Willkommen zur Vorlesung
Softwarekonstruktion
im Wintersemester 2012 / 2013

Prof. Dr. Jan Jürjens

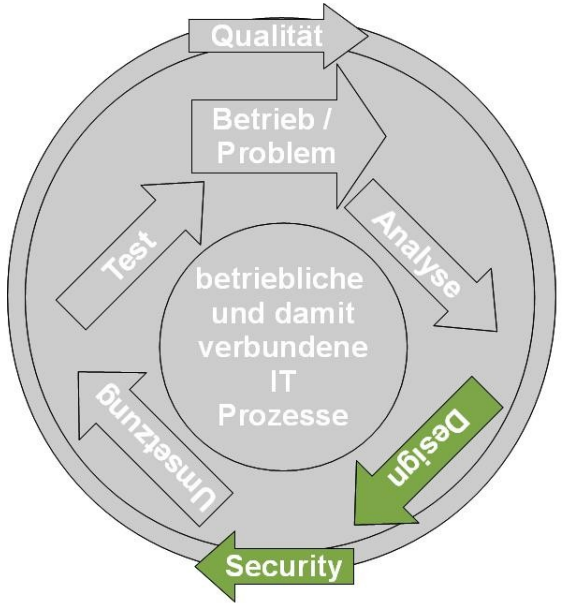
TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

3.0 Einführung in OCL

[Diese Folien basieren auf der Vorlesung ``Muster- und Komponenten-basierte Softwareentwicklung" von Prof. Dr. Maritta Heisel]

Einordnung OCL

- Qualitätsmanagement
- Testen
- Modellgetriebene SW-Entwicklung
  - OCL
  - Algebraische Spezifikation



3.0
OCL

- Motivation & Einführung
- Assoziationen, Navigationen, Operationen
- Vor- und Nachbedingungen
- Anhang: OCL Typen

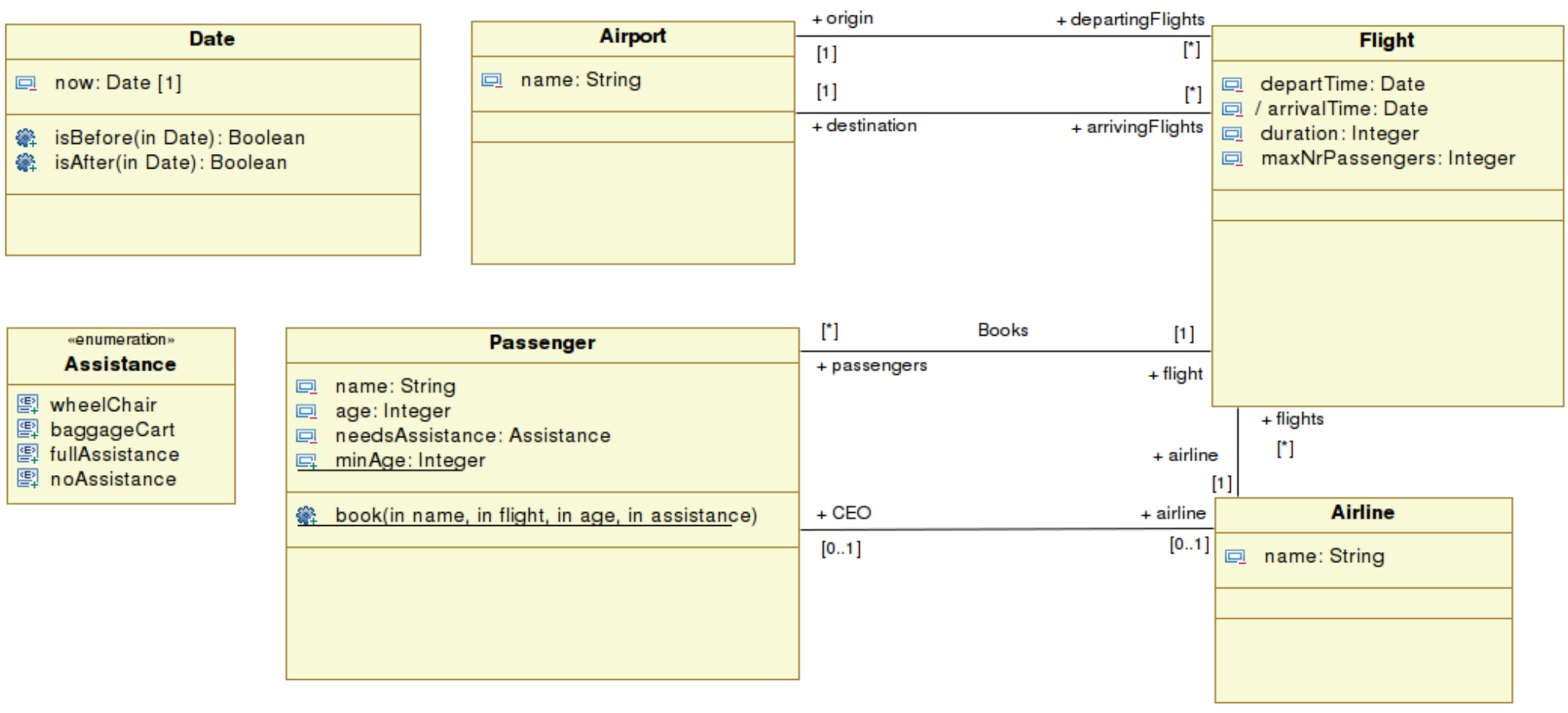
Motivation für die formale modellbasierte Spezifikation

