Software Engineering I (02161)

Week 4

Assoc. Prof. Hubert Baumeister

Informatics and Mathematical Modelling
Technical University of Denmark

Spring 2011

Recap: Software Testing

- Two purposes: Validation testing and defect testing
- Systematic tests: Design your tests by looking at all possible input classes
  - Black box test: test cases are generated from the problem description
  - White box test: test cases are generated by looking into the implementation
- Manual vs automatic tests: automatic tests are preferred
- Acceptance tests: tests defined with the customer to define when a user story / use case is implemented: Best automatic
- Test driven development:
  - → repeat: 1) choose new functionality 2) Define test 3) Implement functionality 4) Refactor
- Refactoring: improve the desing of the code without adding new functionality
  - Essential for the design process in XP

Class Diagram I

Class diagrams can be used for different purposes

1. to give an overview over the domain concepts
   - as part of the requirements analysis (e.g. a graphical form representation supplementing the glossary)
2. to give an overview over the system
   - as part of the design of the system
3. to give an overview over the systems implementation
4. ...

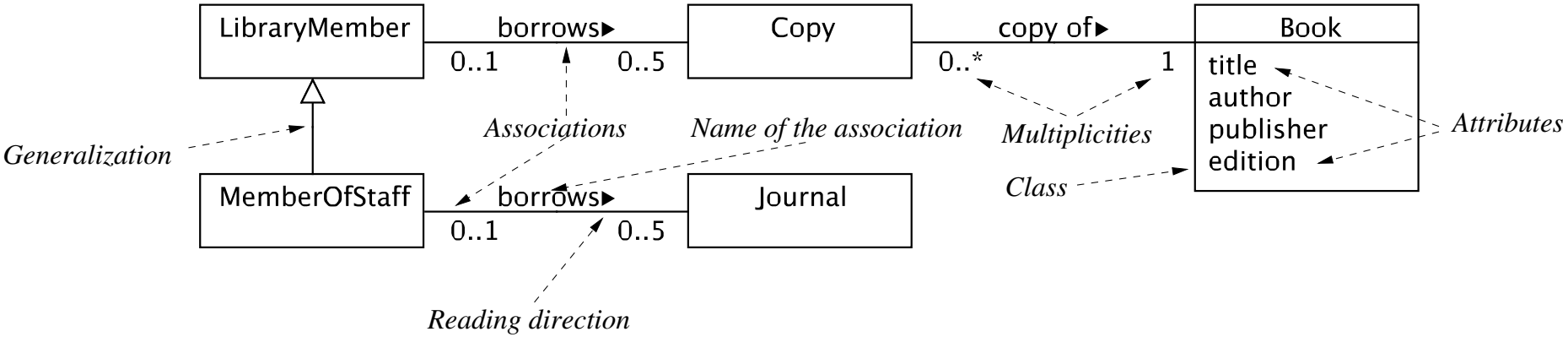
Level of detail of a class description depends on the purpose of the class diagram

Domain Modelling : typically low level of detail

⋮

Implementation : typically high level of detail

Class Diagram Example



Basic concepts of class diagrams

- Classes with attributes and operations
- Associations with multiplicities and possibly navigability
- Generalization of classes (corresponds in principle to subclassing in Java)

Why class diagrams?

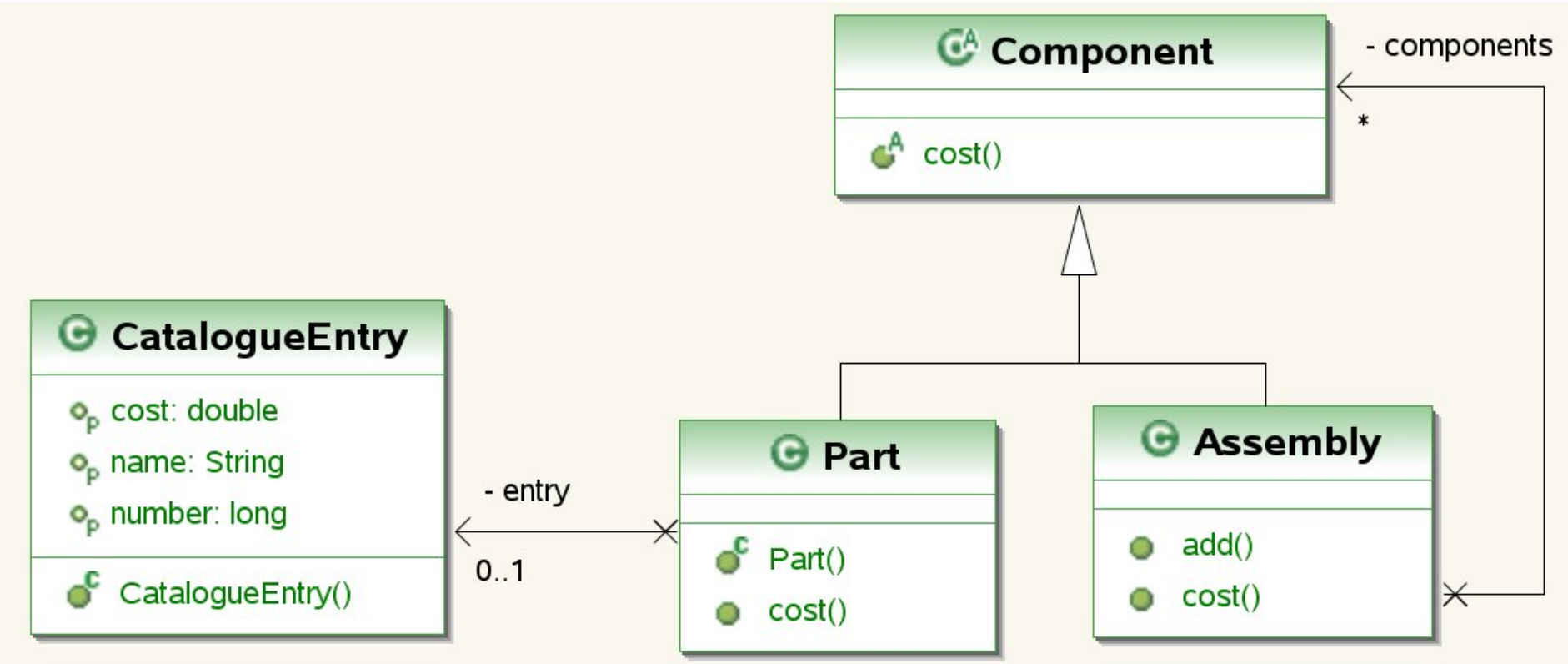
- What is the structure of this small Java program?

```
public class Assembly
             extends Component {
 public double cost() {    }
 public void add(Component c) {}
 private Collection<Component>
      components;
}

public class CatalogueEntry {
 private String name = "";
 public String getName() {}
 private long number;
 public long getNumber() {}
 private double cost;
 public double getCost() {}
}
```

```
public abstract class Component {
 public abstract double cost();
}

public class Part extends Component {
 private CatalogueEntry entry;
 public CatalogueEntry getEntry() {}
 public double cost(){}
 public Part(CatalogueEntry entry){}
```

