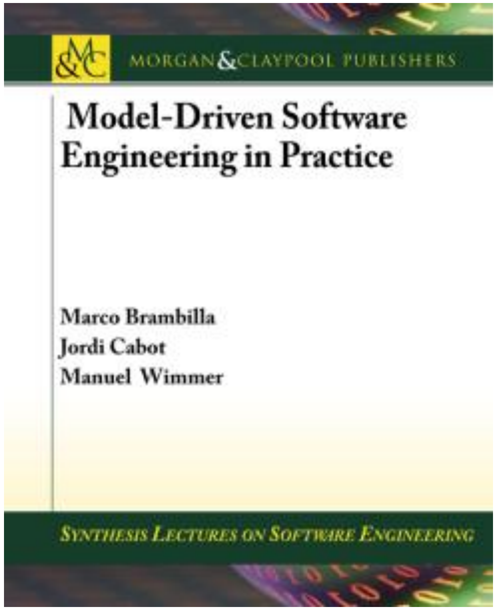
Chapter #6

MODELING LANGUAGES AT A GLANCE

Teaching material for the book
Model-Driven Software Engineering in Practice
by Marco Brambilla, Jordi Cabot, Manuel Wimmer.
Morgan & Claypool, USA, 2012.



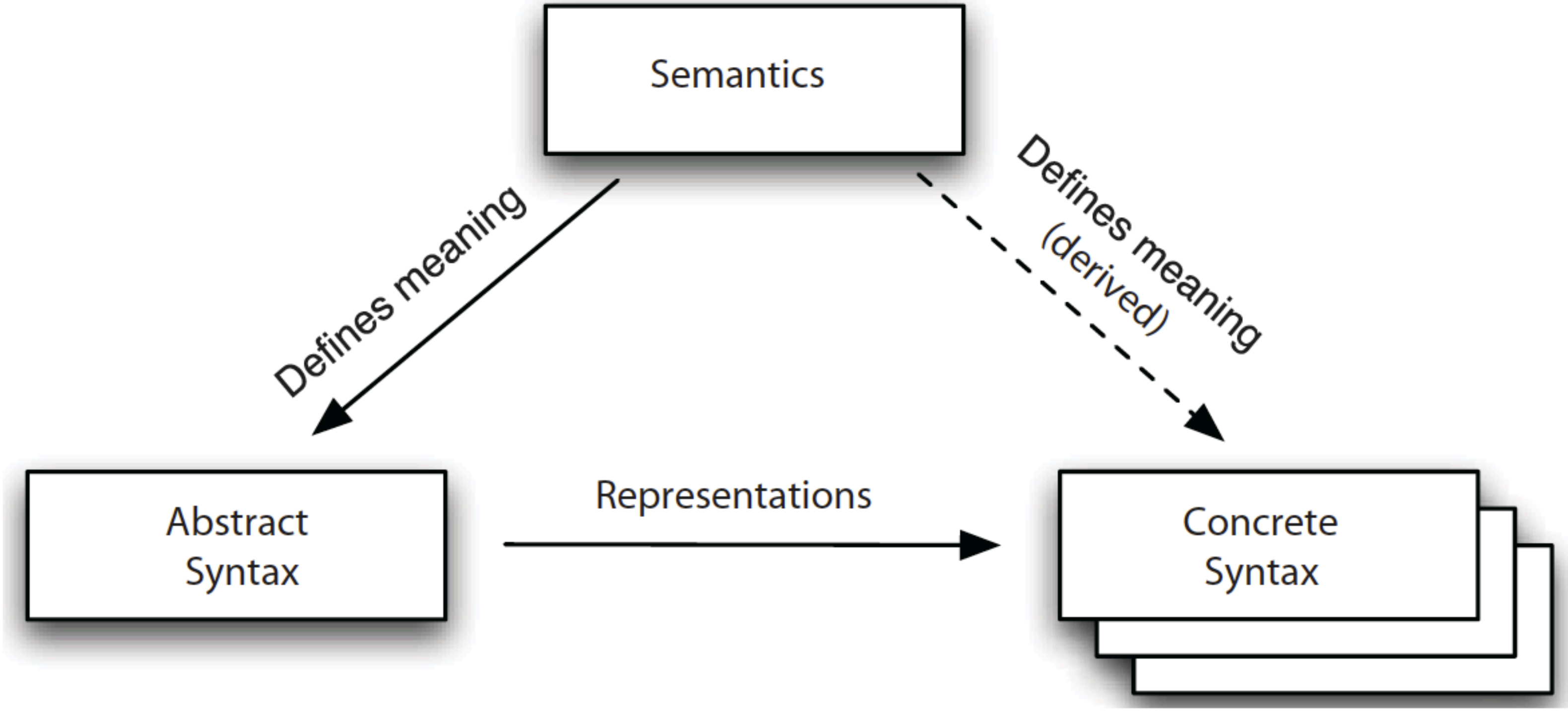
Contents

- DSL vs. GPL
- Example of GPL: UML
- DSL principles and dimensions
- OCL

Anatomy of a Modeling Language

- **Abstract syntax:** Describes the structure of the language and the way the different primitives can be combined together, independently of any particular representation or encoding.
- **Concrete syntax:** Describes specific representations of the modeling language, covering encoding and/or visual appearance.
- **Semantics:** Describing the meaning of the elements defined in the language and the meaning of the different ways of combining them.

Anatomy of a Modeling Language



DSL vs. GPL

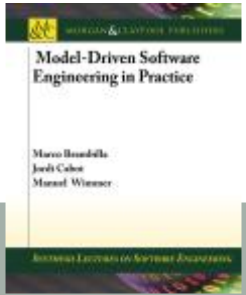
First distinction is between

- General Purpose languages (GPL or GPML) and
- Domain Specific languages (DSL or DSML)

(already discussed in Chapter 2)

- We take UML as an exemplary case of GPL

UML – UNIFIED MODELING LANGUAGE



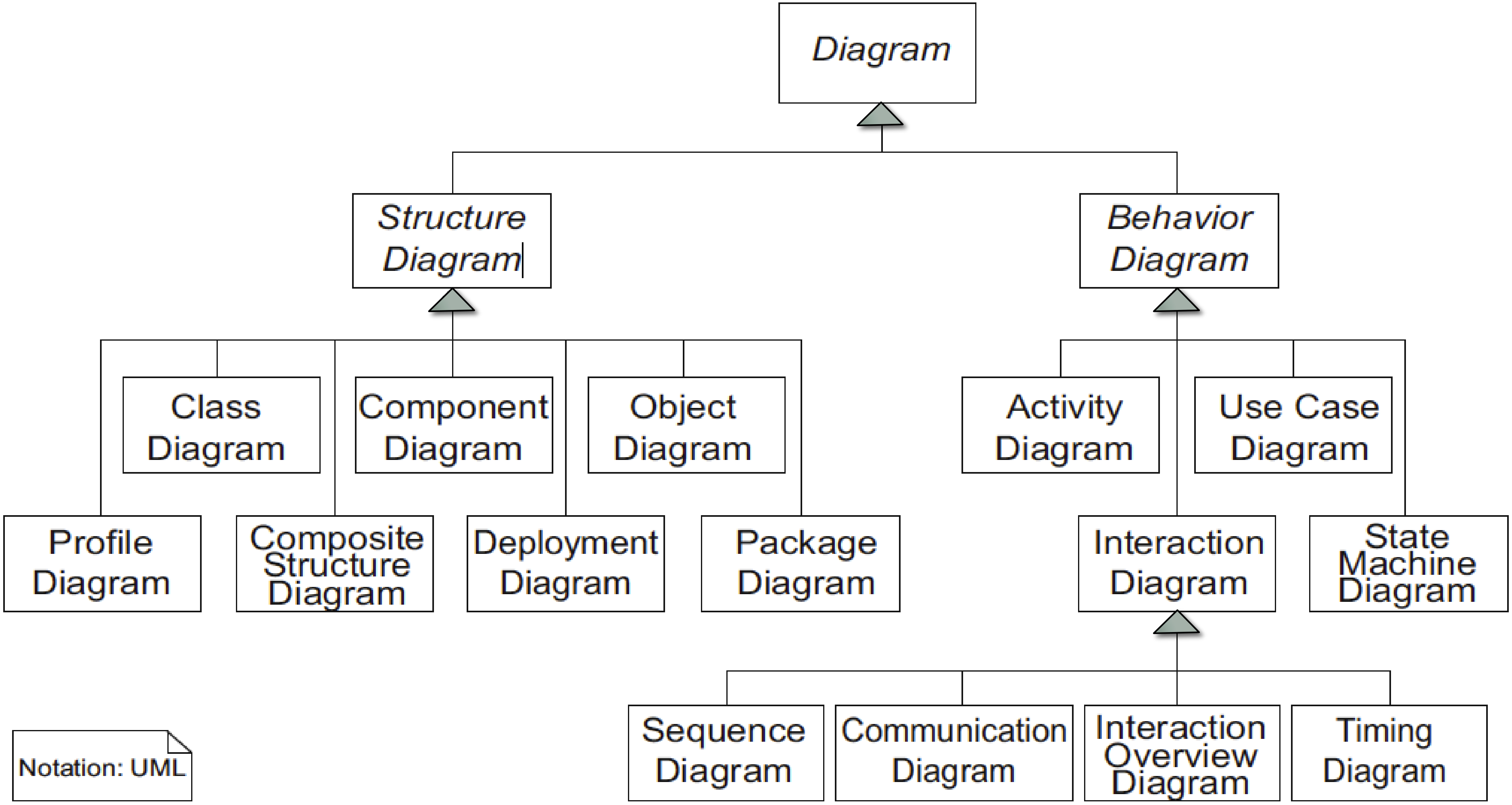
Overview of UML Diagrams

- There is no official UML diagram overview or diagram grouping.
- Although UML models and the repository underlying all diagrams are defined in UML, the definition of diagrams (i.e., special views of the repository) are relatively free.

Overview of UML Diagrams

- In UML a diagram is actually more than a collection of notational elements.
- For example, the package diagram describes the package symbol, the merge relationship, and so on.
- A class diagram describes a class, the association, and so on.
- Nevertheless, we can actually represent classes and packages together in one diagram.

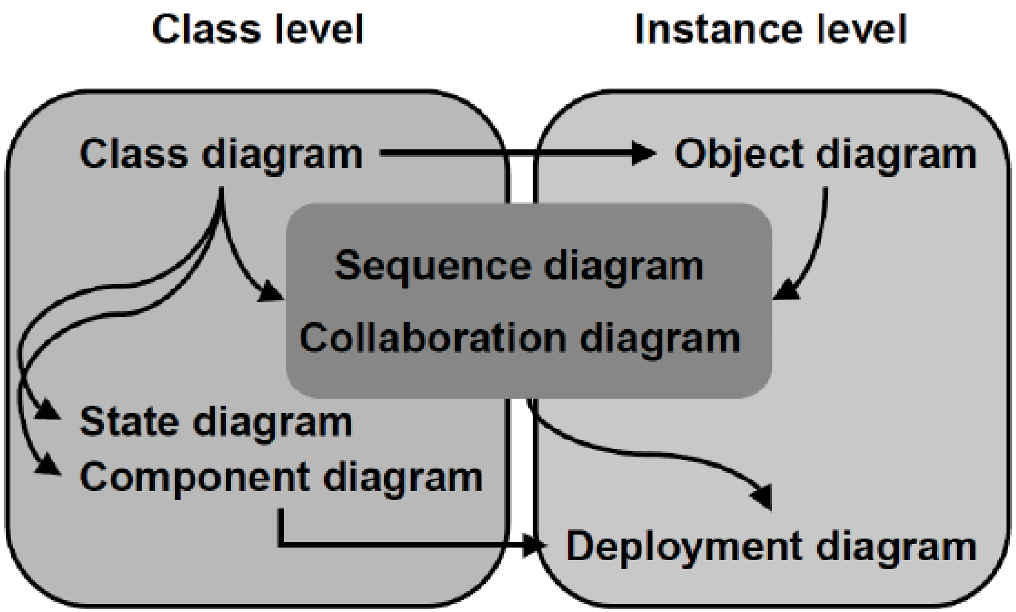
Overview of the UML diagrams



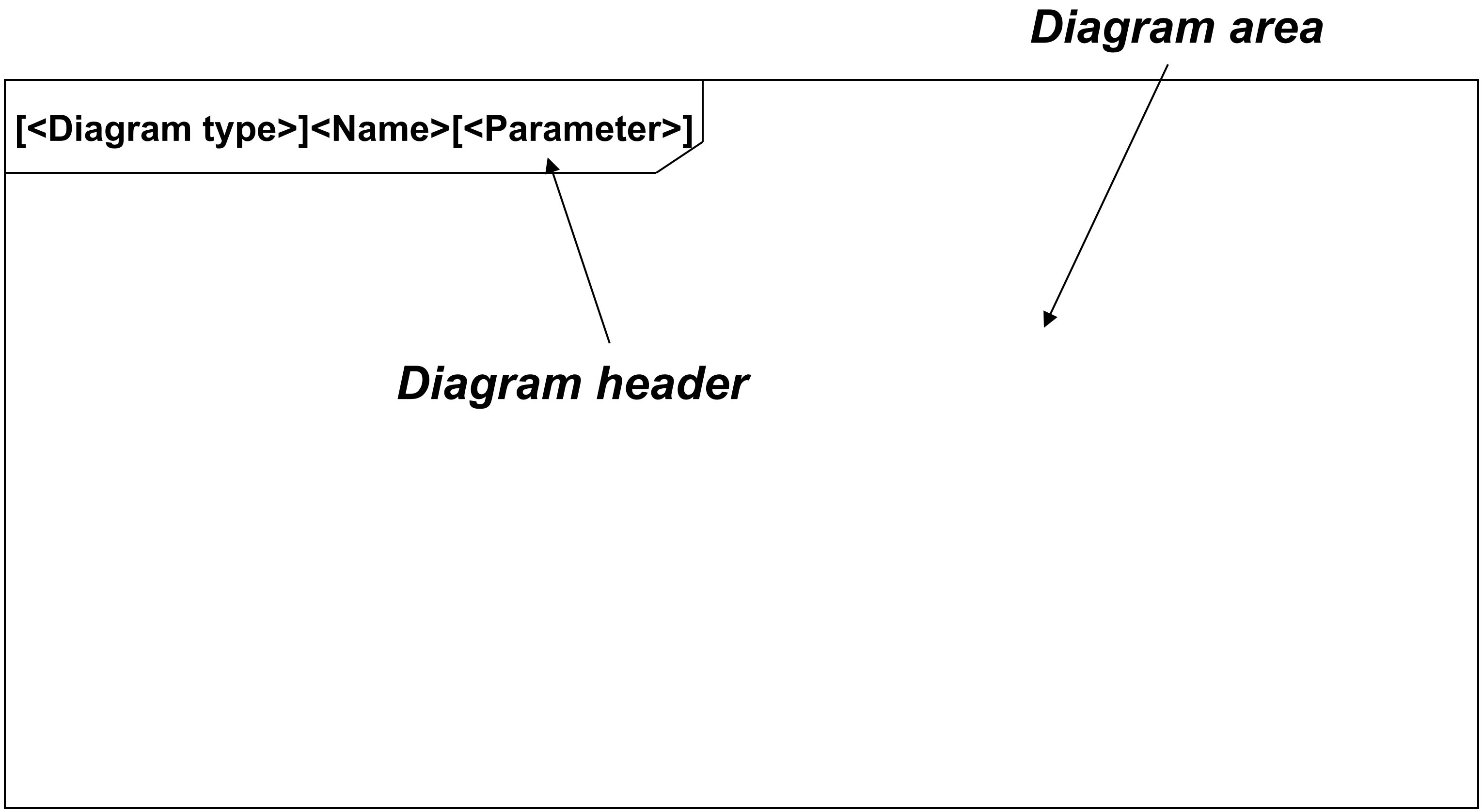
UML Design practices

- **Pattern-based design:** A set of very well-known design patterns, defined by the so-called Gang of Four
- **Using several integrated and orthogonal models together:** UML comprises a suite of diagrams that share some symbols and allow cross-referencing
- **Modeling at different levels of detail:** UML allows eliding details in diagrams when needed. Choose the right quantity of information to include in diagrams
- **Extensibility:** UML provides a good set of extensibility features which allow to design customized modeling languages if needed

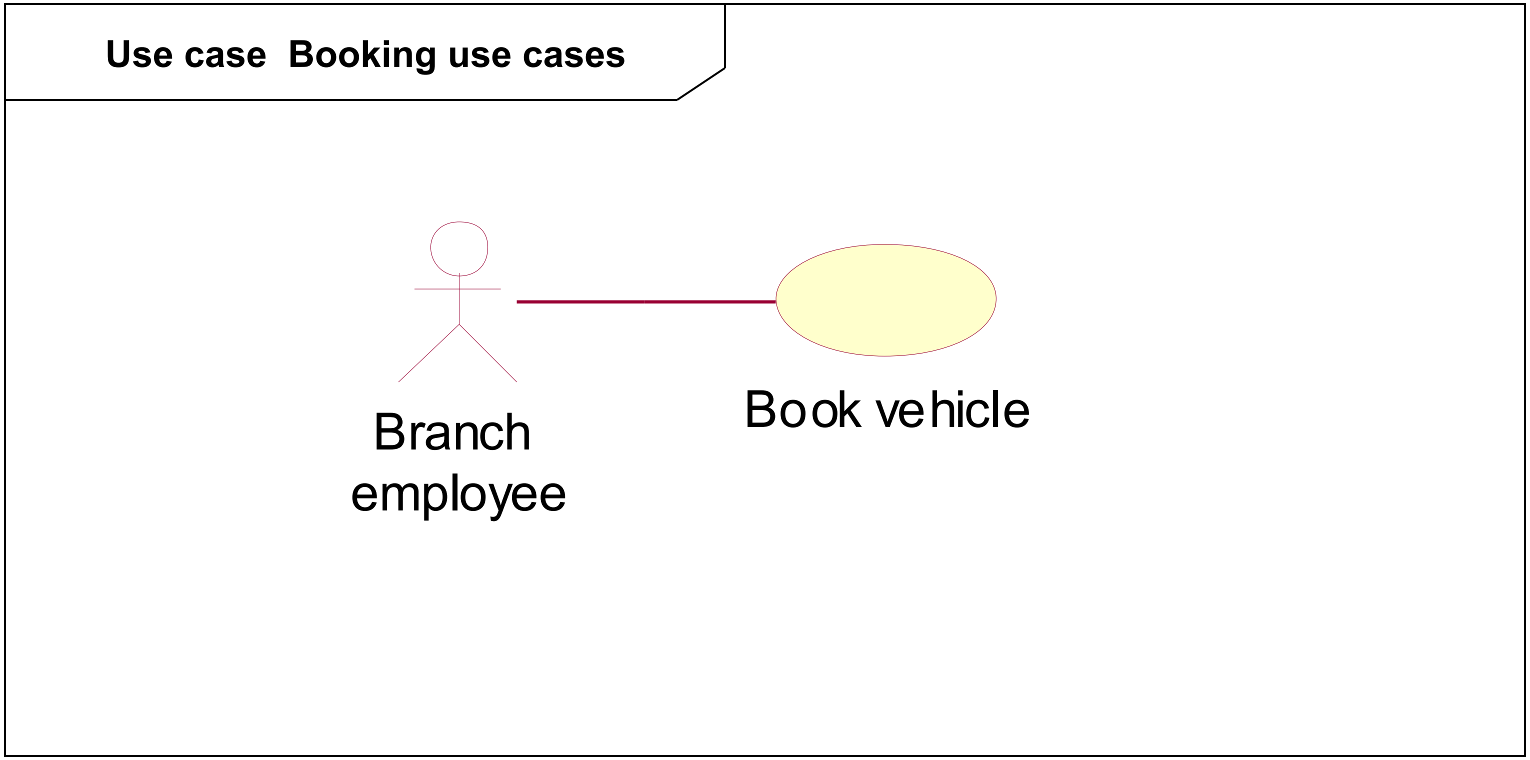
Class vs. instance in diagrams



Basic notation for diagrams



Example of a use case diagram



UML STRUCTURE DIAGRAMS (OR STATIC DIAGRAMS)