- UML (Unified Modeling Language) 2.0 [UML09] ist eine von der OMG (Object Management Group)[1] vorgestellte (semi formale) Modellierungssprache
- UML ist die de facto Industriestandard-Notation um Softwareanalyse und -design Artefakte zu modellieren
- Die UML- Spezifikation 2.2[2] beschreibt 14 (semi) formale Diagrammtypen wie z.B Klassen- und Use-Case-Diagramme
- Einschränkungen
  - Nicht präzise genug, automatische Verifikation nur schwer umsetzbar
  - Schlechte Möglichkeiten Code zu generieren (normalerweise nur Code-Gerüst, kein voll funktionierender Code)

---

[1] http://www.omg.org/
[2] http://www.omg.org/spec/UML/2.2

Praktisches Beispiel: Klassendiagramm für einen Flughafen



Wie viele Fluggäste können sich für einen Flug anmelden?

Formale Modelle

- (semi-formale) visuelle Modelle können mit folgenden formalen Spezifikationen angereichert werden
  - Zustandsbeschränkung (mit Invarianten)
  - Operationelle Semantik (mit Vor- und Nachbedingungen)
- UML definiert eine Sprache, die mit diesem Ziel genutzt werden kann: Object Constraint Language (OCL)
- Vorteile:
  - UML Diagramme, die OCL- Ausdrücke enthalten führen zu präzisen Spezifikationen, die automatisch verifiziert werden können
  - Formale Spezifikationen beseitigen die Mehrdeutigkeit, die charakteristisch für informelle Spezifikationen ist
  - Es existieren Werkzeuge, die aus OCL-Spezifikationen für Zustands-Invarianten und für Vor- und Nachbedingungen von Operationen Code und Aussagen in Java erzeugen

Was ist OCL

- OCL ist eine formale Sprache für die Definition von Einschränkungen in UML-Modellen
- OCL ist keine Programmiersprache; deshalb ist es nicht möglich Programmlogik oder Kontrollflüsse in OCL zu modellieren
- OCL-Ausdrücke sind garantiert ohne Seiteneffekte
  - Wenn ein OCL-Ausdruck evaluiert wird, liefert er einen Wert zurück; er kann nichts im Modell verändern
  - Der Zustand des Systems wird sich wegen der Evaluation eines OCL-Ausdrucks nie ändern, auch wenn ein OCL-Ausdruck eine Zustandsänderung spezifizieren kann (z.B. in einer Nachbedingung)
- OCL unterstützt Prüfung von strenger Typisierung

Literatur

OMG Spezifikation

- „Object Constraint Language 2.0“ [UML10]
  http://www.omg.org/spec/OCL/2.2/PDF

(Teilweise) Basis für unsere „Einführung in OCL“

- The Object Constraint Language:

  Getting Your Models Ready for MDA“ [WK03]

  http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

OCL-Ausdrücke

- Sind immer an ein UML-Modell gebunden
- Beschreiben Einschränkungen für Elemente des Modells zu dem sie gehören; dieses Modell beschreibt welche Klassen genutzt werden können und welche Attribute, Operationen und Assoziationen für die Objekte dieser Klassen vorhanden sind
- Werden im zugehörigen UML-Modell oder in einem separaten Dokument angegeben

Definitionen

Constraint (Einschränkung)
Eine Einschränkung auf einem oder mehreren Teilen eines UML-Modells

Class Invariant (Klasseninvariante)
Eine Bedingung, die (fast) immer von allen Instanzen einer Klasse erfüllt werden muss

Pre-condition (Vorbedingung)
Eine Bedingung, die erfüllt sein muss, bevor eine Operation ausgeführt werden kann

Post-condition (Nachbedingung)
Eine Bedingung, die nach dem Ausführen einer Operation erfüllt sein muss

Guard condition (Abdeckungsbedingung)
Eine Bedingung, die vor einer Transition in einem Zustandsdiagramm, vor einer Nachricht in einem Sequenzdiagramm oder vor anderen UML-Modellen, die Verhalten modellieren, erfüllt sein muss

Basisform eines OCL-Ausdrucks

context <identifier><constraintType>
[<constraintName>]:<boolean expression>

Context Ein Schlüsselwort um das zugehörige Modellelement, welches mittels <identifier> angegeben wird, zu markieren. Dieses kann andere Modellelemente referenzieren. Das Schlüsselwort selbst kann innerhalb <boolean expression> genutzt werden, um den Kontext zu betreten.

