Introduction to OCL

Semester Ganjil 2021/2022

Kurnia Saputra, ST, M.Sc.
kurnia.saputra@unsyiah.ac.id

Jurusan Informatika
Fakultas Matematika dan Ilmu Pengetahuan Alam
Universitas Syiah Kuala

Motivation for Formal Model-Based Specification

- **UML (Unified Modeling Language) 2.0** [UML15] is a (semi-formal) modeling language proposed by the OMG (Object Management Group)[1].
- UML is the de facto **industry standard** notation to model software analysis and design artifacts.
- UML Superstructure specification 2.5[2] describes 14 (semi-)formal diagram types, e.g., class and use-case diagrams.
- Limits:
  - **not precise** and **automatic verification** hardly possible
  - **weak code generation capabilities** (usually only code skeletons, not fully functional code)

[1] http://www.omg.org/
[2] http://www.omg.org/spec/UML/2.5/

Formal Models

- (Semi-formal) visual models can be **enriched with formal specifications** of
  - **state constraints** (with invariants)
  - **operation semantics** (with pre- and post-conditions)
- UML defines a language that can be used with this purpose: Object Constraint Language (OCL)
- Advantages:
  - UML diagrams enriched with OCL expressions lead to **precise specifications** that can be **verified automatically**
  - formal specifications **remove the ambiguity** that characterizes informal specifications
  - formal specifications can be **automatically verified**
  - tools exist that **generate code and assertions** in Java from OCL specifications of state invariants and operations' pre- and post conditions

What is OCL?

- OCL is a formal language used to describe **constraints** on UML models.
- OCL is **not** a programming language; therefore, it is not possible to write program logic or flow control in OCL.
- OCL expressions are guaranteed to be **without side effects**:
  - when an OCL expression is evaluated, it simply returns a value; it cannot change anything in the model
  - the state of the system will never change because of the evaluation of an OCL expression, even though an OCL expression can be used to specify a state change (e.g., in a post-condition)
- OCL supports **strong type checking**.

Basics of OCL

Specification of OCL Expressions

OCL expressions

- are always **bound to a UML model**
- always put constraints on the elements of the UML model they belong to; this model describes which classes may be used and which attributes, operations, and associations are available for objects from these classes
- are given in the UML model they belong to or in a separate document

Definitions

Constraint A **restriction** on one or more parts of a UML model.

Class invariant A constraint that must (almost) **always** be met by all instances of a class.

Pre-condition A constraint that must be true **before** the execution of an operation.

Post-condition A constraint that must be true **after** the execution of an operation.

Basic Format of an OCL Expression

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

context a keyword to mark the relative model element indicated by <identifier> from which other model elements can be referenced. The keyword **self** can be used within <boolean expression> to access the *context*.

<identifier> is a class or operation name

<constraintType> is one of the keywords **inv**, **pre**, or **post**

<constraintName> is an optional name for the constraint

<boolean expression> is some boolean expression, often an equation

OCL Types

The following **types** can be used in an OCL expression:

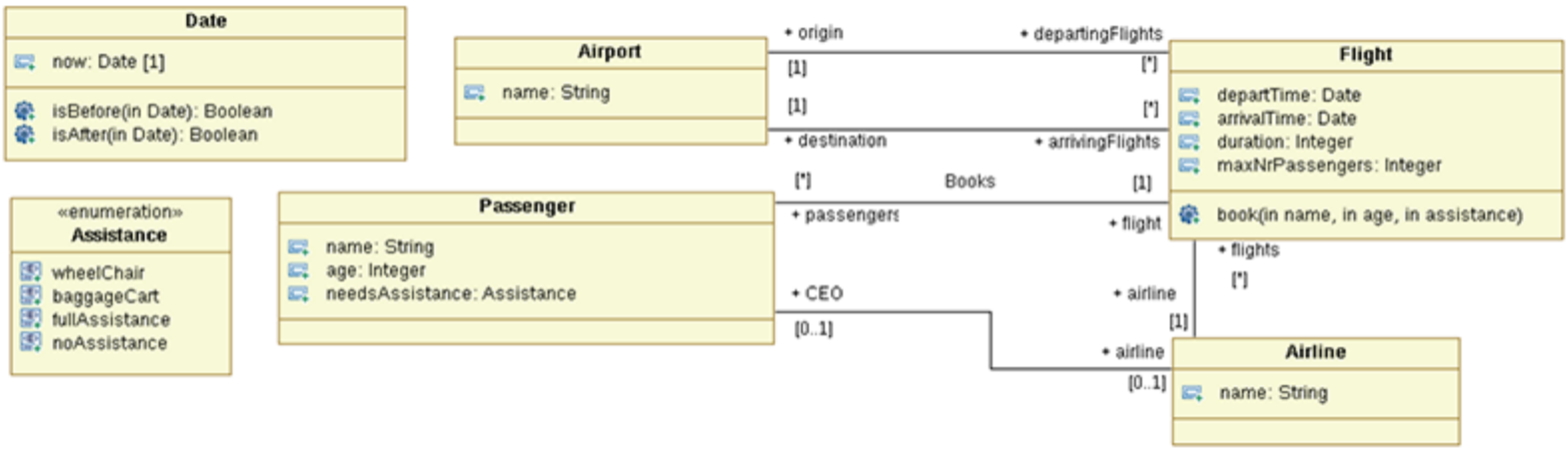
- **predefined types**
  - primitive types: String, Integer, Real, Boolean
  - collection types: Set, Bag, Sequence, OrderedSet
  - tuple types: Tuple
  - special types: OclAny (supertype for all types except for collection and tuple types), ...
- **classiffiers** from the UML model and their features
  - **classes**, **enumeration classes**, and **role names**
  - **attributes** and **operations**

OCL Keywords

The following **keywords** can be used in an OCL expression:

- **if – then – else – endif**: conditional expression
- **not**, **or**, **and**, **xor**, **implies**: boolean operators
- **def**: global definitions
- **let–in**: local definitions

Running Example: Airport Class Diagram



Invariants

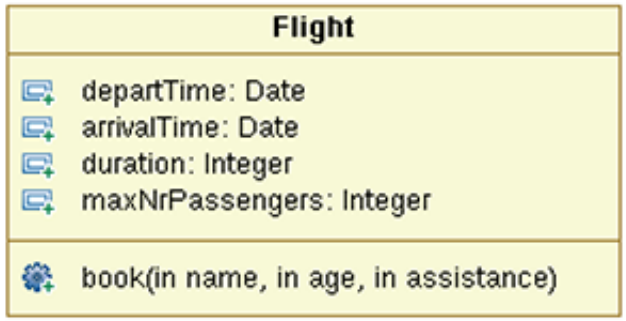
Invariants in OCL I

- An invariant is specified with the keyword **inv**.
- The class to which the invariant refers is the context of the invariant.
- It is followed by a boolean expression that states the invariant.
- All attributes of the context class may be used in this invariant.

