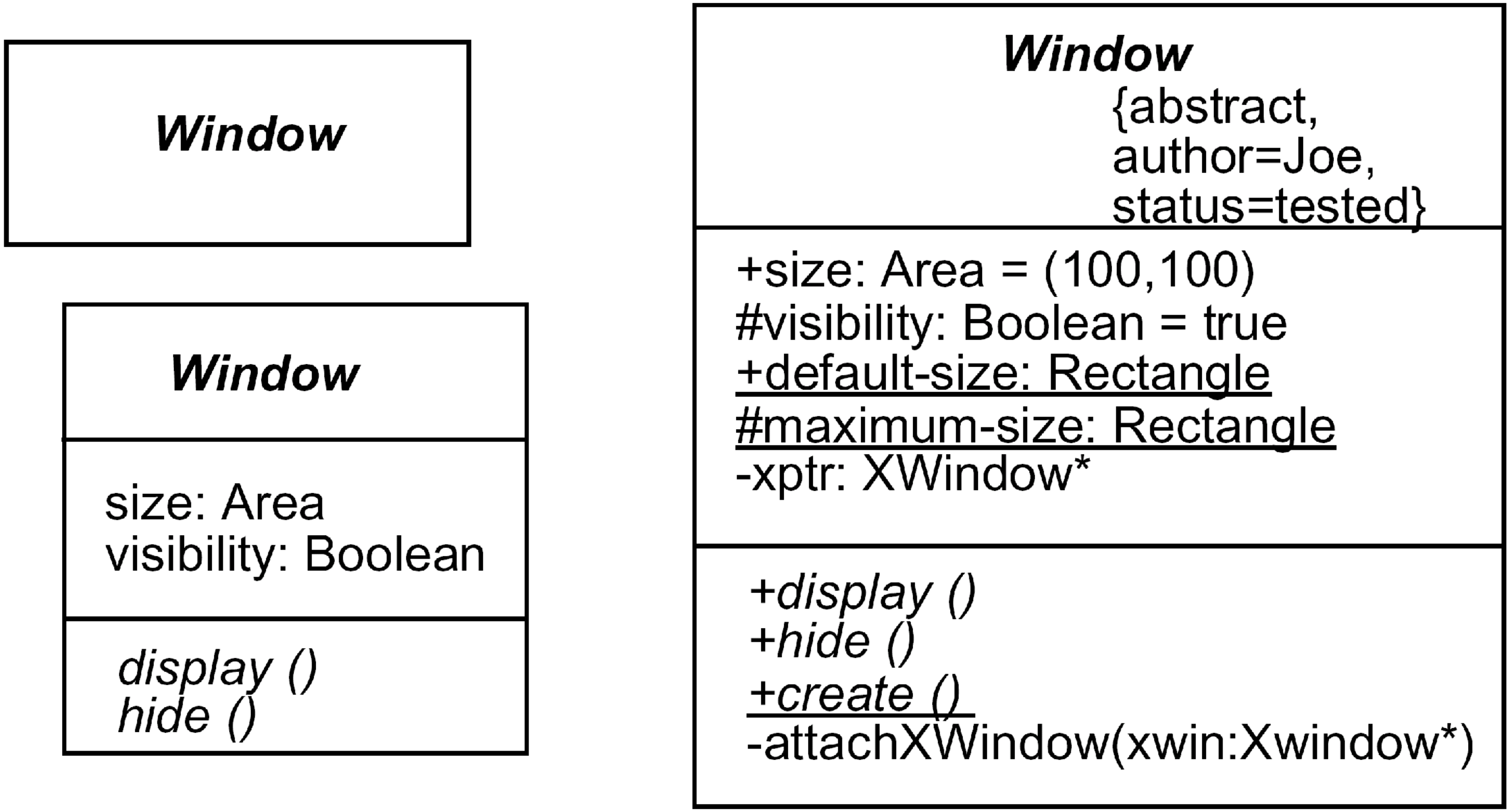
4. Class Diagrams

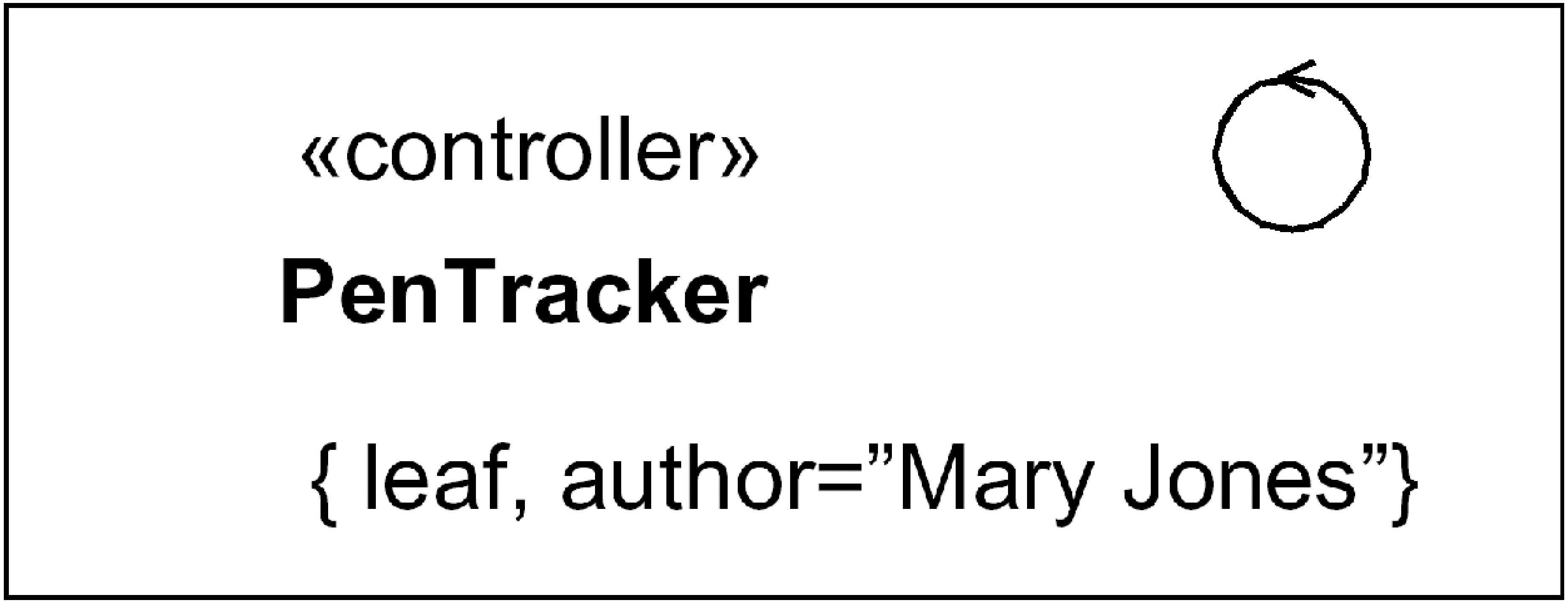
4.1 Examples for Static Structure Diagrams

- To follow: examples are taken from the UML notation guide
- Static structure diagrams include class diagrams and object diagrams

Static structure diagram (3-20)



Static structure diagram (3-21)



Static structure diagram (3-22)

Rectangle
p1:Point p2:Point
«constructor» Rectangle(p1:Point, p2:Point) «query» area (): Real aspect (): Real . . . «update» move (delta: Point) scale (ratio: Real) . . .

Static structure diagram (3-23)

Reservation

operations

guarantee()
cancel ()
change (newDate: Date)