<identifier> ist ein Klassen- oder Operationsname

<constraintType> ist eins der Schlüsselwörter inv, pre oder post

<constraintName> ist ein optionaler Name für die Einschränkung

<boolean expression> ist irgendein boolescher Ausdruck, oft ein Vergleich

OCL Typen und Schlüsselwörter

Die folgenden Typen können in einem OCL-Ausdruck benutzt werden

Vordefinierte Typen
- Primitive Typen: String, Integer, Real, Boolean
- Collection Types: Set, Bag, Sequence, OrderedSet
- Tupel Typen: Tuple
- Spezielle Typen: OclType, OclAny, …

Klassifikatoren von einem UML-Modell und seiner Eigenschaften
- Klassen, Enumerationsklassen und Rollennamen
- Attribute und Operationen

Folgende Schlüsselwörter können in einem OCL-Ausdruck benutzt werden:
- *If-then-else-endif* Konditionalausdrücke
- *Not, or, and xor, implies* boolesche Operatoren
- *Def* globale Definitionen
- *Let-in* lokale Definitionen

Klasseninvarianten

- Eine Invariante
  - Ist eine Bedingung, die fortwährend erfüllt sein muss: „An … invariant ... must be true for all instances of that type at any time“ („Object Constraint Language 2.0“ p. 8)
  - Wird anhand des Schlüsselwortes *inv* im Kontext der Instanz eines Klassifikators (Klasse, Rollenname...) spezifiziert

Gebräuchliche Typen von Invarianten

- Beschränkung von Domänen:
  - Beschränkung der Werte, die ein Attribut annehmen kann
- Beschränkung auf Einmaligkeit:
  - Ein Attribut oder eine Attributmenge einer Klasse für die gilt, dass für zwei unterschiedliche Instanzen dieser Klasse diesen Attributen keine gleichen Werte zugewiesen werden dürfen
- Zeitliche Beschränkung
- Bedingungen, die abgeleitete Modellelemente definieren (z.B abgeleitet Attribute)
- Regeln für die Existenz:
  - Regeln, die angeben, dass bestimmte Objekte/Werte existieren/ definiert sein müssen, bevor andere Objekte/ Werte definiert/ erzeugt werden

Invarianten auf Attributen I

- Die Klasse, die von einer Invarianten referenziert wird, ist der Kontext der Invarianten
- Es ist gefolgt von einem booleschen Wert, der die Invariante angibt
- Alle Attribute der Kontextklasse können in der Invarianten genutzt werden

Beispiel

context: Flight

Inv: duration < 4

- Bedeutung: ?

Invarianten auf Attributen I

- Die Klasse, die von einer Invarianten referenziert wird, ist der Kontext der Invarianten
- Es ist gefolgt von einem booleschen Wert, der die Invariante angibt
- Alle Attribute der Kontextklasse können in der Invarianten genutzt werden

Beispiel

context: Flight

Inv: duration < 4

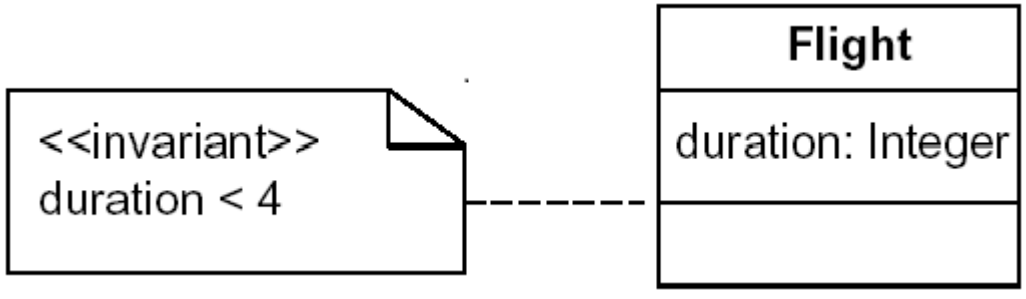
- Bedeutung: Jeder Flug dauert weniger als 4 Stunden

Gebrauch von self

Die folgenden Notationen sind äquivalent

Context Flug
Inv: self.duration < 4

Context Flug
Inv: duration < 4



Invarianten auf Attributen II

- Wenn der Attributtyp eine Klasse ist, können die Attribute und Abfrageoperationen dieser Klasse für die Erstellung der Invarianten genutzt werden (anhand der Punkt-Noatation)
- Abfrageoperation:
  Eine Operation, die den Wert von Attributen nicht ändert

Beispiel

context: Flight

Inv: departTime.isBefore(arrivalTime)

- Bedeutung: ?

Invarianten auf Attributen II

- Wenn der Attributtyp eine Klasse ist, können die Attribute und Abfrageoperationen dieser Klasse für die Erstellung der Invarianten genutzt werden (anhand der Punkt-Noatation)
- Abfrageoperation:
  Eine Operation, die den Wert von Attributen nicht ändert

Beispiel

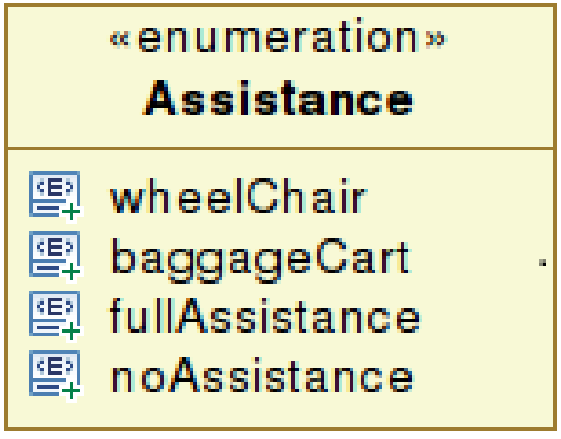
context: Flight

Inv: departTime.isBefore(arrivalTime)

- Bedeutung: Das Abflugdatum ist vor dem Ankunftsdatum.