Structure diagrams (1)

Emphasize the **static description** of the elements that must be present in the system being modeled:

1. The **conceptual items** of interest for the system

- **Class diagram**: Describes the structure of a system by showing the classes of the systems, their attributes, and the relationships among the classes
- **Composite structure diagram**: Describes the internal structure of a class and the collaborations
- **Object diagram**: A view of the structure of example instances of modeled concepts

Structure diagrams (2)

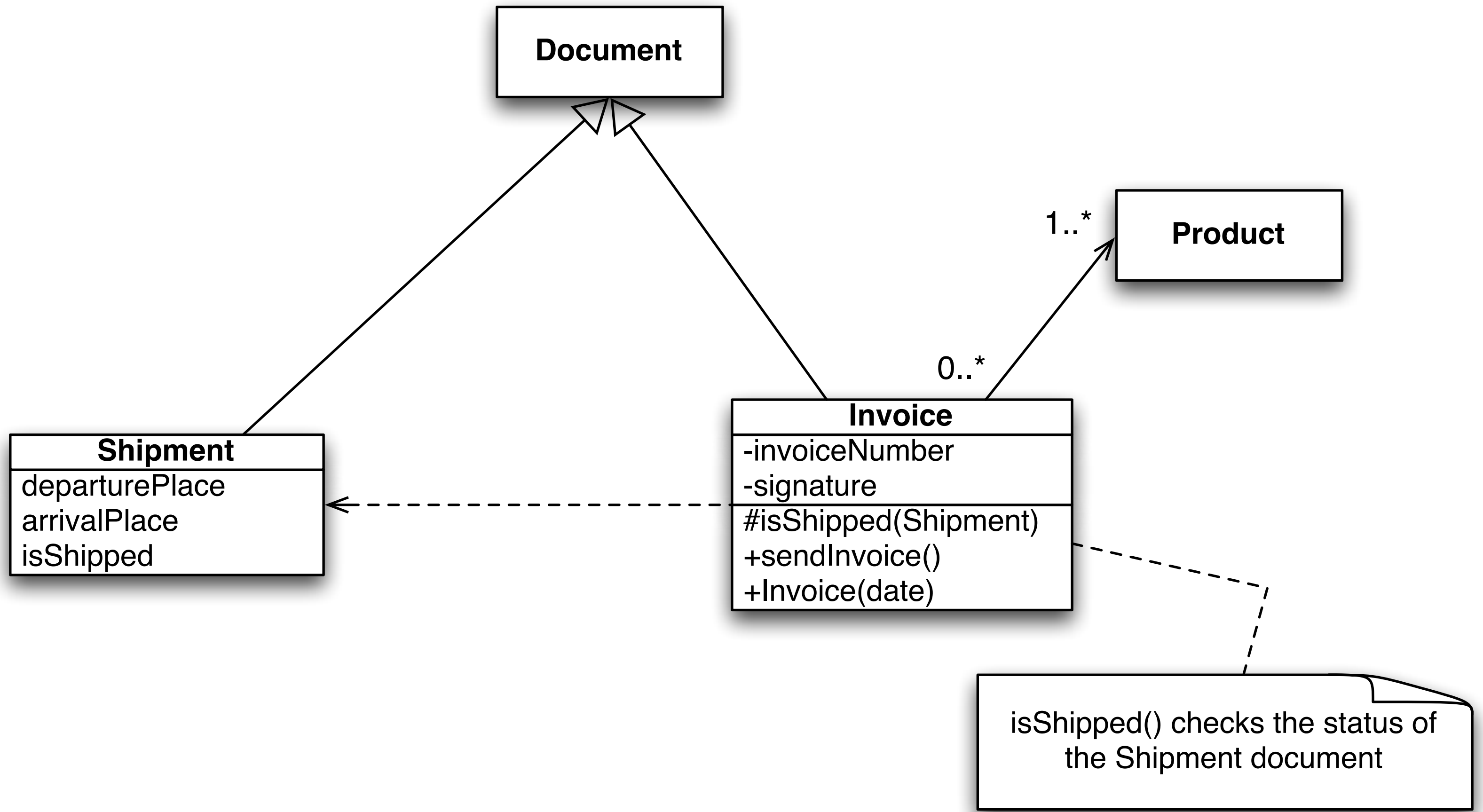
2. The **architectural organization and structure** of the system.

- **Package diagram**: Describes how a system is split up into logical groupings
- **Component diagram**: Describes how a software system is split up into components
- **Object diagram**: A view of the structure of example instances of modeled concepts

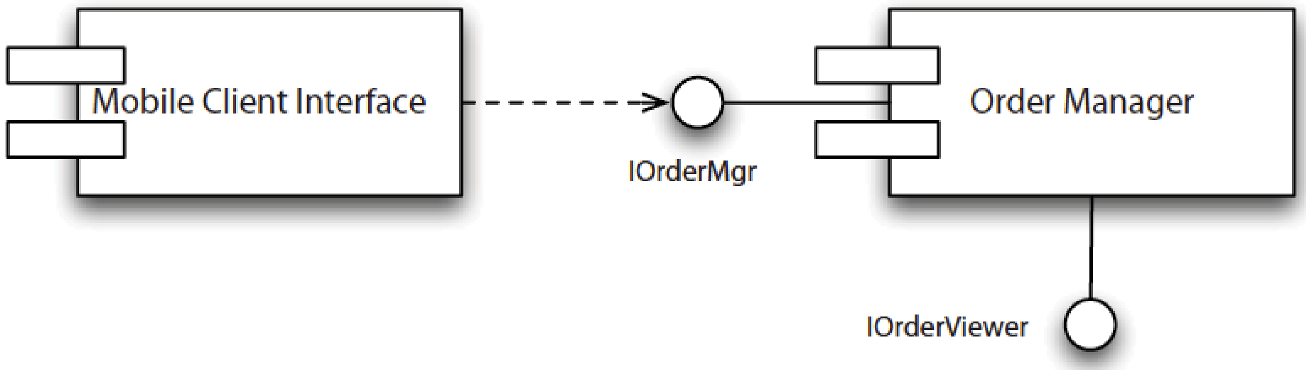
Class diagrams basic concepts

- The basis of UML is described in the **Kernel** package of the UML metamodel.
- Most class models have the superclass ***Element*** and has the ability to own other elements, shown by a composition relationship in the metamodel.
- That's the only ability an element has.

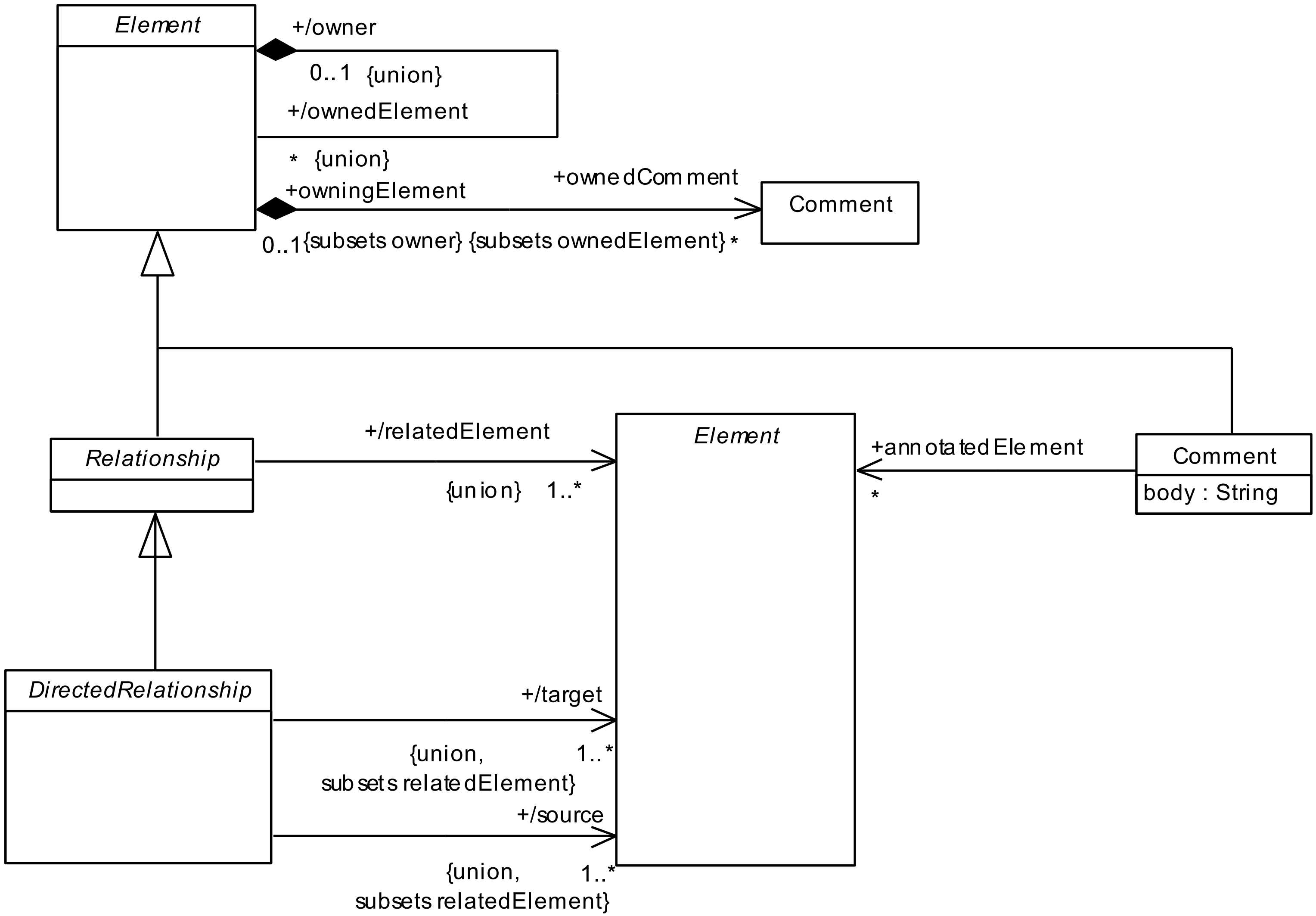
Class diagram example



Component Diagram example



The basic concepts in the UML metamodel for class diagrams



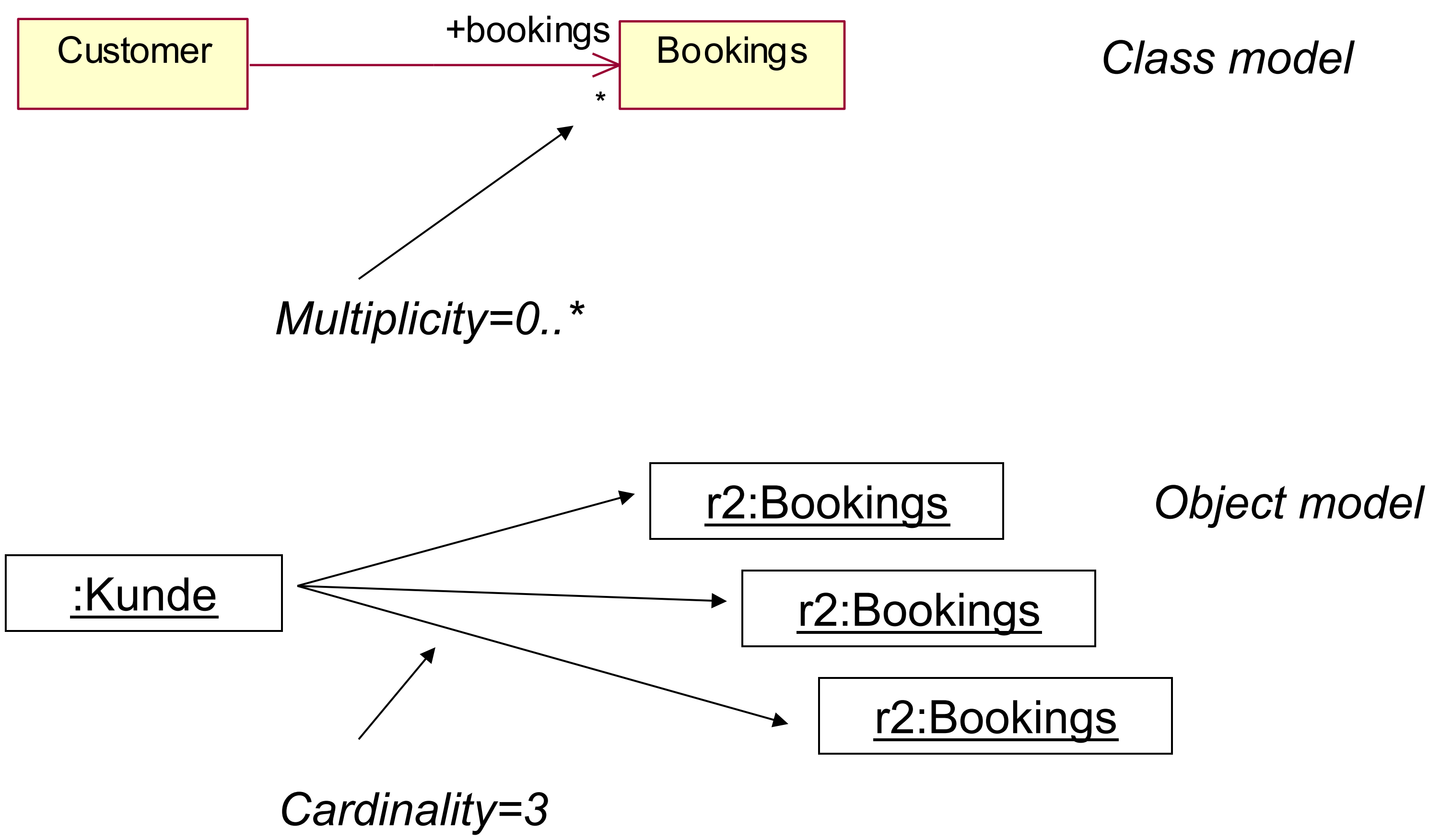
Named elements

- **Def**. A *named element* is an element that can have a name and a defined *visibility* (*public*, *private*, *protected*, *package*):
  - +=public
  - -=private
  - #=protected
  - ~=package
- The name of the element and its visibility are optional.

Multiplicities

- A ***multiplicity element*** is the definition of an interval of positive integers to specify allowable cardinalities.
- A ***cardinality*** is a concrete number of elements in a set.
- A multiplicity element is often simply called ***multiplicity***; the two terms are synonymous.

Example Multipicity & Cardinality



UML BEHAVIOURAL OR DYNAMIC DIAGRAMS

UML Behavioural Diagrams (1)

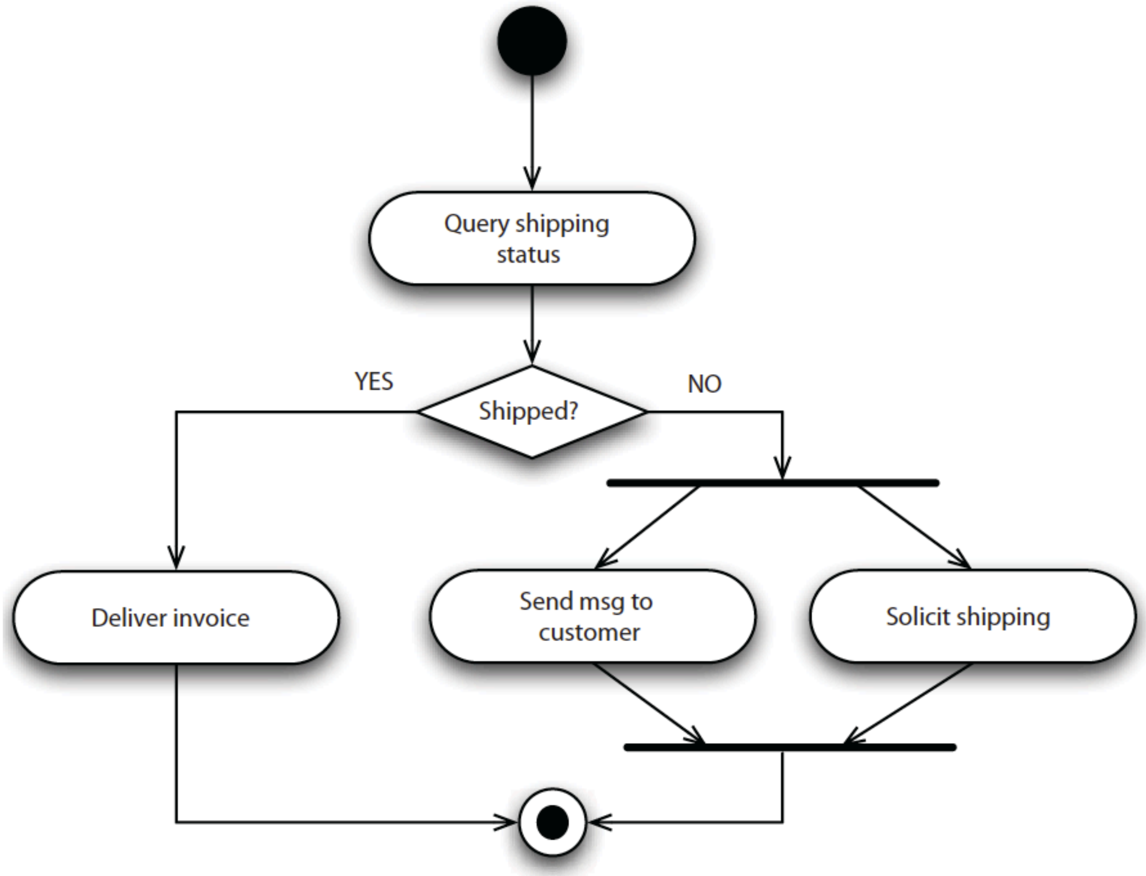
- **Use case diagram**: Describes the functionality provided by a system in terms of actors external to the system and their goals in using the system
- **Activity diagram**: Describes the step-by-step workflows of activities to be performed in a system for reaching a specific goal
- **State machine diagram (or statechart)**: Describes the states and state transitions of the system, of a subsystem, or of one specific object.