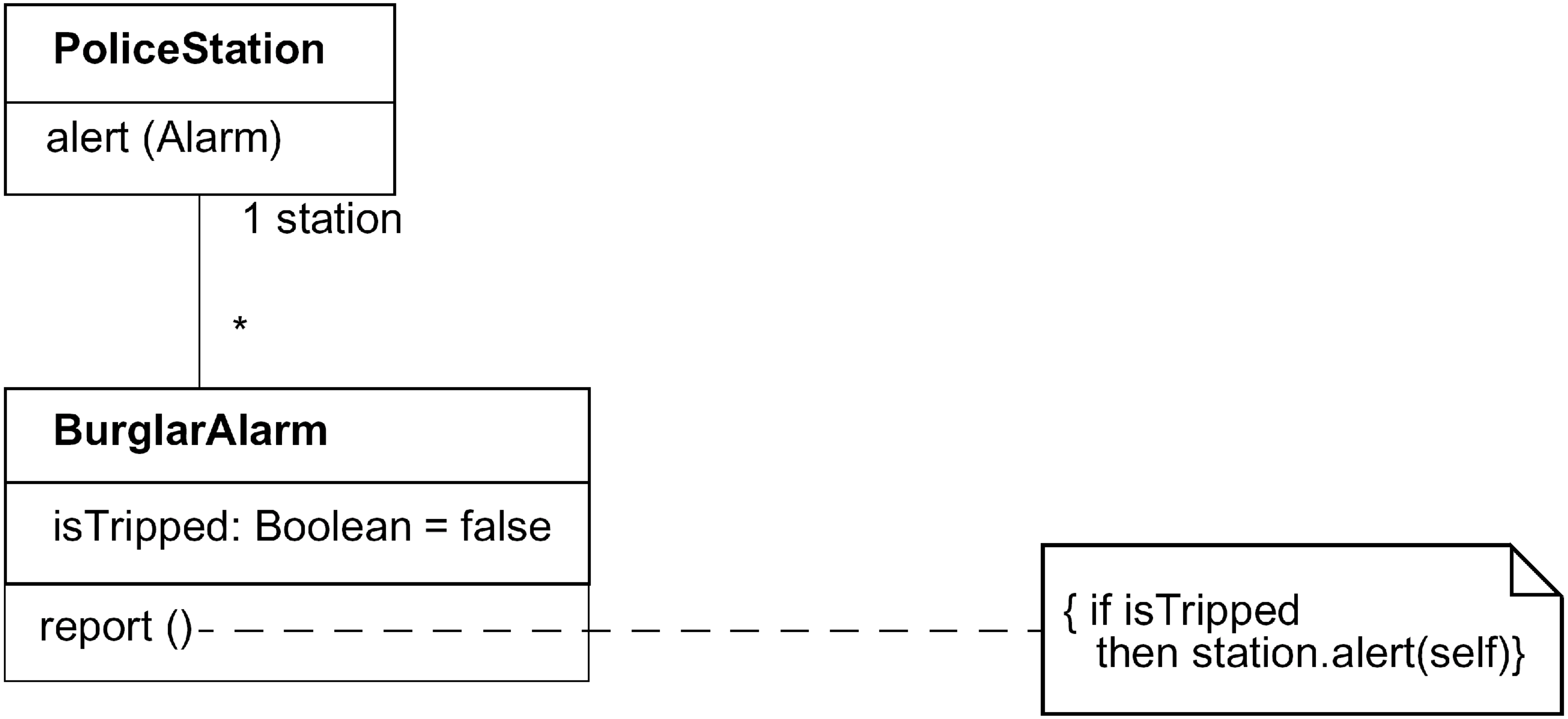
responsibilities

bill no-shows
match to available rooms

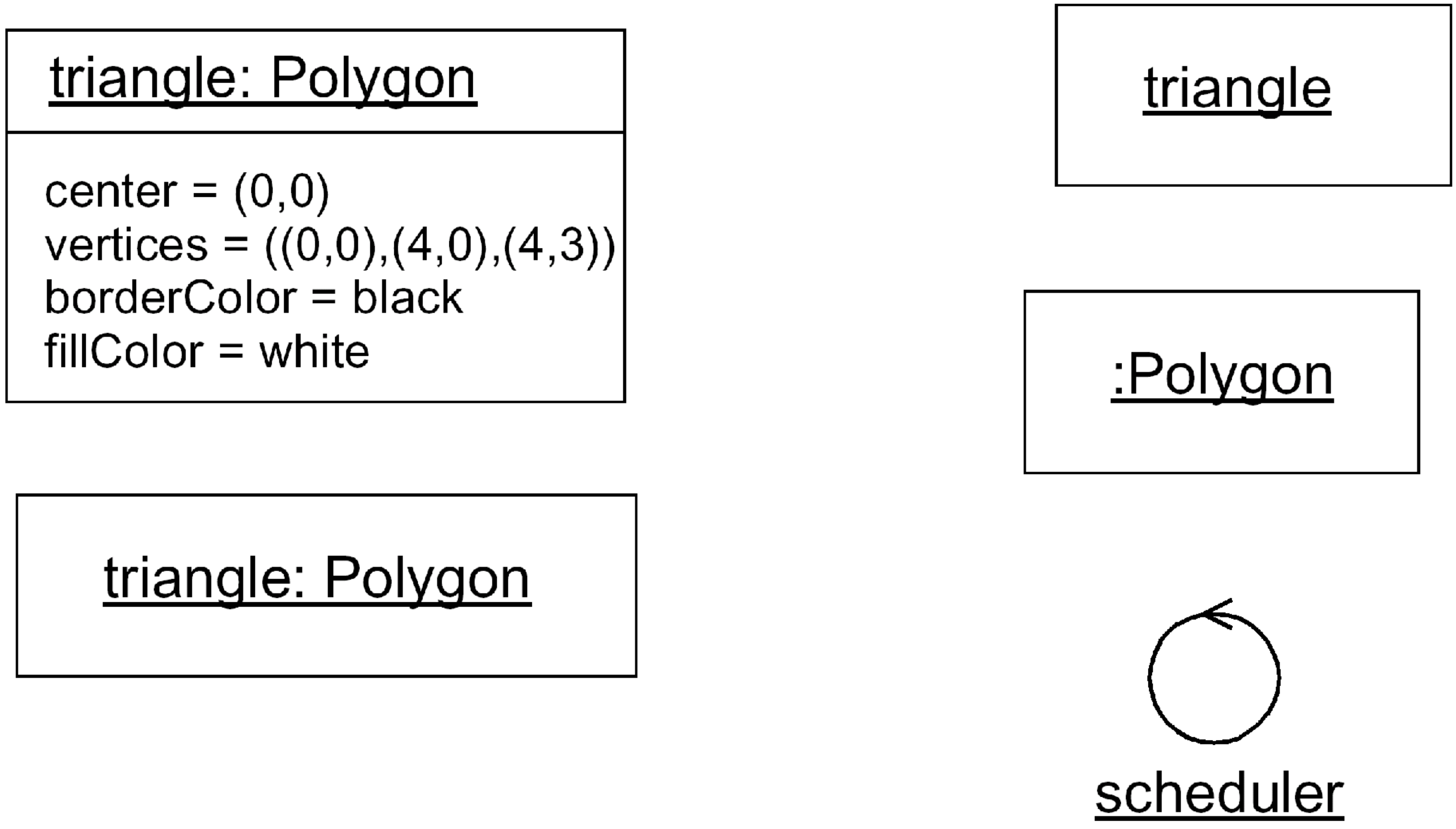
exceptions

invalid credit card

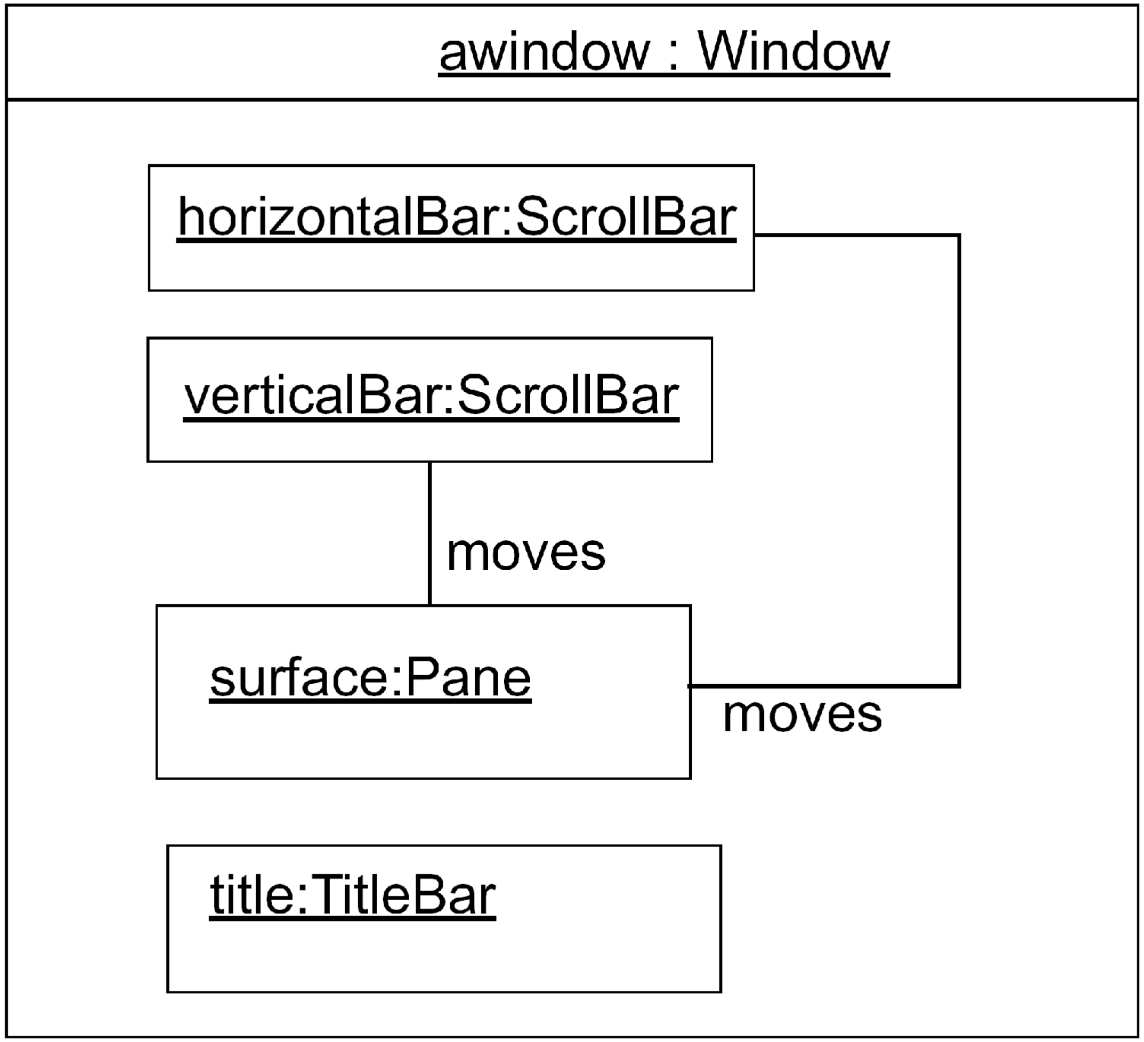
Static structure diagram (3-24)



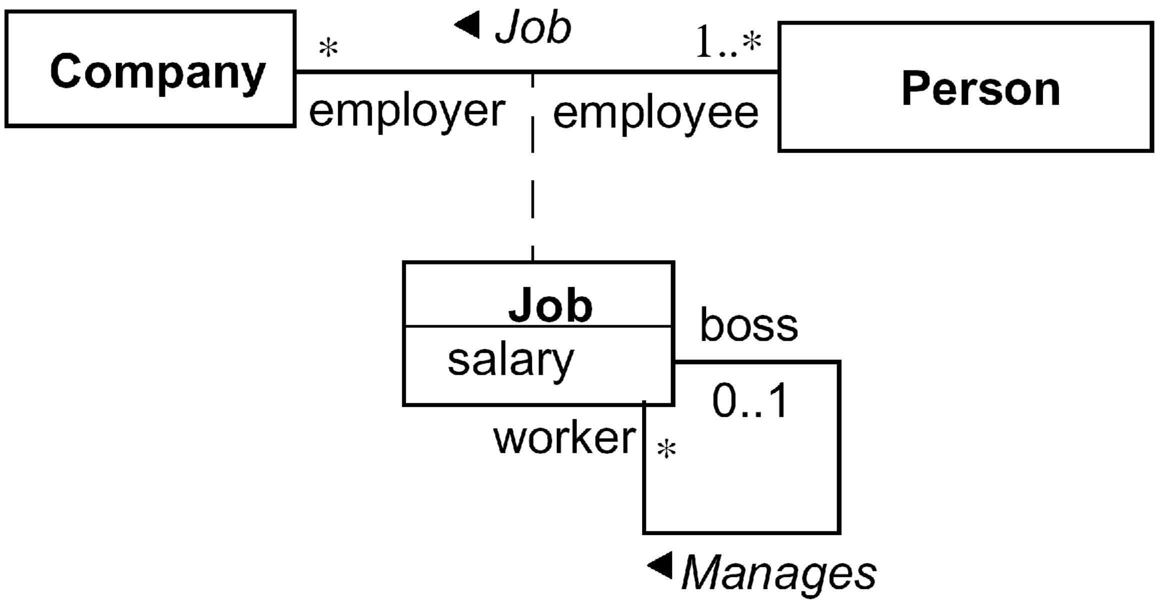
Static structure diagram (3-38)

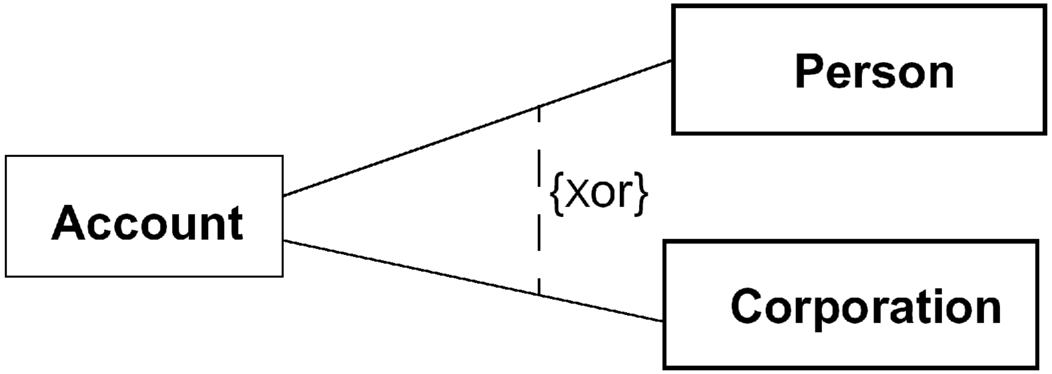


Static structure diagram (3-39)

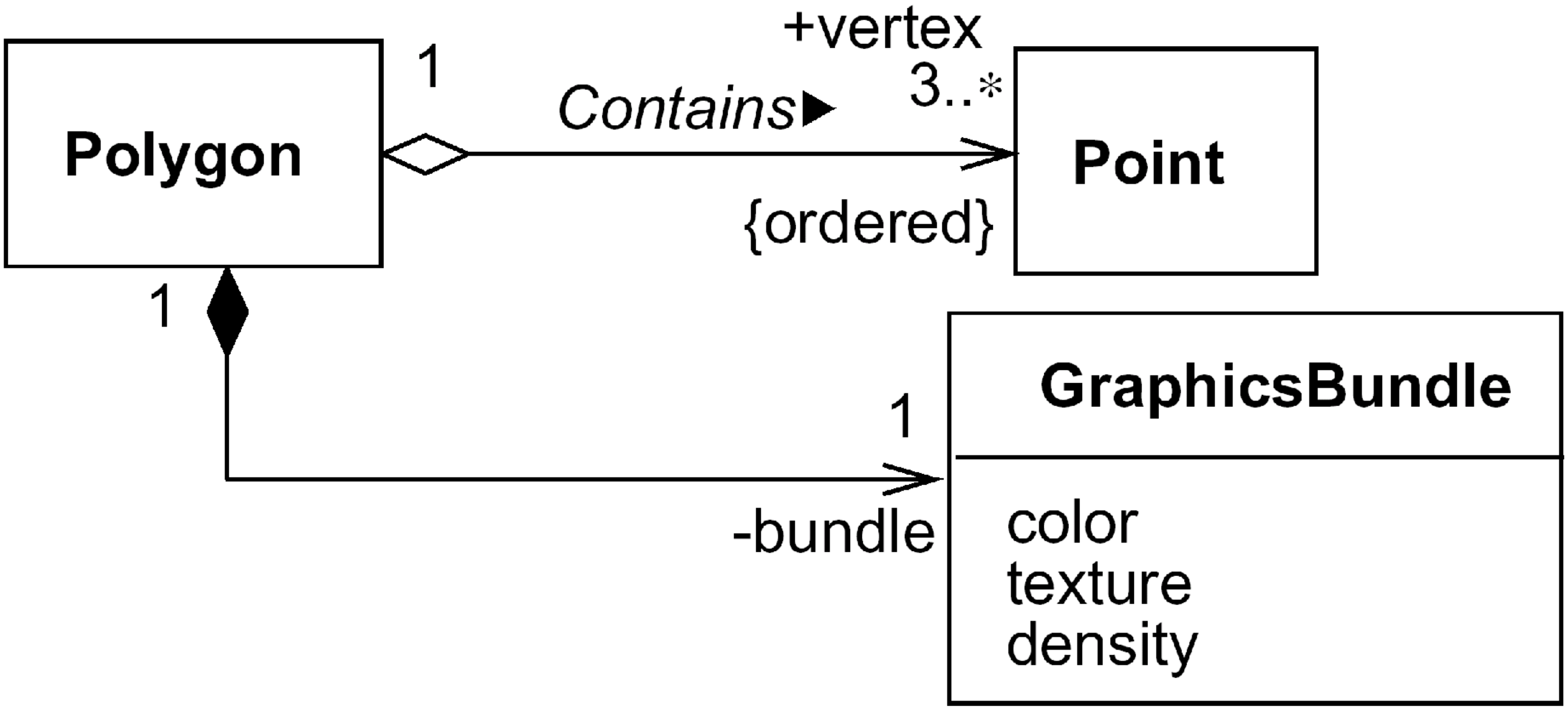


Static structure diagram (3-40)

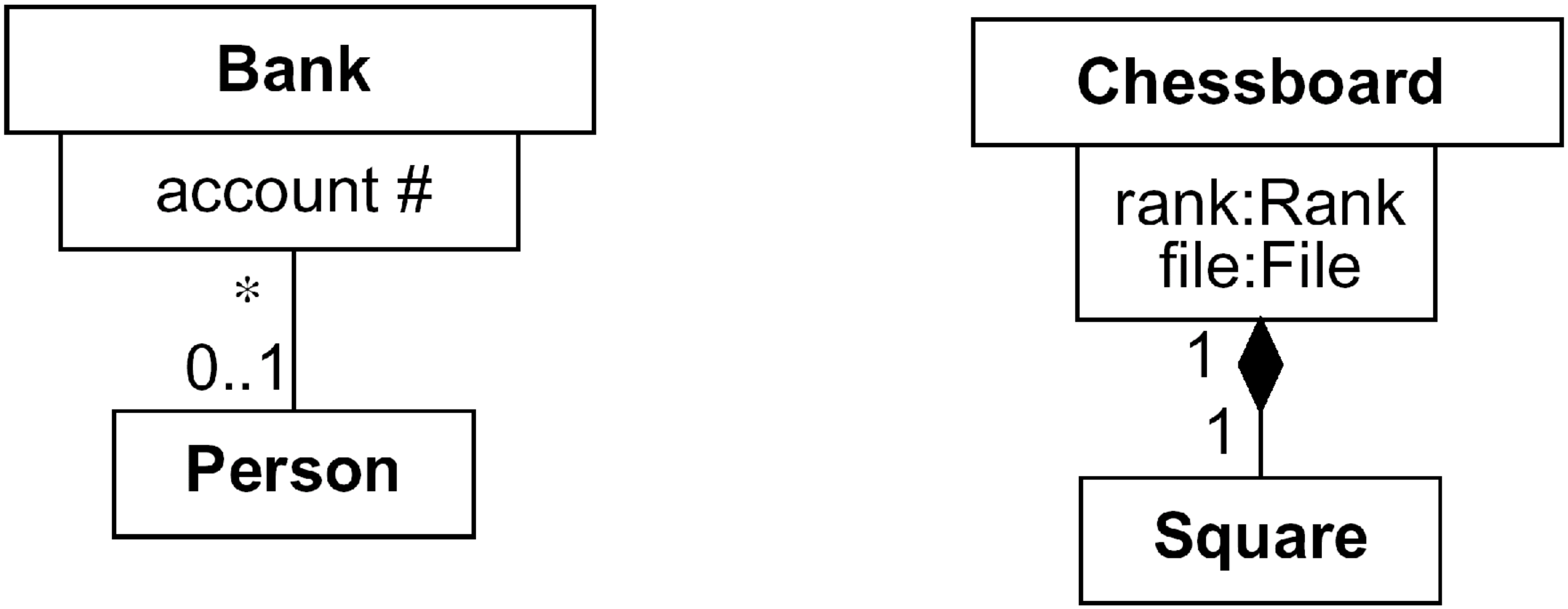




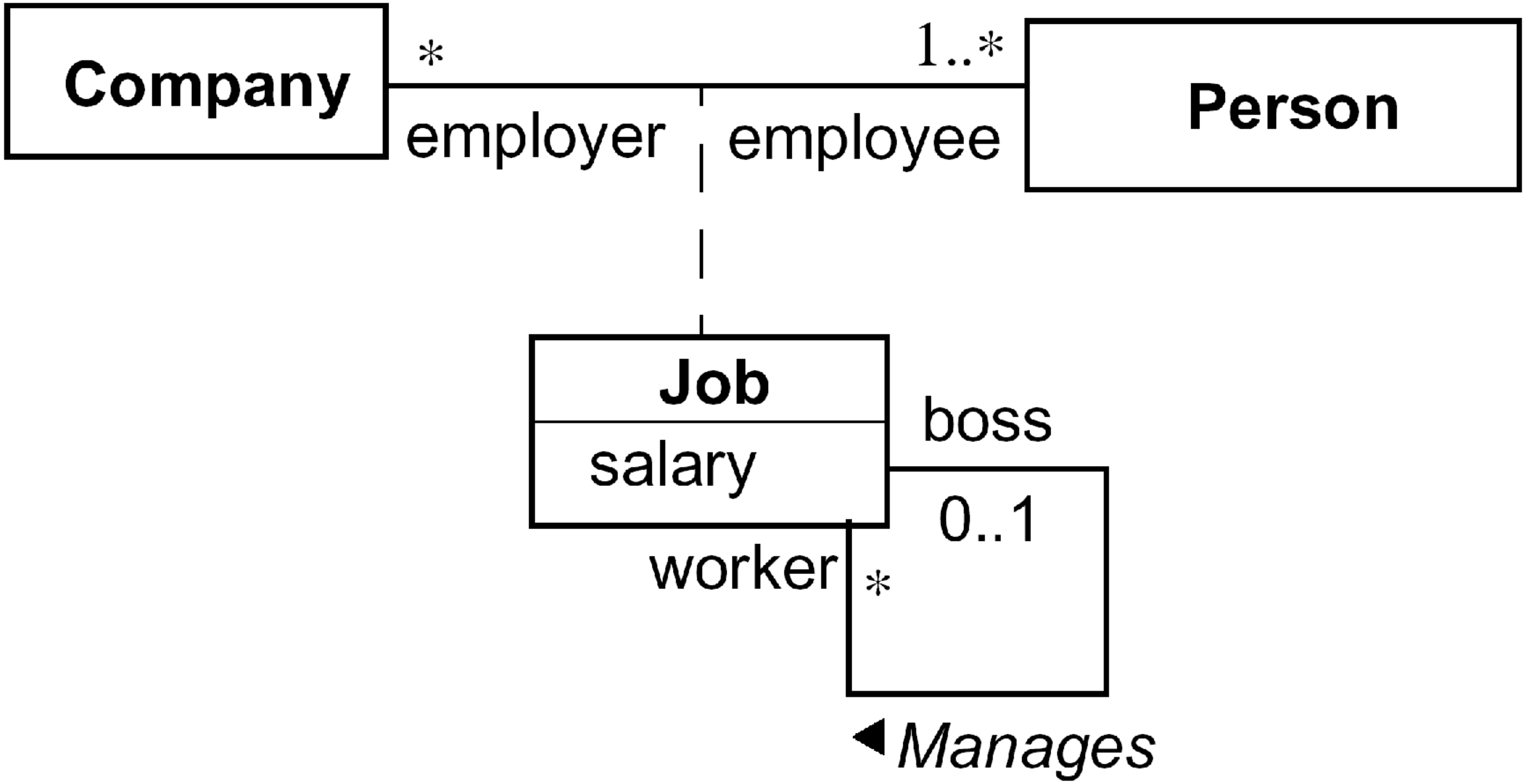
Static structure diagram (3-41)