Invariants in OCL II

context Flight
inv : **self**.duration < 4

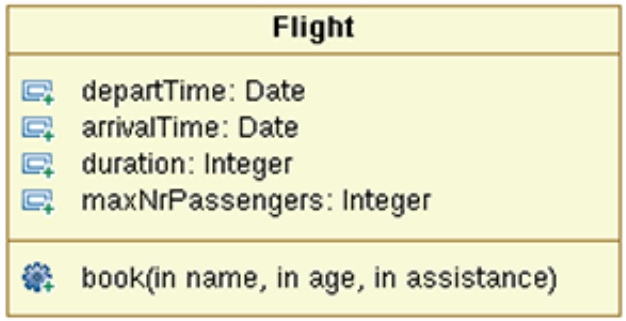
- Meaning: ?



Invariants in OCL II

context Flight
inv : **self**.duration < 4

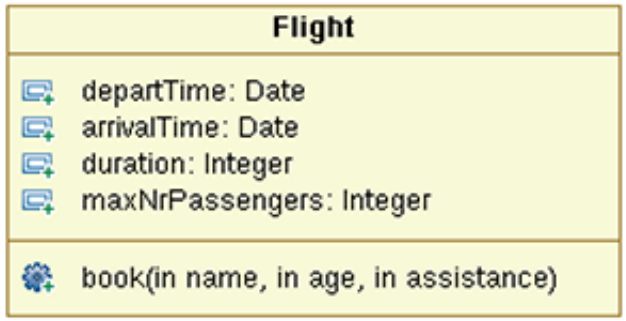
- Meaning:
  - Each flight has a duration of less than 4h.



Invariants in OCL II

context Flight
inv : **self**.duration < 4

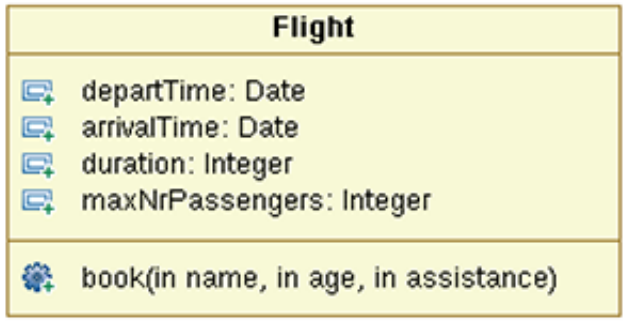
- Meaning:
  - Each flight has a duration of less than 4h.
  - **self** is an instance of type *Flight*.



Invariants in OCL II

context Flight
inv : **self**.duration < 4

- Meaning:
  - Each flight has a duration of less than 4h.
  - **self** is an instance of type *Flight*.
  - **self** can be viewed as the object from where the evaluation of the expression starts.

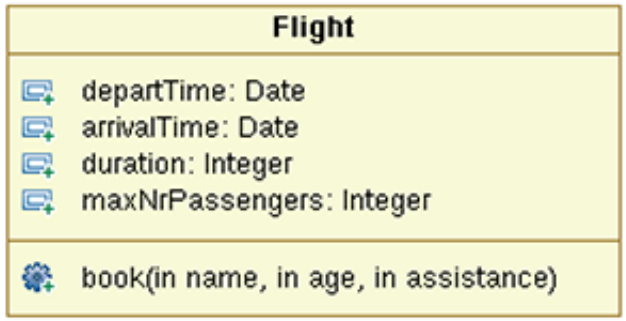


Invariants in OCL II

context Flight
inv : **self**.duration < 4

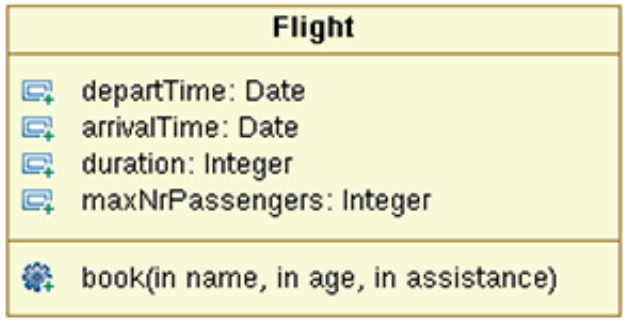
- Meaning:
  - Each flight has a duration of less than 4h.
  - **self** is an instance of type *Flight*.
  - **self** can be viewed as the object from where the evaluation of the expression starts.

Equivalent:



Invariants in OCL II

context Flight
inv : **self**.duration $<$ 4



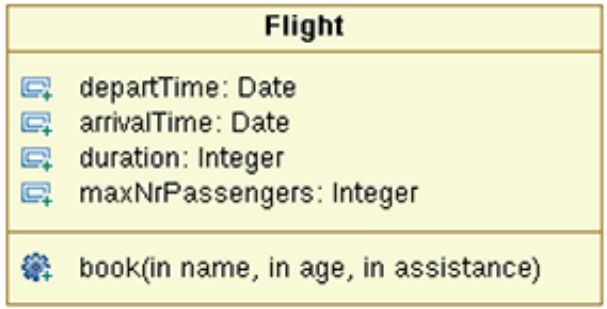
- Meaning:
  - Each flight has a duration of less than 4h.
  - **self** is an instance of type *Flight*.
  - **self** can be viewed as the object from where the evaluation of the expression starts.