Why class diagrams? (cont.)

- Same information as in the Java program
- However, the structure is visible immediately and better to understand



Class Diagrams and Program Code

- Class Diagrams were indented as a means to graphically show object-oriented programs
- As a consequence: Class Diagrams allow one to model all the structural features of a Java class
  - e.g. classes, (static) methods, (static) fields, inheritance, ...
- However, class diagrams are more abstract than programs
  - Concepts of associations, aggregation/composition, ...

→ Modelling with class diagrams is more expressive and abstract than programming in Java

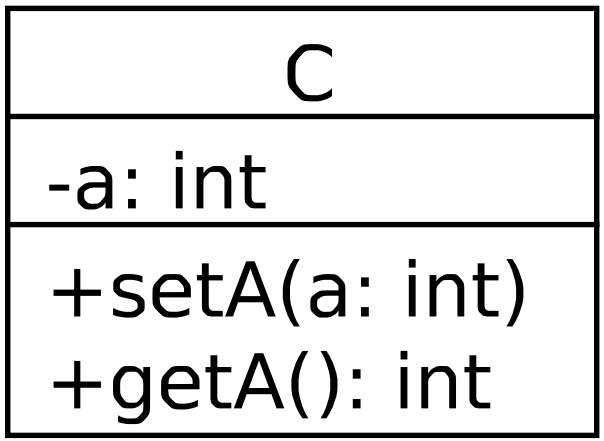
→ It is important to learn who these abstract, object-oriented concepts embodied in class diagrams are implemented in Java programs

  → Improves your object-oriented design skills

Example

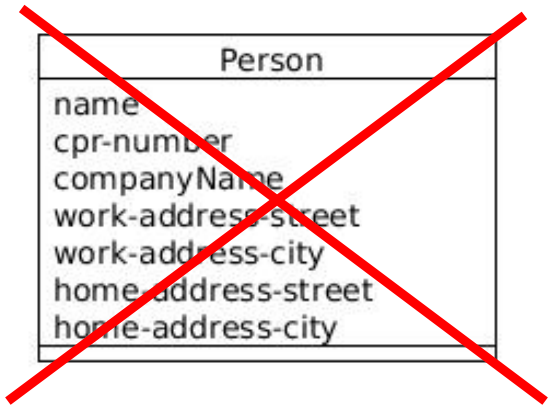
What is the class diagram for the following program?

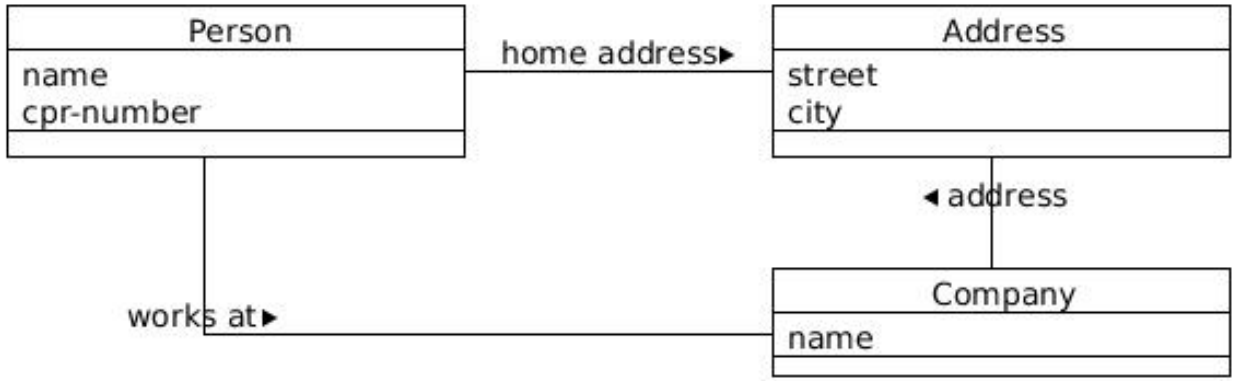
```
public class C {
  private int a;
  public int getA() { return a; }
  public void setA(int a) { this.a = a; }
}
```



Classes

- A class describes a collection of objects that have a common characteristics regarding
  - state (attributes)
  - behaviour (operations)
  - relations to other classes (associations and generalisations)
- A class ideally should represent only one concept
  - All the attributes should related to that concept
  - All the operations should make sense for that concept