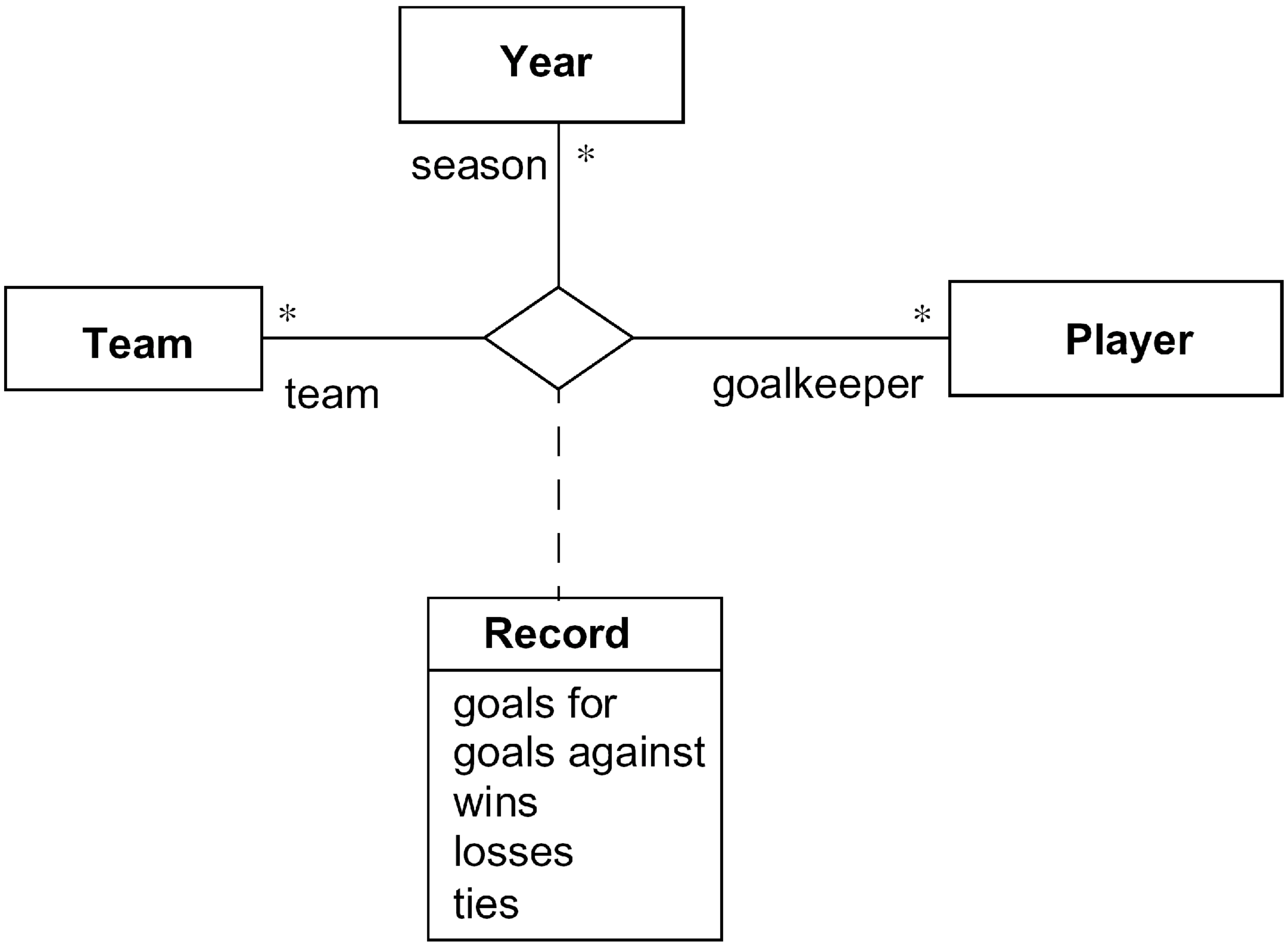
Static structure diagram (3-42)



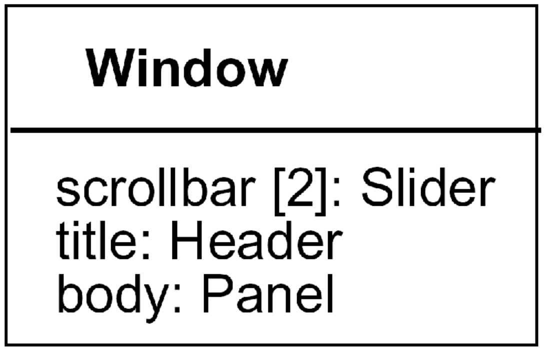
Static structure diagram (3-43)

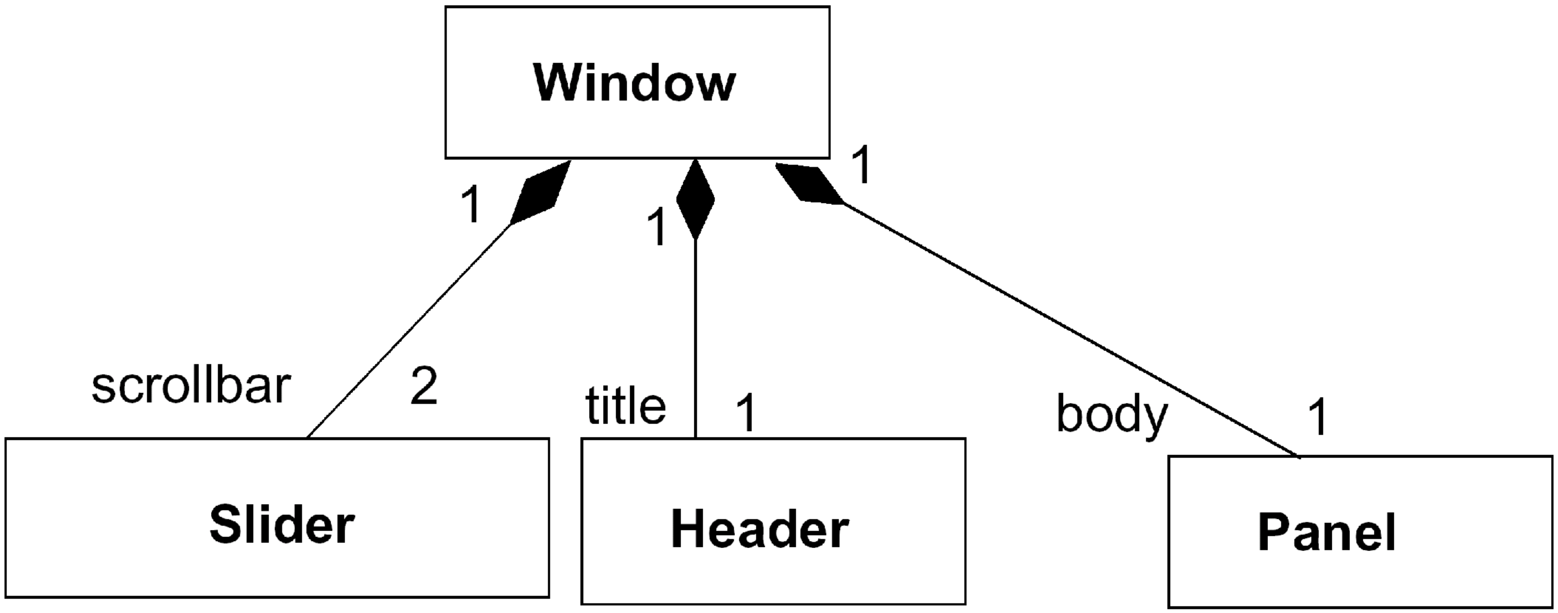


Static structure diagram (3-44)

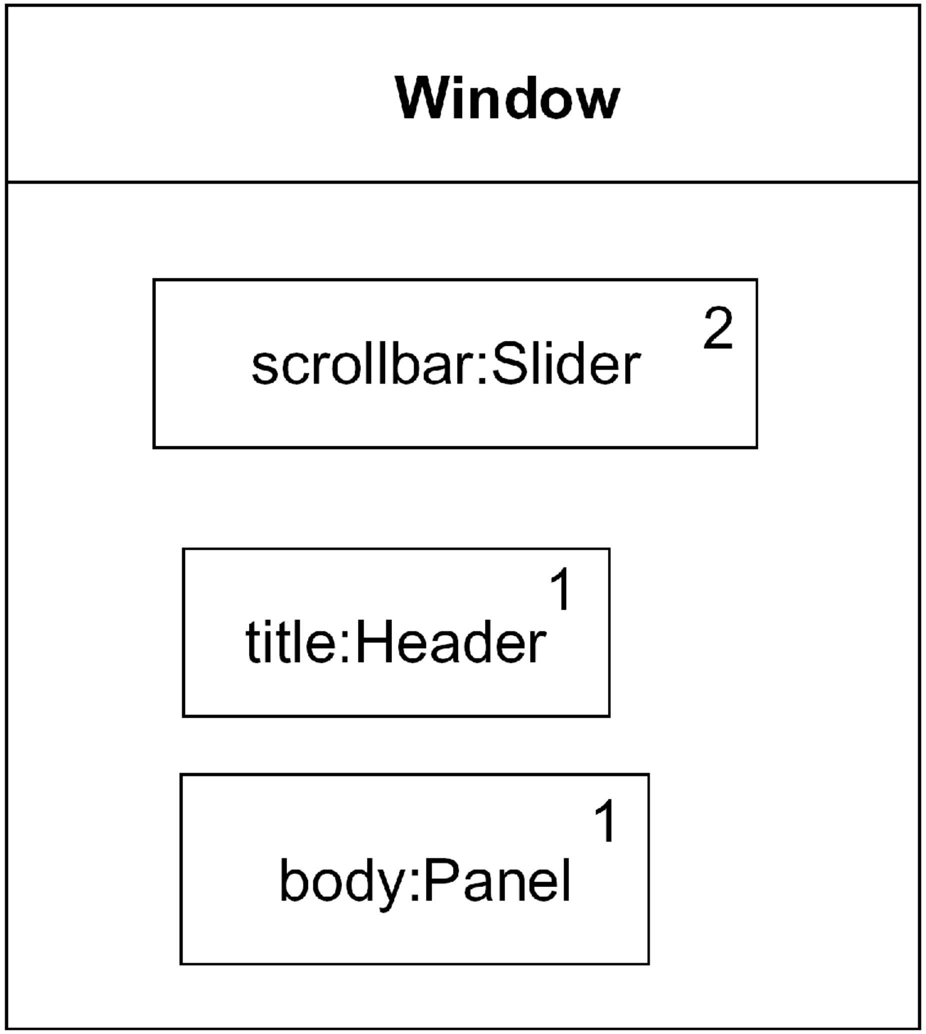


Static structure diagram (3-45-1)

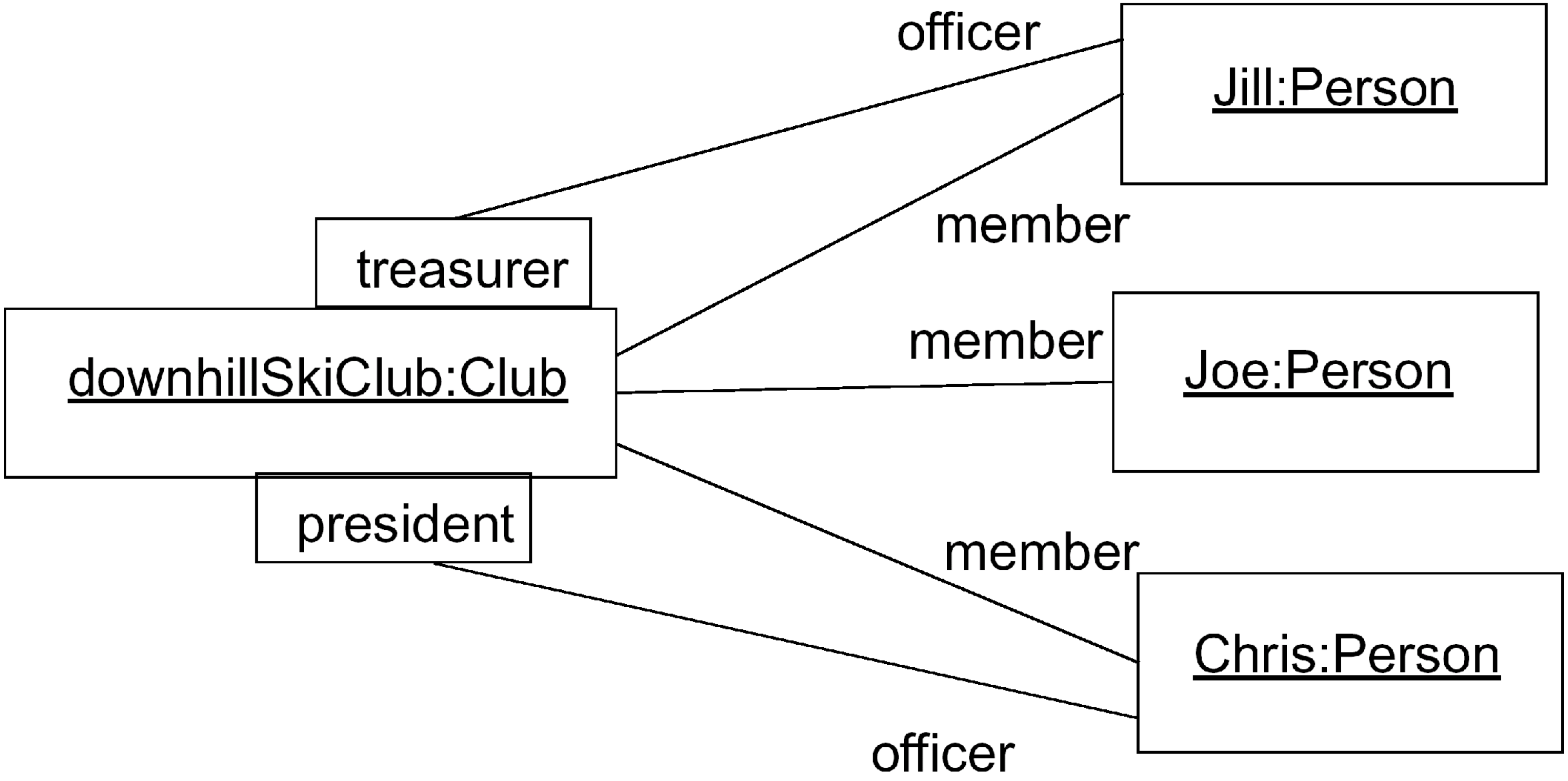




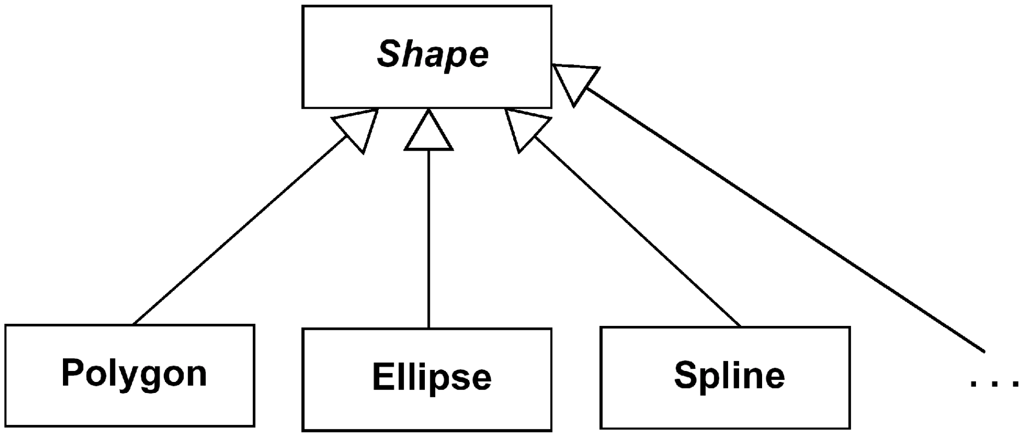
Static structure diagram (3-45-2)