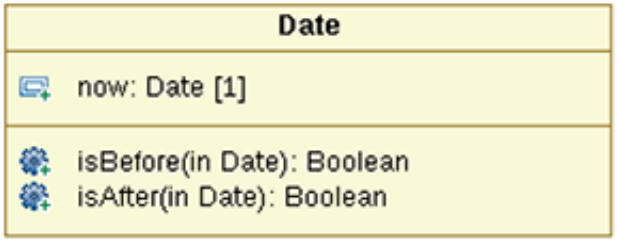
Equivalent:

context Flight
inv : duration $<$ 4

Invariants in OCL III

- If the type of the attribute is a class, the attributes or **query operations** defined on that class can be used to write the invariant (using a **dot notation**).
- Query operation:
  An operation that does not change the value of any attributes.



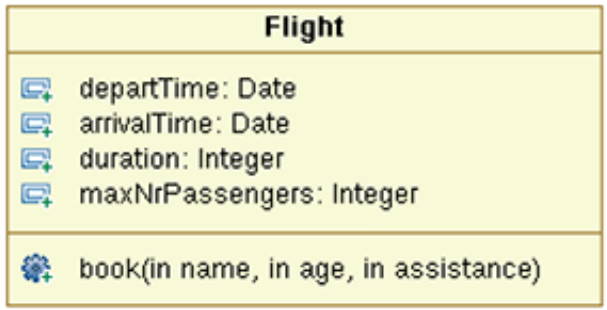


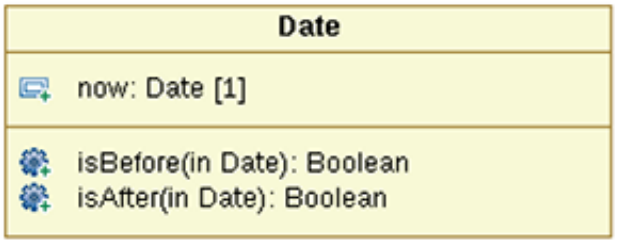
context Flight
inv : departTime.isBefore(arrivalTime)

- Meaning: ?

Invariants in OCL III

- If the type of the attribute is a class, the attributes or **query operations** defined on that class can be used to write the invariant (using a **dot notation**).
- Query operation:
  An operation that does not change the value of any attributes.





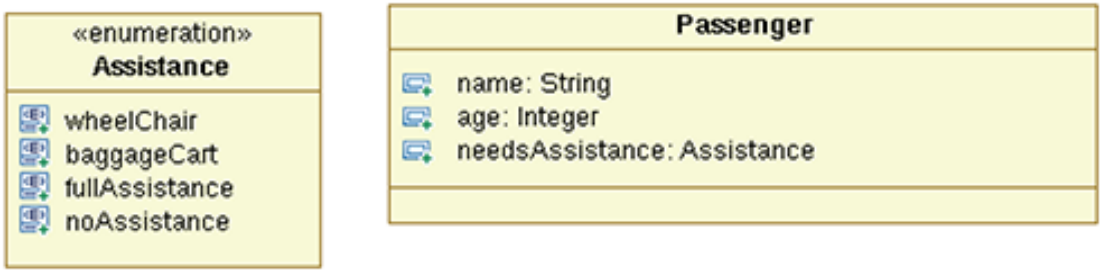
context Flight
inv : departTime.isBefore(arrivalTime)

- Meaning:
  The departure date is earlier than the arrival date.

Enumerations

Enumeration Types

- Enumeration are datatypes in UML.
- Within OCL one can refer to the value of an enumeration by using the datatype followed by :: and the value.

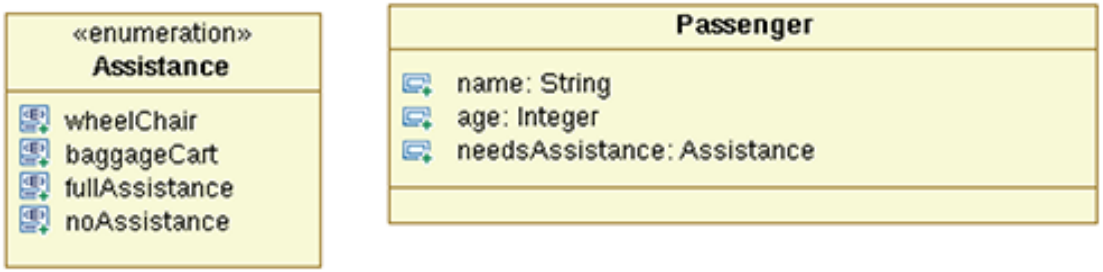


context Passanger
inv : **self**.age > 95 **implies**
self.needsAssistance = Assistance :: wheelchair

- Meaning: ?

Enumeration Types

- Enumeration are datatypes in UML.
- Within OCL one can refer to the value of an enumeration by using the datatype followed by :: and the value.



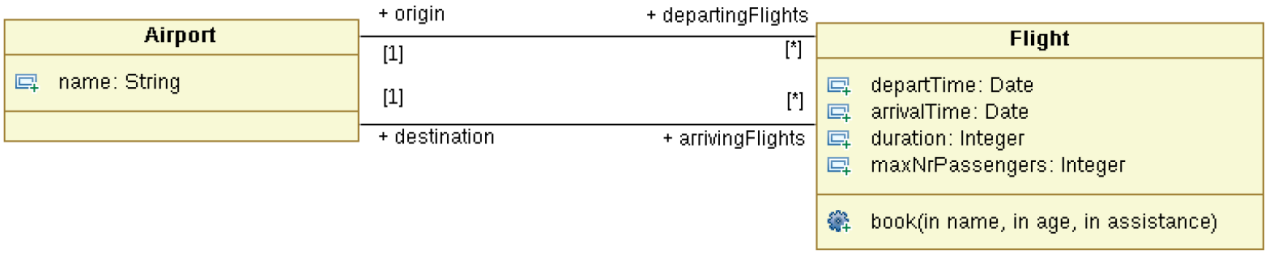
context Passanger
inv : **self**.age $>$ 95 **implies**
self.needsAssistance = Assistance :: wheelchair

- Meaning:
  - Each passenger with an age above 95 needs assistence by a wheelchair.