UML Behavioural Diagrams (2): Interaction diagrams

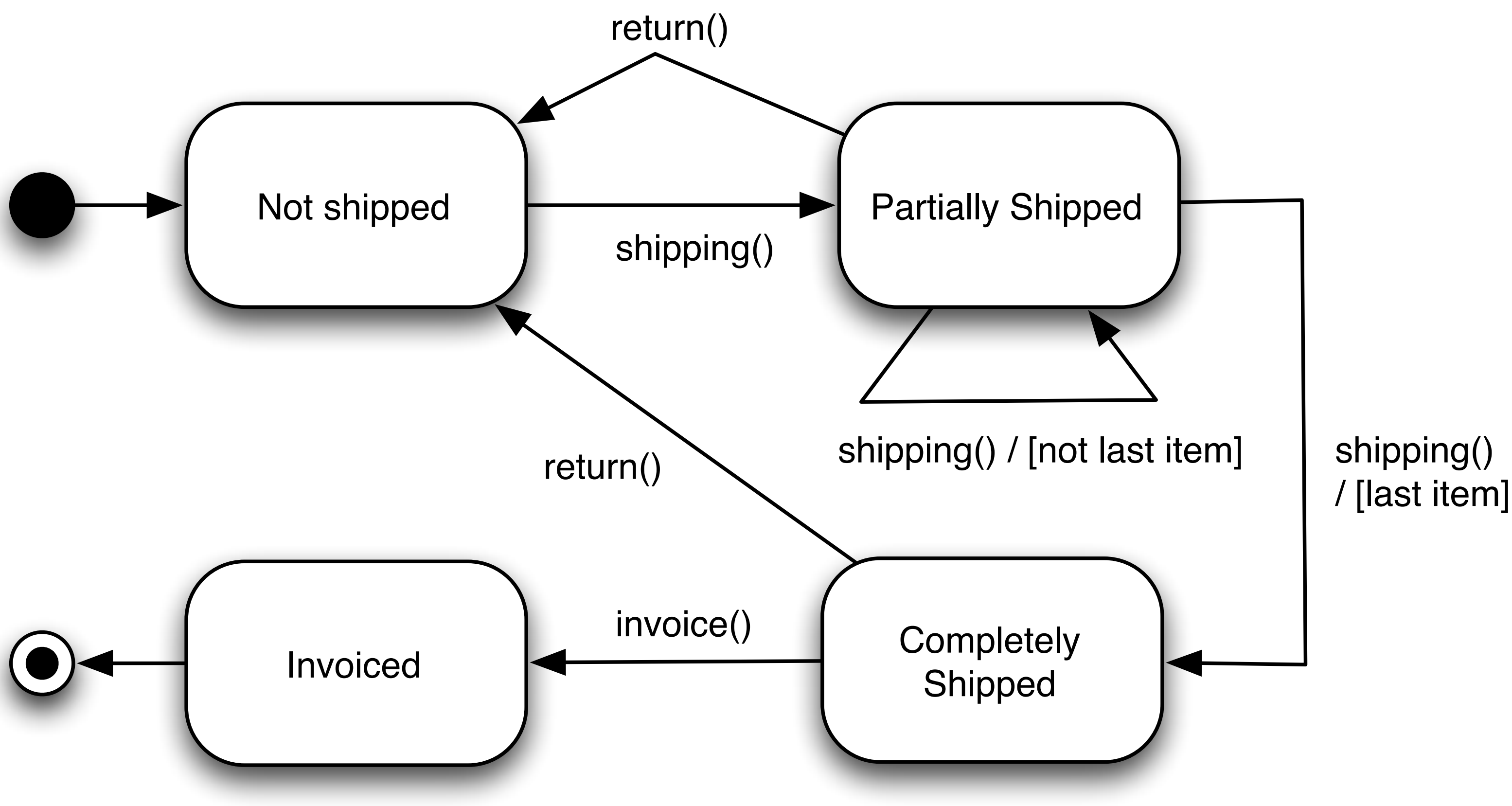
A subset of behavior diagrams, emphasize the **flow of control and data** among the elements of the system.

- **Sequence diagram**: Shows how objects communicate with each other in terms of a temporal sequence of messages
- **Communication or collaboration diagram**: Shows the interactions between objects or classes in terms of links and messages that flow through the links
- **Interaction overview diagram**: Provides an overview in which the nodes represent interaction diagrams
- **Timing diagrams**: A specific type of interaction diagram where the focus is on timing constraints

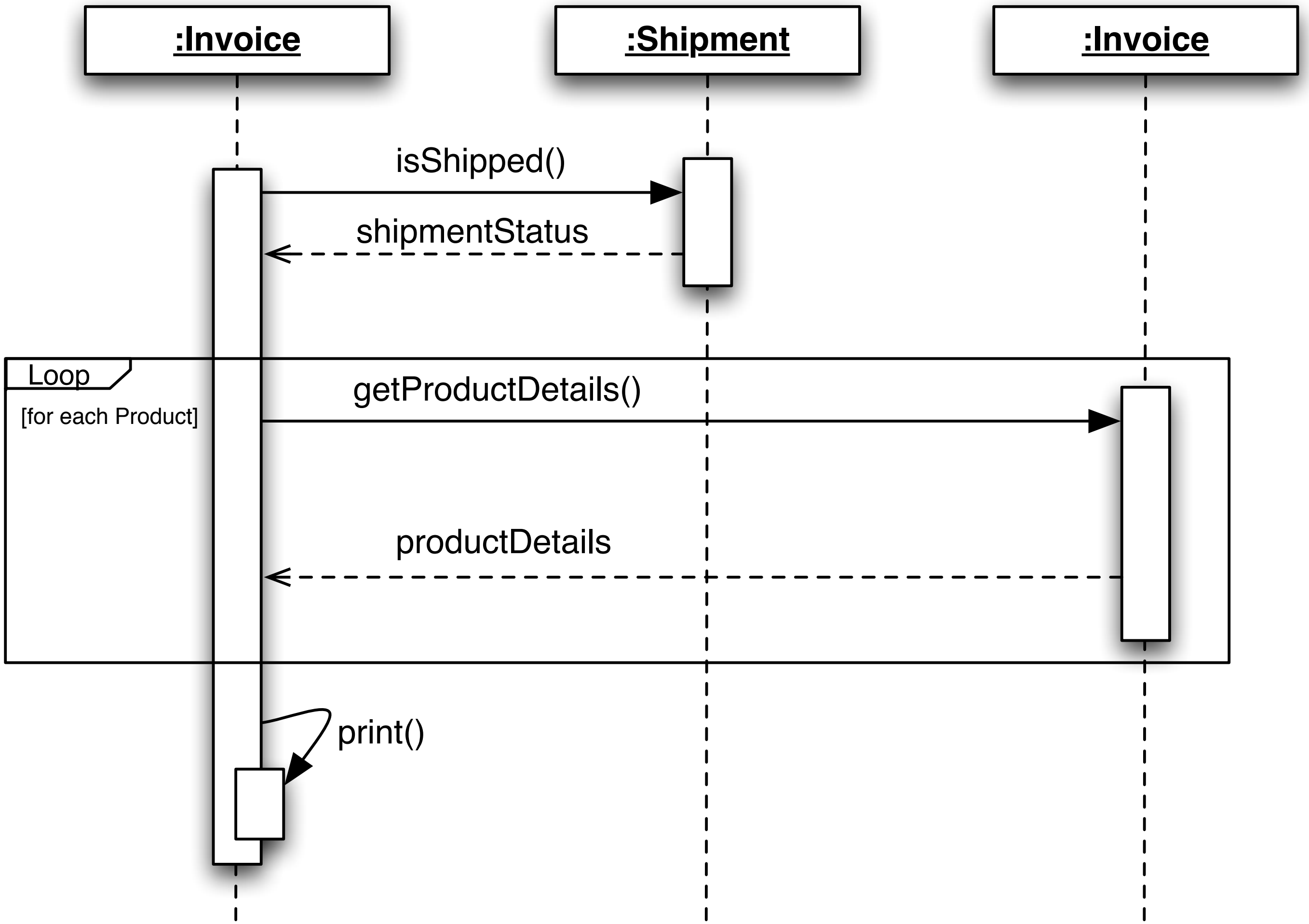
Activity diagram example



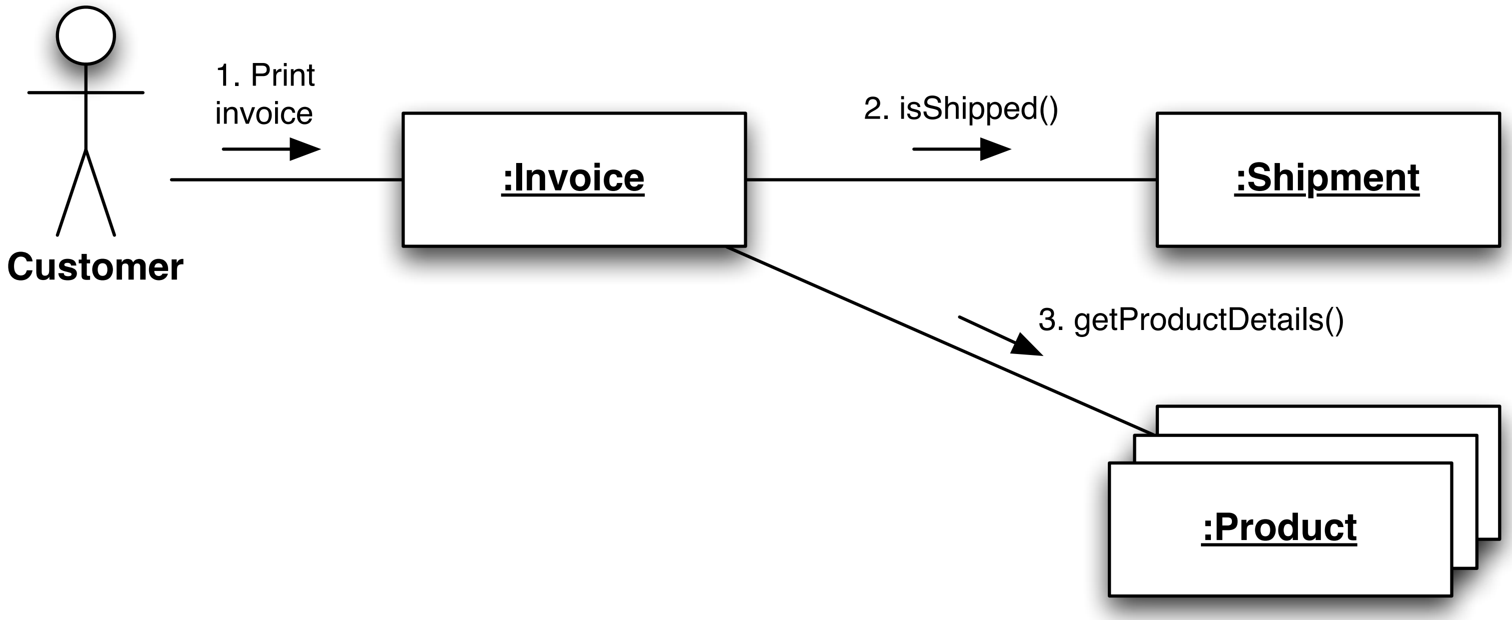
State Diagram example



Sequence diagram example



Collaboration diagram example



UML EXTENSIBILITY

Extensibility: Stereotype definition

- Stereotypes are formal extensions of existing model elements within the UML metamodel, that is, ***metamodel extensions***.
- The modeling element is directly influenced by the **semantics** defined by the extension.
- Rather than introducing a new model element to the metamodel, **stereotypes add semantics to an existing model element**.

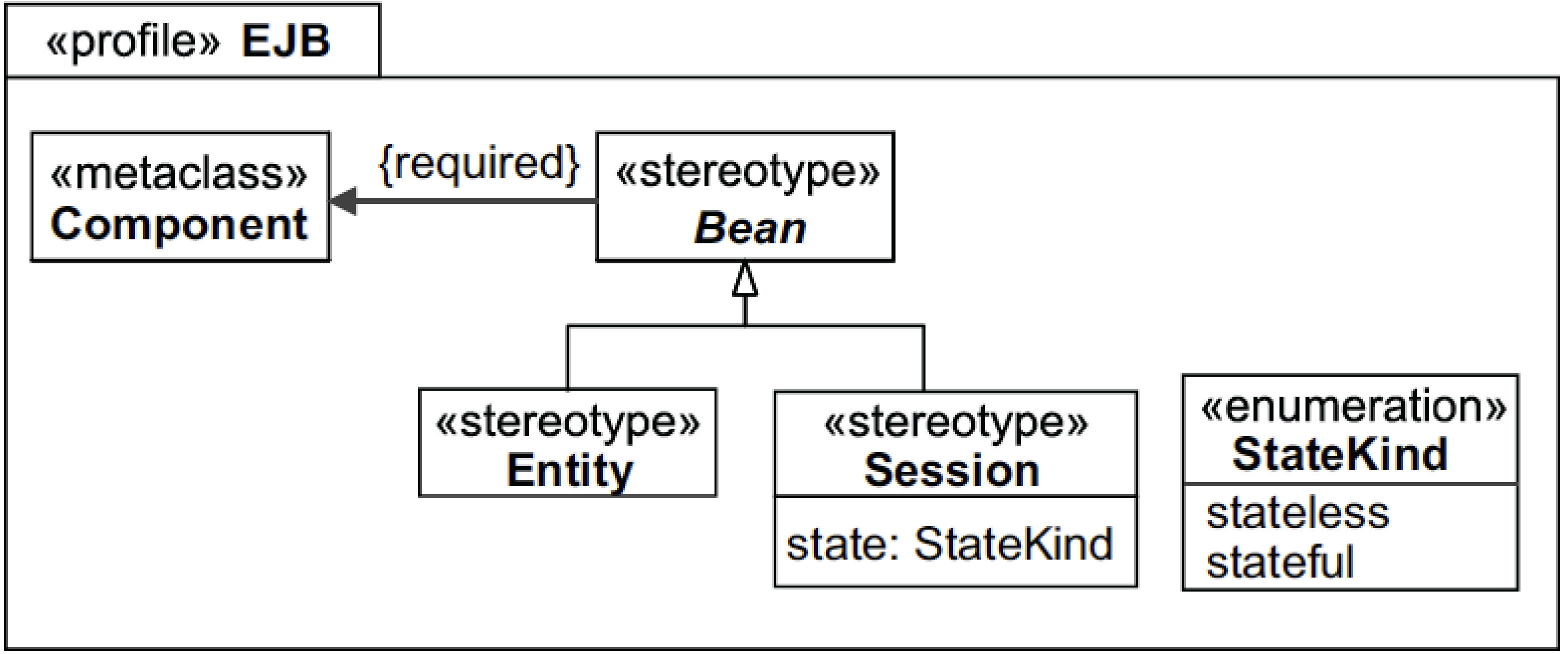
Multiple stereotyping

- **Several stereotypes** can be used to classify one single modeling element.
- Even **the visual representation** of an element can be influenced by allocating stereotypes.
- Moreover, stereotypes **can be added to** attributes, operations and relationships.
- Further, stereotypes **can have attributes** to store additional information.

Stereotypes Notation

- A stereotype is placed before or above the element name and enclosed in guillemets (**<<,>>**).
- **Important**: not every ocurrence of this notation means that you are looking at a stereotype. **Keywords** predefined in UML are also enclosed in guillemets.

UML Extensibility: profile example



DOMAIN-SPECIFIC LANGUAGES

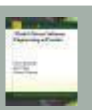
Principles for Domain Specific Languages

- The language must **provide good abstractions** to the developer, must be intuitive, and **make life easier**, not harder
- The language must **not depend on one-man expertise** for its adoption and usage. Its definition must be **shared and agreed** upon
- The language must **evolve** and must **be kept updated** based on the user and context needs, otherwise it is doomed to die.
- The language must come together with supporting **tools and methods**
- The language should be **open for extensions and closed for modifications** (open-close principle)

Classification of DSLs (1): FOCUS. Horizontal vs. Vertical

- **Vertical DSLs** aim at a specific industry or field.
- Examples: configuration languages for home automation systems, modeling languages for biological experiments, analysis languages for financial applications.

- **Horizontal DSLs** have a broader applicability and their technical and cover concepts that apply across a large set of fields. They may refer to a specific technology but not to a specific industry.
- Examples: SQL, Flex, WebML.



Classification of DSLs (2): STYLE. Declarative vs. Imperative

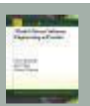
- **Declarative DSLs:** specification paradigm that **expresses the logic of a computation without describing its control flow**.
- The language defines what the program should accomplish, rather than describing how to accomplish it.
- Examples Web service choreography, SQL.

- **Imperative DSLs:** define an **executable algorithm** that states the steps and control flow that needs to be followed.
- Examples: service orchestrations (start-to-end flows), BPMN process diagrams, programming languages like Java or C/C++.

Classification of DSLs (3): NOTATION. Graphical vs. Textual

- **Graphical DSLs:** the outcomes of the development are visual models and the development primitives are graphical items such as blocks, arrows and edges, containers, symbols, and so on.

- **Textual DSLs** comprise several categories, including XML-based notations, structured text notations, textual configuration files, and so on.



Classification of DSLs (4): INTERNALITY. Internal vs. External

- **External DSLs** have their own custom syntax, with a full parser and self-standing, **independent models/ programs.**

- **Internal DSLs** consist in using a **host language** and give it the feel of a particular domain or objective, either by embedding pieces of the DSL in the host language or by providing abstractions, structures, or functions upon it.