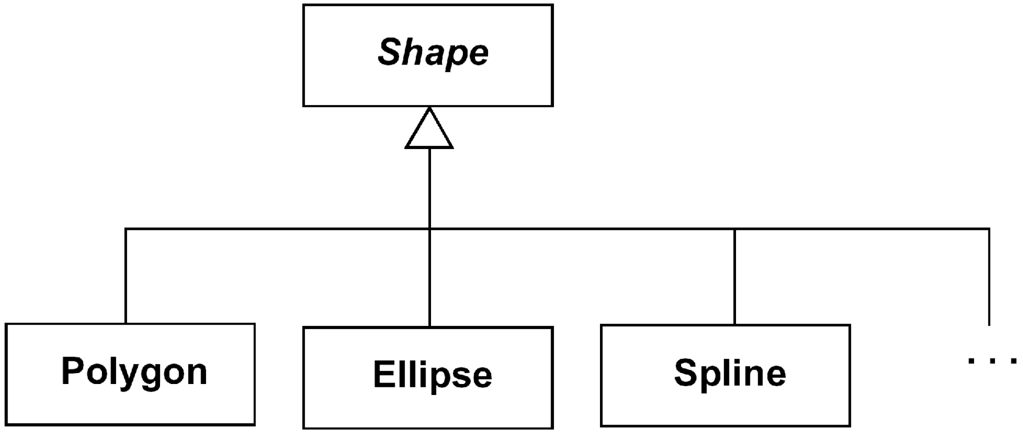
Static structure diagram (3-46)



Static structure diagram (3-47)

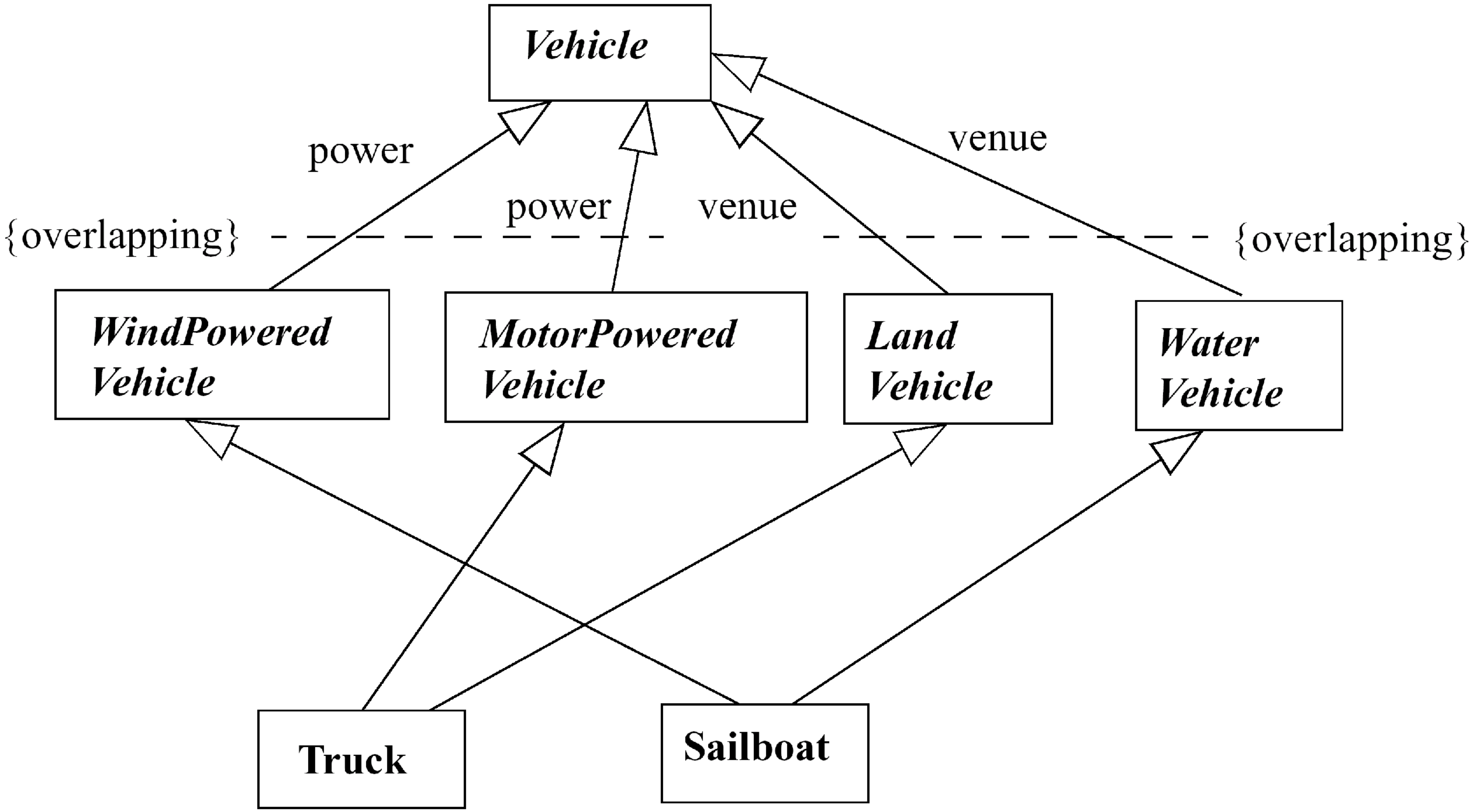


Separate Target Style

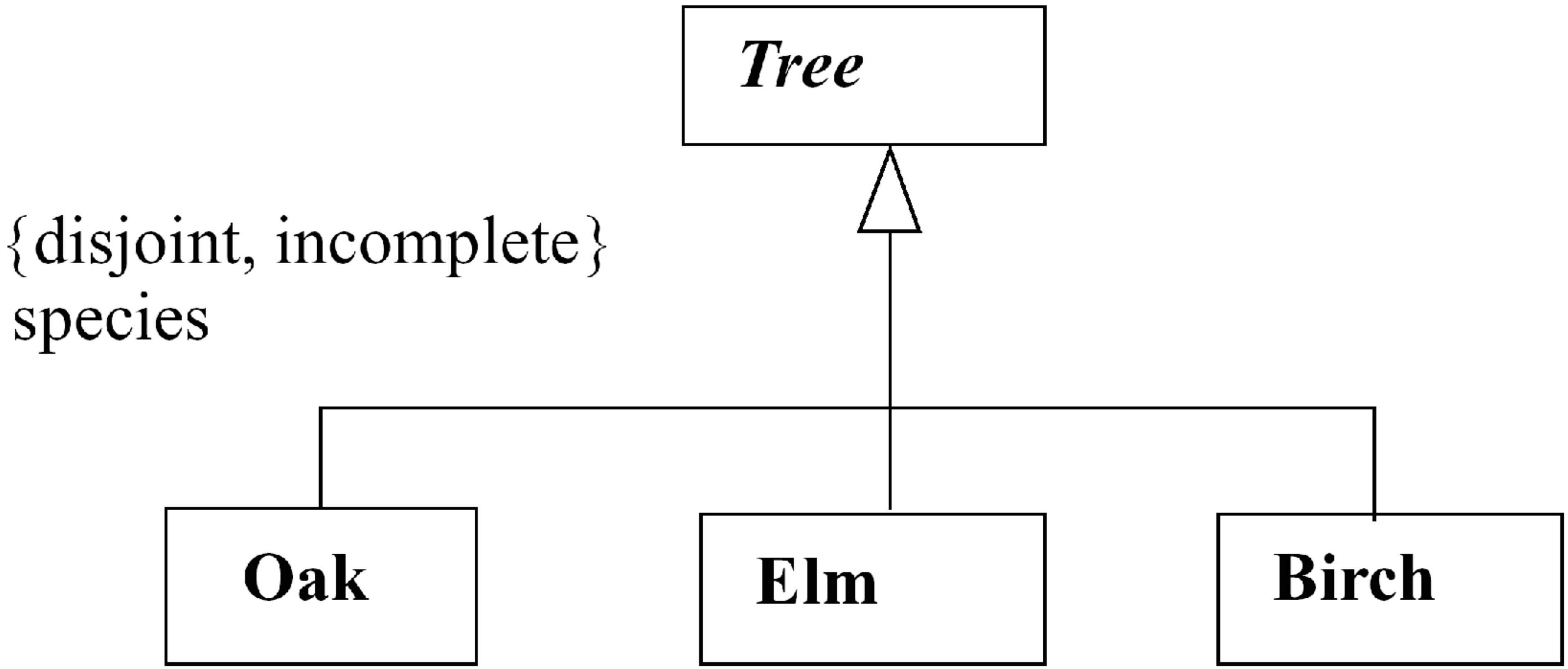


Shared Target Style

Static structure diagram (3-48)



Static structure diagram (3-49)



4.2 Advanced Concepts in Class Diagrams

Starting situation

- UML complex language with many modeling features
- no overall agreed semantic foundation
- how to deal with the language element variety?

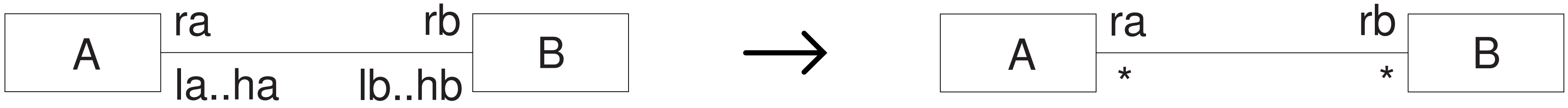
Study UML class and statechart diagrams and compare language elements

- explain advanced UML features by some more basic UML constructs
- use Object Constraint Logic OCL to express these advanced features

Benefits

- small set of features reduces complexity of design and facilitates communication
- smooth transition from utilizing basic set of language features to more sophisticated ones
- definition in terms of simple features helps identifying certain repeating patterns

Equivalence Rule for Cardinality Notation



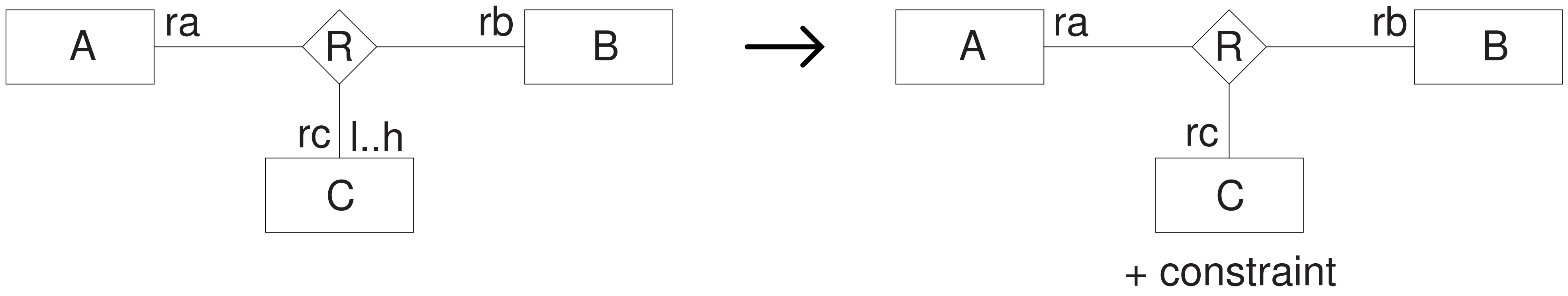
+ constraint

```
A.allInstances->forAll( a |
                          a.rb->size>=lb and a.rb->size<=hb )
and
B.allInstances->forAll( b |
                          b.ra->size>=la and b.ra->size<=ha )
```

Example for Cardinality Notation


```
Rook->forAll( r | r.protectedSquare->size>=2 and
                  r.protectedSquare->size<=14 )
and
Square->forAll( s | s.protectingRook->size>=0 and
                    s.protectingRook->size<=2 )
```

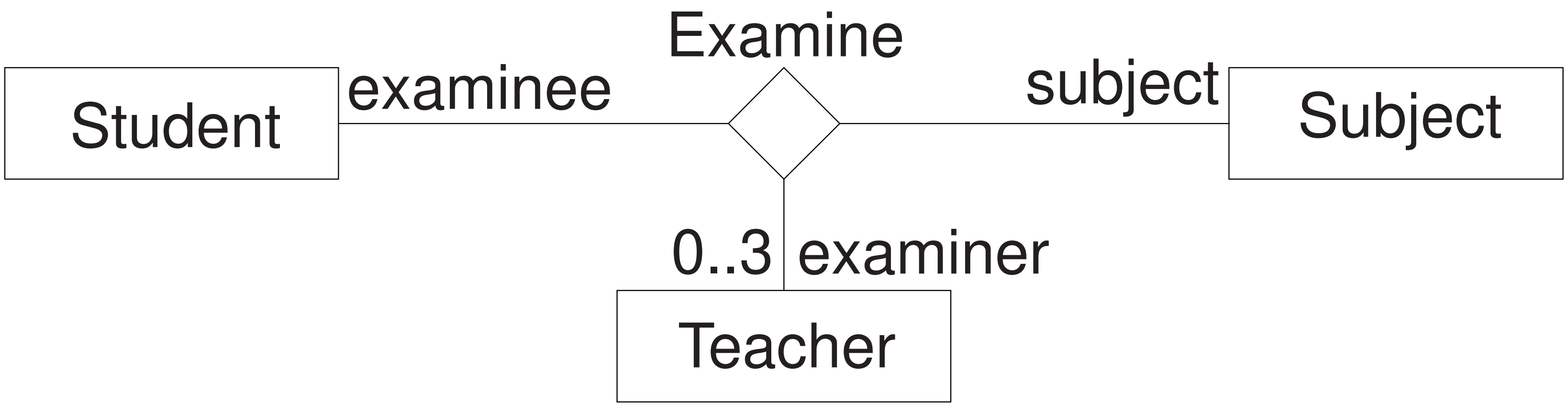
Equivalence Rule for Ternary Association Cardinality




```
A->forAll( a |
  B->forAll( b |
    C->select( c | R(a,b,c) )->size >= l and    -- (*)
    C->select( c | R(a,b,c) )->size <= h ) ) )
```