Associations and Navigations

Associations and Navigation I

- Every association is a navigation path.
- The context of the expression is the starting point.
- Role names (or association ends) are used to identify the navigated associations.

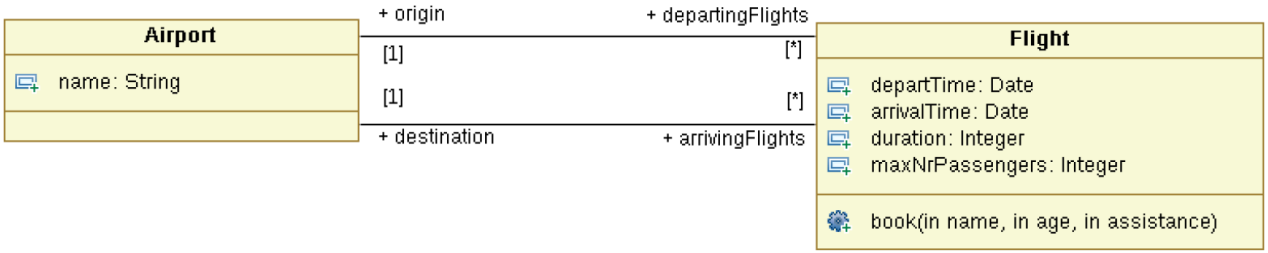


context Flight
inv : origin <> destination

- Meaning: **?**

Associations and Navigation I

- Every association is a navigation path.
- The context of the expression is the starting point.
- Role names (or association ends) are used to identify the navigated associations.



context Flight
inv : origin <> destination

- Meaning:
  The origin of each flight is unequal to the destination.

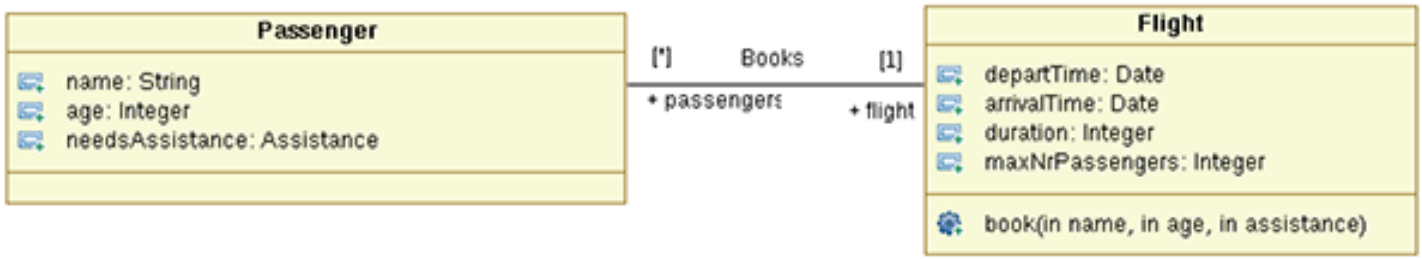
Associations and Navigation II

- Often associations are one-to-many or many-to-many, which means that constraints on a collection of objects are necessary.
- OCL expressions either state a fact about all objects in the collection or states facts about the collection itself.

Collections

Using Collection Operations I

- One of the collection operations can be used whenever navigation results in a **collection of objects**.
- An arrow (–>) between the rolename and the operation indicates the use of one of the predefined collection operations.

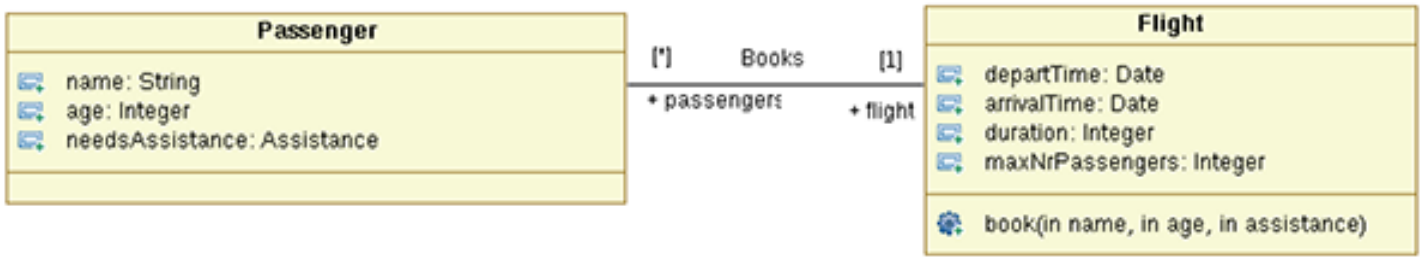


context Flight
inv : passengers –> size() <= maxNrPassengers

- Meaning: **?**

Using Collection Operations I

- One of the collection operations can be used whenever navigation results in a **collection of objects**.
- An arrow (–>) between the rolename and the operation indicates the use of one of the predefined collection operations.



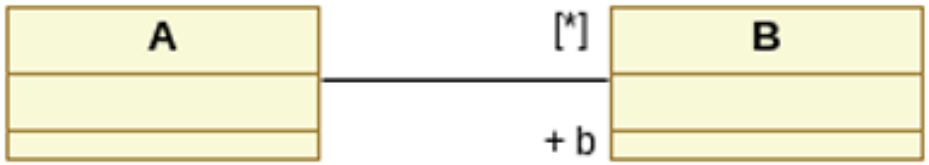
context Flight
inv : passengers –> size() <= maxNrPassengers

- Meaning:
  - The number of passengers is less or equal to the maximum number of seats.