Classification of DSLs (5): EXECUTABILITY. Model Interpretation vs. Code Generation

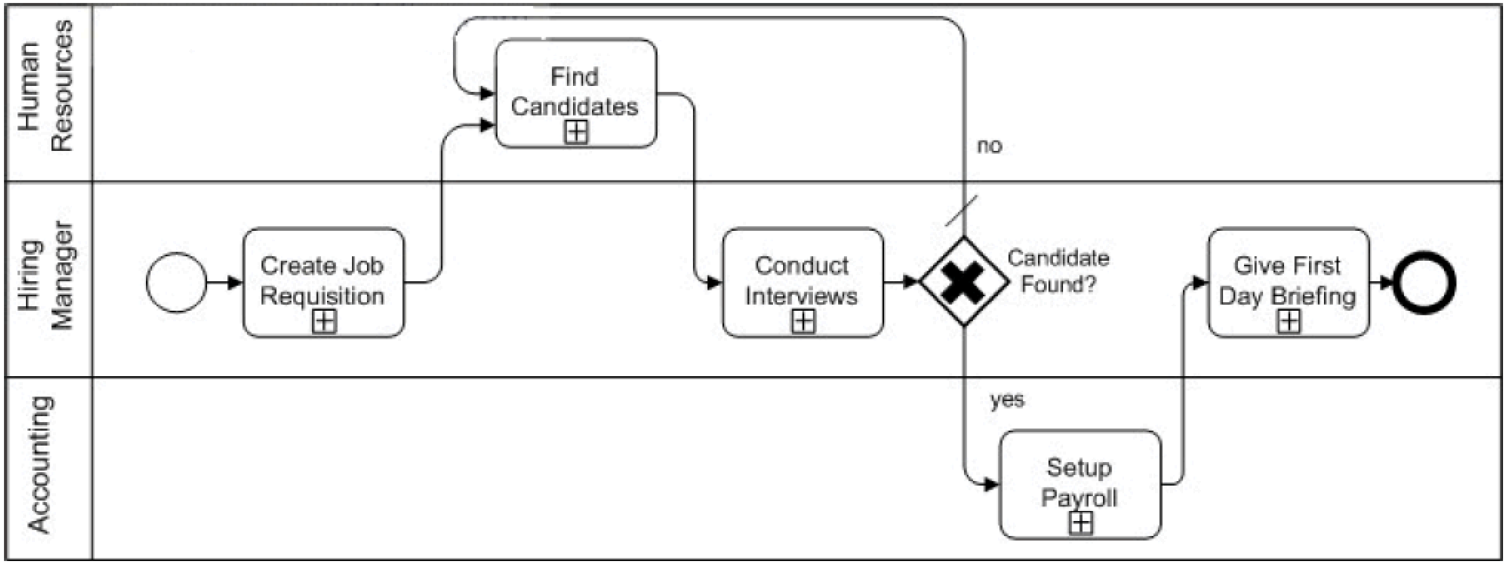
- **Model interpretation:** reading and executing the DSL script at runtime one statement at a time, exactly as programming languages interpreters do.

- **Code-generation:** applying a complete model-to-text (M2T) transformation at deployment time, thus producing an executable application, as compilers do for programming languages.

- See Chapter 2 for model executability details.

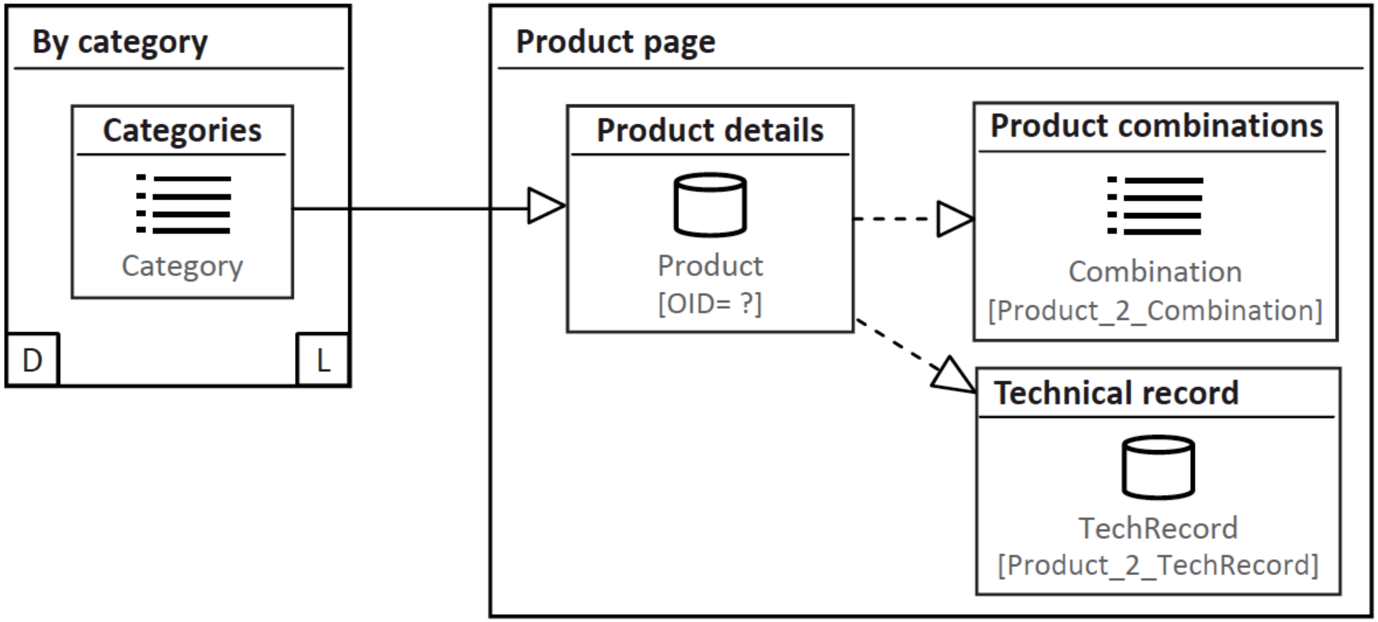
DSL example (1): BPMN process model

- Graphical, external, imperative, horizontal DSL for specifying business processes



DSL example (2): WebML hypertext model

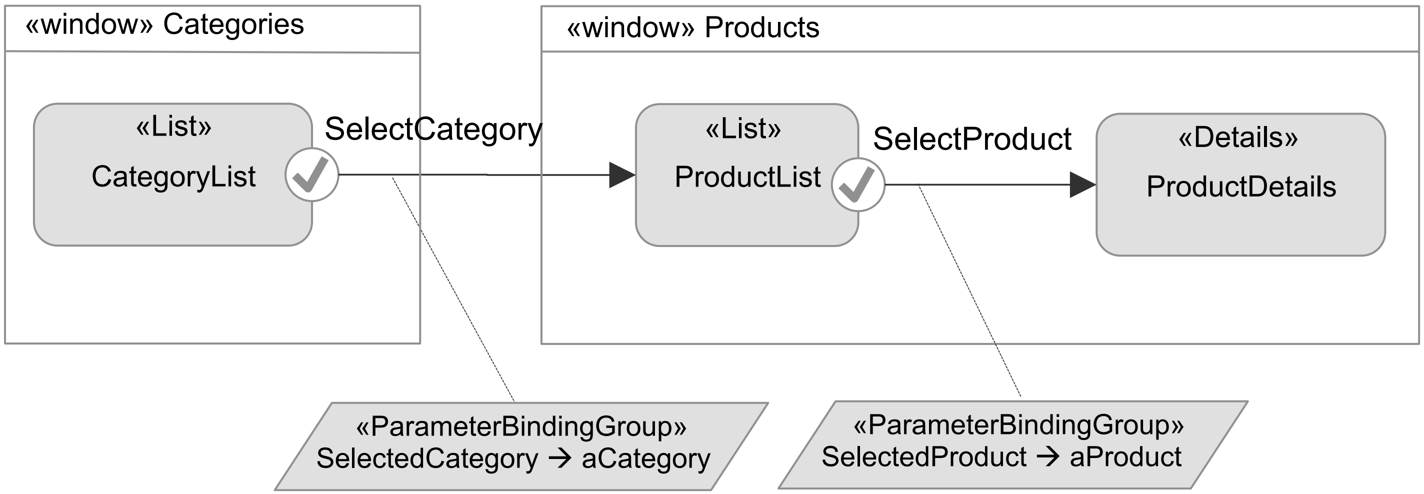
- Declarative, graphical, horizontal DSL for modeling Web navigation Uis. Supporting tool WebRatio applies a full code generation approach for executing the models.



- See also: www.webml.org

DSL Example (3): IFML by OMG

- Interaction Flow Modeling Language
- Defines content and navigation of user interfaces



- See also: http://www.ifml.org
  and IFML book: http://amzn.to/1mcgYuo

DSL example (4): VHDL specs

- Textual, external, declarative, vertical DSL for specifying the behaviour of electronic components.
- Example: definition of a Multiplexer in VHDL

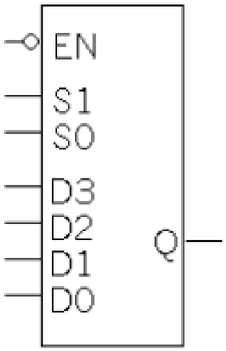
```
entity mux4_to_1 is
      port (I0,I1,I2,I3: in std_logic_vector(7 downto 0);
      SEL: in std_logic_vector (1 downto 0);
       OUT1: out std_logic_vector(7 downto 0));
end mux4_to_1;
```

- Multiplexer (MUX): electronic component that selects one of several analog or digital input signals and forwards the selected input into a single output line.
- See more at: http://en.wikipedia.org/wiki/Multiplexer

DSL Example (4): VHDL Cont.d

Alternative representation of the multiplexer, according to different notations (which could be seen as DSLs themselves):

- Electronic block diagram, truth table, output boolean expression



EN'	S1	S0	Q
0	0	0	D0
0	0	1	D1
0	1	0	D2
0	1	1	D3
1	x	x	1

$$Q = S1'\ S0'\ D0 + S1'\ S0\ D1 + S1\ S0'\ D2 + S1\ S0\ D3$$

OCL – OBJECT CONSTRAINT LANGUAGE

OCL Topics

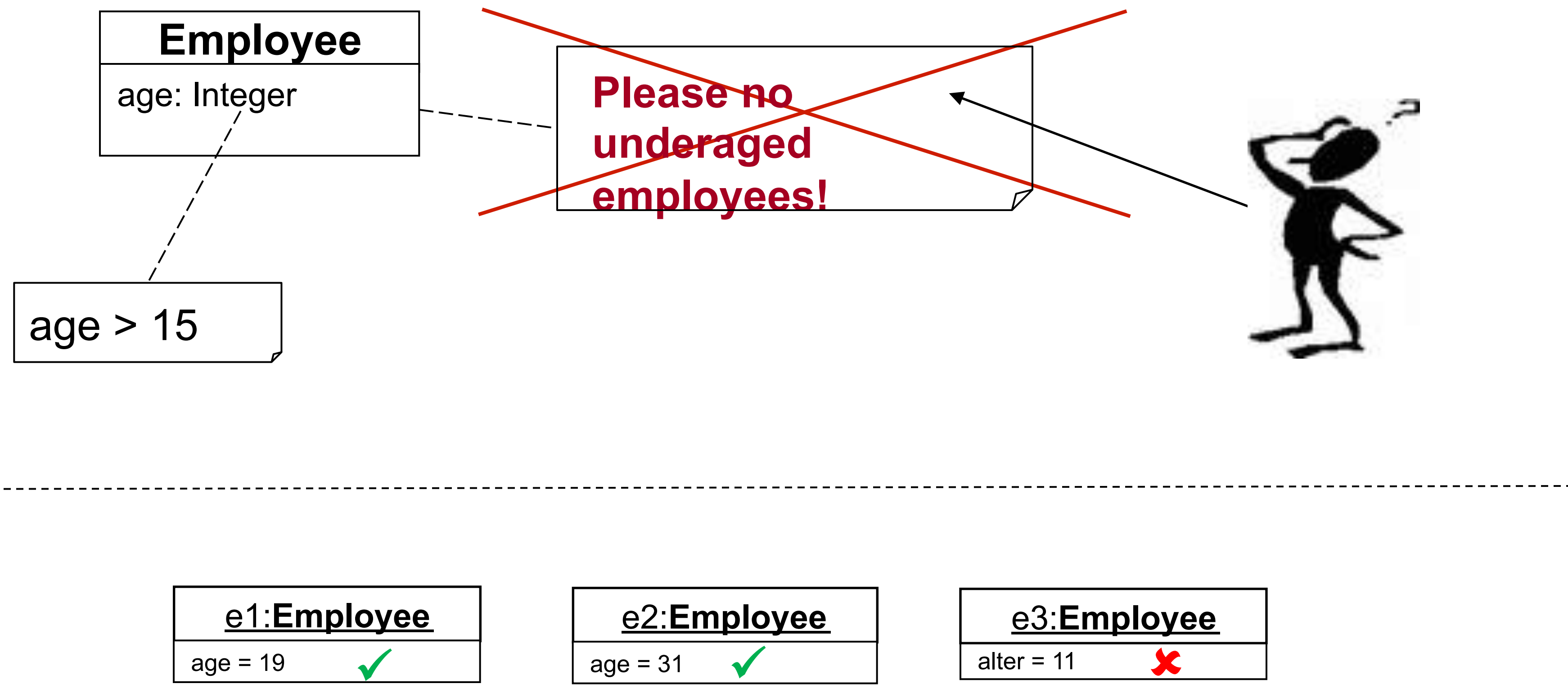
- Introduction
- OCL Core Language
- OCL Standard Library
- Tool Support
- Examples

Motivation

- Graphical modeling languages are generally not able to describe all facets of a problem description
  - *MOF, UML, ER, …*
- Special **constraints** are often (if at all) added to the diagrams in **natural language**
  - Often **ambiguous**
  - Cannot be validated **automatically**
  - No **automatic** code generation
- Constraint definition also crucial in the definition of new modeling languages (DSLs).

Motivation

- Example 1

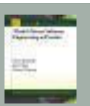


Additional question: How do I get all Employees younger than 30 years old?

Motivation

- **Formal specification languages** are the solution
  - Mostly based on **set theory** or **predicate logic**
  - Requires good mathematical understanding
  - Mostly used in the academic area, but hardly used in the industry
  - Hard to learn and hard to apply
  - Problems when to be used in big systems

- ***Object Constraint Language (OCL)*: Combination of modeling language and formal specification language**
  - Formal, precise, unique
  - Intuitive syntax is key to **large group of users**
  - No programming language (no algorithms, no technological APIs, …)
  - Tool support: *parser, constraint checker, code generation,…*

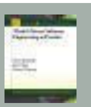


OCL usage

- Constraints in UML-models
  - Invariants for classes, interfaces, stereotypes, …
  - Pre- and postconditions for operations
  - Guards for messages and state transition
  - Specification of messages and signals
  - Calculation of derived attributes and association ends

- Constraints in meta models
  - Invariants for Meta model classes
  - Rules for the definition of well-formedness of meta model

- Query language for models
  - In analogy to SQL for DBMS, XPath and XQuery for XML
  - Used in transformation languages



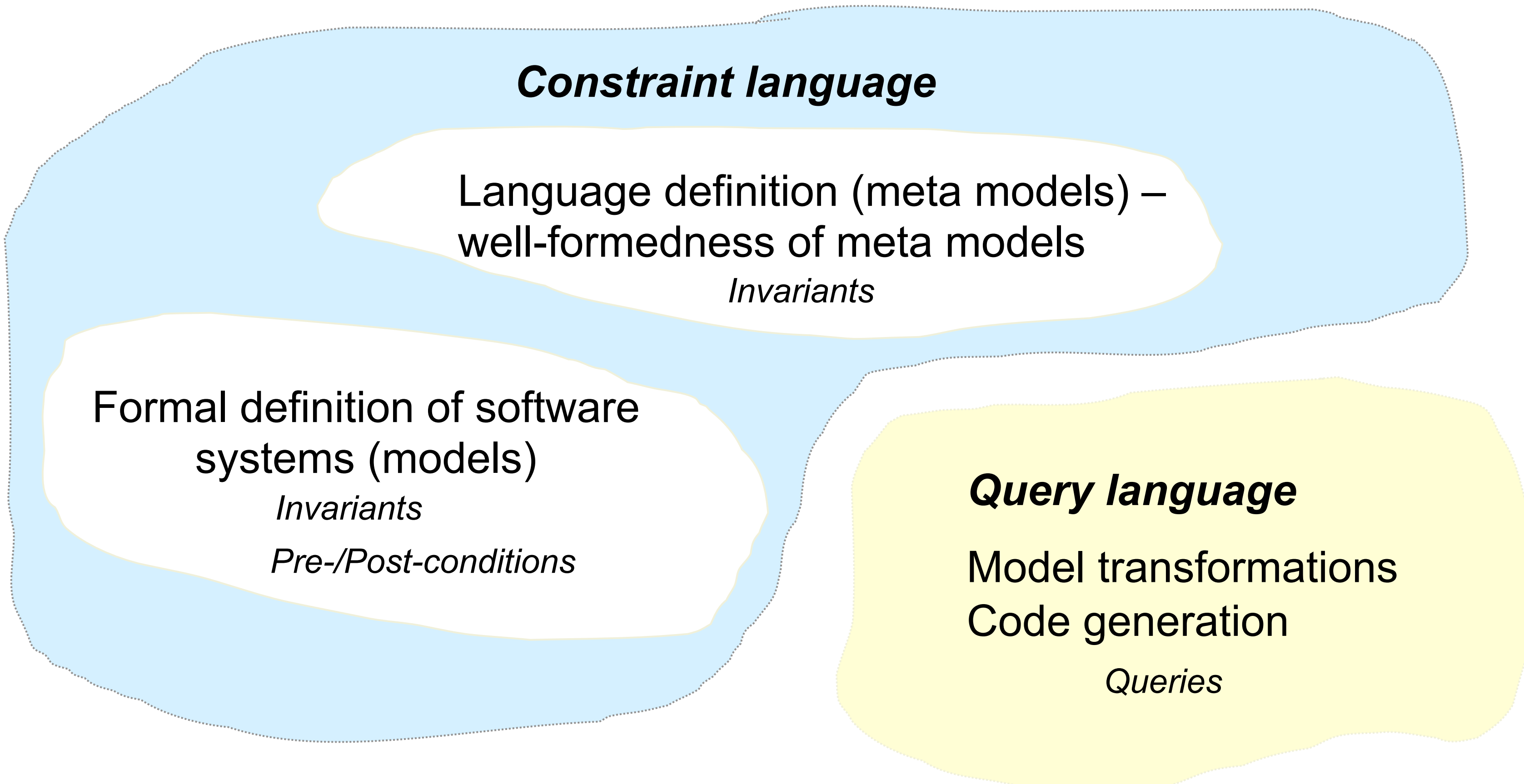
OCL usage

- OCL field of application
  - Invariants **context** *C* **inv:** *I*
  - Pre-/Postconditions **context** *C::op() : T* **pre:** *P* **post:** *Q*
  - Query operations **context** *C::op() : T* **body:** *e*
  - Initial values **context** *C::p : T* **init:** *e*
  - Derived attributes **context** *C::p : T* **derive:** *e*
  - Attribute/operation definition **context** *C* **def:** *p : T = e*