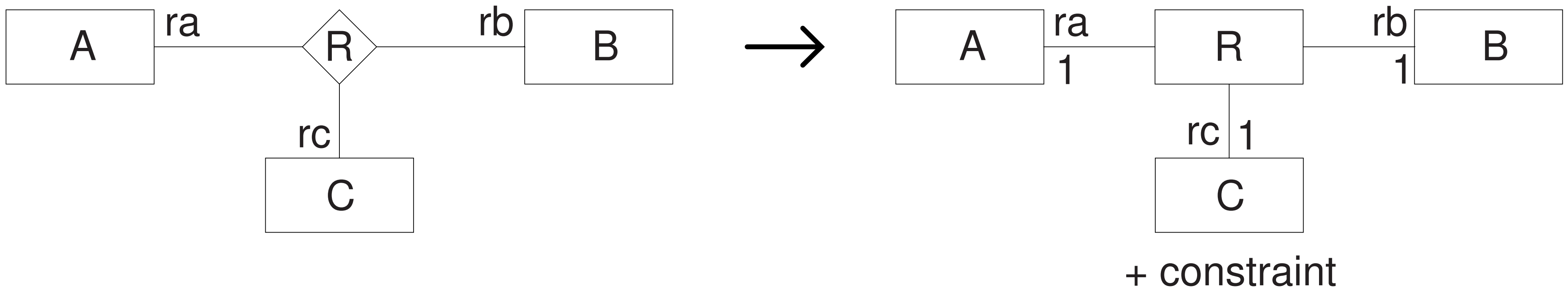
(*) Syntax `R(a,b,c)` not supported by OCL

Example for Cardinality Notation in N-ary Association



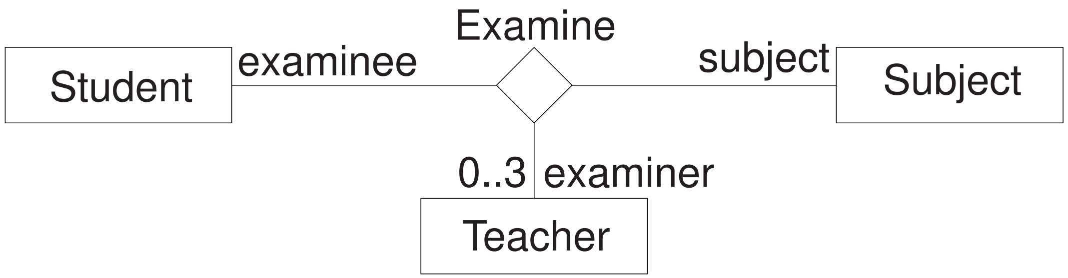

```
Student->forAll( st |
  Subject->forAll( su |
    Teacher->select( t | Examine(st,su,t) )->size >= 0
    and
    Teacher->select( t | Examine(st,su,t) )->size <= 3 ) )
```

Equivalence Rule for Ternary Association

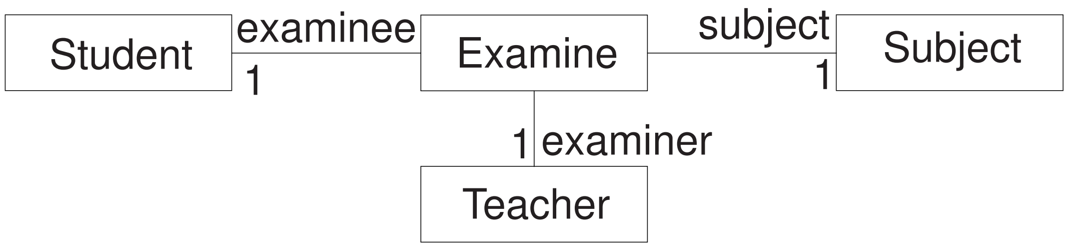



```
R->forAll( r, r' |
  (r.ra=r'.ra and r.rb=r'.rb and r.rc=r'.rc ) implies r=r')
```

Example for Equivalence Rule for Ternary Association

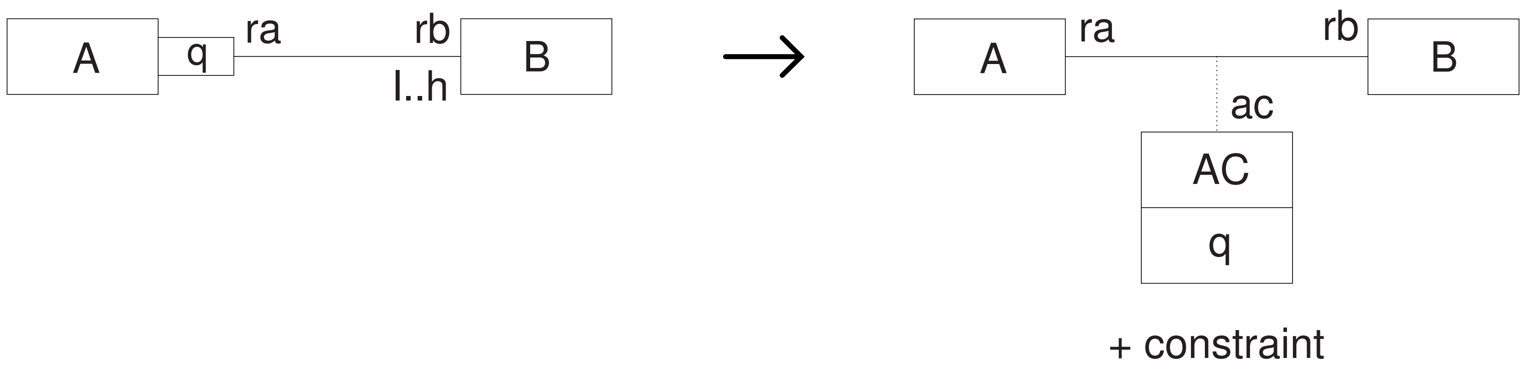


↓



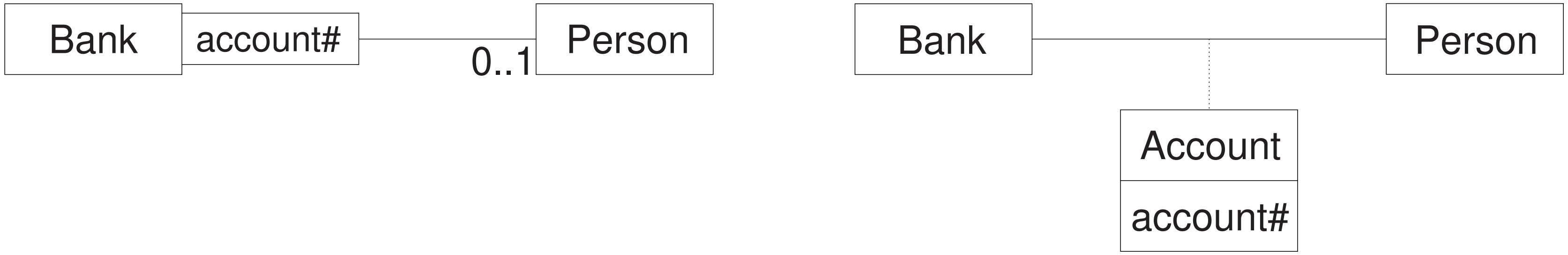

```
Examine->forAll( e, e’ |
  ( e.examinee=e’.examinee and e.subject=e’.subject and
    e.examiner=e’.examiner ) implies e=e’ )
```

Equivalence Rule for Qualifier



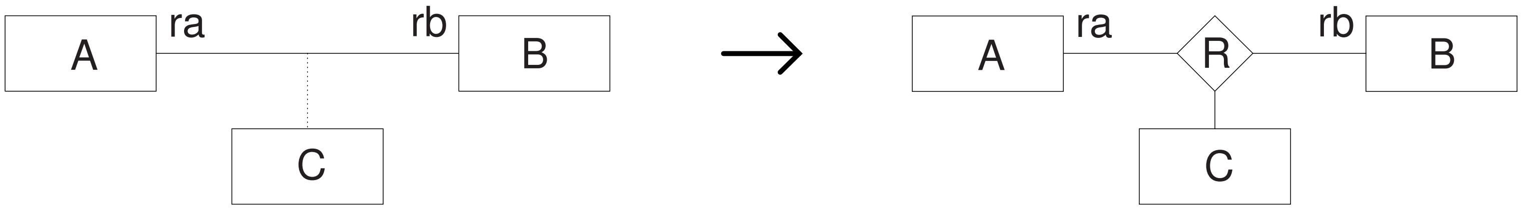

```
A->forAll( a |
  a.ac->forAll( ac |
    a.rb->select( b | b.ac->exists( ac’ | ac’.q=ac.q
                                and ac’.ra=a ) )->size >= l
    and
    a.rb->select( b | b.ac->exists( ac’ | ac’.q=ac.q
                                and ac’.ra=a ) )->size <= h ) )
```

Example for Qualifier




```
Bank->forAll( b |
  b.account->forAll( a |
    b.person->select( p |
      p.account->exists( a’ | a’.account#=a.account#
        and a’.bank=b ) )->size >= 0
    and
    b.person->select( p |
      p.account->exists( a’ | a’.account#=a.account#
        and a’.bank=b ) )->size <= 1 ) )
```

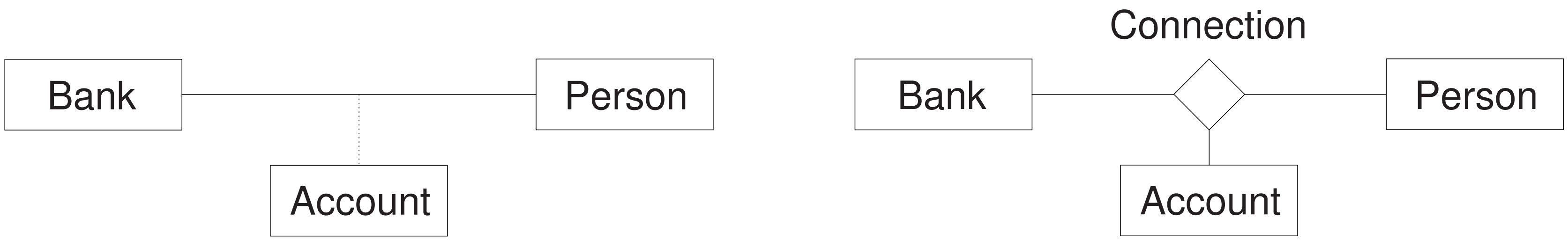
Equivalence Rule for Association Class



+ constraint

```
C->forAll( c |
  c.ra->size=1 and c.rb->size=1 and
  C->forAll( c' |
    ( c.ra=c'.ra and c.rb=c'.rb ) implies c=c'))
```

Example for Association Class




```
Account->forAll( a |
  a.bank->size=1 and a.person->size=1
  and
  Account->forAll( a' |
    ( a.bank=a'.bank and a.person=a'.person ) implies a=a'))
```

Problems with UML Semantics document

Problem: Verbal semantics in UML Semantics document leads to ambiguities

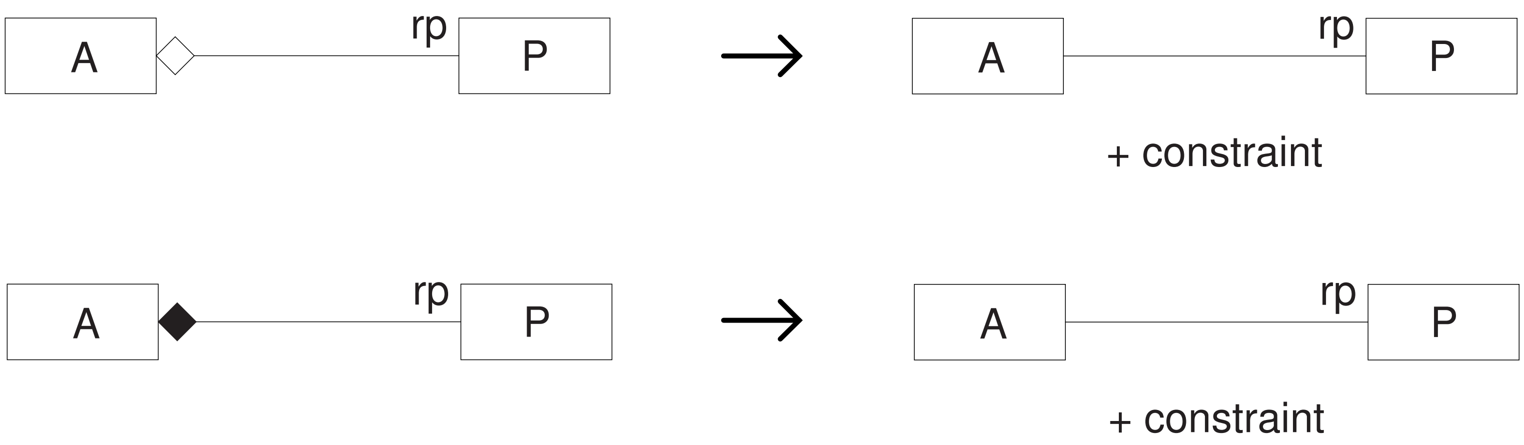
Example: Aggregation and composition

An association may represent an aggregation (i.e., a whole/part relationship). In this case, the association-end attached to the whole element is designated, and the other association-end of the association represents the parts of the aggregation. Only binary associations may be aggregations. Composite aggregation is a strong form of aggregation which requires that a part instance be included in at most one composite at a time, although the owner may be changed over time. Furthermore, a composite implies propagation semantics (i.e., some of the dynamic semantics of the whole is propagated to its parts). For example, if the whole is copied or deleted, then so are the parts as well. A shared aggregation denotes weak ownership (i.e., the part may be included in several aggregates) and its owner may also change over time. However, the semantics of a shared aggregation does not imply deletion of the parts when one of its containers is deleted. Both kinds of aggregations define a transitive, antisymmetric relationship (i.e., the instances form a directed, non-cyclic graph). Composition instances form a strict tree (or rather a forest).

Text above is subject of many papers!

Further observation: Text above also mentions syntactical requirements ("Only binary associations may be aggregations.")

Equivalence Rules for Aggregation and Composition (Part One)



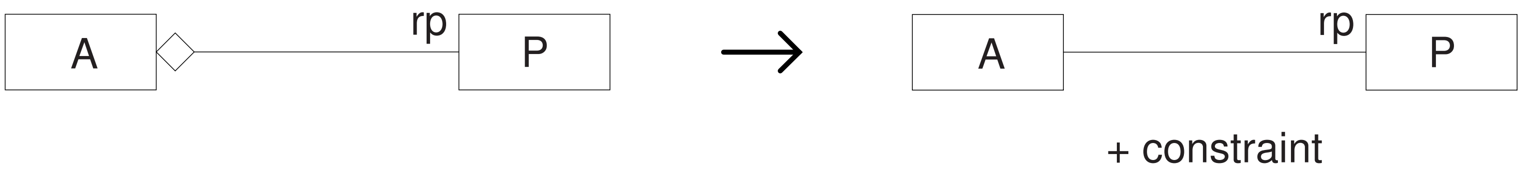
Existential Dependency for the Part

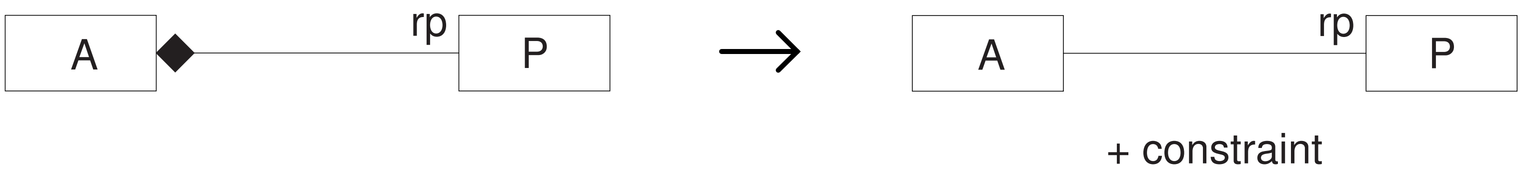
```
P->forAll( p |
  A->exists( a | a.rp->includes(p) ) )
```

Existential Dependency for the Aggregate

```
A->forAll( a |
  P->exists( p | a.rp->includes(p) ) )
```

Equivalence Rules for Aggregation and Composition (Part Two)