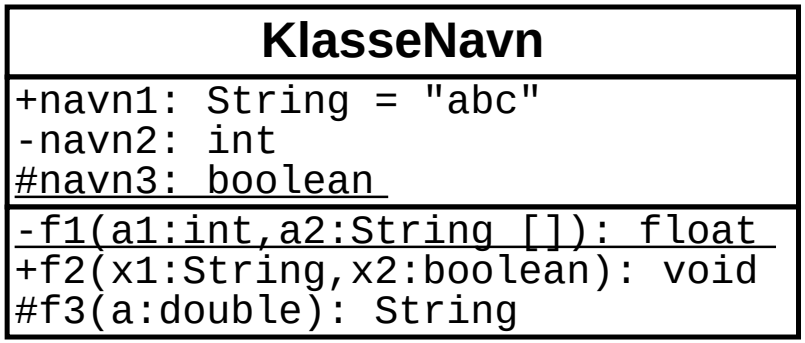


General correspondence between Classes and Programs

'-' : private

'+' : public

'#': protected



Klassens navn

Attributter

Operationer

'navn3' og 'f1' er statiske størrelser

```
public class KlasseNavn
{
   private String navn1 = "abc";
   private int navn2;
   protected static boolean navn3;

   private static float f1(int a1, String[] a2) { ... }
   public void f2(String x1, boolean x2) { ... }
   protected String f3(double a) { ... }
   public String getNavn1(); {...}
   public void setNavn1(String n) {...}
}
```

Generalisation

- Each instance of the specific classifier is also an indirect instance of the general classifier

→ Substitutability

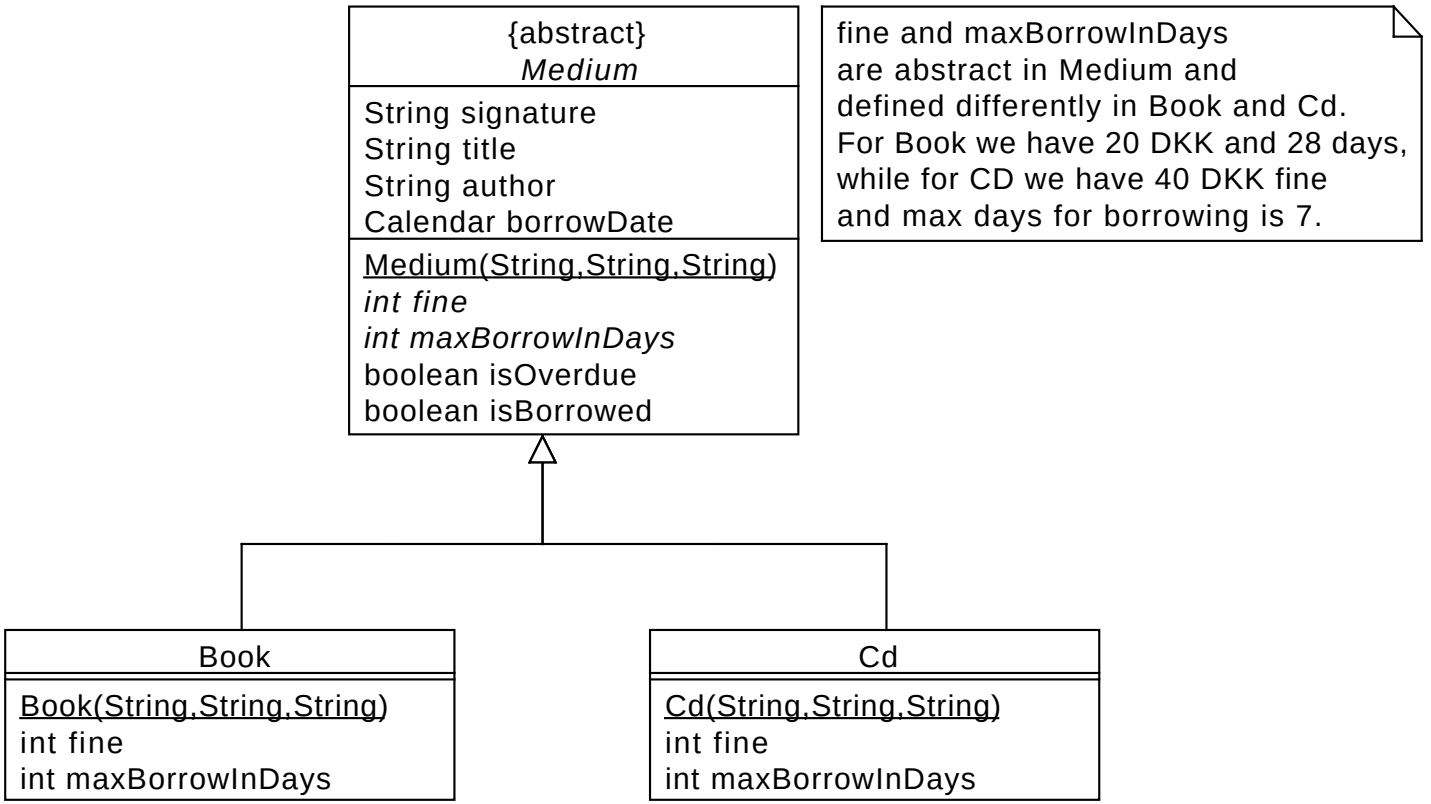
  - One can use objects of the subclass instead of objects of the superclass

→ this is called the Liskov-Wing Substitution Principle

  "*If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g., correctness).*"