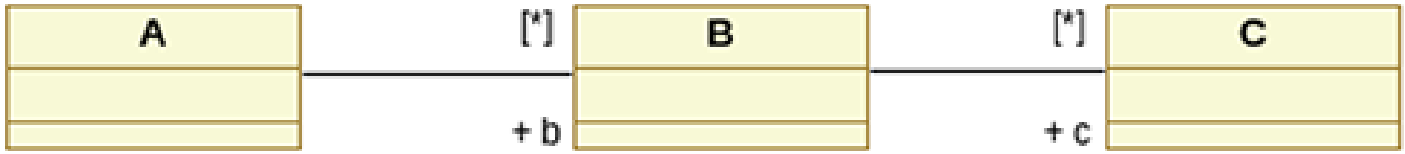
Using Collection Operations II

A collection of objects may be:

- Set:
  - Each element may occur only once.
  - Single navigation of an association results in a Set.

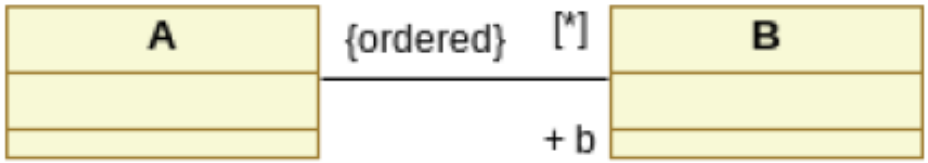


- Bag:
  - Elements may be present more than once.
  - Combined navigation results in a Bag.



Using Collection Operations III

- OrderedSet:
  - A set in which the elements are ordered.
  - Single navigation of an association that is marked as $\{ordered\}$ results in an OrderedSet.



- Sequence:
  - A Bag in which the elements are ordered.
  - Combined navigation of associations, at least one of which is marked as $\{ordered\}$, results in an Sequence.

Acessing the Elements of a Collection I

Conversion operations:

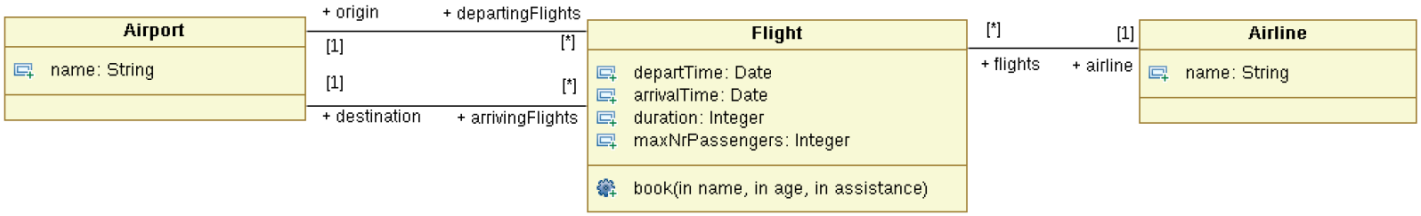
- A Set can be converted into
  - a Bag (*asBag*())
  - an OrderedSet (*asOrderedSet*())
  - a Sequence (*asSequence*())
- A Bag can be converted into
  - a Set (*asSet*())
  - an OrderedSet (*asOrderedSet*())
  - a Sequence (*asSequence*())
- A Sequence can be converted into
  - a Set (*asSet*())
  - a Bag (*asBag*())
  - an OrderedSet (*asOrderedSet*())

Acessing the Elements of a Collection II

- OrderedSet and Sequence provide operations to
  - access the first element ($first()$)
  - access the last element ($last()$)
  - access the i-th element ($at(i : Integer)$)
- All kind of collections provide iterator operations:
  - *collect* operation
  - *select* and *reject* operations
  - *forAll* operation
  - *exists* and *one* operations
  - *any* operation
  - ...

The *collect* Operation I

- The *collect* operation can be used to specify a collection that is derived from some other collection, but which contains different objects from the original collection (i.e., it is not a sub-collection).

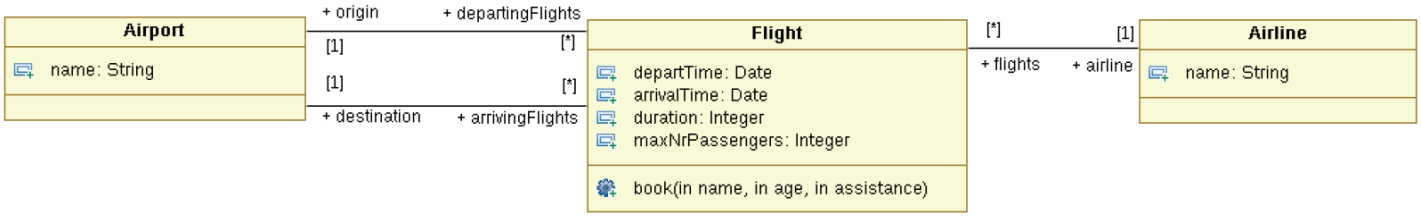


context Airport
inv : arrivingFlights –> size() =
arrivingFlights –> collect(airline) –> size()

- Meaning: **?**

The *collect* Operation I

- The *collect* operation can be used to specify a collection that is derived from some other collection, but which contains different objects from the original collection (i.e., it is not a sub-collection).



context Airport
inv : arrivingFlights –> size() =
arrivingFlights –> collect(airline) –> size()

- Meaning:
  Each arriving flight is carried out by an airline.

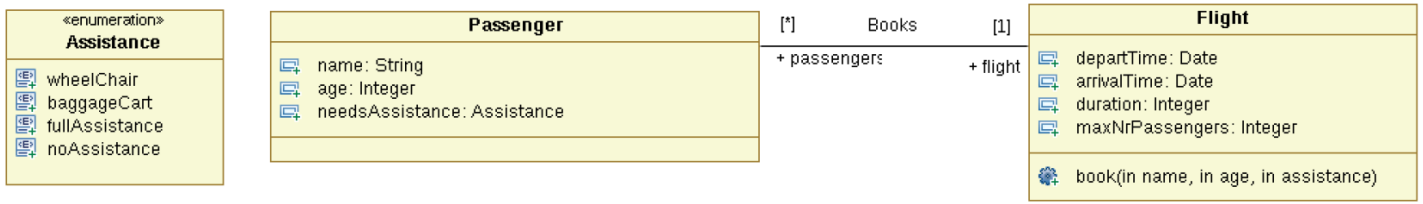
The *collect* Operation II

- When the source collection is
  - a Set the resulting collection is not a Set but a Bag
  - a Sequence or an OrderedSet, the resulting collection is a Sequence
- The dot notation is an abbreviation for applying the *collect* operation:

```
context Airport
inv : arrivingFlights -> size() =
      arrivingFlights.airline -> size()
```