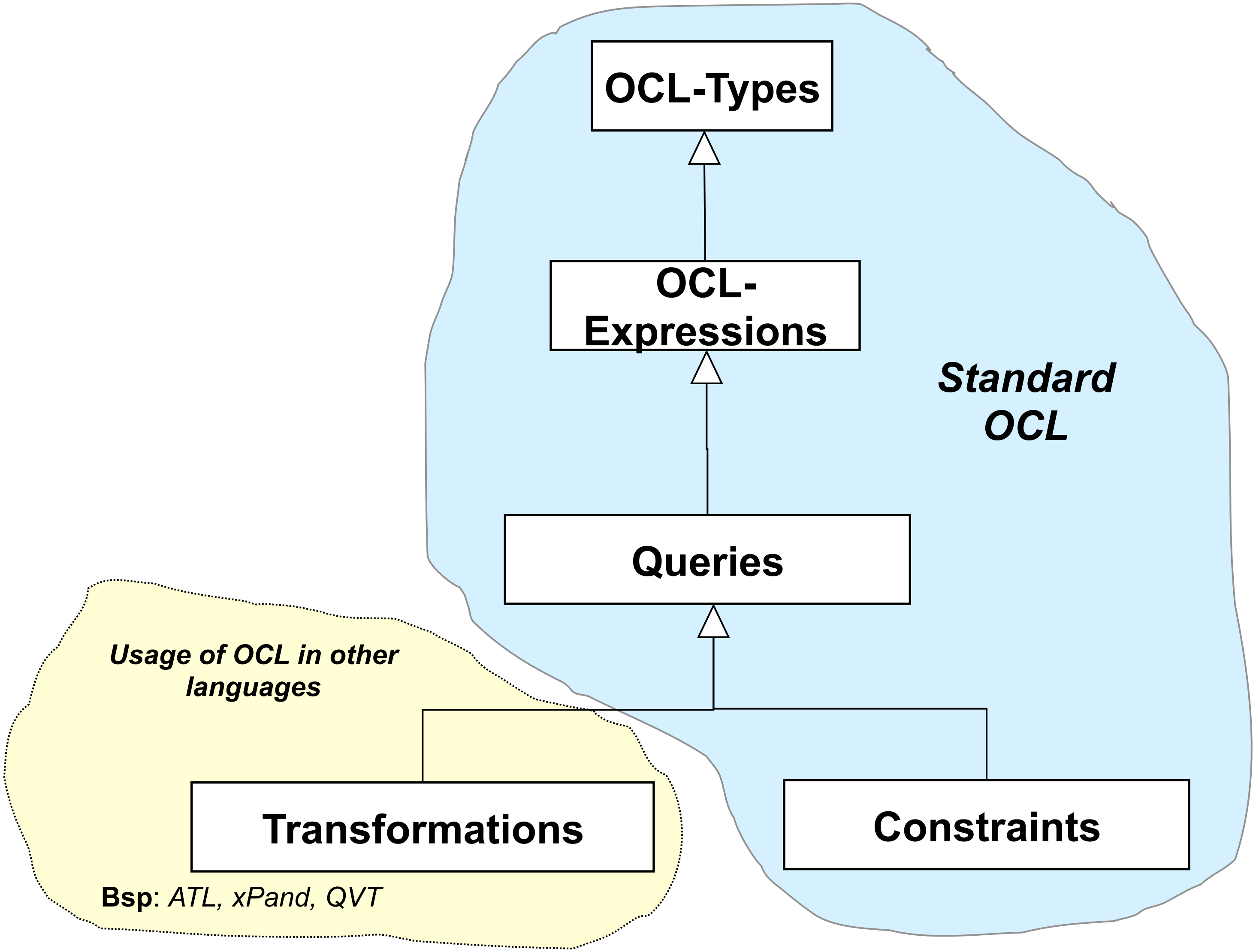
- Caution: Side effects are not allowed!
  - Operation `C::getAtt : String body: att` allowed in OCL
  - Operation `C::setAtt(arg) : T body: att = arg` **not** allowed in OCL

OCL usage

- **Field of application** of OCL in model driven engineering



OCL usage

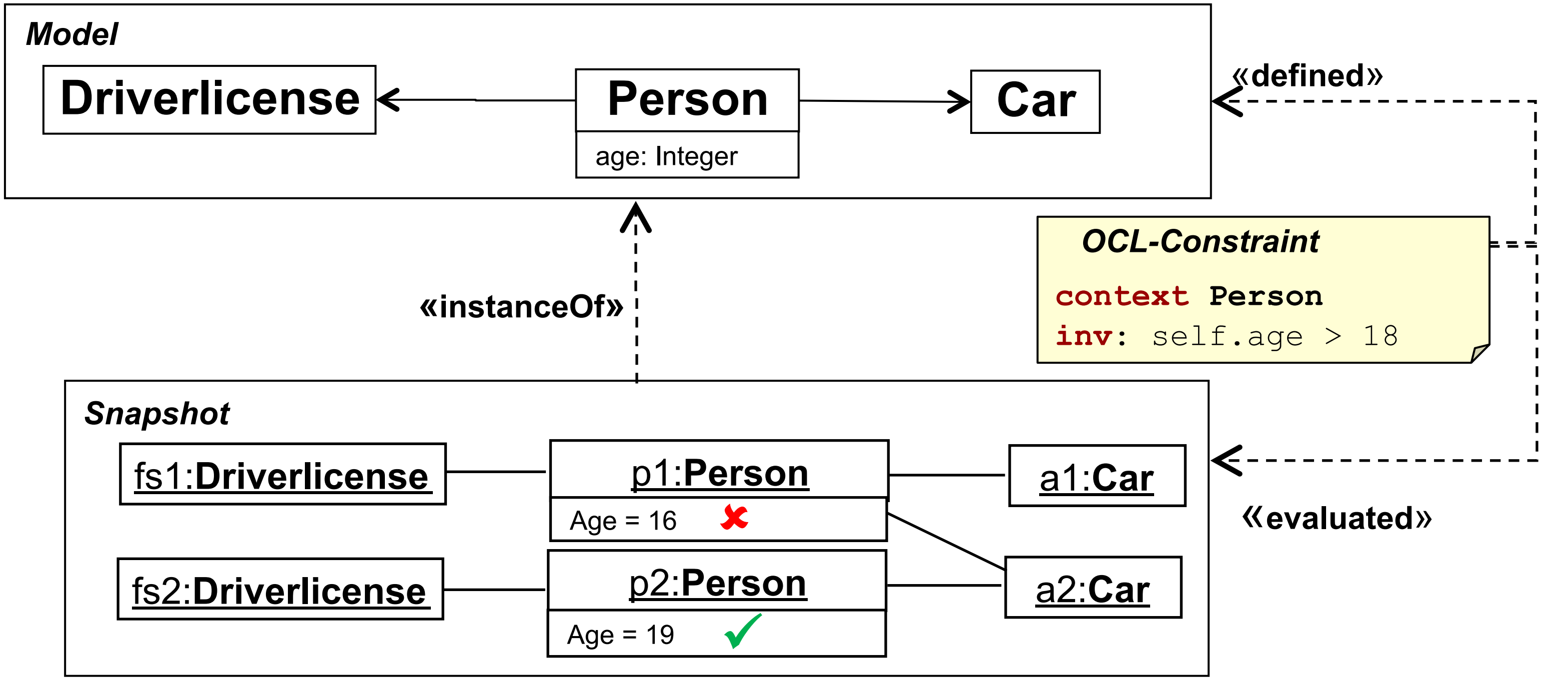


OCL-Types
OCL-Expressions
Standard OCL
Queries
Usage of OCL in other languages
Transformations
Constraints
Bsp: ATL, xPand, QVT

OCL usage

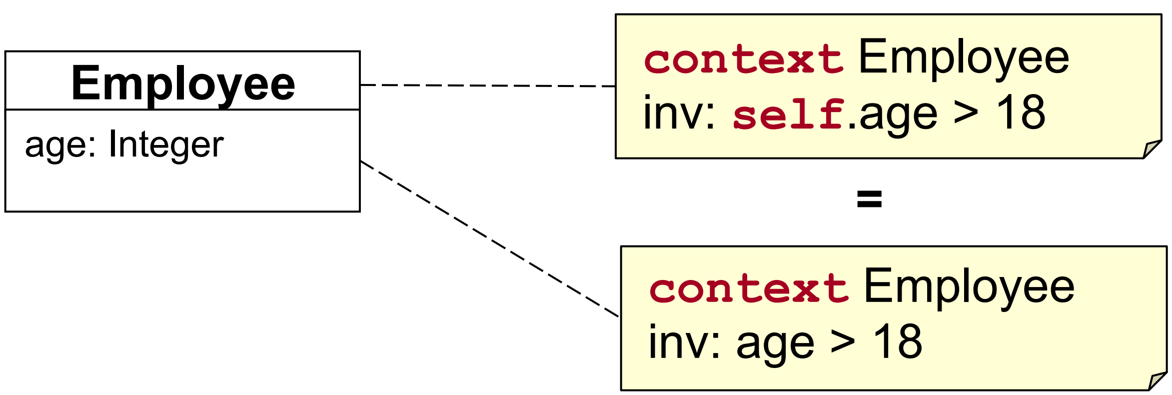
How does OCL work?

- **Constraints** are defined on the modeling level
  - Basis: Classes and their properties
- Information of the **object graph** are **queried**
  - Represents system status, also called ***snapshot***
- **Anaology** to XML query languages
  - XPath/XQuery query XML-documents
  - Scripts are based on XML-schema information
- Examples



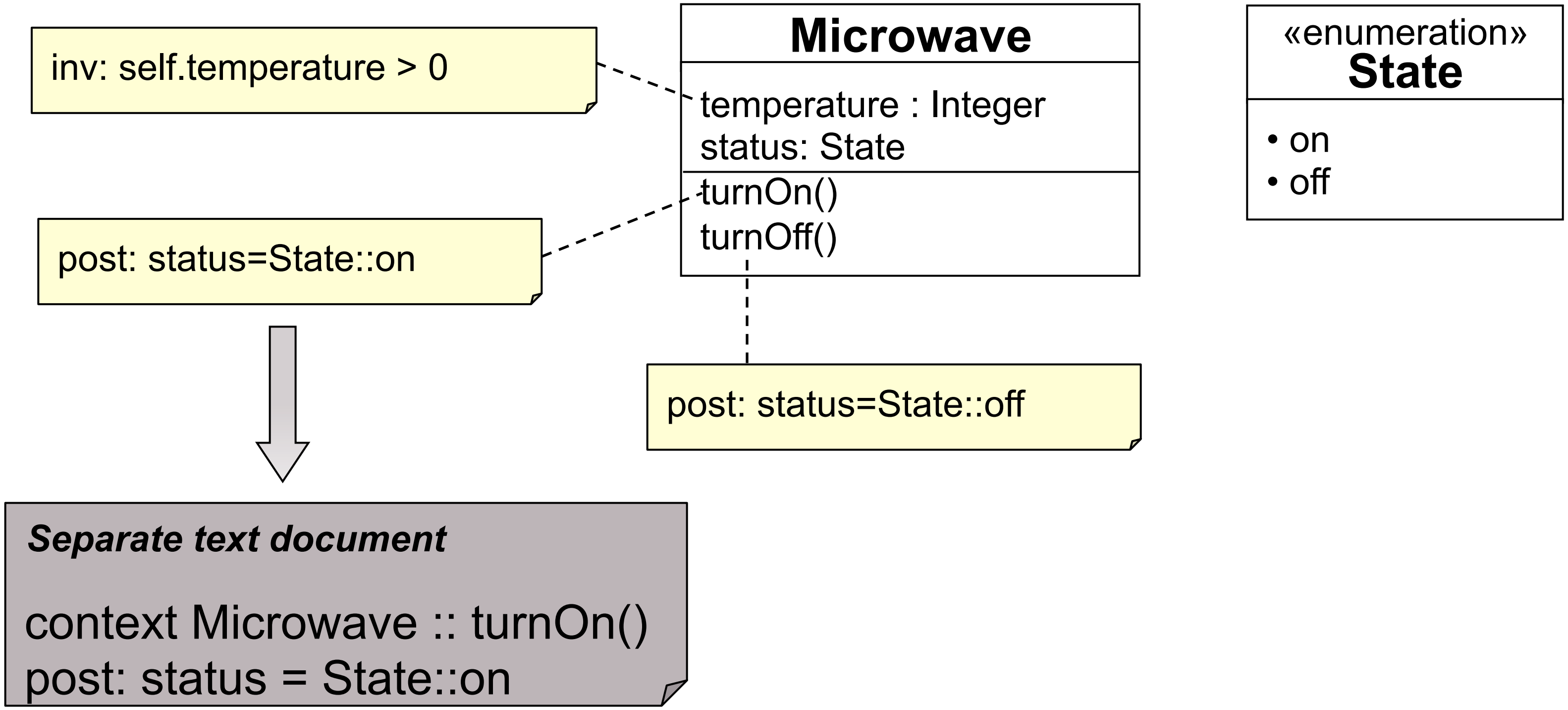
Design of OCL

- A context has to be assigned to each OCL-statement
  - **Starting address –** which model element is the OCL-statement defined for
  - Specifies which model elements can be reached using path expressions
- The context is specified by the keyword `context` followed by the name of the model element (mostly class names)
- The keyword `self` specifies the current instance, which will be evaluated by the invariant (context instance).
  - `self` can be omitted if the context instance is unique
- Example:



Design of OCL

- OCL can be specified in **two** different ways
  - As a comment **directly** in the class diagram (context described by connection)
  - Separate document file



Types

- **OCL** is a typed language
  - Each **object, attribute,** and **result** of an operation or navigation is assigned to a **range of values** (type)

- **Predefined types**
  - **Basic types**
    - Simple types: *Integer, Real, Boolean, String*
    - OCL-specific types: *AnyType, TupleType, InvalidType, …*
  - **Set-valued, parameterized Types**
    - Abstract supertyp: Collection(T)
    - Set(T) – no duplicates
    - Bag(T) – duplicates allowed
    - Sequence(T) – Bag with ordered elements, association ends {*ordered*}
    - OrderedSet(T) – Set with ordered elements, association ends {*ordered, unique*}

- **Userdefined Types**
  - Instances of *Class* in MOF and indirect instances of *Classifier* in UML are types
  - *EnumerationType* – user defined set of values for defining constants

Types
Examples

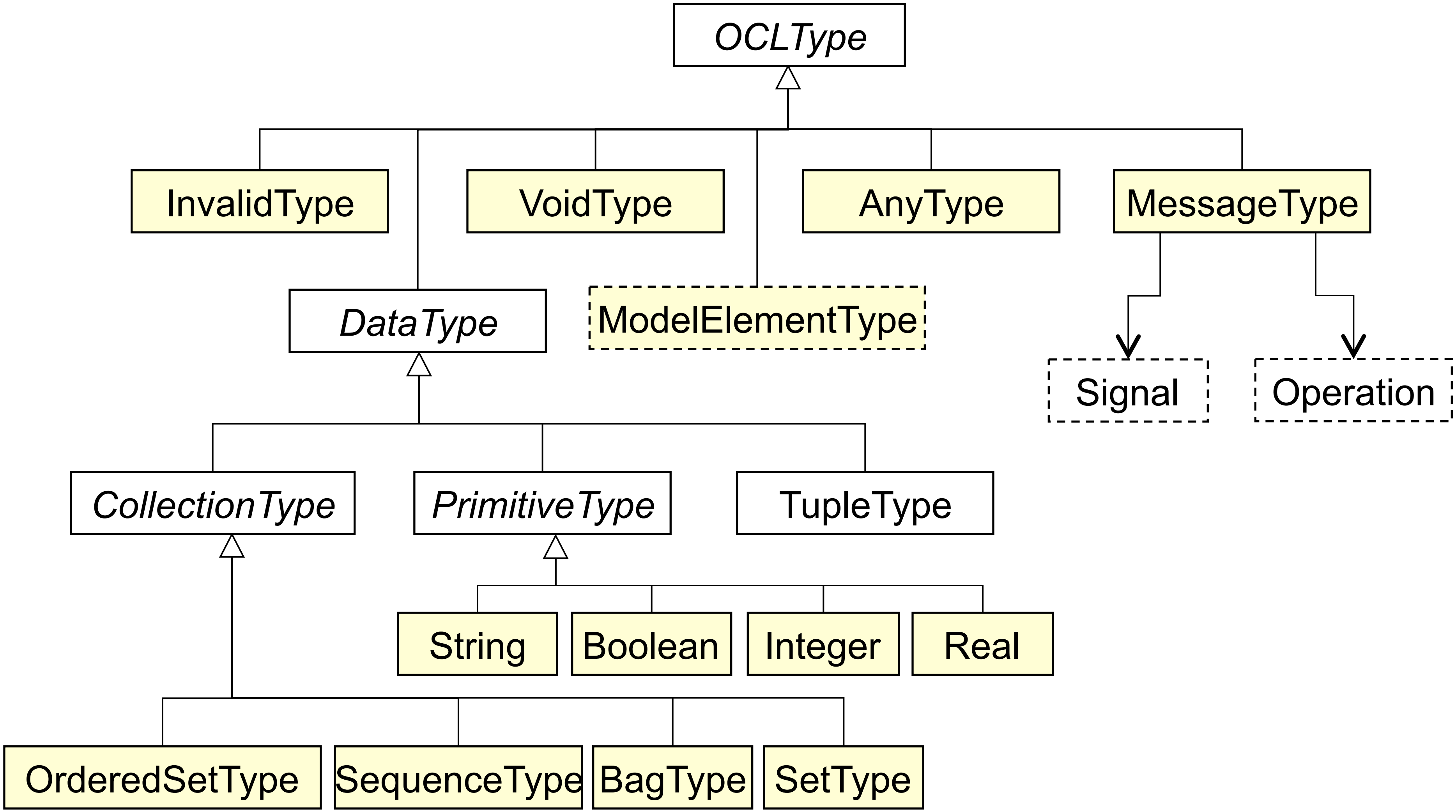
- **Basic types**
  - true, false : *Boolean*
  - -17, 0, 1, 2 : *Integer*
  - -17.89, 0.01, 3.14 : *Real*
  - “Hello World” : *String*

- **Set-valued, parameterized types**
  - Set{ Set{1}, Set{2, 3} } : *Set(Set(Integer))*
  - Bag{ 1, 2.0, 2, 3.0, 3.0, 3 } : *Bag(Real)*
  - Tuple{ x = 5, y = false } : *Tuple{x: Integer, y : Boolean}*

- **Userdefined types**
  - Passenger : *Class*, Flight : *Class*, Provider : *Interface*
  - Status::started - enum Status {started, landed}

Types

OCL meta model (extract)



Expressions

- Each OCL expression is an indirect instance of *OCLExpression*
  - Calculated in certain environment – cf. context
  - Each OCL expression has a **typed return value**
  - **OCL Constraint is an OCL expression with return value Boolean**

- **Simple OCL expressions**
  - *LiteralExp, IfExp, LetExp, VariableExp, LoopExp*

- **OCL expressions for querying model information**
  - *FeatureCallExp* – abstract superclass
  - *AttributeCallExp* – querying attributes
  - *AssociationEndCallExp* – querying association ends
    - Using role names; if no role names are specified, lowercase class names have to be used (if unique)
  - *AssociationClassCallExp* – querying association class (only in UML)
  - *OperationCallExp* – Call of query operations
    - Calculate a value, but do **not** change the system state!