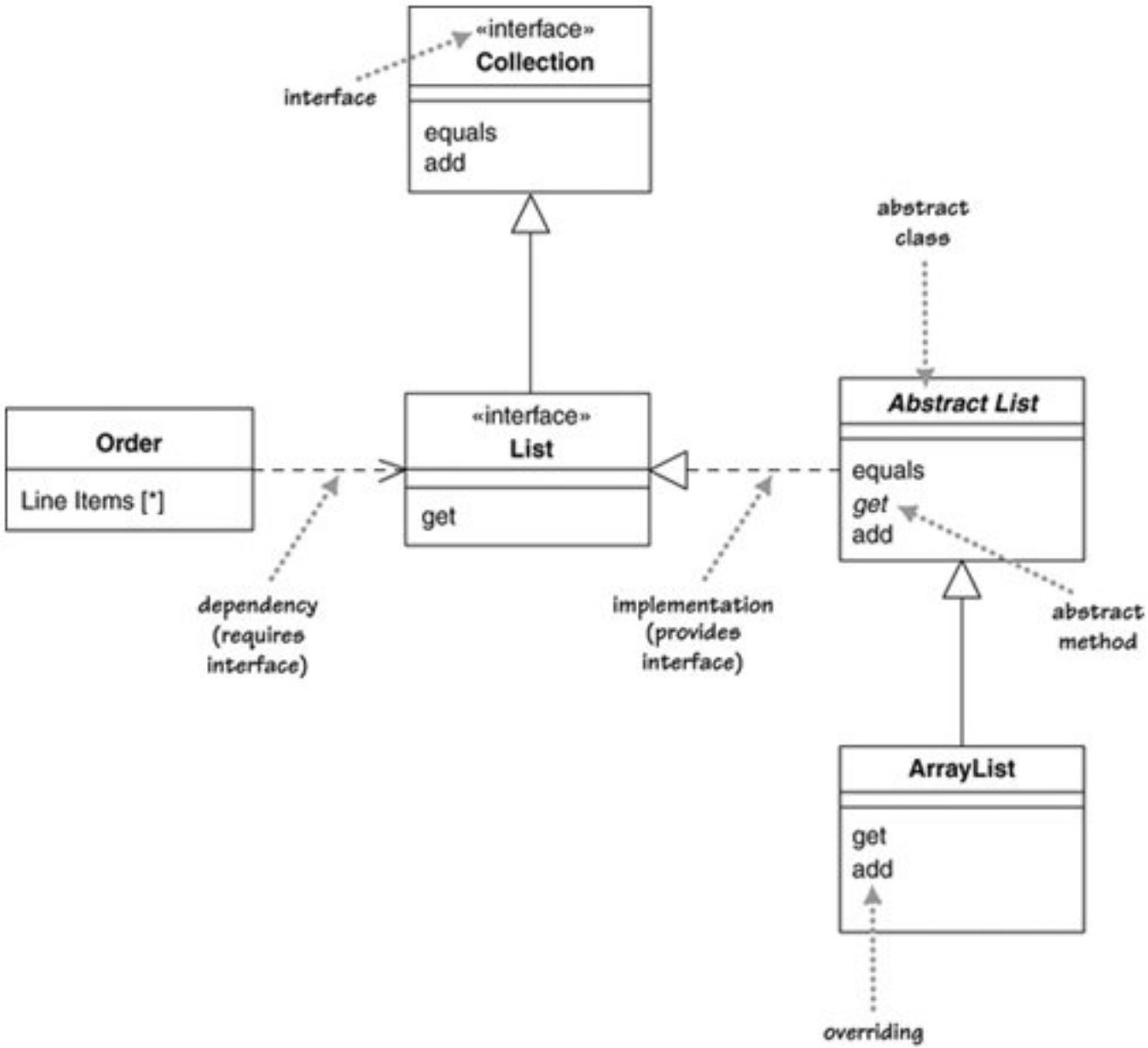
- Achieved by inheritance
  - The structure (i.e. fields and operations) of the superclass are inherited by the subclass

Generalisation Example



Side remark: Constructors are not inherited

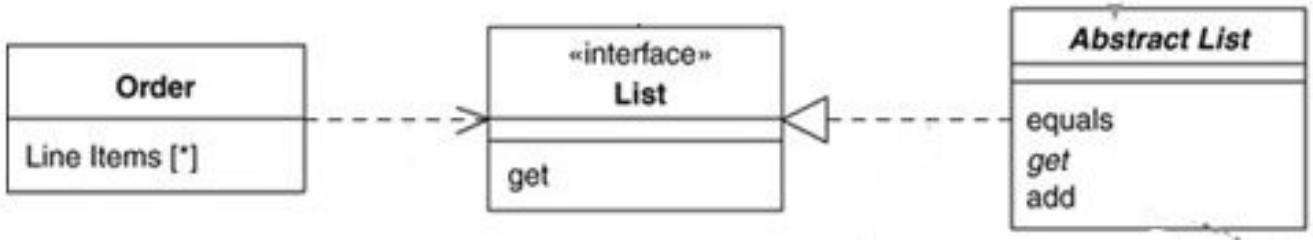
Interfaces



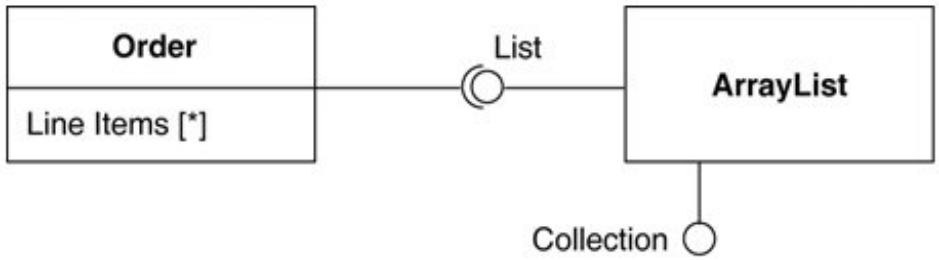
- Correspond to interfaces in Java
- Define a contract that a class that realizes the interface has to fulfil
- Interface descriptions in UML can contain operations and attributes