Aufzählungstypen

Eine Aufzählung nutzt Datentypen gefolgt von :: und einem Wert



Beispiel

context Passenger

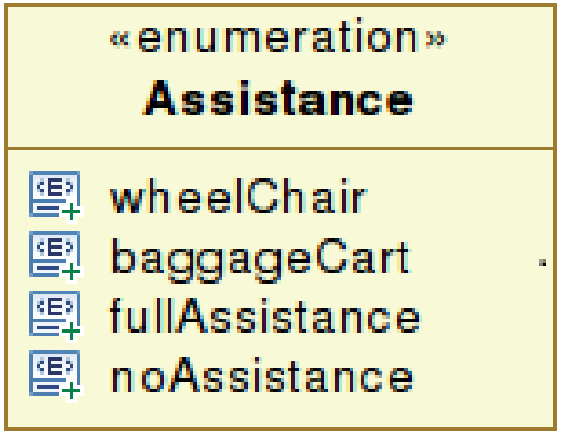
Inv: self.age > 95 implies

Self. NeedsAssistance = Assistance :: wheelchair

Bedeutung: ?

Aufzählungstypen

Eine Aufzählung nutzt Datentypen gefolgt von :: und einem Wert



Beispiel

context Passenger

Inv: self.age > 95 implies

Self. NeedsAssistance = Assistance :: wheelchair

Bedeutung: Jeder Passagier über 95 braucht einen Rollstuhl.

3.0
OCL

- Motivation & Einführung
- **Assoziationen, Navigationen, Operationen**
- Vor- und Nachbedingungen
- Anhang: OCL Typen

Assoziation und Navigation I

- Jede Assoziation ist ein Navigationspfad
- Der Kontext des Ausdrucks ist der Startpunkt
- Rollennamen (oder Assoziationsenden) werden genutzt, um navigierte Assoziationen zu identifizieren

Assoziation und Navigation II

Beispiel

context Flight

Inv: origin <> destination

Bedeutung: ?

Beispiel

context Flight

Inv: origin.name = 'Duisburg'

Bedeutung:?

Assoziation und Navigation II

Beispiel

context Flight

Inv: origin <> destination

Bedeutung: Der Abflugort ist immer ungleich dem Flugziel

Beispiel

context Flight

Inv: origin.name = 'Duisburg'

Bedeutung:?

Assoziation und Navigation II

Beispiel

context Flight

Inv: origin <> destination

Bedeutung: Der Abflugort ist immer ungleich dem Flugziel

Beispiel

context Flight

Inv: origin.name = 'Duisburg'

Bedeutung: Der Abflugort ist immer Duisburg

Assoziation und Navigation III

- Assoziationen sind oft one-to-many oder many-to-many Beziehungen, deshalb werden Beschränkungen für Collections benötigt
- OCL Ausdrücke geben entweder einen Fakt über alle Objekte in der Collection an oder über die Collection selbst

Nutzung von Collection-Operationen I

- Eine Collection-Operationen kann immer dann verwendet werden, wenn eine Navigation in einer Menge von Objekten endet
- Ein Pfeil (→) zwischen dem Rollennamen und der Operation weist auf die Nutzung einer der vordefinierten Operation für Collections hin (z.B *Passanger → size()*)
- Ein Punkt (.) zwischen dem Rollennamen und der Operation gibt an, dass eine Operation aus dem UML-Modell verwendet wird (z.B. *departTime.isBefore(arrivalTime)*)

Beispiel

context Flight

Inv: passengers → size() <= maxNrPassengers

Bedeutung: ?

Nutzung von Collection-Operationen I

- Eine Collection-Operationen kann immer dann verwendet werden, wenn eine Navigation in einer Menge von Objekten endet
- Ein Pfeil (→) zwischen dem Rollennamen und der Operation weist auf die Nutzung einer der vordefinierten Operation für Collections hin (z.B *Passanger → size()*)
- Ein Punkt (.) zwischen dem Rollennamen und der Operation gibt an, dass eine Operation aus dem UML-Modell verwendet wird (z.B. *departTime.isBefore(arrivalTime)*)

Beispiel

context Flight

Inv: passengers → size() <= maxNrPassengers

Bedeutung: Die Anzahl der Passagiere ist kleiner oder gleich der maximalen Anzahl von Sitzen

Nutzung von Collection-Operationen II

- Eine Collection von Objekten ist:
  - Ein Set:
    - Jedes Element kommt nur einmal vor
    - Einfaches Navigieren einer Assoziation liefert ein Set zurueck.
  - Ein Bag:
    - Gleiche Elemente dürfen mehrmals vorkommen
  - Ein OrderedSet:
    - Ein Satz von geordneten Elementen
  - Eine Sequence:
    - Ein Bag in dem die Elemente geordnet sind

Die *collect* Operation I

- Operation um Attributwerte zu sammeln, z. B.
  passengers → *collect(name)*

Bedeutung (in Pseudocode)

```
Collection<String> c = new Collection();
foreach (p: passengers) {c.add(p.name);}
return c;
```

- Die Operation kann auch benutzt werden, um eine neue Collections aus den objekten am Ende der Assoziation zu bilden, z.B. *arrivingFlights → collect(airline)*

Bedeutung (in Pseudocode)

```
Collection<Airline> c = new Collection();
foreach (f: arrivingFlights) {c.add(f.airline);}
return c;
```

Die *collect* Operation II

- Die resultierende Collection beinhaltet andere Objekte als die ursprüngliche Collection
- Wenn die Quelle ein Set ist, so ist die resultierende Collection kein Set sonder ein Bag
- Wenn die Quelle eine Sequence oder ein OrderedSet ist, so ist die neue Collection eine Sequence
- Die Punktnotation ist eine verkürzte Schreibweise:
  passengers.name
  arrivingFlights.airline

Die *collect* Operation III

Beispiel

context Airport

Inv: arrivingFlights → size() =
arrivingFlights → collect(Airline) → size()

- Bedeutung: ?