Expressions

- Examples for *LiteralExp, IfExp, VariableExp, AttributeCallExp*

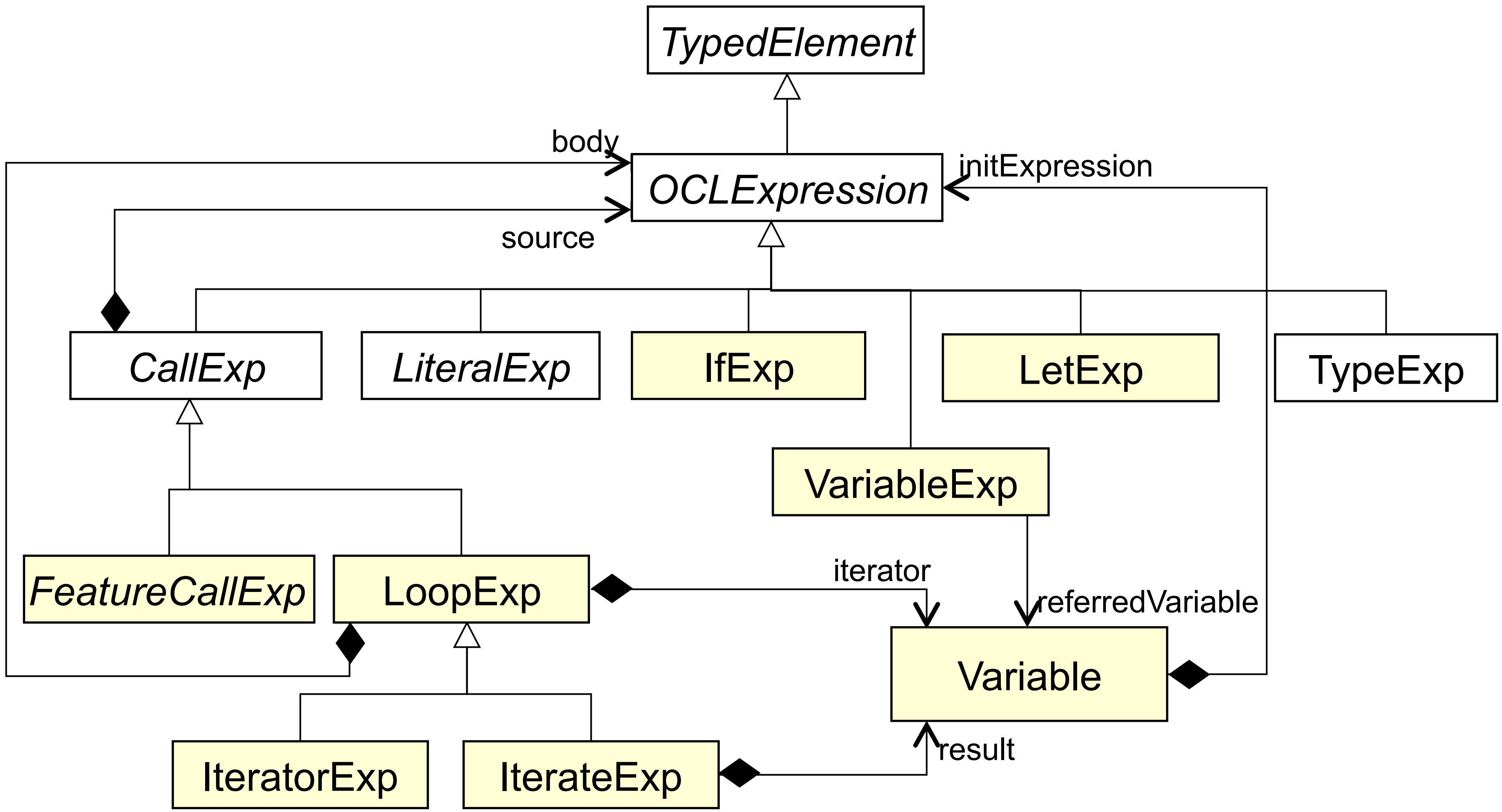
LetExp VariableExp AttributeCallExp IntegerLiteralExp IfExp

```
let annualIncome : Real = self.monthlyIncome * 14 in
   if self.isUnemployed then
        annualIncome < 8000
   else
        annualIncome >= 8000
   endif
```

- **Abstract syntax** of OCL is described as **meta model**
- **Mapping from abstract syntax to concrete syntax**
  - *IfExp -> **if** Expression **then** Expression **else** Expression **endif***

Expressions

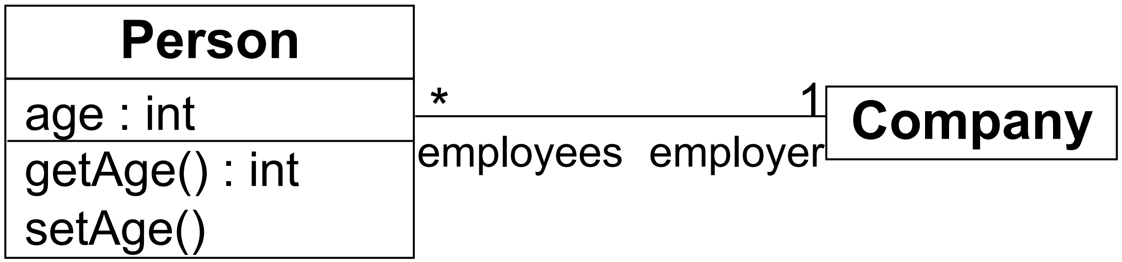
OCL meta model (extract)



LiteralExp: *CollectionLiteralExp*, *PrimitiveLiteralExp*, *TupleLiteralExp*, *EnumLiteralExp*

Query of model information

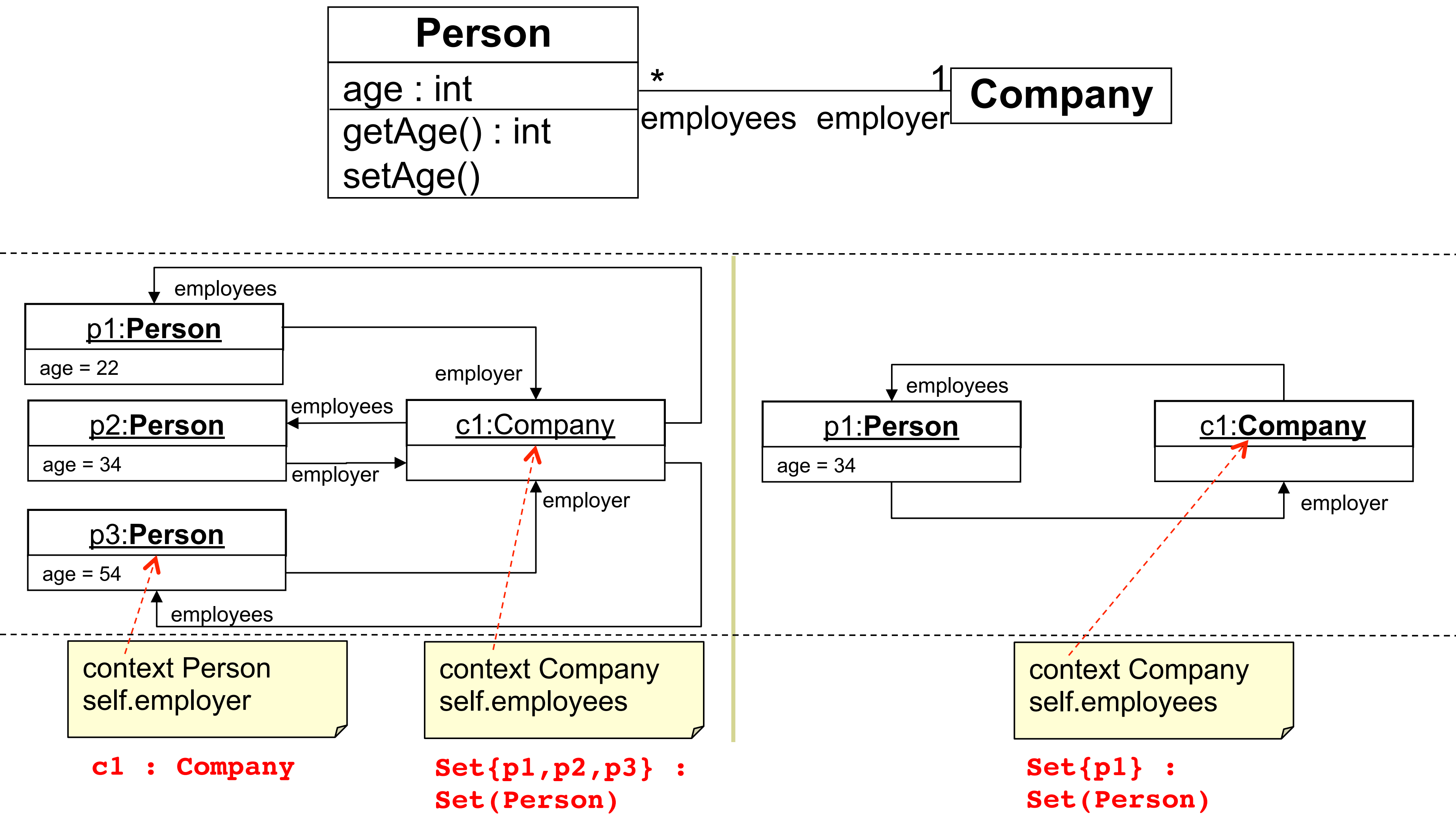
- Context instance
  - `context` **`Person`**



- AttributeCallExp
  - `self.age : int`
- OperationCallExp
  - Operations must not have **side effects**
  - Allowed: `self.getAge() : int`
  - **Not allowed**: `self.setAge()`
- AssociationEndCallExp
  - Navigate to the opposite association end using role names
    `self.employer` – Return value is of type ***Company***
  - Navigation often results into a set of objects – Example
    `context` **`Company`**
    `self.employees` – Return value is of type **Set (Person)**

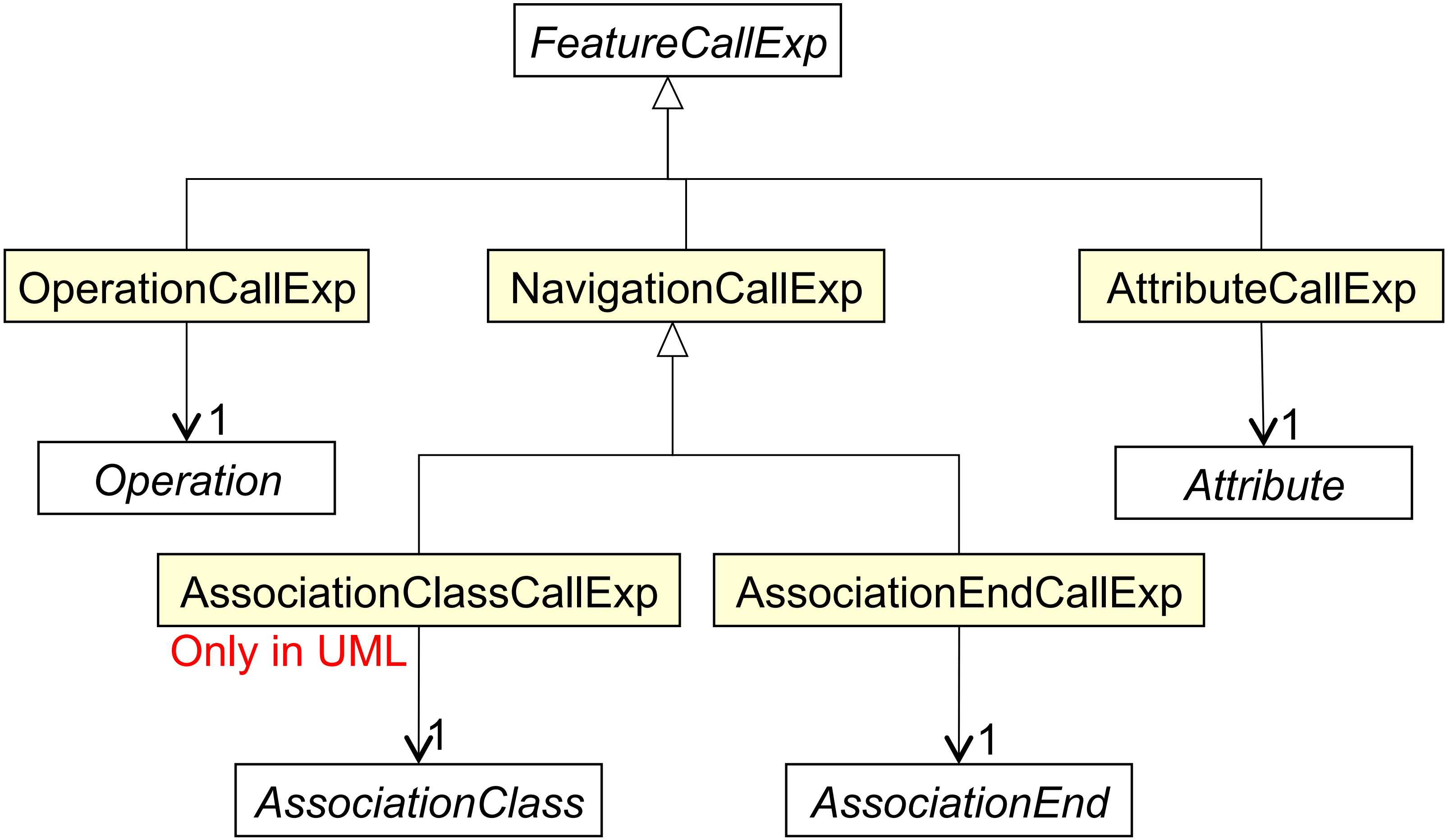
Query of model information

Example



Query of model information

OCL meta model (extract)



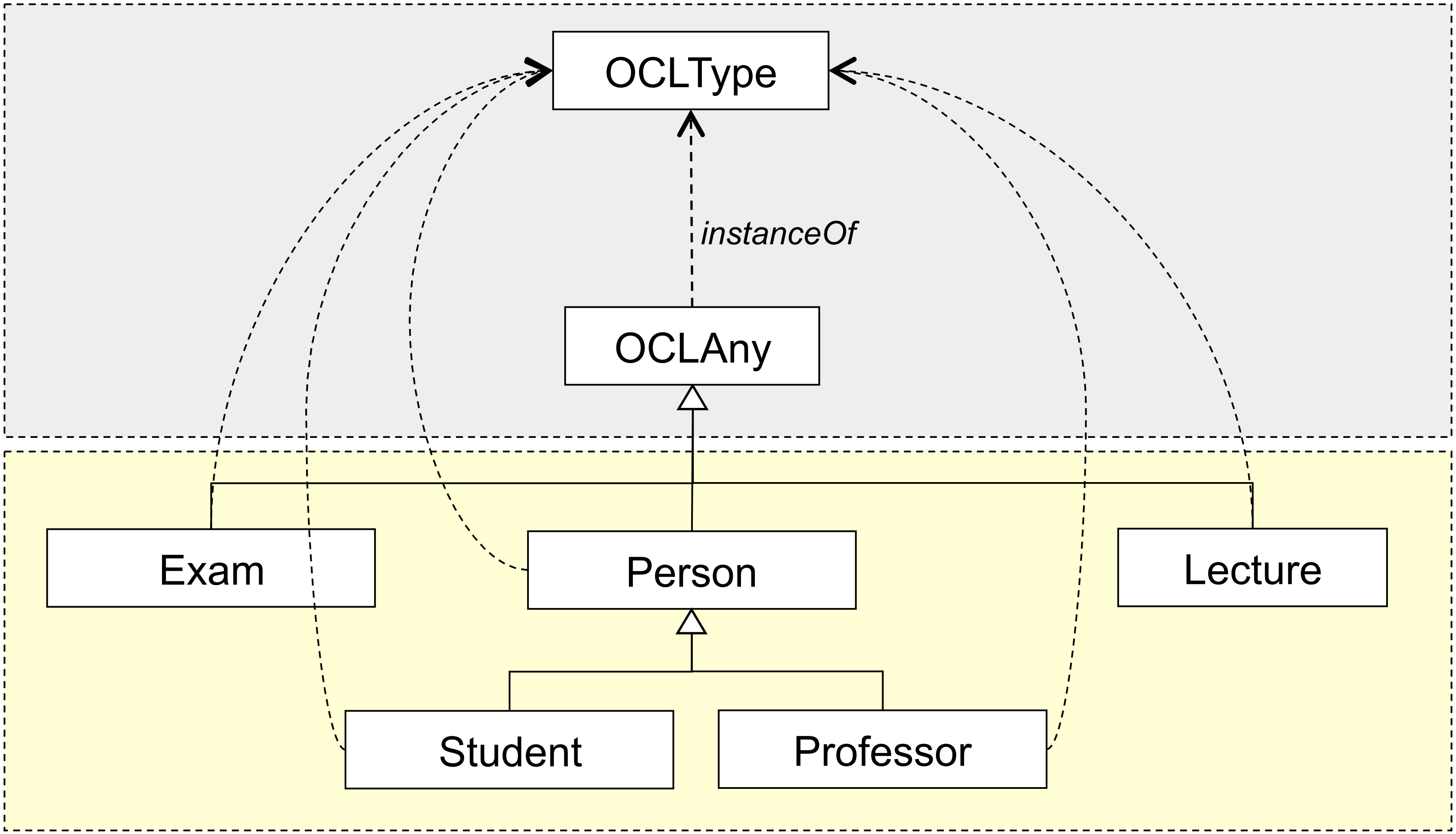
OCL Library: Operations for OclAny

- *OclAny* - **Supertype** of all other types in OCL
  - **Operations** are **inherited** by all other types.
- **Operations** of *OclAny* (extract)
  - Receiving object is denoted by *obj*

Operation	Explanation of result
=(obj2:OclAny):Boolean	True, if *obj2* and *obj* reference the same object
oclIsTypeOf(type:OclType):Boolean	True, if *type* is the type of *obj*
oclIsKindOf(type:OclType): Boolean	True, if *type* is a direct or indirect supertype or the type of *obj*
oclAsType(type:Ocltype): Type	The result is *obj* of type *type*, or *undefined*, if the current type of *obj* is not *type* or a direct or indirect subtype of it (casting)