Requires and implements interface dependency

- As dependencies



- Lollipop notation



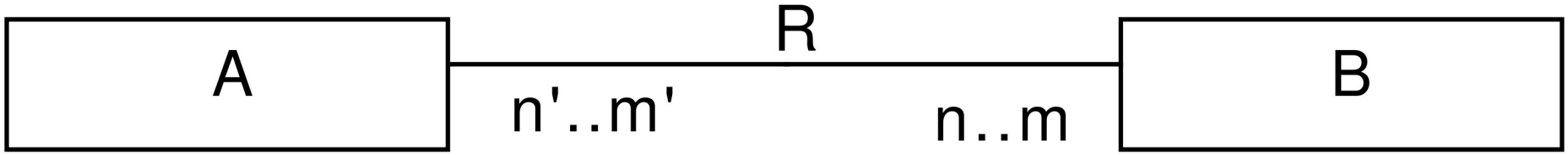
Associatons between classes I

Example: Persons and their employers:

- every person is associated to an employer (company/firma)
- every company has 0, 1, or more ('*') employees (persons)

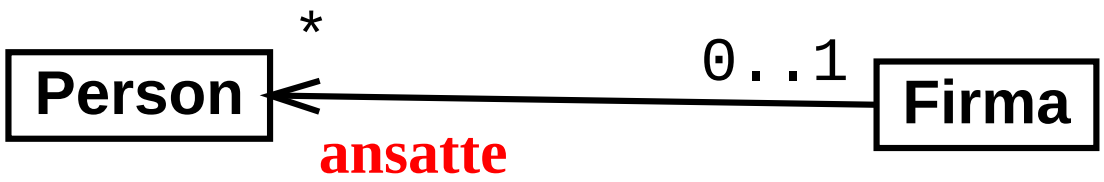
Associations between classes II

- A association between classes means, that the objects belonging to the two classes have knowledge of each other



- Mathematically an association describes a relation $R \subseteq A \times B$
- $R(a, \_) = \{b \mid (a, b) \in R\}$ and $R(\_, b) = \{a \mid (a, b) \in R\}$
- Multiplicity
  - $\forall a \in A : n \leq |R(a, \_)| \leq m$ and $\forall b \in B : n' \leq |R(\_, b)| \leq m'$
  - Ex: 0..1, 1, *, ...

Associations between classes III

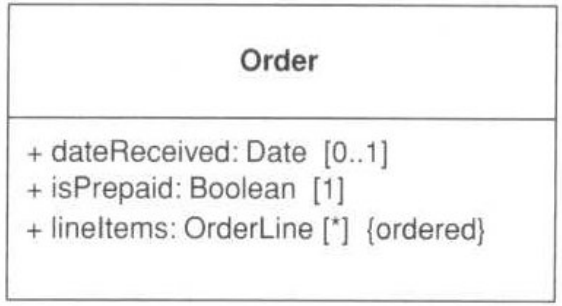


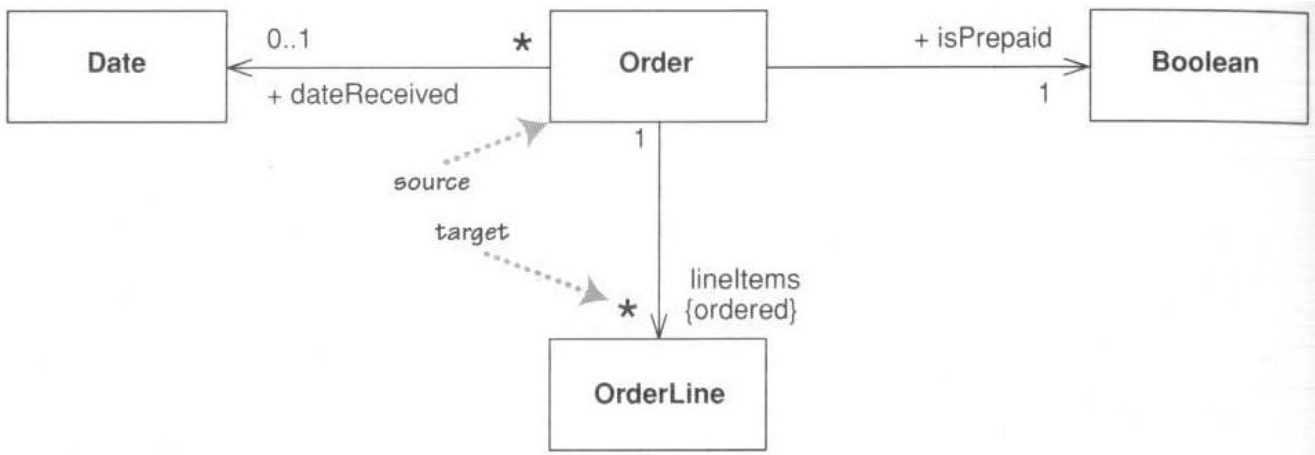
- a *role* (here **ansatte**) describes objects (here persons) at the end of an association, seen from the objects belonging to the classes at the opposite end of the (here company)
- default role name: name of the associated class (e.g. person)
- in an implementation a role name is typically a variable. For example:

```
public class Firma
{    ....
     private Collection<Person> ansatte;
     ....
}
```

Attributes and Associations

- There is in principle no distinction between attributes and associations
- Associations can be drawn as attributes and vice versa





Attributes versus Associations

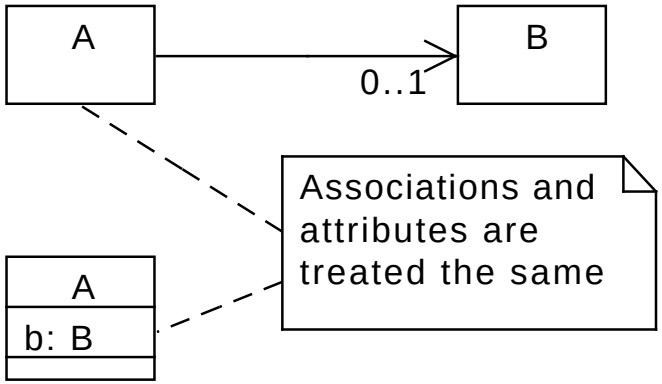
When to use attributes and when to use associations?