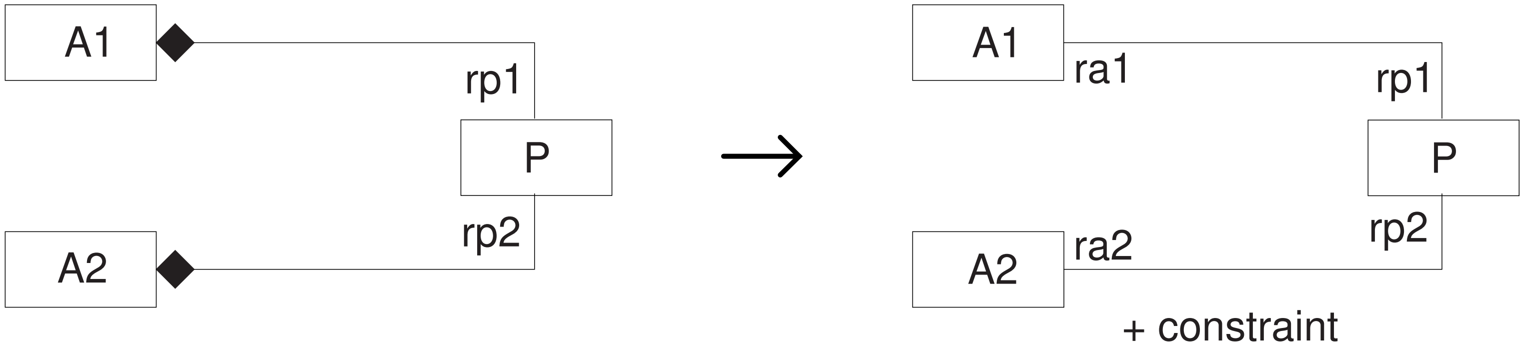




Weak Form of Forbidding Sharing

```
P->forAll( p |
  A->forAll( a, a' |
    ( a.rp->includes(p) and a'.rp->includes(p) )
    implies a=a'))
```

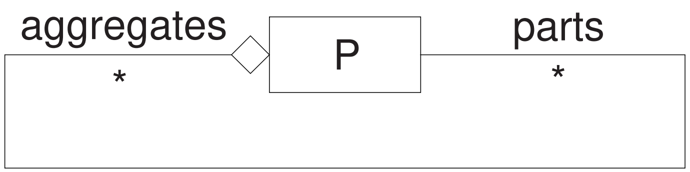
Equivalence Rules for Aggregation and Composition (Part Three)



Strong Form of Forbidding Sharing

```
P->forAll( p |
  A1->forAll( a1 |
    A2->forAll( a2 |
      ( a1.rp1->includes(p) and a2.rp2->includes(p) )
          implies a1=a2)))
```

Equivalence Rules for Aggregation and Composition (Part Four)




```
parts: P -> Set(P)
aggregates: P -> Set(P)
```

Now possible but unwanted: `p.parts->includes(p)`

```
partsClosure(p) =
  parts(p) + ‘parts(parts(p))’
  + ‘parts(parts(parts(p)))’ + ...
```

Forbidding Instance Reflexivity

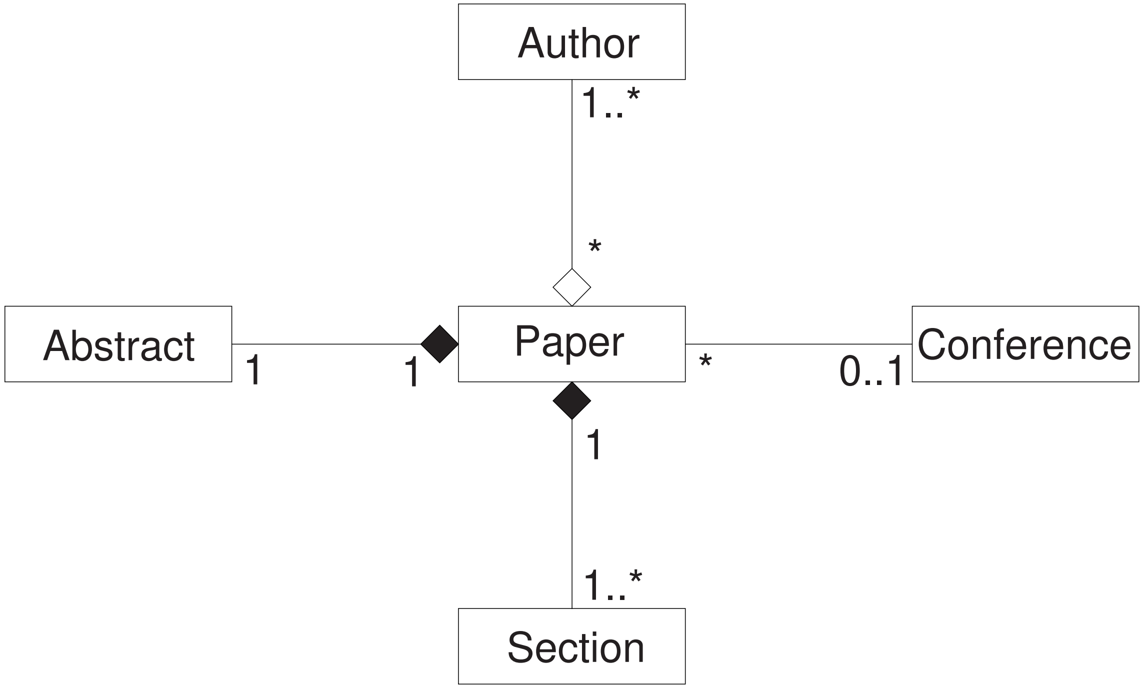
```
P.forAll( p |
  not( p.partsClosure->includes(p) ) )
```

Due to generalizations, reflexive associations may occur frequently!

Association vs. Aggregation vs. Composition

	Association	Aggregation	Composition
Existential Dependency for the Part	-	-	+
Existential Dependency for the Aggregate	-	-	-
Weak Form of Forbidding Sharing	-	-	+
Strong Form of Forbidding Sharing	-	-	+
Forbidding Instance Reflexivity	-	+	+

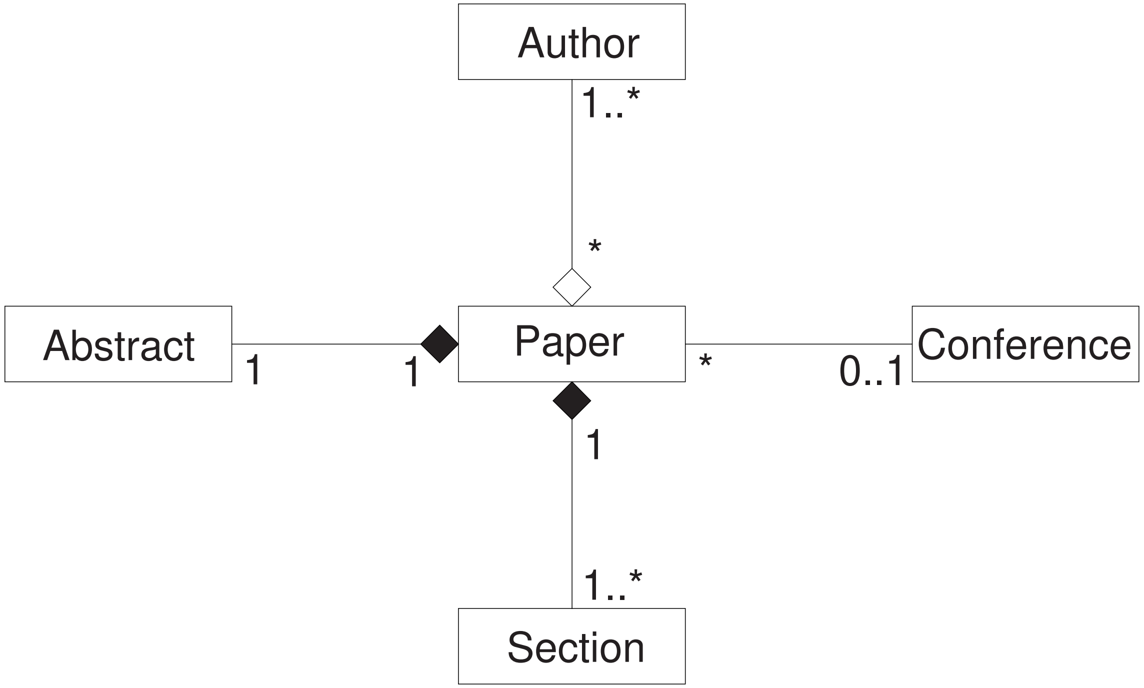
‘Paper’ Example – Existantial Dependency



Existential Dependency for the Part

```
Abstract->forAll( a |
  Paper->exists( p | p.abstract->includes(a) ) )
```

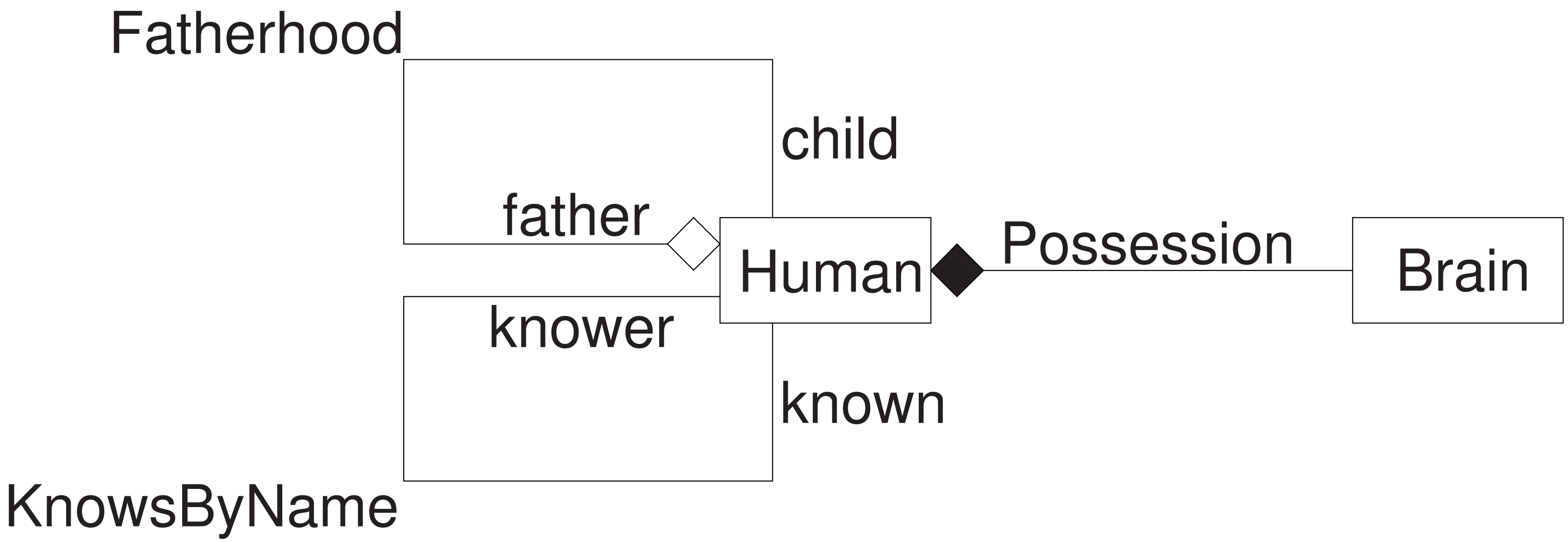
‘Paper’ Example – Sharing



Forbidding Sharing for Composition

```
Abstract->forAll( a |
  Paper->forAll( p, p’ |
    ( p.abstract->includes(a) and
      p’.abstract->includes(a) ) implies p=p’ ) )
```

‘Brain’ Example – Class Diagram



‘Brain’ Example – Induced Requirements

Existential Part Dependency for Composition

```
Brain->forAll( b |
  Human->exists( h | h.brain->includes(b) ) )
```

Forbidding Sharing for Composition

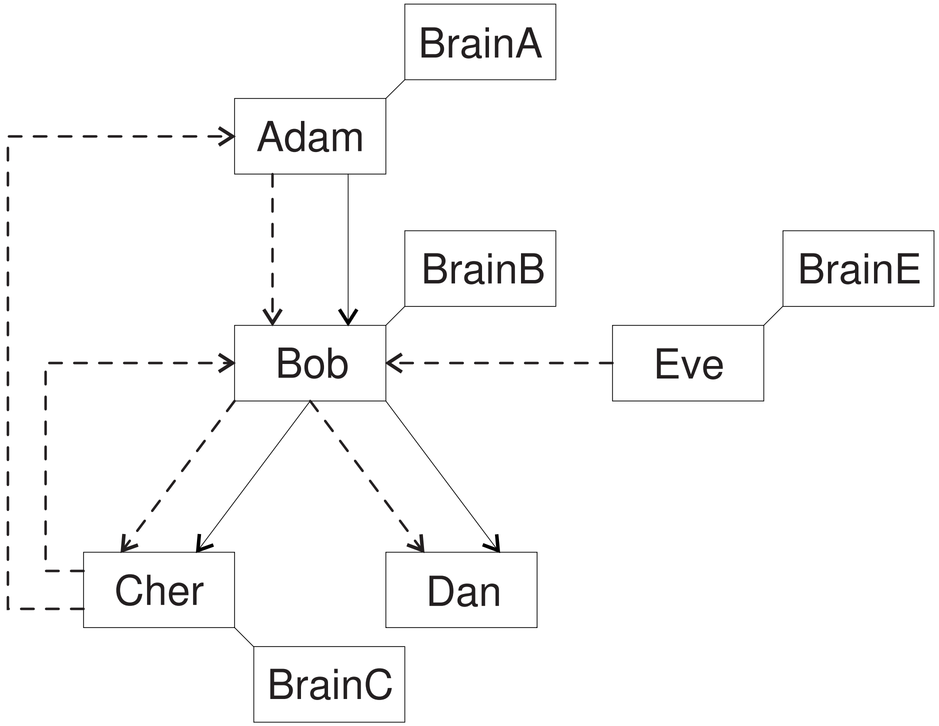
```
Brain->forAll( b |
  Human->forAll( h, h’ |
    (h.brain->includes(b) and
       h’.brain->includes(b)) implies h=h’ ) )
```

Forbidding Instance Reflexivity for Aggregation

```
child:  Human -> Set(Human)
childClosure:  Human -> Set(Human) = offsprings
Human.forAll( h | not( h.childClosure->includes(h) ) )
Human.forAll( h | not( h.offsprings->includes(h) ) )
```

No restriction for Association!