The *select* Operation

- The *select* operation
  - specifies a subset of a collection
  - takes a boolean expression as parameter
  - selects all elements from the collection for which the expression evaluates to **true**

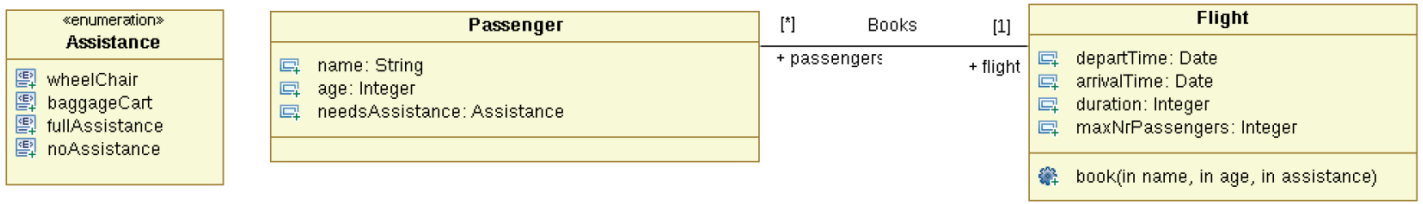


context Flight
inv : passangers –> **select**(p : Passanger | p.needsAssistance <>
Assistance :: noAssistance) –> size() <= 10

- Meaning: **?**

The *select* Operation

- The *select* operation
  - specifies a subset of a collection
  - takes a boolean expression as parameter
  - selects all elements from the collection for which the expression evaluates to **true**

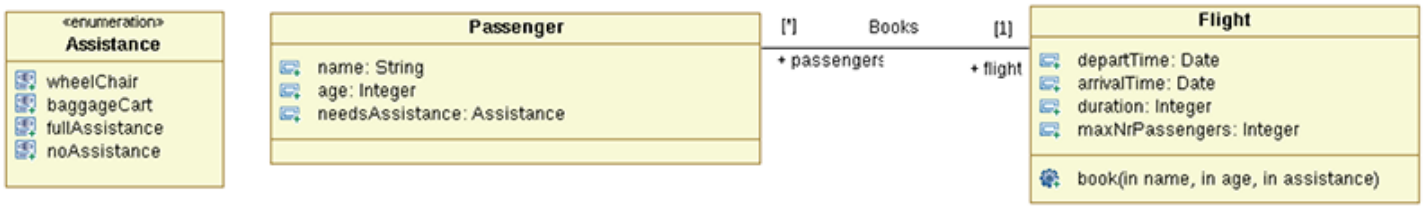


context Flight
inv : passangers –> **select**(p : Passanger | p.needsAssistance <>
Assistance :: noAssistance) –> size() <= 10

- Meaning:
  The number of passengers who need assistance is less or equal to 10.

The *reject* Operation

- The *reject* operation is similar to *select* operation.
- *reject* selects all elements from the collection for which the boolean expression evaluates to **false**.

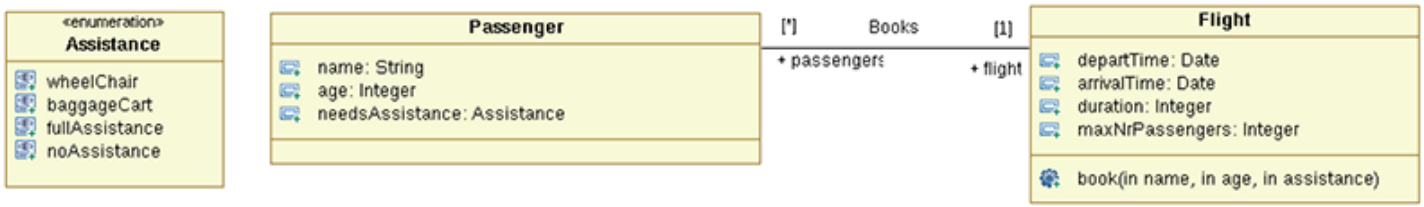


context Flight
inv : passangers –> **reject**(p : Passanger | p.needsAssistance = Assistance :: noAssistance) –> size() <= 10

- Meaning: **?**

The *reject* Operation

- The *reject* operation is similar to *select* operation.
- *reject* selects all elements from the collection for which the boolean expression evaluates to **false**.



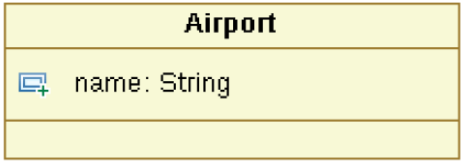
context Flight
inv : passangers –> **reject**(p : Passanger | p.needsAssistance = Assistance :: noAssistance) –> size() <= 10

- Meaning:
  The number of passengers who need assistance is less or equal to 10.

The *forAll* Operation I

- The *forAll* operation allows one to specify a boolean expression, which must hold for all elements in a collection.
- The result of the operation is a boolean value:
  - **true** if the expression evaluates to true for all elements in the collection
  - otherwise **false**

The *forAll* Operation II

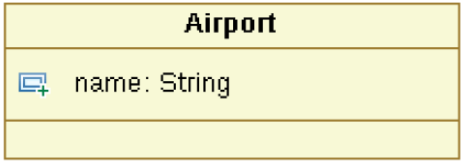


Example

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

- *class.allInstances*(): collection of all instances of the class
- Meaning: **?**

The *forAll* Operation II

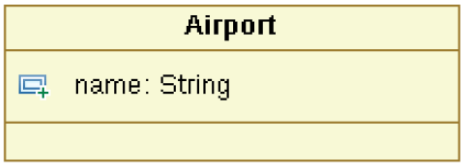


Example

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

- *class.allInstances*(): collection of all instances of the class
- Meaning:
  Each airport name is unique.