- Associations
  - When the target class of an association is shown in the diagram
  - The target class of an association is a major class of the model
    - e.g. Part, Assembly, Component, ...
- Attributes
  - When the target class of an associations is not shown in the diagram
  - With datatypes / Value objects
    - Datatypes consists of a set of values and set of operations on the values
    - In contrast to classes are datatypes stateless
    - e.g. int, boolean, String ...
  - Library classes
- However, final choice depends on what one wants to express with the diagram
  - E.g. Is it important to show a relationship to another class?

Implementing Associations: Cardinality 0..1

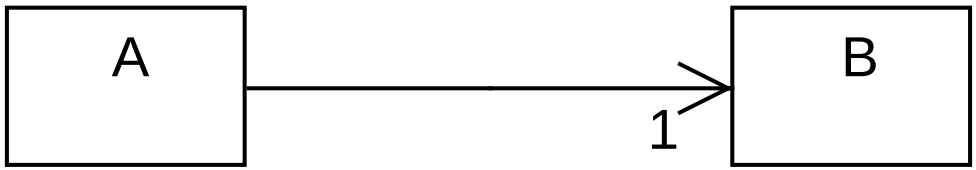
- Associations can be navigable in one direction or two directions (a.k.a. bidirectional association)

→ this means that knowledge can be only on one side or on two sides




```
public class A {

   private B b;

   public B getB() { return b; }
   public void setB(B b) { this.b = b; }
}
```

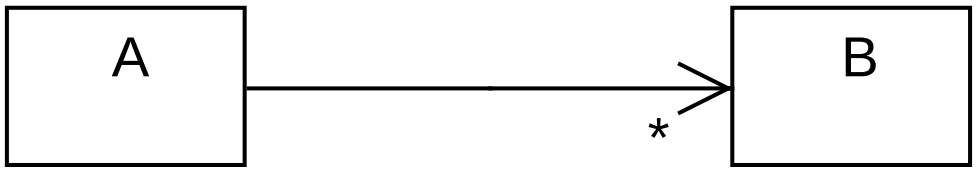
Implementing Associations: Cardinality 1



- When requesing the field b, the implementation needs to ensure that always a B is returned

```
public class A {

   private B b = new B(); // 1. possibility
   public A(B b) { this.b = b;} // 2. possibility
   public B getB() {  // 3. possibility
     if (b == null) {b = computeB();}
     return b;
   }
   public void setB(B b) { if (b != null) {this.b = b;} }
}
```

Implementing Associations: Cardinality *



- Uses an attribute of type Collection

```
public class A {

    private Collection<B> bs = new java.util.ArrayList<B>();

    public void addB(B b) { bs.add(b);}
    public void contains(B b) { return bs.contains(b); }
    public void removeB(B b) { bs.remove(b); }
}
```

- If the multiplicity is >1, one adds a plural s to the role name: b → bs

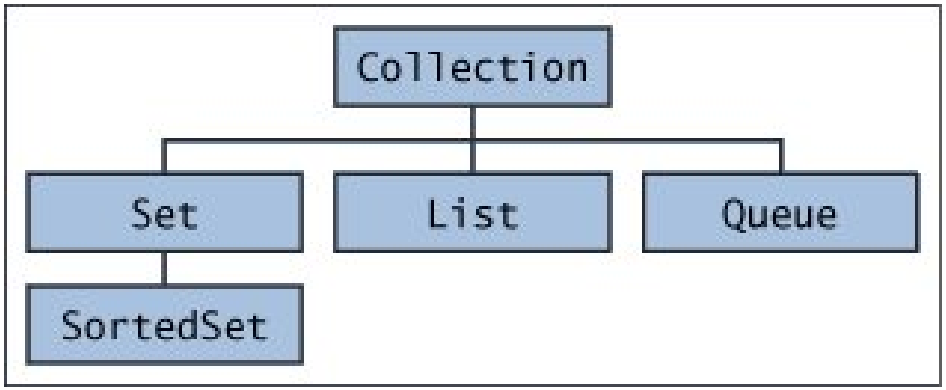
Interface *Collection<E>*

Operation	Description
`boolean add(E e)`	returns false if e is in the collection
`boolean remove(E e)`	returns true if e is in the collection
`boolean contains(E e)`	returns true if e is in the collection
`Iterator<E> iterator()`	allows to iterate over the collection
`int size()`	number of elements

Example of iterating over a collection

```
Collection<String> names = new HashSet<String>() ;
names.add("Hans");
...
for (String name : names) {
   // Do something with name, e.g.
   System.out.println(name);
}
```

Hierarchy of collection interfaces

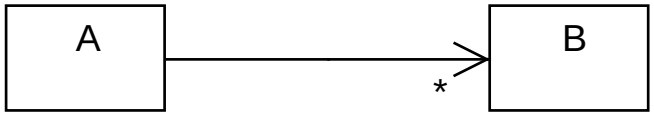


- Collection: Superinterface of all collections
- Set: Order is irrelevant; no duplicates allowed
- List: Order is relevant; duplicates are allowed; allows positional access in addition to Collection operations
  - E get(int index);
  - E set(int index, E element);
  - void add(int index, E element);
  - E remove(int index);

Collection and their subinterfaces cannot be instantiated directly

→ One needs to use concrete implementation classes like HashSet or ArrayList

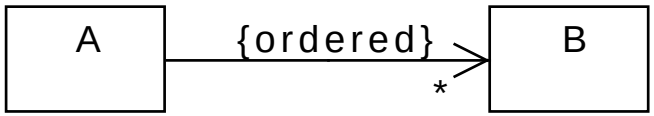
Implementing Associations: Cardinality *