Die *collect* Operation III

Beispiel

context Airport

Inv: arrivingFlights → size() =
arrivingFlights → collect(Airline) → size()

- Bedeutung: Jeder ankommende Flug gehört zu einer Airline

Die *select* Operation

- Die select Operation bekommt einen OCL-Ausdruck als Parameter übergeben
- Das Ergebnis dieser Operation ist eine Subcollection der verwendeten Collection
- *Select* liefert alle Elemente einer Collection, für die der Ausdruck wahr ist

Beispiel

context Flight

inv: passengers → select(needsAssistance <>
Assistance::noAssistance) → size() <= 10

- Bedeutung: ?

Die *select* Operation

- Die select Operation bekommt einen OCL-Ausdruck als Parameter übergeben
- Das Ergebnis dieser Operation ist eine Subcollection der verwendeten Collection
- *Select* liefert alle Elemente einer Collection, für die der Ausdruck wahr ist

Beispiel

context Flight

inv: passengers → select(needsAssistance <>
Assistance::noAssistance) → size() <= 10

- Bedeutung: Die Anzahl der Passagiere, die Hilfe brauchen, ist kleiner oder gleich 10

Die *reject* Operation

- Die *reject* Operation verhält sich analog zu select
- *Reject* liefert alle Elemente einer Collection, für die der Ausdruck falsch ist

Beispiel

context Flight

inv: passengers → reject(needsAssistance = Assistance::noAssistance) → size() <= 10

- Bedeutung: ?

Die *reject* Operation

- Die *reject* Operation verhält sich analog zu select
- *Reject* liefert alle Elemente einer Collection, für die der Ausdruck falsch ist

Beispiel

context Flight

inv: passengers → reject(needsAssistance = Assistance::noAssistance) → size() <= 10

- Bedeutung: Die Anzahl der Passagiere, die Hilfe brauchen, ist kleiner oder gleich 10

Die forAll Operation I

- Die forAll Operation kann genutzt werden, um eine Bedingung zu definieren, die von allen Elementen in einer Collection eingehalten werden muss
- Die forAll Operation erhält einen OCL-Ausdruck als Parameter
- Diese Operation wird verwendet, wenn es ein (Sub-)Set von allen Instanzen einer Klasse gibt und dieses Set überprüft werden soll
- Diese Operation liefert einen booleaschen Wert zurück:
  - Wahr, wenn die Bedingung von allen Elementen erfüllt wird
  - Ansonsten falsch

Die forAll Operation II

- *class.allInstances():* Collection mit allen Elementen einer Klasse

Beispiel

```
context Airport
  inv :  Airport.allInstances()->forAll(a1, a2 |
           a1 <> a2 implies a1.name <> a2.name)
```

- Bedeutung: ?

Die forAll Operation II

- *class.allInstances():* Collection mit allen Elementen einer Klasse

Beispiel

```
context Airport
  inv :  Airport.allInstances()->forAll(a1, a2 |
           a1 <> a2 implies a1.name <> a2.name)
```

- Bedeutung: Jeder Flughafenname ist einzigartig
- Äquivalent:

context Airport
inv : Airport.allInstances() → isUnique(name)

3.0
OCL

- Motivation & Einführung
- Assoziationen, Navigationen, Operationen
- **Vor- und Nachbedingungen**
- Anhang: OCL Typen

Vor- und Nachbedingungen I

- In Klassendiagrammen kann nur die Syntax und Signatur einer Operation definiert werden
- Die Semantik einer Operation kann mittels Vor- und Nachbedingungen in OCL spezifiziert werden
- Vorbedingung:
  - Bedingungen, die von Argumenten und dem initialen Objektzustand erfüllt werden müssen, damit ein Operationsaufruf gültig ist.

Ein Beispiel für eine Vorbedingung

Beispiel

context Passenger :: book(name: String, flight: Flight, age: Integer, assistance: Assistance)

pre: flight.passengers → size() < flight.maxNrPassengers

- Bedeutung: ?

Ein Beispiel für eine Vorbedingung

Beispiel

context Passenger :: book(name: String, flight: Flight, age: Integer, assistance: Assistance)

pre: flight.passengers → size() < flight.maxNrPassengers

- Bedeutung: Die Anzahl der registrierten Passagiere für einen Flug muss vor einer Buchung kleiner als *maxNrPassengers* sein

Vor- und Nachbedingungen II

- Nachbedingung:
  - Bedingung die vom Rückgabewert, finalem Objektzustand, den Argumenten und dem initialem Objektzustand erfüllt sein müssen am Ende einer Operationsausführung, in der Annahme, das die Vorbedingungen erfüllt sind
  - Spezifiziert beabsichtigte Ergebnisse und Zustandsänderungen (was), aber nicht wie sie geschehen (wie)
  - Der initiale Zustand eines Objektfelds wird mit *@pre* angegeben
  - Der Rückgabewert wird mit dem Schlüsselwort *result* angegeben

Ein Beispiel für eine Nachbedingung

Beispiel

context Passenger :: book(name: String, flight: Flight, age: Integer, assistance: Assistance)

post: flight.passengers → size() - flight.passengers@pre → size() = 1
and
flight.passengers → exists (p: Passenger |
p.age =age and p.name = name
and p.needsAssistance = assistance)

Bedeutung:

- ?
- ?