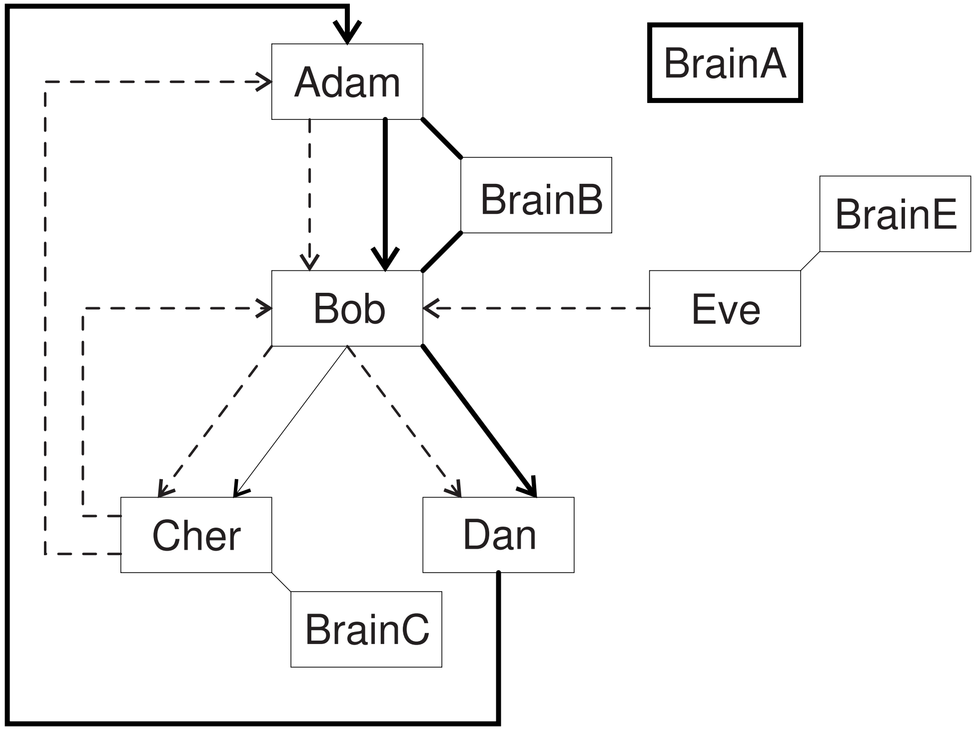
‘Brain’ Example – Valid Object Diagram



Fatherhood: solid arc, KnowsByName: dashed arc, Possession: undirected line

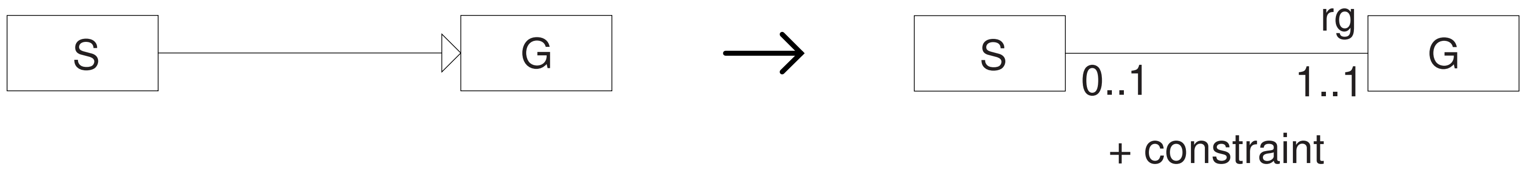
‘Brain’ Example – Invalid Object Diagram

Objects and links drawn fat contribute to invalidity



Fatherhood: solid arc, KnowsByName: dashed arc, Possession: undirected line

Equivalence Rule for Generalization

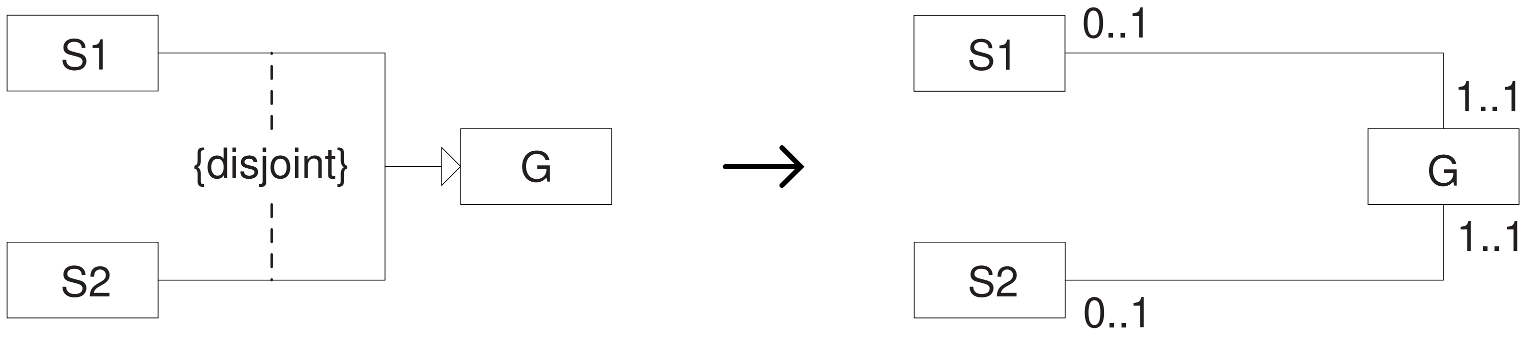



```
S->forAll( s, s' | s<>s' implies s.rg<>s'.rg )
```

or equivalently

```
S->forAll( s, s' | s.rg=s'.rg implies s=s')
```

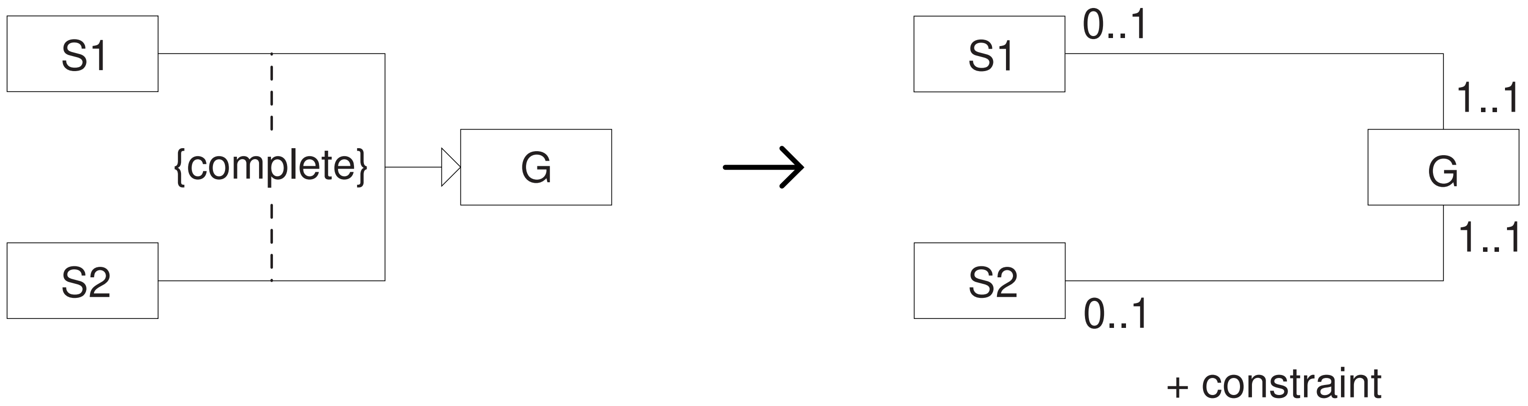
Equivalence Rule for Disjoint Generalization



+ constraint

```
S1->forAll( s1 | S2->forAll( s2 | s1.g<>s2.g ) )
```

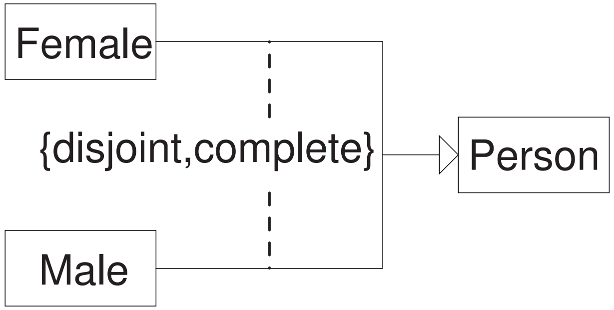
Equivalence Rule for Complete Generalization




```
G->forAll( g' |
  S1->exists( s1 | s1.g=g' ) or S2->exists( s2 | s2.g=g' ) )
```

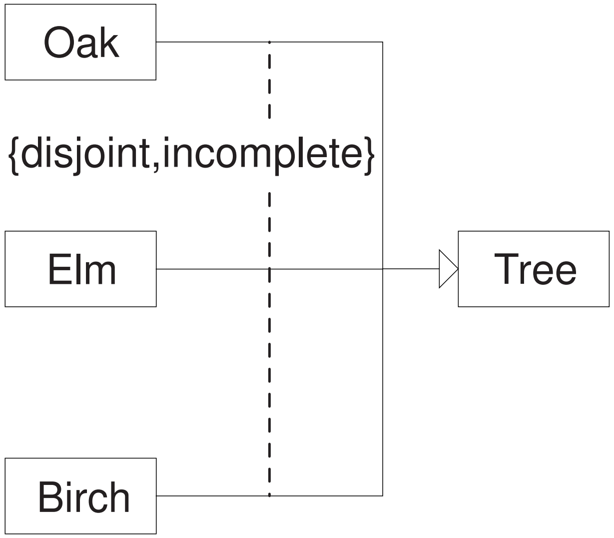
Examples for (1) Plain Generalization and (2) Disjoint and Complete Generalization


```
Car->forAll( c, c' | c<>c' implies c.vehicle<>c'.vehicle )
```



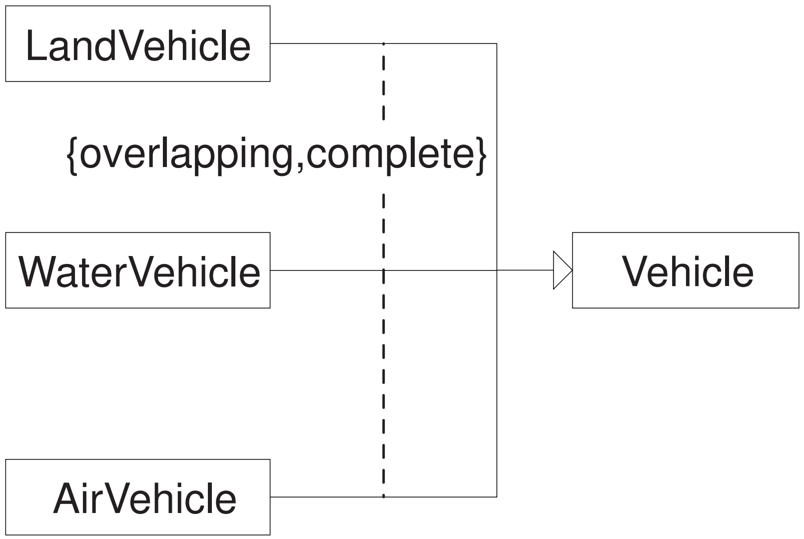

```
Female->forAll( f |
  Male->forAll( m | f.person<>m.person ) )
Person->forAll( p |
  Female->exists( f | f.person=p ) or
  Male->exists( m | m.person=p ) )
```

Example for Disjoint and Incomplete Generalization



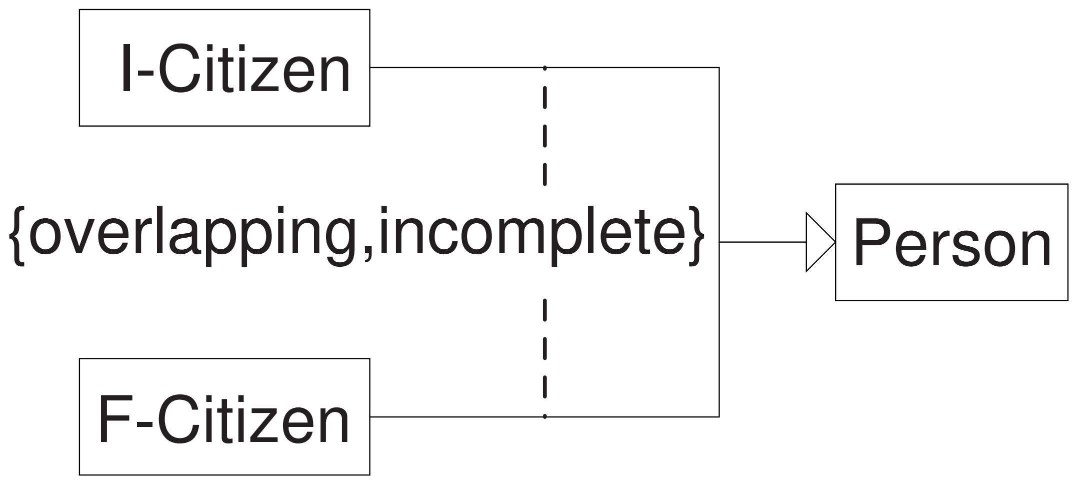

```
Oak->forAll( o |
  Elm->forAll( e |
    Birch->forAll( b |
      o.tree<>e.tree and
      o.tree<>b.tree and
      e.tree<>b.tree ) ) )
```

Example for Overlapping and Complete Generalization




```
Vehicle->forAll( v | LandVehicle->exists( l | l.vehicle=v )
         or
         WaterVehicle->exists( w | w.vehicle=v )
         or
         AirVehicle->exists( a | a.vehicle=v ) )
```

Example for Overlapping and Incomplete Generalization

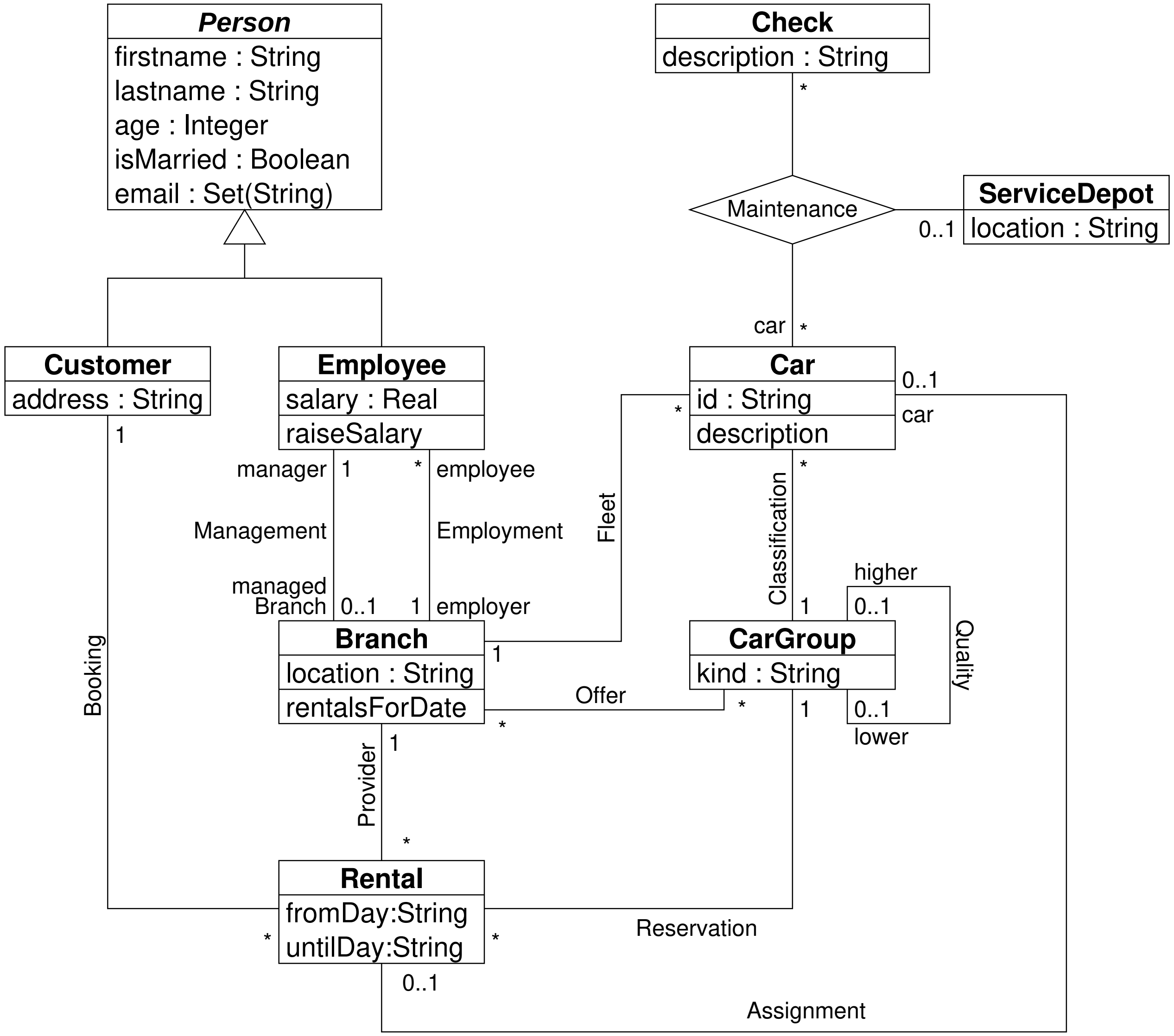


No constraints!