With UML the default for n-ary associations is: unordered and no duplicates

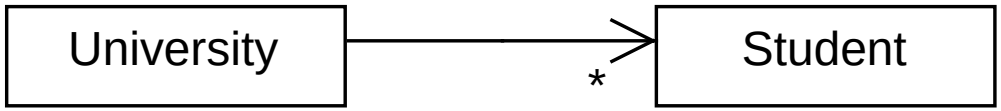
```
public class A {
   private Set<B> bs = new HashSet<B>();
   ...
}
```

If one wants the collection to be ordered with duplicates one has to use {ordered}




```
public class A {
   private List<B> bs = new ArrayList<B>();
   ...
}
```

Encapsulation problem: getBs




```
University dtu = new University("DTU");
..
Student hans = new Student("Hans");
Collection<Student> students = dtu.getStudents();
```

- Access to the association using getBs (e.g. getStudents()) poses encapsulation problems: Why?
- A client of A can change the association without A knowing it!

```
Student hans = new Student("Hans");
students.add(hans);
students.remove(ole);
...
```

- Solution: getStudents should return an unmodifiable collection

```
   public void Collection<Student> getStudents() {
      students = Collections.unmodifiableCollection();
}
```

Encapsulation problem: setBs

```
University dtu = new University("DTU");
..
Collection<Student> students = new HashSet<Student>();
dtu.setStudents(students);
```

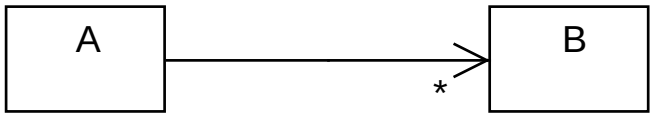
- Providing a setBs (e.g. setStudents(aCollection)) method poses also encapsulation problems. Why?
- Again the client can change the association without the university knowing about it

```
Student hans = new Student("Hans");
students.add(hans);
...
```

→ Solution: Here setStudents should make a copy of the collection

```
   public void setStudents(Collection<Student> stds) {
      students = new HashSet<Student>(stds);
}
```

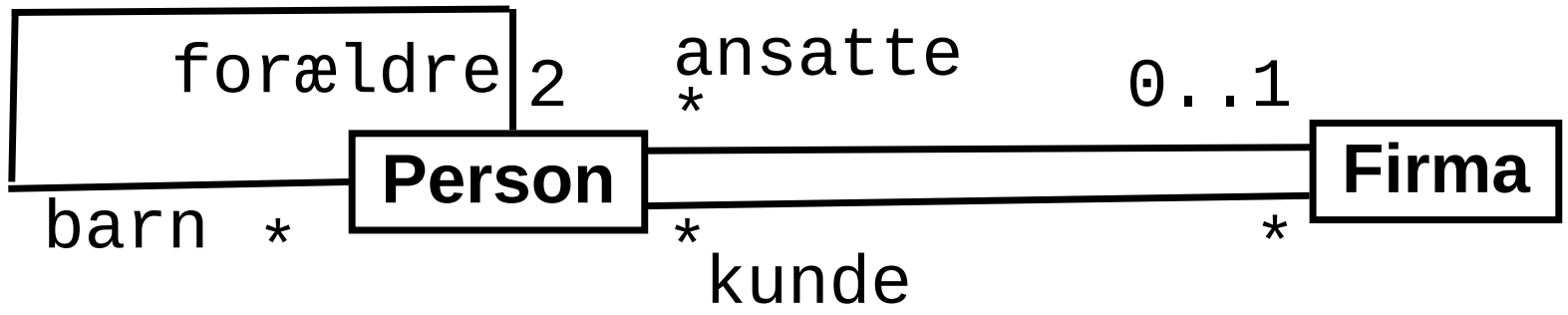
Solution: How to change the association?




```
public class A {

   private Collection<B> bs = new java.util.HashSet<B>();

   public void addB(B b) { bs.add(b);}
   public void contains(B b) { return bs.contains(b); }
   public void removeB(B b) { bs.remove(b); }
}
```

- addB, removeB, . . . control the access to the association
- The methods could have more intention revealing names, like registerStudent for addStudent
  - addB, removeB, . . . would normally not shown in class diagrams
  - intention revealing methods like registerStudent would be shown in class diagrams

Bi-directional associations



when associations don't have any arrows, can this be understood

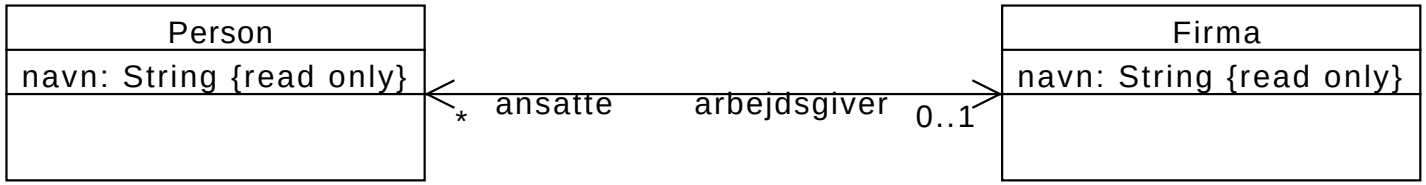
- as bi-directional, i..e. navigable in both directions, where on has decided not to show navigability, e.g.
  - every person object has a reference to his employer
  - every company object has a reference to his employee
- or as an under specification of navigability

