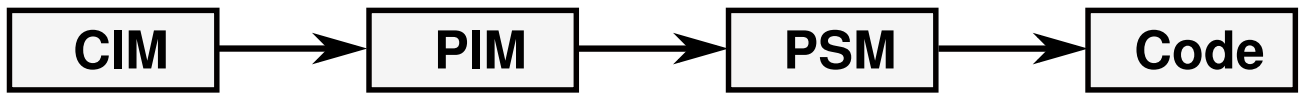
```
context Flight::avgAge() : Real
pre:  self.pass->exists(p| True)
post: result =
        self.pass->iterate(p:Passenger; sum: Real= 0
                                | sum+ p.age)
                               / self.pass->size()
```

Style Guide

- Komplexe Navigation vermeiden (“Loose coupling”)
- Adäquaten Kontext auswählen
- “Use of allInstances is discouraged”
- Invarianten aufspalten
- Hilfsoperationen definieren

MDA + OCL

- MDA: Model-driven architecture
- Entwicklung durch Modelltransformation



- Rolle der OCL:
  - Metasprache
  - Codegenerierung
  - Laufzeitchecks
- Beispiele für Werkzeuge: MDT/OCL
  - MDT/OCL: EMF mit OCL-Unterstützung

Zusammenfassung

- OCL erlaubt Einschränkungen auf Modellen
- Programmbegriff: abstrakter Zustandsübergang
  - Relation zwischen Vor- und Nachtzustand
- Erlaubt mathematisch präzisere Modellierung
- Frage:
  - Werkzeugunterstützung?
  - Ziel: Beweise, Codegenerierung, . . . ?
- Kritik UML:
  - “OO built-in”
  - Adäquat für eingebettete Systeme, CPS, . . . ?