NB: obiges flight bezieht sich auf das Argument von book, nicht das gleichnamige Assoziationsende.

Ein Beispiel für eine Nachbedingung

Beispiel

context Passenger :: book(name: String, flight: Flight, age: Integer, assistance: Assistance)

post: flight.passengers → size() - flight.passengers@pre → size() = 1
and
flight.passengers → exists (p: Passenger |
p.age =age and p.name = name
and p.needsAssistance = assistance)

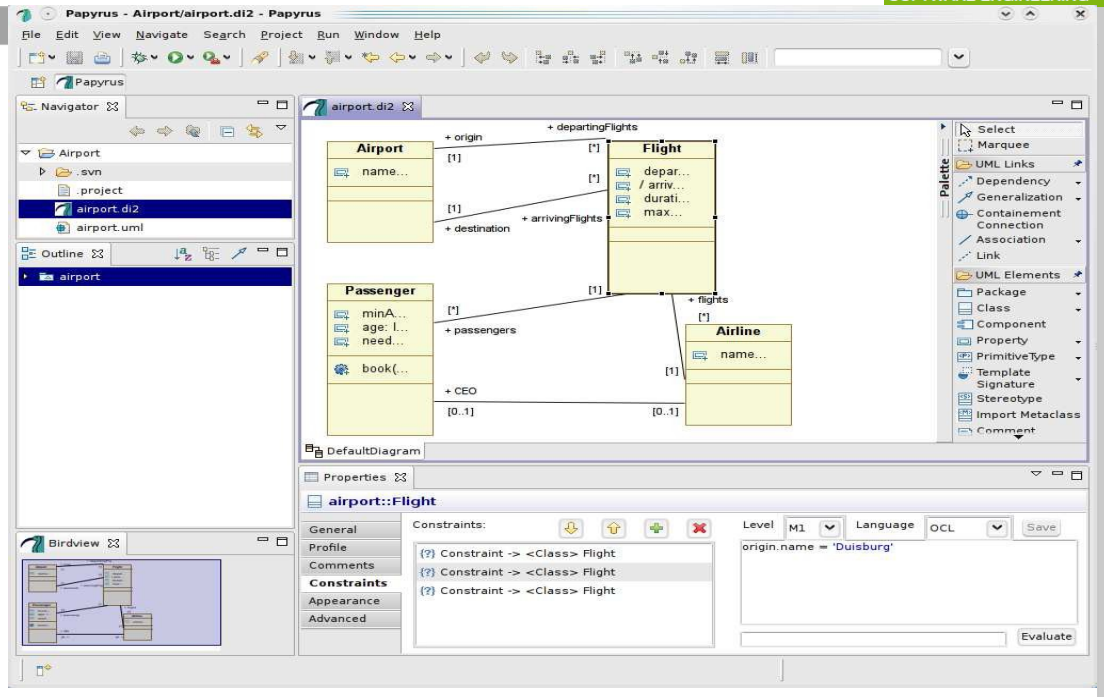
Bedeutung:

- Nach der Ausführung existiert ein zusätzliches Objekt
- Die Attribute eines Objekts wurden anhand der Werte aus *book* initialisiert

NB: obiges flight bezieht sich auf das Argument von book, nicht das gleichnamige Assoziationsende.

Papyrus

- Papyrus ist ein frei erhältliches Open-Source Werkzeug für die Modellierung mit UML 2.0
- Download: http://www.papyrusuml.org/
- Basiert auf der Entwicklungs-umgebung Eclipse



- Gänzliche Einhaltung des von OMG definierten UML 2.0 Standards (gemäß Webseite)
- Erweiterbare Architektur von Papyrus erlaubt das Hinzufügen von Diagrammen, neuen Codegeneratoren, etc.
- Erlaubt das Einbinden von OCL-Bedingungen

3.0
OCL

- Motivation & Einführung
- Assoziationen, Navigationen, Operationen
- Vor- und Nachbedingungen
- Anhang: OCL Typen

Primitive Typen

entnommen aus http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Type	Description	Values	Operators and Operations
Boolean		true, false	=, <>, and, or, xor, not, implies, if-then-else-endif (note 2)
Integer	A whole number of any size	-1, 0, 1, ...	=, <>, >, <, >=, <=, *, +, - (unary), - (binary), / (real), abs(), max(b), min(b), mod(b), div(b)
Real	A real number of any size	1.5, ...	=, <>, >, <, >=, <=, *, +, - (unary), - (binary), /, abs(), max(b), min(b), round(), floor()
String	A string of characters	'a', 'John'	=, <>, size(), concat(s2), substring(lower, upper) (1<=lower<=upper<=size), toReal(), toInteger()

Notes:
1) Operations indicated with parenthesis are applied with ".", but the parenthesis may be omitted.
2) Example: title = (if isMale then 'Mr.' else 'Ms.' endif)

Collections und Tuples

entnommen aus http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Description	Syntax	Examples
Abstract collection of elements of type T	Collection(T)	
Unordered collection, no duplicates	Set(T)	Set{1 , 2}
Ordered collection, duplicates allowed	Sequence(T)	Sequence {1, 2, 1} Sequence {1..4} (same as {1,2,3,4})
Ordered collection, no duplicates	OrderedSet(T)	OrderedSet {2, 1}
Unordered collection, duplicates allowed	Bag(T)	Bag {1, 1, 2}
Tuple (with named parts)	Tuple(field1: T1, ... fieldn : Tn)	Tuple {age: Integer = 5, name: String = 'Joe' } Tuple {name = 'Joe', age = 5}

Note 1: They are *value types*: "=" and "<>" compare values and not references.