The *forAll* Operation II

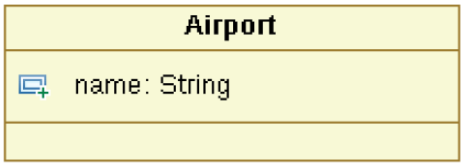


Example

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

- *class.allInstances*(): collection of all instances of the class
- Meaning:
  Each airport name is unique.
- Equivalent:

The *forAll* Operation II



Example

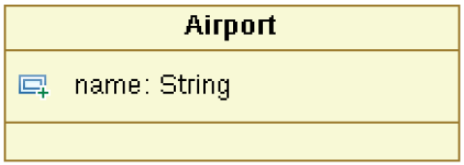
context Airport
inv : Airport.allInstances() –> **forAll**(a1 : Airport, a2 : Airport |
a1 <> a2 **implies** a1.name <> a2.name)

- *class.allInstances*(): collection of all instances of the class
- Meaning:
  Each airport name is unique.
- Equivalent:

context Airport
inv : Airport.allInstances() –> isUnique(name)

The *exists* Operation

- The *exists* operation allows one to specify a boolean expression, which must hold for at least one element in a collection.
- The result of the operation is a boolean value:
  - **true** if the expression evaluates to true for **at least** one element in the collection
  - otherwise **false**

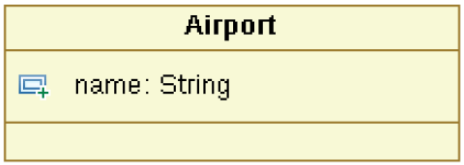



```
context Airport
inv : Airport.allInstances() -> exists(a : Airport |
      a.name = 'Sultan Iskandar Muda')
```

- Meaning: ?

The *exists* Operation

- The *exists* operation allows one to specify a boolean expression, which must hold for at least one element in a collection.
- The result of the operation is a boolean value:
  - **true** if the expression evaluates to true for **at least** one element in the collection
  - otherwise **false**

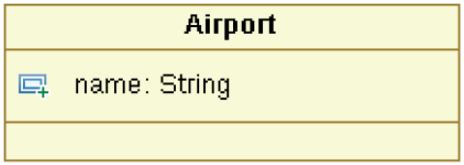



```
context Airport
inv : Airport.allInstances() -> exists(a : Airport |
      a.name = 'Sultan Iskandar Muda')
```

- Meaning:
  There exists an airport with the name 'Sultan Iskandar Muda'.

The *one* Operation

- The *one* operation allows one to specify a boolean expression, which must hold for **exactly** one element in a collection.
- The result of the operation is a boolean value:
  - **true** if the expression evaluates to true for **exactly** one element in the collection
  - otherwise **false**

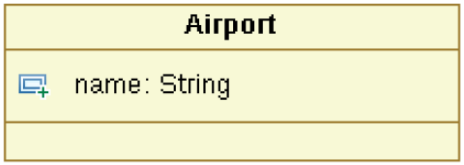



```
context Airport
inv : Airport.allInstances() -> one(a : Airport |
      a.name = 'Sultan Iskandar Muda')
```

- Meaning: **?**

The *one* Operation

- The *one* operation allows one to specify a boolean expression, which must hold for **exactly** one element in a collection.
- The result of the operation is a boolean value:
  - **true** if the expression evaluates to true for **exactly** one element in the collection
  - otherwise **false**




```
context Airport
inv : Airport.allInstances() -> one(a : Airport |
      a.name = 'Sultan Iskandar Muda')
```

- Meaning:
  There exists exactly one airport with the name 'Sultan Iskandar Muda'.

The *any* Operation

- The *any* operation returns the first element in the source collection for which a specified boolean expression evaluates to **true**.

context Airport
inv : Airport.allInstances() –> **any**(a : Airport |
a.name = 'Sultan Iskandar Muda').departingFlights –> size() < 10

- Meaning: **?**

The *any* Operation

- The *any* operation returns the first element in the source collection for which a specified boolean expression evaluates to **true**.

context Airport
inv : Airport.allInstances() –> **any**(a : Airport |
a.name = 'Sultan Iskandar Muda').departingFlights –> size() < 10

- Meaning:
  The number of flights that depart from the airport name 'Sultan Iskandar Muda' is less than 10.

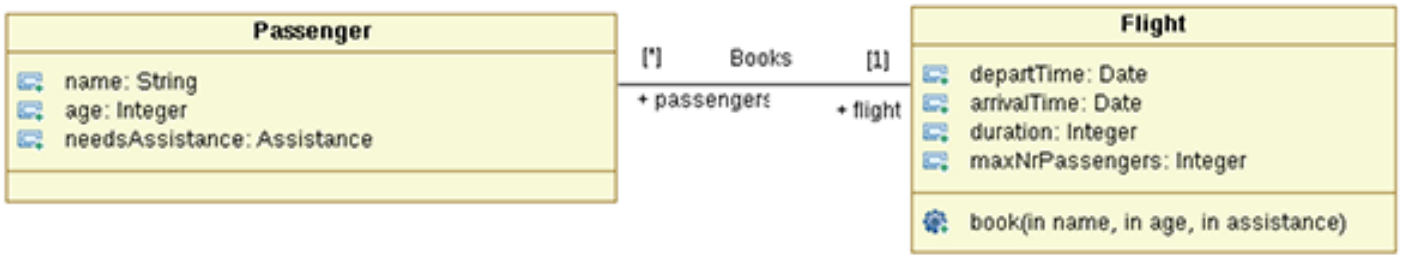
Pre- and Post-Conditions

Pre- and Post-Conditions

- In class diagrams only the syntax and signature of operations can be defined.
- Operation **semantics** can be specified through **pre- and post-conditions** in OCL.
- The **context** declaration for a pre- and postcondition uses the keyword context followed by the class name and the operation declaration. Class name and operation declaration are separated by "::".
- The names of parameters of the operation can be used in the pre- and postcondition.

Precondition

- Condition on the parameters and initial object state that must hold for the operation call to be valid.