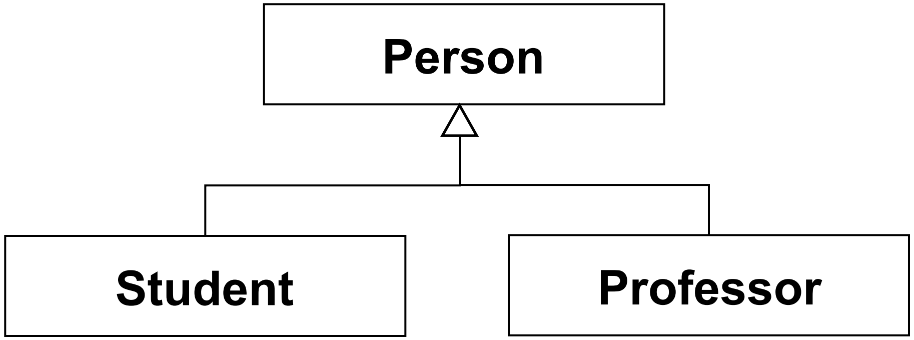
Operations for OclAny

Predefined environment for model types



Operations for OclAny

- ***oclIsKindOf vs. oclIsTypeOf***



context **Person**
self.oclIsKindOf(Person) : *true*
self.oclIsTypeOf(Person) : *true*
self.oclIsKindOf(Student) : *false*
self.oclIsTypeOf(Student) : *false*

context **Student**
self.oclIsKindOf(Person) : *true*
self.oclIsTypeOf(Person) : *false*
self.oclIsKindOf(Student) : *true*
self.oclIsTypeOf(Student) : *true*
self.oclIsKindOf(Professor) : *false*
self.oclIsTypeOf(Professor) : *false*

Operations for simple types

- **Predefined** simple types
  - Integer {Z}
  - Real {R}
  - Boolean {true, false}
  - String {ASCII, Unicode}

- Each simple type has predefined operations

Simple type	**Predefined operations**
Integer	*, +, -, /, abs(), …
Real	*, +, -, /, floor(), …
Boolean	and, or, xor, not, implies
String	concat(), size(), substring(), …

Operations for simple types

- Syntax
  - v.operation(para1, para2, …)
    - Example: “bla”.**concat**(“bla”)
  - Operations without brackets (Infix notation)
    - Example: 1 **+** 2, true **and** false

Signature	**Operation**
Integer **X** *Integer* → *Integer*	{+, -, *}
t1 **X** *t2* → *Boolean*	{<,>,≤,≥}, *t1, t2* typeOf {Integer or Real}
Boolean **X** *Boolean* → *Boolean*	{and, or, xor, implies}

Operations for simple types

Boolean operations - semantic

- OCL is based on a **three-valued (trivalent) logic**
  - Expressions are mapped to the three values {true, false, undefined}
- Semantic of the operations
  - $\mathcal{M}$(I, *exp*)= *I(exp*), if exp not further resolvable
  - $\mathcal{M}$(I, **not** *exp*)= ¬$\mathcal{M}$ (I, *exp*)
  - $\mathcal{M}$(I,(*exp1* **and** *exp2*)) = $\mathcal{M}$(I, *exp1*) ∧ $\mathcal{M}$(I, *exp2*)
  - $\mathcal{M}$(I,(*exp1* **or** *exp2*)) = $\mathcal{M}$(I, *exp1*) ∨ $\mathcal{M}$(I, *exp2*)
  - $\mathcal{M}$(I,(*exp1* **implies** *exp2*)) = $\mathcal{M}$(I, *exp1*) → $\mathcal{M}$(I, *exp2)*

Undefined: Return value if an expression fails

1. Access on the first element of an empty set
2. Error during *Type Casting*
3. …

- Truth table: true(1), false (0),undefined (?)

¬	
0	1
1	0
?	?

∧	0	1	?
0	0	0	0
1	0	1	?
?	0	?	?

∨	0	1	?
0	0	1	?
1	1	1	1
?	?	1	?

→	0	1	?
0	1	1	1
1	0	1	?
?	?	1	?

Operations for simple types

Boolean operations - semantic

- Simple example for an **undefined** OCL expression
  - *1/0*
- **Query** if undefined– OCLAny.oclIsUndefined()
  - *(1 / 0).oclIsUndefined() : true*
- Examples for the evaluation of Boolean operations
  - *(1/0 = 0.0)* ***and*** *false : false*
  - *(1/0 = 0.0)* ***or*** *true : true*
  - *false* ***implies*** *(1.0 = 0.0) : true*
  - *(1/0 = 0.0) implies true : true*

Operations for collections

- Collection is an **abstract supertype** for all set types
  - Specification of the **mutual** operations
  - *Set, Bag, Sequence, OrderedSet* inherit these operations
- **Caution:** Operations with a return value of a set-valued type create a new collection (no side effects)
- Syntax: v -> op(…) – Example: {1, 2, 3} -> size()

- Operations of collections (extract)
  - Receiving object is denoted by *coll*

Operation	**Explanation of result**
size():Integer	Number of elements in *coll*
includes(obj:OclAny):Boolean	True, if *obj* exists in *coll*
isEmpty:Boolean	True, if *coll* contains no elements
sum:T	Sum of all elements in *coll* Elements have to be of type Integer or Real

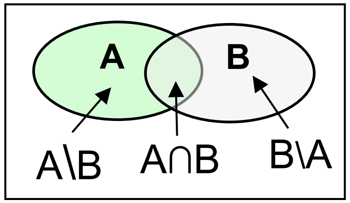
Operations for collections

- **Model operations vs. OCL operations**

OCL-Constraint	***Semantic***
context Container inv: self.content -> first().isEmpty()	Operation *isEmpty()* always has to return true
context Container inv: self.content -> isEmpty()	Container instances must not contain bottles

Operationen for Set/Bag

- *Set* and *Bag* define additional operations
  - Generally based on **theory of set concepts**
- **Operations** of **Set** (extract)
  - Receiving object is denoted by set



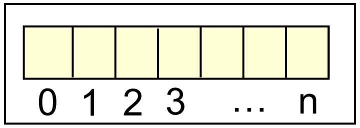
Operation	Explanation of result
union(set2:Set(T)):Set(T)	Union of *set* and *set2*
intersection(set2:Set(T)):Set(T)	Intersection of *set* and *set2*
difference(set2:Set(T)):Set()	Difference set; elements of *set*, which do not consist in *set2*
symmetricDifference(set2:Set(T)): Set(T)	Set of all elements, which are either in *set* or in *set2*, but do not exist in both sets at the same time

- **Operations** of **Bag** (extract)
  - Receiving object is denoted by bag

Operation	Explanation of result
union(bag2:Bag(T)):Bag(T)	Union of *bag* and *bag2*
intersection(bag2:Bag(T)): Bag(T)	Intersection of *bag* and *bag2*

Operations for OrderedSet/Sequence

- *OrderedSet* and *Sequences* define additional operations
  - Allow access or modification through an **Index**



- **Operations** of **OrderedSet** (extract)
  - Receiving object is denoted by *orderedSet*

Operation	Explanation of result
first:T	First element of *orderedSet*
last:T	Last element of *orderedSet*
at(i:Integer):T	Element on index i of *orderedSet*
subOrderedSet(lower:Integer, upper:Integer):OrderedSet(T)	Subset of *orderedSet,* all elements of *orderedSet* including the element on position *lower* and the element on position *upper*
insertAt(index:Integer,object:T) :OrderedSet(T)	Result is a copy of the *orderedSet*, including the element *object* at the position *index*

- **Operations** of **Sequence**
  - Analogous to the operations of OrderedSet

Iterator-based operations

- OCL defines operations for *Collections* using *Iterators*
  - Expression Package: LoopExp
  - **Projection** of new *Collections* out of existing ones
  - Compact **declarative specification** instead of imperative algorithms

- Predefined Operations
  - select(exp) : *Collection*
  - reject(exp) : *Collection*
  - collect(exp) : *Collection*
  - forAll(exp) : *Boolean*
  - exists(exp) : *Boolean*
  - isUnique(exp) : *Boolean*

- iterate(…) – Iterate over all elements of a *Collection*
  - Generic operation
  - Predefined operations are defined with iterate(…)

Iterator-based operations

Select-/Reject-Operation

- **Select** and **Reject** return subsets of collections
  - Iterate over the complete collection and collect elements
- Select
  - **Result**: Subset of collection, including elements where *booleanExpr* is **true**

```
collection -> select( v : Type | booleanExp(v) )
collection -> select( v | booleanExp(v) )
collection -> select( booleanExp )
```

- Reject
  - **Result**: Subset of collection, including elements where *booleanExpr* is **false**
  - Just *Syntactic* Sugar, because each *reject-Operation* can be defined as a *select-Operation* with a negated expression

```
collection-> reject(v : Type | booleanExp(v))
```

=

```
collection-> select(v : Type | not (booleanExp(v)))
```

Iterator-based operations

Select-/Reject-Operation

- Semantic of the *Select-Operation*

OCL

```
context Company inv:
   self.employee -> select(e : Employee | e.age>50)  ->
notEmpty()
```

Java

```
List persons<Person> = new List();
for ( Iterator<Person> iter = comp.getEmployee();
iter.hasNext() ){
     Person p = iter.next();
     if ( p.age > 50 ){
          persons.add(p);
     }
}
```

Iterator-based operations

Collect-Operation

- *Collect-Operation* returns a new collection from an existing one. It collects the **Properties** of the objects and not the objects itself.
  - Result of *collect* always **Bag<T>.T** defines the type of the property to be collected

```
collection -> collect( v : Type | exp(v) )
collection -> collect( v | exp(v) )
collection -> collect( exp )
```

- Example
  - *self.employees -> collect(age)* – Return type: Bag(Integer)
- Short notation for collect
  - *self.employees.age*

Iterator-based operations

Collect-Operation

- Semantic of the *Collect-Operator*

OCL

```
context Company inv:
   self.employee -> collect(birthdate) -> size() > 3
```

Java

```
List birthdate<Integer> = new List();
for ( Iterator<Person> iter = comp.getEmployee();
iter.hasNext() ){
    birthdate.add(iter.next().getBirthdate()); }
```

- Use of *asSet()* to eliminate duplicates

OCL

```
context Company inv:
   self.employee -> collect(birthdate) -> asSet()
```

Set
(without duplicates)

Bag
(with duplicates)

Iterator-based operations

ForAll-/Exists-Operation

- **ForAll** checks, if all elements of a collection evaluate to true

```
collection -> forAll( v : Type | booleanExp(v) )
collection -> forAll( v | booleanExp(v) )
collection -> forAll( booleanExp )
```

- **Example:** self.employees -> forAll(age > 18)

- **Nesting** of forAll-Calls (*Cartesian Product*)

```
context Company inv:
self.employee->forAll (e1 | self.employee -> forAll (e2 |
        e1 <> e2 implies e1.svnr <> e2.svnr))
```

- ***Alternative***: Use of multiple iterators

```
context Company inv:
self.employee -> forAll (e1, e2 | e1 <> e2 implies e1.svnr <> e2.svnr))
```

- **Exists** checks, if at least one element evaluates to true
  - Beispiel: employees -> exists(e: Employee | e.isManager = true)

Iterator-based operations

Iterate-Operation

- **Iterate** is the generic form of all iterator-based operations

- **Syntax**
  collection -> iterate(**elem** : Typ; **acc** : Typ =
  <initExp> | **exp(elem**, **acc**))
  - Variable **elem** is a typed ***Iterator***
  - Variable **acc** is a typed ***Accumulator***
  - Gets assigned initial value initExp
  - **exp(elem, acc)** is a function to calculate **acc**

- **Example**
  collection -> collect(x : T | x.property)

  -- semantically equivalent to:

  collection -> iterate(x : T; acc : T2 = Bag{} | acc -> including(x.property))

Iterator-based operations

Iterate-Operator

- Semantic of the *Iterate-Operator*

OCL

```
collection -> iterate(x : T; acc : T2 = value | acc -> u(acc, x)
```

Java

```
iterate (coll : T, acc : T2 = value){
      acc=value;
      for( Iterator<T> iter =
coll.getElements(); iter.hasNext(); ){
            T elem = iter.next();
            acc = u(elem, acc);
      }
}
```

- Example
  - Set{1, 2, 3} -> iterate(i:Integer, a:Integer=0 | a+i)
  - Result: 6

Tool Support

- **Wishlist**
  - Syntactic analysis: Editor support
  - Validation of logical consistency (Unambiguous)
  - Dynamic validation of invariants
  - Dynamic validation of Pre-/Post-conditions
  - Code generation and test automation

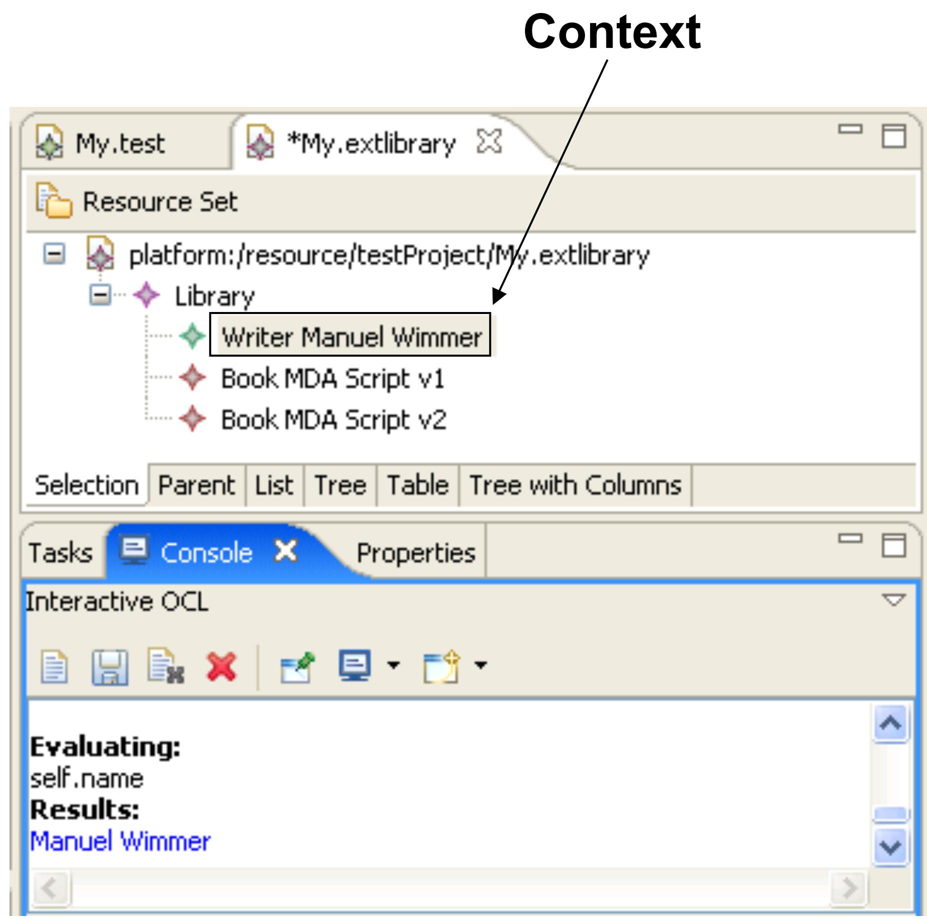
- **Today**
  - UML tools provide OCL editors
  - MDA tools provide code generation of OCL expressions
  - Meta modeling platforms provide the opportunity to define OCL Constraints for meta models.
    - The editor should dynamically check constraints or restrict modeling, respectively.

OCL Tools

- Some OCL-parsers, which check the syntax of OCL-constraints and apply them to the models, are for free.
  - IBM Parser
- Dresden OCL Toolkit 2.0
  - Generation of Java code out of OCL-constraints
  - Possible integration with ArgoUML
- USE: UML-based Specification Environment
  - http://sourceforge.net/projects/useocl
- OCL frameworks are originated in the areas of EMF and the UML2 project of Eclipse
  - Octopus
  - Frauenhofer Toolkit
  - OSLO
  - EMFT OCL-Framework/Query-Framework

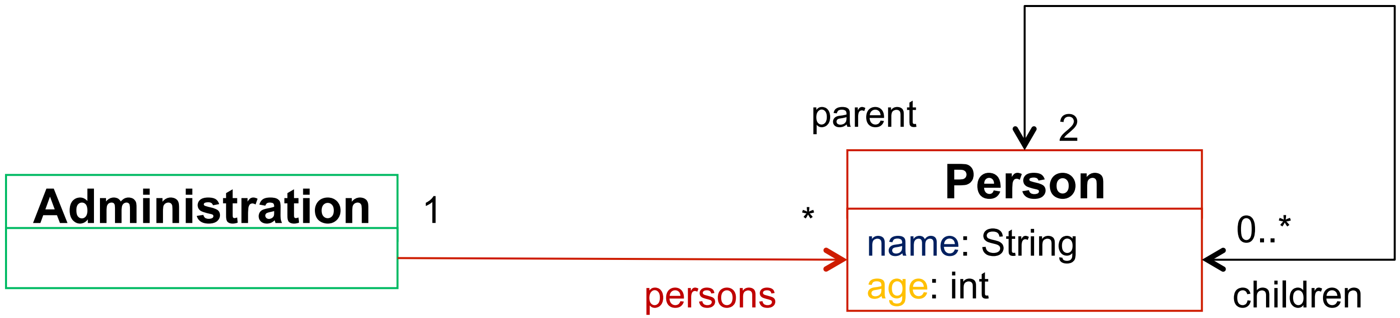
OCL Tools

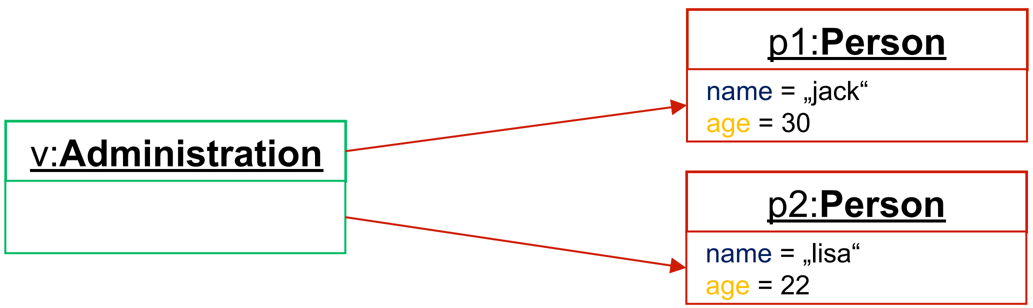
- **EMFT OCL-Framework**
  - Based on EMF
  - *OCL-API* – Enables the use of OCL in Java programs
  - *Interactive OCL Console* – Enables the definition and evaluation of OCL-constraints



- **EMFT Query-Framework**
  - **Goal**: SQL-like query of model information
  - **select** exp **from** exp **where** *oclExp*

Example 1: Navigation (1)