Note 2: Tuple components can be accessed with "." as in "t1.name"

Operations on Collection (T)

entnommen aus http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Operation	Description
size(): Integer	The number of elements in this collection (*self*)
isEmpty(): Boolean	size = 0
notEmpty(): Boolean	size > 0
includes(object: T): Boolean	True if *object* is an element of *self*
excludes(object: T): Boolean	True if *object* is not an element of *self*
count(object: T): Integer	The number of occurrences of *object* in *self*
includesAll(c2: Collection(T)): Boolean	True if *self* contains all the elements of *c2*
excludesAll(c2: Collection(T)): Boolean	True if *self* contains none of the elements of *c2*
sum(): T	The addition of all elements in *self* (T must support "+")
product(c2: Collection(T2)) : Set(Tuple(first:T, second:T2))	The cartesian product operation of *self* and *c2*.

Note: Operations on collections are applied with "->" and not "."

Iterator Expressions on Collection (T)

entnommen aus http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Iterator expression	Description
iterate(iterator: T; accum: T2 = init | body) : T2	Returns the final value of an accumulator that, after initialization, is updated with the value of the *body* expression for every element in the *source* collection.
exists(iterators | body) : Boolean	True if *body* evaluates to true for at least one element in the *source* collection. Allows multiple iterator variables.
forAll(iterators | body): Boolean	True if *body* evaluates to true for each element in the source collection. Allows multiple iterator variables.
one(iterator| body): Boolean	True if there is exactly one element in the *source* collection for which *body* is true
isUnique(iterator| body): Boolean	Results in true if *body* evaluates to a different value for each element in the *source* collection.
any(iterator | body): T	Returns any element in the source collection for which *body* evaluates to true. The result is null if there is none.
collect(iterator| body): Collection(T2)	The Collection of elements resulting from applying *body* to every member of the *source* set.

Note: The iterator variable declaration can be omitted when there is no ambiguity.

Iterator expression on Collection (T)

entnommen aus http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Iterator expression	Description
select(iterator | body): Collection(T)	The Collection of elements of the *source* collection for which *body* is true. The result collection is of the same type of the *source* collection.
reject(iterator | body): Collection(T)	The Collection of elements of the *source* collection for which *body* is false. The result collection is of the same type of the *source* collection.
collectNested(iterator | body): CollectionWithDuplicates(T2)	The Collection of elements (allowing duplicates) that results from applying *body* (of type T2) to every member of the *source* collection. The result is not flattened. Conversions: Set -> Bag, OrderedSet -> Sequence.
sortedBy(iterator | body): OrderedCollection(T)	Returns an ordered Collection of all the elements of the *source* collection by ascending order of the value of the *body* expression. The type T2 of the *body* expression must support "<". Conversions: Set -> OrderedSet, Bag -> Sequence.

Operations on Set (T) I

entnommen aus http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Operation	Description
=(s: Set(T)) : Boolean	Do *self* and *s* contain the same elements?
union(s: Set(T)): Set(T)	The union of *self* and *s*.
union(b: Bag(T)): Bag(T)	The union of *self* and bag *b*.
intersection(s: Set(T)): Set(T)	The intersection of *self* and *s*.
intersection(b: Bag(T)): Set(T)	The intersection of *self* and *b*.
-(s: Set(T)) : Set(T)	The elements of *self*, which are not in *s*.
including(object: T): Set(T)	The set containing all elements of *self* plus *object*.
excluding(object: T): Set(T)	The set containing all elements of *self* minus *object*.
symmetricDifference(s: Set(T)): Set(T)	The set containing all the elements that are in *self* or *s*, but not in both.

Operations on Set (T) II

entnommen aus http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Operation	Description
flatten() : Set(T2)	If T is a collection type, the result is the set with all the elements of all the elements of *self*; otherwise, the result is *self*.
asOrderedSet(): OrderedSet(T)	OrderedSet with elements from *self* in undefined order.
asSequence(): Sequence(T)	Sequence with elements from *self* in undefined order.
asBag(): Bag(T)	Bag will all the elements from *self*.

Operations on Bag (T)

entnommen aus http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Operation	Description
=(bag: Bag(T)) : Boolean	True if *self* and *bag* contain the same elements, the same number of times.
union(bag: Bag(T)): Bag(T)	The union of *self* and *bag*.
union(set: Set(T)): Bag(T)	The union of *self* and *set*.
intersection(bag:Bag(T)): Bag(T)	The intersection of *self* and *bag*.
intersection(set: Set(T)): Set(T)	The intersection of *self* and *set*.
including(object: T): Bag(T)	The bag with all elements of *self* plus *object*.
excluding(object: T): Bag(T)	The bag with all elements of *self* without *object*.
flatten() : Bag(T2)	If T is a collection type: bag with all the elements of all the elements of *self*; otherwise: *self*.
asSequence(): Sequence(T)	Seq. with elements from *self* in undefined order.
asSet(): Set(T)	Set with elements from *self*, without duplicates.
asOrderedSet(): OrderedSet(T)	OrderedSet with elements from *self* in undefined order, without duplicates.