4.3 Basic Concepts in Class Diagrams

- To follow: formal definition of basic concepts in class diagrams based on set theory
- Formal abstract syntax for the these basic concepts
- Formal interpretation for these basic concepts

Example Class Diagram



Definition 1 (Names)
Let $\mathcal{N}$ be a set of Names.

Definition 2 (Basic Types)
We assume that there is a signature $\Sigma = (T, \Omega)$ with T being a set of basic type names, and Ω being a set of operations over types in T:

$$\omega : t_1 \times \cdots \times t_n \to t$$

Example 1 (Basic Types and Operations)
Predefined OCL Basic Types:

$$\{Integer, Real, Boolean, String\} \subset T$$

Operations in Ω *include, for example, the usual arithmetic operations* $+, -, *, /$*, etc. for integers, e.g.*

$$+ : \textit{Integer} \times \textit{Integer} \rightarrow \textit{Integer}$$

Definition 3 (Collection Types)
Collection types over the basic types are available for describing collections of values, for example, *Set*(*String*), *Bag*(*Integer*), *and* *Sequence*(*Real*), *Set*(*Sequence*(*Integer*)).

Example 2 (Collection Types)
In our example model, the attribute email of class Person has the type Set(String) to allow multiple email addresses for each person.

Definition 4 (Classes)
The set of classes is given by a finite set of names

$$\textsc{Class} \subset \mathcal{N}$$

Example 3 (Class Definition)
The example class diagram defines the following set of classes:

$$\textsc{Class} = \begin{array}{l} \{Person, Check, ServiceDepot, Customer, \\ Employee, Car, Branch, CarGroup, Rental\} \end{array}$$

Definition 5 (Attributes)

Let $t \in T$ be a basic or collection type. The attributes of a class $c \in$ CLASS are defined as a set ATT_c of signatures $a : t_c \to t$ where the attribute name a is an element of $\mathcal{N}$, and $t_c \in T$ is the type of class c.

Example 4 (Attribute)

$$\text{ATT}_{Person} = \begin{array}{c} \{firstname : Person \to String, \\ lastname : Person \to String, \\ age : Person \to Integer, \\ isMarried : Person \to Boolean, \\ email : Person \to Set(String)\} \end{array}$$

Definition 6 (Operations)
Let t and $t_1, \ldots, t_n \in T$ be types. Operations of a class $c \in$ CLASS with type $t_c \in T$ are defined by a set of OP_c of signatures $\omega : t_c \times t_1 \times \cdots \times t_n \to t$ with operation symbols $\omega \in \mathcal{N}$.

Example 5 (Operation)

$$\mathrm{OP}_{Car} = \{description : Car \to String\}$$

$$\mathrm{OP}_{Branch} = \{rentalsForDay : Branch \times String \to Set(Rental)\}$$
$$\mathrm{OP}_{Employee} = \{raiseSalary : Employee \times Real \to Real\}$$

Definition 7 (Assocations)
The set of associations is given by:

1. *a finite set of names* $\textsc{Assoc} \subset \mathcal{N}$
2. *a function*

$$\begin{aligned} associates: \quad \textsc{Assoc} &\rightarrow \textsc{Class}^{+} \\ as &\mapsto \langle c_1, \ldots, c_n \rangle \; with \; (n \geq 2) \end{aligned}$$

Example 6 (Association)

$$\textsf{Assoc} = \begin{array}{c} \{Booking, Management, Employment, \\ Fleet, Classification, Assignment, \\ Offer, Quality, Provider, Reservation\} \end{array}$$

associates(Assignment) = ⟨Rental, Car⟩
associates(Booking) = ⟨Rental, Customer⟩
associates(Classification) = ⟨CarGroup, Car⟩
associates(Employment) = ⟨Branch, Employee⟩
associates(Fleet) = ⟨Branch, Car⟩
associates(Maintenance) = ⟨ServiceDepot, Check, Car⟩
...

Definition 8 (Role names)

Let $as \in \textsc{Assoc}$ be an association with associates(as) = $\langle c_1, \ldots, c_n \rangle$.

Role names for an association are defined by a function

$$\begin{aligned} roles: \quad \textsc{Assoc} &\rightarrow \mathcal{N}^+ \\ as &\mapsto \langle r_1, \ldots, r_n \rangle \end{aligned}$$

where all role names must be distinct, i. e.,

$$\forall i, j \in \{1, \ldots, n\} : i \neq j \Longrightarrow r_i \neq r_j \quad .$$

If roles names are not specified in the class diagram, then implicit role names consisting of the class name with a lower case first letter are used

In case of ambiguities, for example, in the case where two associations between two classes are present, roles names must be specified in the class diagram

Example 7 (Role names)

$roles(Assignment) = \langle rental, car \rangle$

$roles(Booking) = \langle rental, customer \rangle$

$roles(Classification) = \langle carGroup, car \rangle$

$roles(Employment) = \langle employer, employee \rangle$

$roles(Fleet) = \langle branch, car \rangle$

$roles(Maintenance) = \langle serviceDepot, check, car \rangle$

$roles(Management) = \langle managedBranch, manager \rangle$

$roles(Offer) = \langle branch, carGroup \rangle$

$roles(Provider) = \langle rental, branch \rangle$

$roles(Quality) = \langle lower, higher \rangle$

$roles(Reservation) = \langle rental, carGroup \rangle$

Definition 9 (Multiplicities)

Let $as \in \textsc{Assoc}$ *be an association with associates(as) =* $\langle c_1, \ldots, c_n \rangle$*.*
The function multiplicities(as) = $\langle M_1, \ldots, M_n \rangle$ *assigns each class* c_i *participating in the association a non-empty set* $M_i \subset \mathbb{N}_0$ *with* $M_i \neq \{0\}$ *for all* $1 \leq i \leq n$*.*

Example 8 (Multiplicities)

For example, the multiplicities of the Maintenance association are defined as multiplicities(Maintenance) = $\langle \{0, 1\}, \mathbb{N}_0, \mathbb{N}_0 \rangle$*.*

Definition 10 (Generalization hierarchy)
A generalization hierarchy $\prec$ is a partial order (transitive, non-reflexive relation) on the set of classes $\textsc{Class}$*.*

Example 9 (Generalization hierarchy)
The example class diagram contains two generalizations. Person is the parent class of Customer and Employee.

$\prec = \{(\mathit{Customer}, \mathit{Person}), (\mathit{Employee}, \mathit{Person})\}$

Definition 11 (Parents)

$$
\begin{aligned}
parents: \quad \mathsf{C}\textsf{LASS} &\rightarrow \mathcal{P}(\mathsf{C}\textsf{LASS}) \\
c &\mapsto \{c' \mid c' \in \mathsf{C}\textsf{LASS} \wedge c \prec c'\}
\end{aligned}
$$

Definition 12 (Full set of attributes)

$$
\mathsf{ATT}^*_c = \mathsf{ATT}_c \cup \bigcup_{c' \in parents(c)} \mathsf{ATT}_{c'}
$$

Definition 13 (Inherited user-defined operations)

$$\mathrm{OP}_c^* = \mathrm{OP}_c \; \cup \bigcup_{c' \in parents(c)} \mathrm{OP}_{c'}$$

Definition 14 (Navigable role names)

$$\mathit{navends}^*(c) = \mathit{navends}(c) \; \cup \bigcup_{c' \in \mathit{parents}(c)} \mathit{navends}(c')$$

Definition 15 (Full descriptor of a class)
The full descriptor of a class $c \in \textsc{Class}$ *is a structure* $\mathsf{FD}_c = (\textsc{Att}^*_c, \textsc{Op}^*_c, \textit{navends}^*(c))$ *containing all attributes, user-defined operations, and navigable role names defined for the class and all of its parents.*

Additional well-formedness rules

1. In a full class descriptor, attributes are defined in exactly one class
2. In a full class descriptor, an operation may only be defined once
3. In a full class descriptor, role names are defined exactly in one class
4. Attribute names and role names must not conflict

Overview on Formal Syntax

Definition 16 (Syntax of object models)
The syntax of an object model is a structure

$$M = (\textsc{Class}, \textsc{Att}_c, \textsc{Op}_c, \textsc{Assoc}, \textit{associates}, \textit{roles}, \textit{multiplicities}, \prec)$$

where

1. $\textsc{Class}$ *is a set of classes.*

2. $\textsc{Att}_c$ *is a set of operation signatures for functions mapping an object of class* c *to an associated attribute value.*

3. $\textsc{Op}_c$ *is a set of signatures for user-defined operations of a class* c.

4. ASSOC *is a set of association names.*
   (a) *associates is a function mapping each association name to a list of participating classes.*
   (b) *roles is a function assigning each end of an association a role name.*
   (c) *multiplicities is a function assigning each end of an association a multiplicity specification.*

5. $\prec$ *is a partial order on* CLASS *reflecting the generalization hierarchy of classes.*

Definition 17 (Object Identifiers)

1. *The set of object identifiers of a class* $c \in \textsf{CLASS}$ *is defined by an infinite set oid(c) =* $\{\underline{c}_1, \underline{c}_2, \ldots\}$.

2. *The domain of a class* $c \in \textsf{CLASS}$ *is defined as* $I_{\textsf{CLASS}}(c) = \bigcup\{\textit{oid(c')} \mid\ c' \in \textsf{CLASS} \wedge c' \preceq c\}$.

Example 10 (Object Identifiers)

I(Branch) = $\{\ \underline{Branch}_1, \underline{Branch}_2, \ldots\}$

I(Customer) = $\{\ \underline{cu}_1, \underline{cu}_2, \ldots\}$

(Class names may be abbreviated)

I(Employee) = $\{\ \underline{e}_1, \underline{e}_2, \ldots\}$

I(Person) = $\{\ \underline{p}_1, \underline{p}_2, \ldots\}$ $\cup$ *I(Customer)* $\cup$ *I(Employee)*

Generalization

The above definition implies that a generalization hierarchy induces a subset relation on the semantic domain of classes.

$$\forall c_1, c_2 \in \textsf{CLASS} : c_1 \prec c_2 \Longrightarrow I(c_1) \subset I(c_2)\ .$$

Example 11 (Inheritance)

$I(Customer) \subset I(Person)$

$I(Employee) \subset I(Person)$

Definition 18 (Links)
Each association $as \in \textsf{Assoc}$ *with associates(as) =* $\langle c_1, \ldots, c_n \rangle$ *is interpreted as the Cartesian product of the sets of object identifiers of the participating classes:*

$$I_{\textsf{Assoc}}(as) = I_{\textsf{Class}}(c_1) \times \ldots \times I_{\textsf{Class}}(c_n)$$

A link denoting a connection between objects is an element $l_{as} \in I_{\textsf{Assoc}}(as)$*.*

Example 12 (Links)

$I(Assignment) = I(Rental) \times I(Car)$

$I(Maintenance) = I(ServiceDepot) \times I(Check) \times I(Car)$

$I(Quality) = I(CarGroup) \times I(CarGroup)$

Definition 19 (System State)

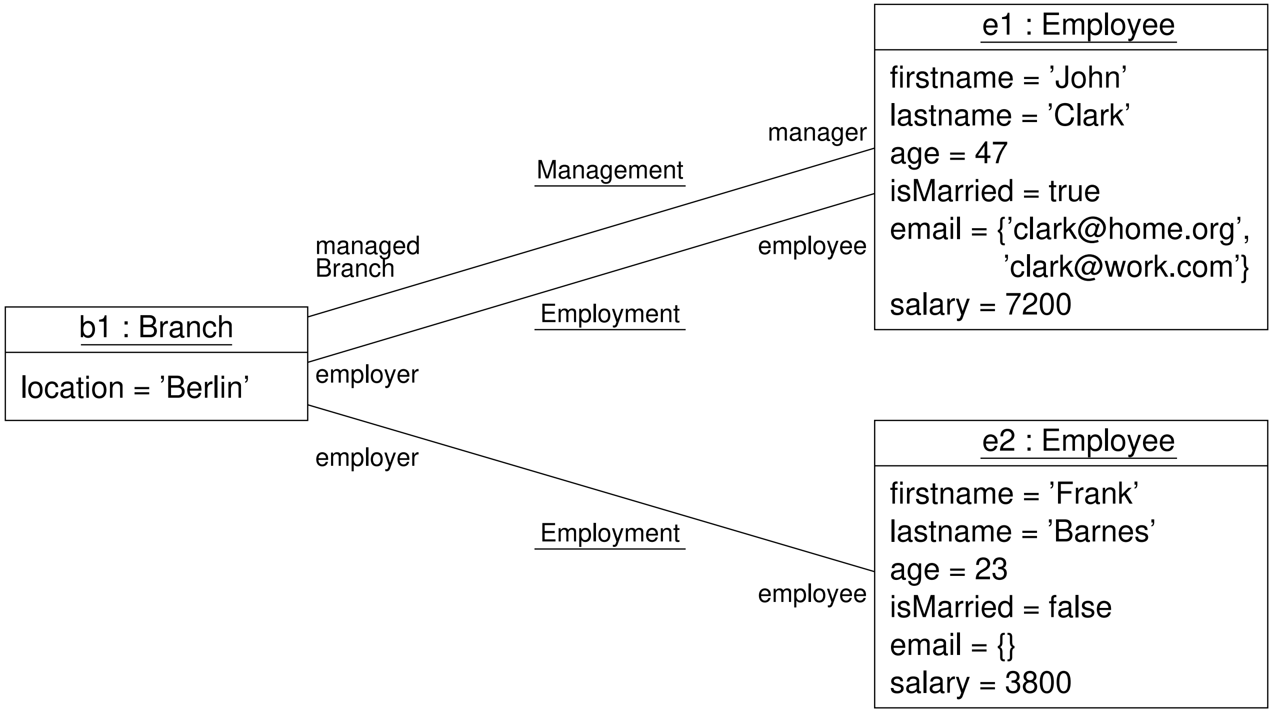
A system state for a model $\mathcal{M}$ is a structure $\sigma(\mathcal{M}) = (\sigma_{\text{CLASS}}, \sigma_{\text{ATT}}, \sigma_{\text{ASSOC}})$.

1. *Finite sets* $\sigma_{\text{CLASS}}(c)$ *contain all objects of a class* $c \in \text{CLASS}$ *existing in the system state:*
   $\sigma_{\text{CLASS}}(c) \subset$ *oid(c).*

2. *Functions* σ_{ATT} *assign attribute values to each object:*
   $\sigma_{\text{ATT}}(a) : \sigma_{\text{CLASS}}(c) \rightarrow I(t)$ *for each* $a : t_c \rightarrow t \in \text{ATT}^*_c$.

3. *Finite sets $\sigma_{\textsf{ASSOC}}$ contain links connecting objects. For each $as \in$ ASSOC: $\sigma_{\textsf{ASSOC}}(as) \subset I_{\textsf{ASSOC}}(as)$. A link set must satisfy all multiplicity specifications defined for an association (the function $\pi_i(l)$ projects the ith component of a tuple or list l, whereas the function $\bar{\pi}_i(l)$ projects all but the ith component):*
$\forall i \in \{1, \ldots, n\}, \forall l \in \sigma_{\textsf{ASSOC}}(as) :$
$|\{l' \mid \; l' \in \sigma_{\textsf{ASSOC}}(as) \wedge (\bar{\pi}_i(l') = \bar{\pi}_i(l))\}|$
$\in \pi_i(multiplicities(() = as))$

Example 13 (System State)



$$\sigma(Employee) = \{\underline{e}_1, \underline{e}_2\}$$
$$\sigma(Branch) = \{\underline{b}_1\}$$
$$\sigma(Employment) = \{(\underline{b}_1, \underline{e}_1), (\underline{b}_1, \underline{e}_2)\}$$
$$\sigma(Management) = \{(\underline{b}_1, \underline{e}_1)\}$$

Definition 20 (Interpretation of object models)

The interpretation of an object model $\mathcal{M}$ is the set of all possible system states $\sigma(\mathcal{M})$.

Syntax and Semantics as Meta-model