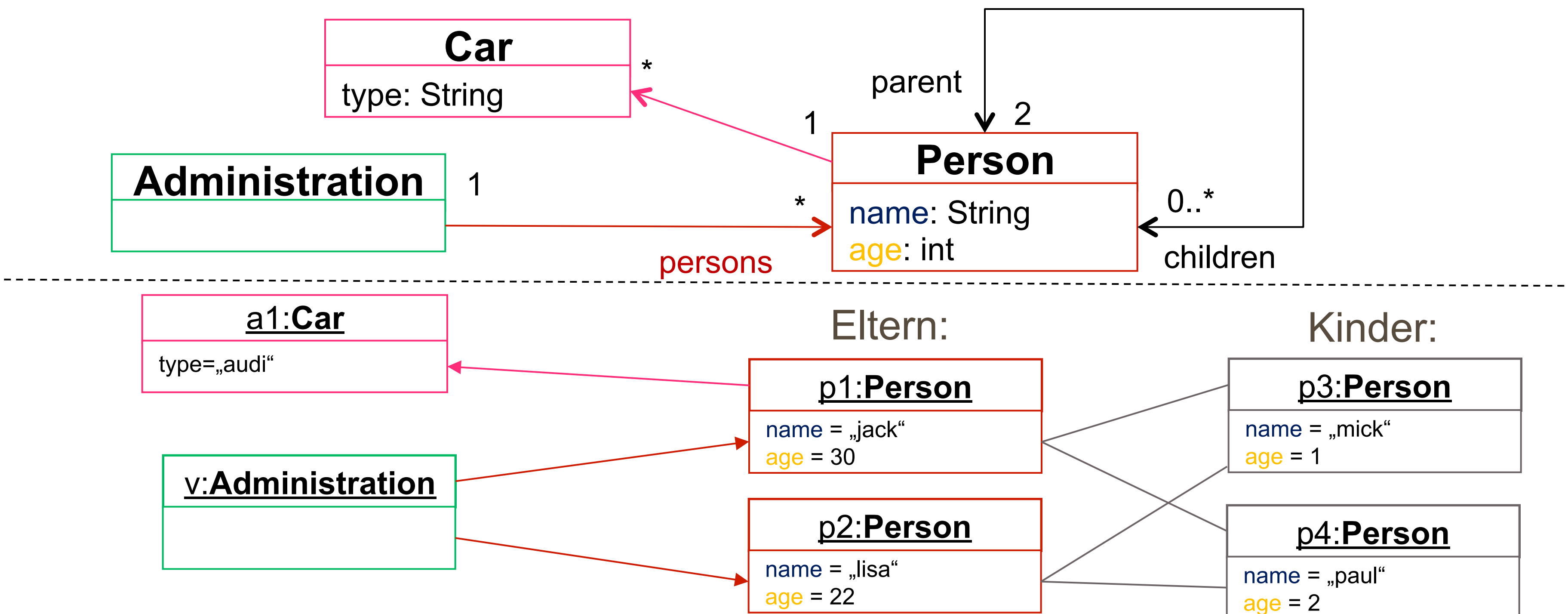




context Administration:

- `self.persons` → {Person p1, Person p2}
- `self.persons.name` → {jack, lisa}
- `self.persons.alter` → {30, 22}

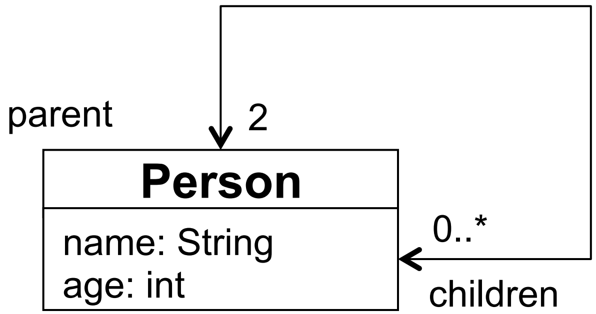
Example 1: Navigation (2)



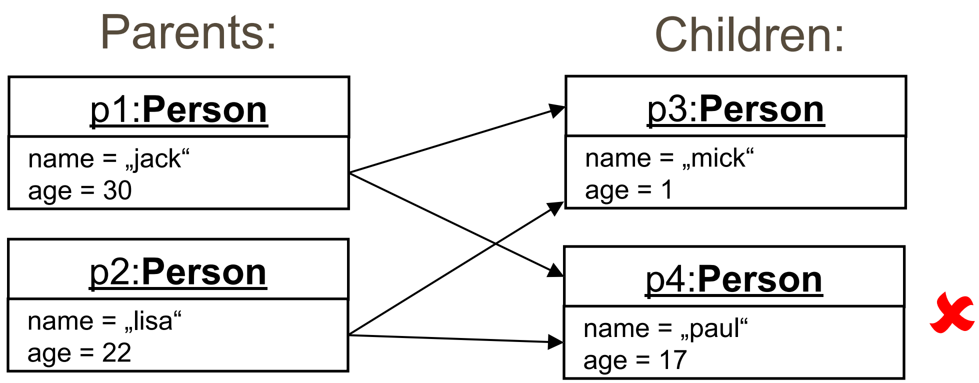
context Administration:

- `self.persons.children` → {{p3, p4}, {p3, p4}}
- `self.persons.children.parent` → {{{p1, p2}, {p1, p2}}, ...}
- `self.persons.car.type` → {{“audi”}}

Example 2: Invariant (1)



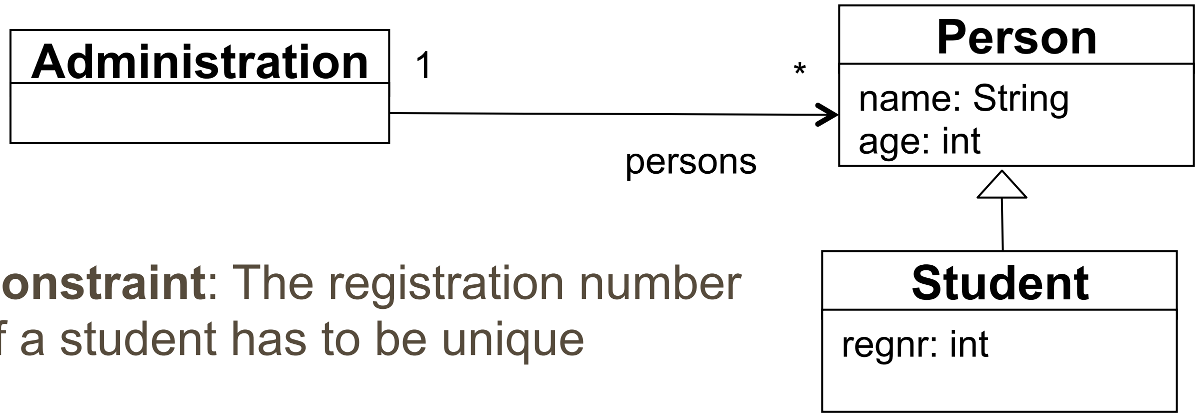
Constraint: A child is at least 15 years younger than his parents.




```
context Person
inv: self.children->forAll(k : Person | k.age
 < self.age-15)
```


Marco Brambilla, Jordi Cabot, Manuel Wimmer.
Model-Driven Software Engineering In Practice. Morgan & Claypool 2012.

Example 2: Invariant (2)

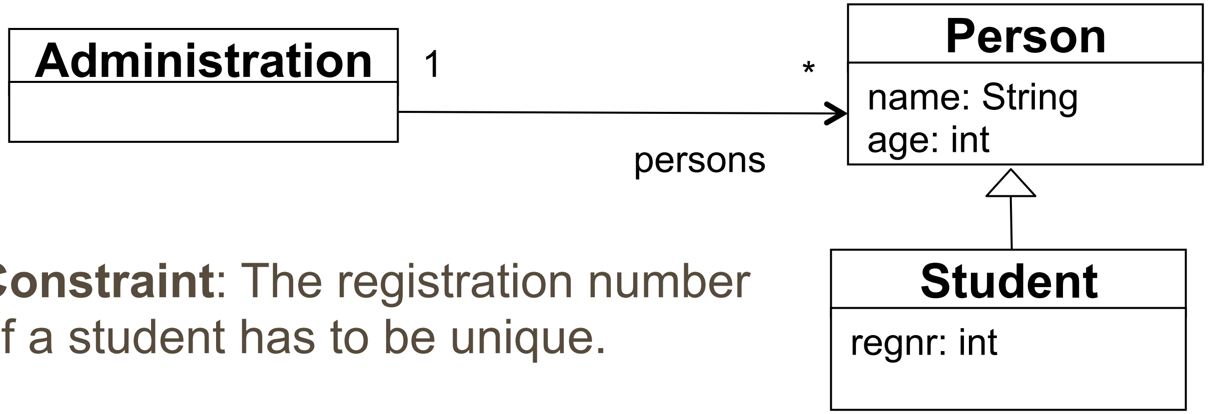


Constraint: The registration number of a student has to be unique

```
context Administration
inv uniqueRegnr :
  self.persons -> select(e : Person| e.oclIsTypeOf(Student))
                  -> forAll(e1 |
   self.persons -> select(e : Person |
  e.oclIsTypeOf(Student))
                  -> forAll(e2 |
  e1 <> e2 implies   e1.oclAsType(Student).regnr      <>

      e2.oclAsType(Student).regnr))
```

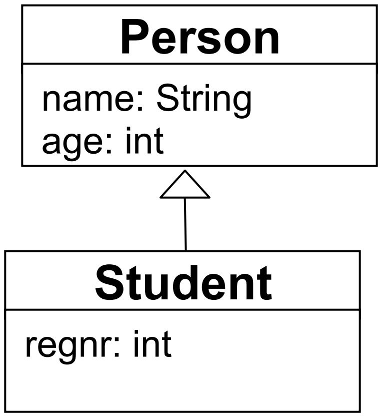
Example 2: Invariant (2) cont.



Constraint: The registration number of a student has to be unique.

```
context Administration
inv uniqueRegnr :
  self.persons -> select(e : Person| e.oclIsTypeOf(Student))
       -> forAll(e1, e1 | e1 <> e2 implies
       e1.oclAsType(Student).regnr <>
       e2.oclAsType(Student).regnr)
       )
```

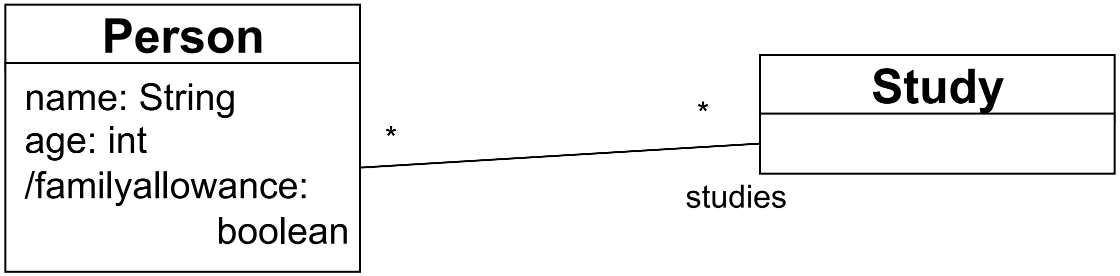
Example 2: Invariant (2) cont.



Constraint: The registration number of a student has to be unique.

```
context Student
inv uniqueRegnr :
  Student.allInstances() -> forAll(e1, e1 | e1 <> e2 implies
        e1.oclAsType(Student).regnr <>
        e2.oclAsType(Student).regnr))
```

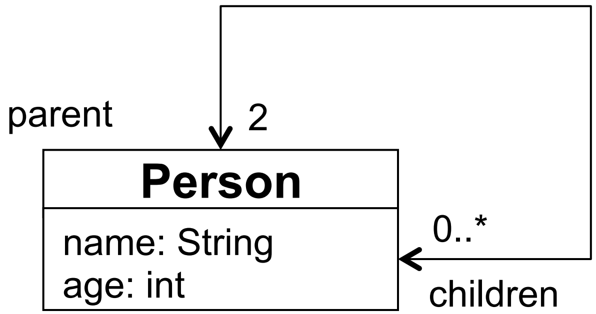

Example 3: Inherited attribute



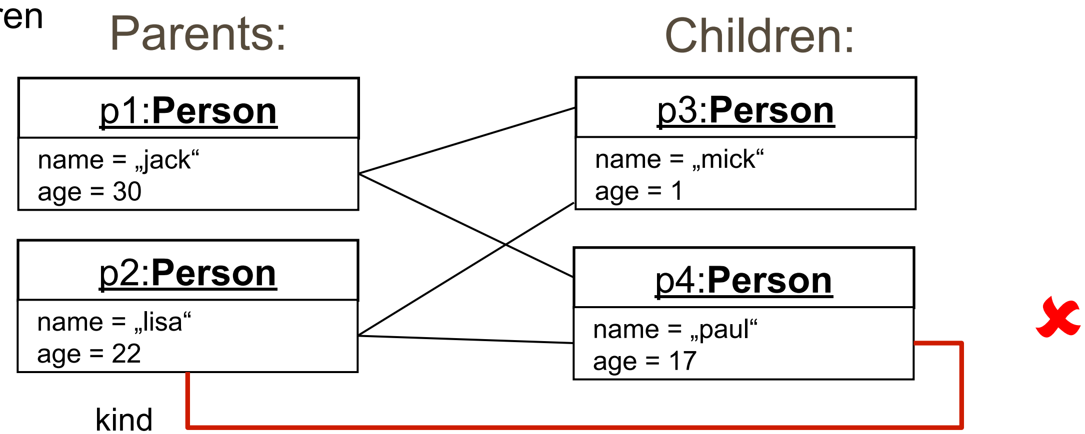
A Person obtains family allowance, if he/she is younger than 18 years, or if he/she is studying and younger than 27 years old.

```
context Person::familyallowance
derive: self.age < 18 or
        (self.age < 27 and self.studies -> size() > 0)
```

Example 4: Definitions



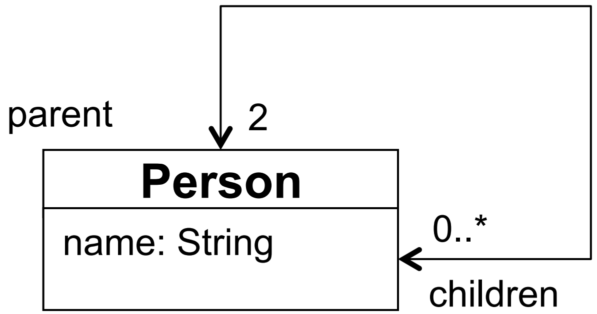
Constraint: A Person is not a relative of itself




```
context Person
def: relative: Set(Person) = children-> union(relative)
inv: self.relative -> excludes(self)
```

Assumption: Fixed-point semantic, otherwise if then else required

Example 5: equivalent OCL-formulations (1)



Constrain: A person is not its own child

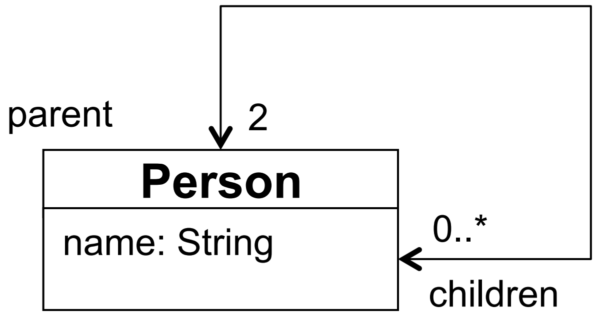
- **`(self.children->select(k | k = self))->size() = 0`**

  The Number of children for each person „self", where the children are the person „self", have to be 0.

- **`(self.children->select(k | k = self))->isEmpty()`**

  The set of children for each person „self, where the children are the person „self", has to be empty.

Example 5: equivalent OCL-formulations (2)



Constrain: A person is not its own child

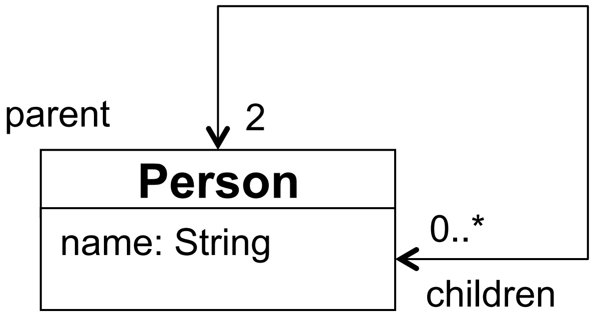
- **`not self.children->includes(self)`**

  It is not possible, that the set of children of each person „self“ contains the person „self“.

- **`self.children->excludes(self)`**

  `The set of children of each person „self“ cannot contain „self“.`

Example 5: equivalent OCL-formulations (3)

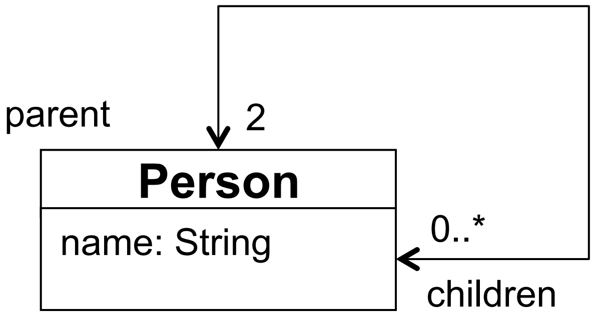


Constrain: A person is not its own child

- **`Set{self}->intersection(self.children)->isEmpty()`**
  The intersection between the one element set, which only includes one person „self“ and the set of the children of „self“ has to be empty.

- **`(self.children->reject(k | k <> self))->isEmpty()`**
  The set of children for each person „self“, for whome does not apply, that they are not equal to the person „self“, has to be empty.

Example 5: equivalent OCL-formulations (4)



Constrain: A person is not its own child

- **`self.children->forAll(k | k <> self)`**

  Each child of the person „self“ is not the person „self“.

- **`not self.children->exists(k | k = self)`**

  There is no child for each person „self“, which is the person „self“

Outlook

ATLAS Transformation Language (ATL)

- **Query-part** (`from`) – What has to be transformed?
  - When **selecting** the relevant model elements
    - additionally to the type indication, constraints on attributes and association ends are required,
    - which are specified in OCL.
- **Generation-part**(`to`) – What has to be created?
  - When **creating** the target structure
    - additionally to the type information, derived information is required,
    - which are calculated in OCL.

Transformation rules

Query-part

OCL-expression

Generation-part

```
rule Property2Attribute {
  from p : UML!Property (
     p.association.oclIsUndefined()
  )
  to a : ER!Attribute (
     name <- p.name.toUpper(),
     entity <- p.owningClass
  )
}
```

References on OCL

- **Literature**
  - Object Constraint Language Specification, Version 2.0
    - http://www.omg.org/technology/documents/formal/ocl.htm
  - Jos Warmer, Anneke Kleppe: The Object Constraint Language - Second Edition, Addison Wesley (2003)
  - Martin Hitz et al: UML@Work, d.punkt, 2. Auflage (2003)

- **Tools**
  - OSLO - http://oslo-project.berlios.de
  - Octopus - http://octopus.sourceforge.net
  - Dresden OCL Toolkit - http://dresden-ocl.sourceforge.net
  - EMF OCL - http://www.eclipse.org/modeling/mdt/?project=ocl
  - USE - http://sourceforge.net/projects/useocl

MORGAN & CLAYPOOL PUBLISHERS

MODEL-DRIVEN SOFTWARE ENGINEERING IN PRACTICE

Marco Brambilla,
Jordi Cabot,
Manuel Wimmer.
Morgan & Claypool, USA, 2012.

www.mdse-book.com
www.morganclaypool.com
or buy it at: www.amazon.com