Operations on Sequence (T) I

entnommen aus http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Operation	Description
=(s: Sequence(T)) : Boolean	True if *self* contains the same elements as *s*, in the same order.
union(s: Sequence(T)): Sequence(T)	The sequence consisting of all elements in *self*, followed by all elements in *s*.
flatten() : Sequence(T2)	If T is a collection type, the result is the set with all the elements of all the elements of *self*; otherwise, it's *self*.
append(object: T): Sequence(T)	The sequence with all elements of *self*, followed by *object*.
prepend(obj: T): Sequence(T)	The sequence with *object*, followed by all elements in *self*.
insertAt(index : Integer, object : T) : Sequence(T)	The sequence consisting of *self* with *object* inserted at position *index* (1<=index<=size+1)
subSequence(lower : Integer, upper: Integer) : Sequence(T)	The sub-sequence of *self* starting at index *lower*, up to and including index *upper* (1<=lower<=upper<=size)

Operations on Sequence (T) II

entnommen aus http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Operation	Description
at(i : Integer) : T	The *i*-th element of *self* (1<=i<=size)
indexOf(object : T) : Integer	The index of *object* in *self*.
first() : T	The first element in *self*.
last() : T	The last element in *self*.
including(object: T): Sequence(T)	The sequence containing all elements of *self* plus *object* added as last element
excluding(object: T): Sequence(T)	The sequence containing all elements of *self* apart from all occurrences of *object*.
asBag(): Bag(T)	The Bag containing all the elements from *self*, including duplicates.
asSet(): Set(T)	The Set containing all the elements from *self*, with duplicates removed.
asOrderedSet(): OrderedSet(T)	An OrderedSet that contains all the elements from *self*, in the same order, with duplicates removed.

Operations on OrderedSet (T)

entnommen aus http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Operation	Description
append(object: T): OrderedSet(T)	The set of elements, consisting of all elements of *self*, followed by *object*.
prepend(object: T): OrderedSet(T)	The sequence consisting of *object*, followed by all elements in *self*.
insertAt(index : Integer, object : T) : OrderedSet(T)	The set consisting of *self* with *object* inserted at position *index*.
subOrderedSet(lower : Integer, upper : Integer) : OrderedSet(T)	The sub-set of *self* starting at number *lower*, up to and including element number *upper* (1<=lower<=upper<=size).
at(i : Integer) : T	The *i*-th element of *self* (1<=i<=size).
indexOf(object : T) : Integer	The index of *object* in the sequence.
first() : T	The first element in *self*.
last() : T	The last element in *self*.

Special Types

entnommen aus http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Type	Description
OclAny	Supertype for all types except for collection and tuple types. All classes in a UML model inherit all operations defined on OclAny.
OclVoid	The type OclVoid is a type that conforms to all other types. It has one single instance called *null*. Any property call applied on *null* results in *OclInvalid*, except for the operation oclIsUndefined(). A collection may have *null's*.
OclInvalid	The type OclInvalid is a type that conforms to all other types. It has one single instance called *invalid*. Any property call applied on *invalid* results in *invalid*, except for the operations oclIsUndefined() and oclIsInvalid().
OclMessage	Template type with one parameter T to be substituted by a concrete operation or signal type. Used in some postconditions that need to constrain the messages sent during the operation execution.
OclType	Meta type.

Operations defined in OclAny

entnommen aus http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Operation	Description
=(object2 : OclAny) : Boolean	True if *self* is the same object as *object2*.
<>(object2 : OclAny) : Boolean	True if *self* is a different object from *object2*.
oclIsNew() : Boolean	Can only be used in a postcondition. True if *self* was created during the operation execution.
oclAsType(t : OclType) : OclType	Cast (type conversion) operation. Useful for downcast.
oclIsTypeOf(t: OclType) : Boolean	True if *self* is of type *t*.
oclIsKindOf(t : OclType) : Boolean	True if *self* is of type *t* or a subtype of *t*.
oclIsInState(s : OclState) : Boolean	True if *self* is in state *s*.
oclIsUndefined() : Boolean	True if *self* is equal to *null* or *invalid*.
oclIsInvalid() : Boolean	True if *self* is equal to *invalid*.
allInstances() : Set(T)	Static operation that returns all instances of a classifier.

Operations defined in OclMessage

entnommen aus http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Operation	Description
hasReturned() : Boolean	True if type of template parameter is an operation call, and the called operation has returned a value.
result()	Returns the result of the called operation, if type of template parameter is an operation call, and the called operation has returned a value.
isSignalSent() : Boolean	Returns true if the OclMessage represents the sending of a UML Signal.
isOperationCall() : Boolean	Returns true if the OclMessage represents the sending of a UML Operation call.
parameterName	The value of the message parameter.

Literatur

- [UML09] UML Revision Task Force: *OMG Unified Modeling Language: Sperstructure*, February 2009 http://www.omg.org/spec/UML/2.2/.2
- [UML10] UML Revision Task Force: *Object Constraint Language Specification*, February 2010 http://www.omg.org/spec/OCL/2.2/.6
- [WK03] Jos Warmer and Anneke Kleppe: *The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Longman Publishing & Co.,Inc., Boston, MA, USA, 2003*