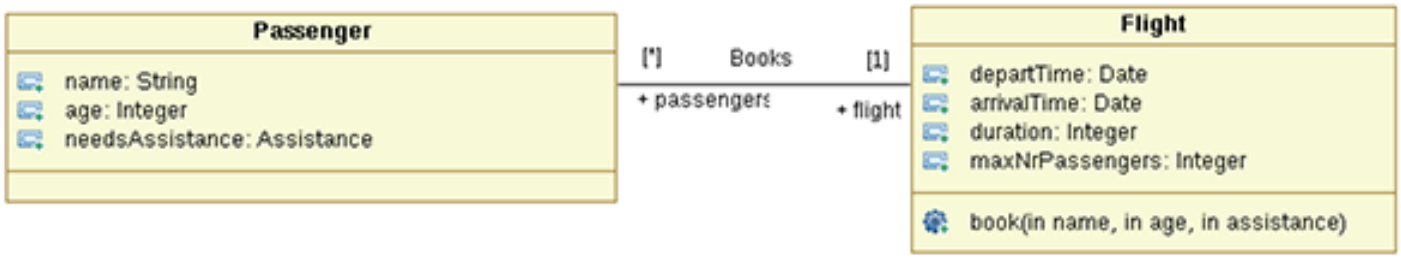
Example

context Flight :: book(name : String, age : Integer,
assistance : Assistance)
pre : passangers –> size() < maxNrPassangers

- Meaning: **?**

Precondition

- Condition on the parameters and initial object state that must hold for the operation call to be valid.



Example

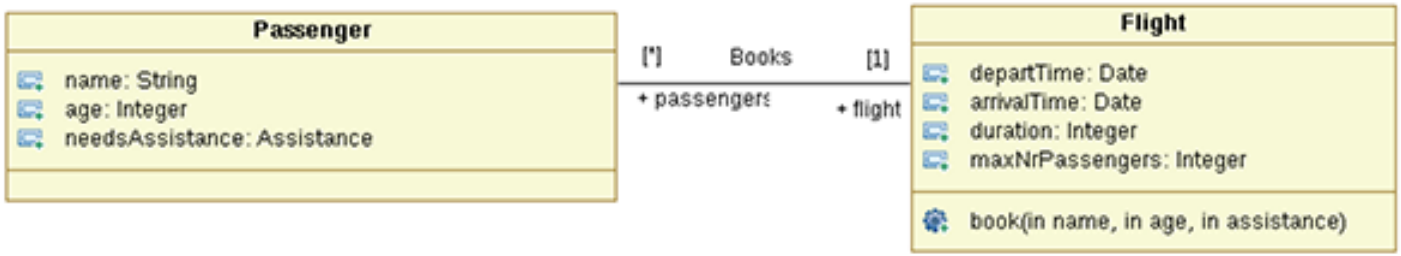
context Flight :: book(name : String, age : Integer, assistance : Assistance)
pre : passangers –> size() < maxNrPassangers

- Meaning:
  The number of passengers registered for *flight* before the execution of *book* must be less than *maxNrPassengers*.

Postcondition I

- Condition on the return value, final object state, parameters, and initial object state that must hold after the operation execution, assuming the precondition is satisfied.
- Specifies intended result and state change (what), but not the steps (how).
- The pre state of an object field is denoted with **@pre**.
- the returned value is denoted with the keyword **result**.

Postcondition II

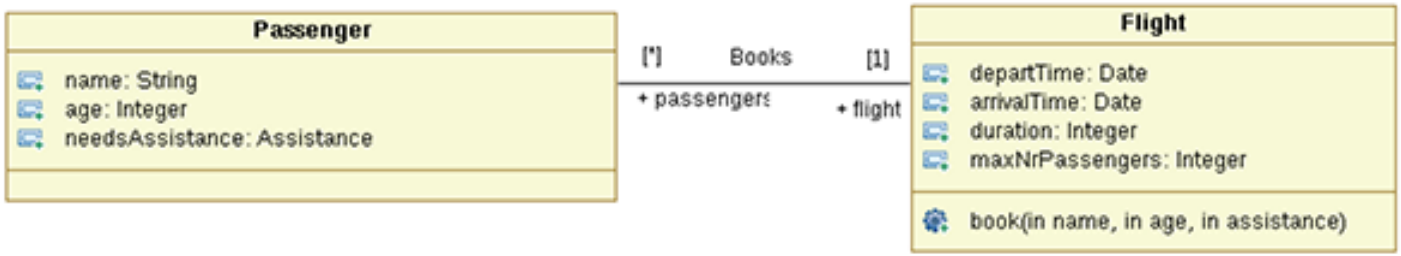


Example

context Flight :: book(name : String, age : Integer,
assistance : Assistance)
post : passengers –> size() – passengers@pre –> size() = 1
and passengers –> exists(p : Passenger | p.age = age
and p.name = name
and p.needsAssistance = assistance)

- Meaning: ?

Postcondition II



Example

context Flight :: book(name : String, age : Integer,
assistance : Assistance)
post : passengers –> size() – passengers@pre –> size() = 1
and passengers –> exists(p : Passenger | p.age = age
and p.name = name
and p.needsAssistance = assistance)

- Meaning:
  - one additional object exists after execution
  - the attributes of one object have been initialized using the parameter values of *book*

Calling Operations

- To specify that communication has taken place, the *hasSent*(^) operator is used.

Example

context Subject :: hasChanged()
post : observer^update(12, 14)

- The *observer^update*(12, 14) results in **true** if an update message with arguments 12 and 14 wes sent to *observer* during the execution of the operaton.
- *update*() is an operation that is defined in the class of *observer*
- The argument(s) of the message expression (12 and 14 in this example) must conform to the parameters of the operation.

Quick References

Primitive Types

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 and Tuples

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)

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) I

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. The result is flattened.

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

Iterator Expressions on Collection(T) II

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

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

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)

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

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

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)

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

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

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

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.

References I

[UML15] UML 2.5 - Object Management Group. http://www.omg.org/spec/UML/2.5/, 2015.