The example shows also

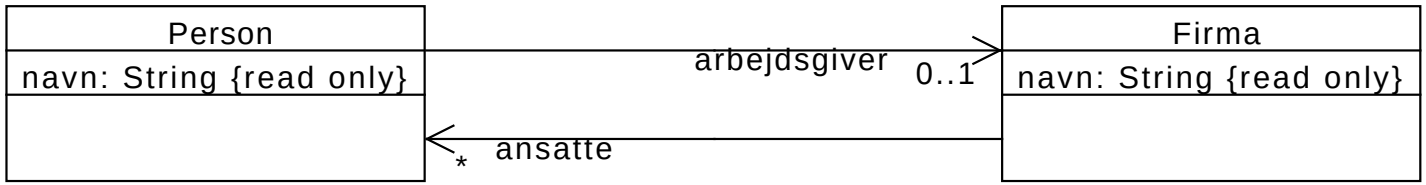
- two different associations between person and companies
- and a self-association

Implementing bi-directional associations

Example:



A bi-directional association is implemented as two uni-directional associations:

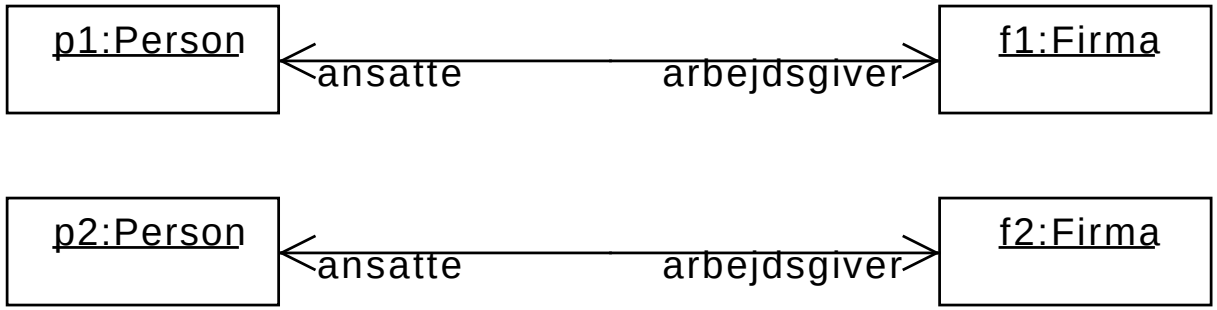


Note:

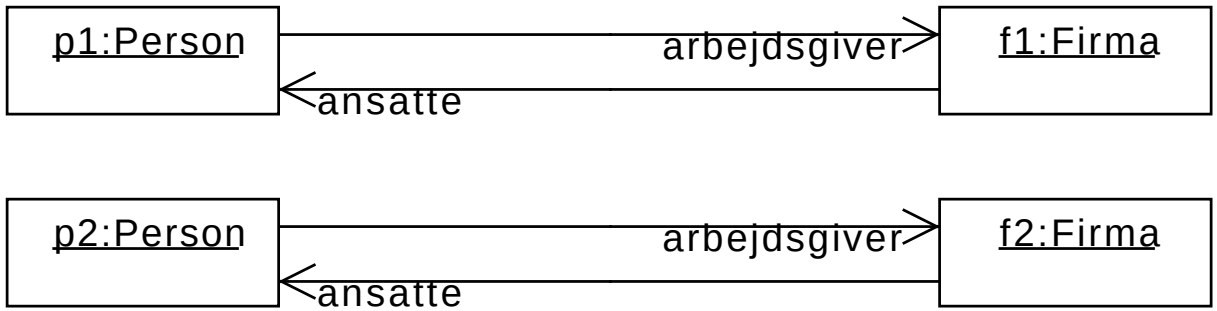
- Changes of a person objects employer gives rise to changes in up to two company objects list of employees
- Changes in the company's objects list of employees gives rise to a change in the person objects employer

→ referential integrity

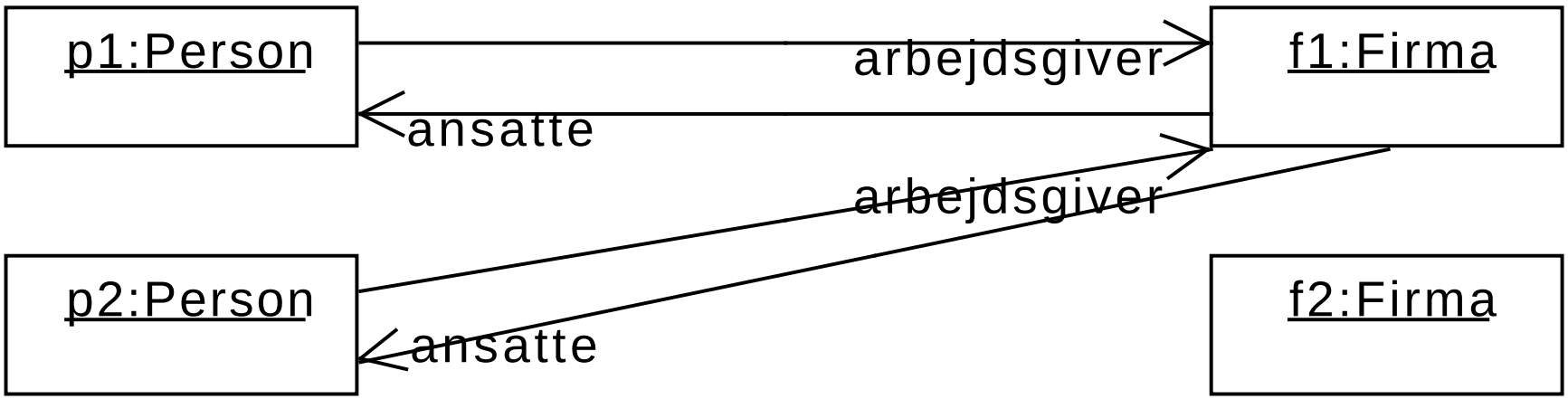
Referential Integrity



- Referential Integrity:
  - For all employees of a company *c*, their company has to be *c* and
  - for all persons *p*,they have to employee of the company they are employed in

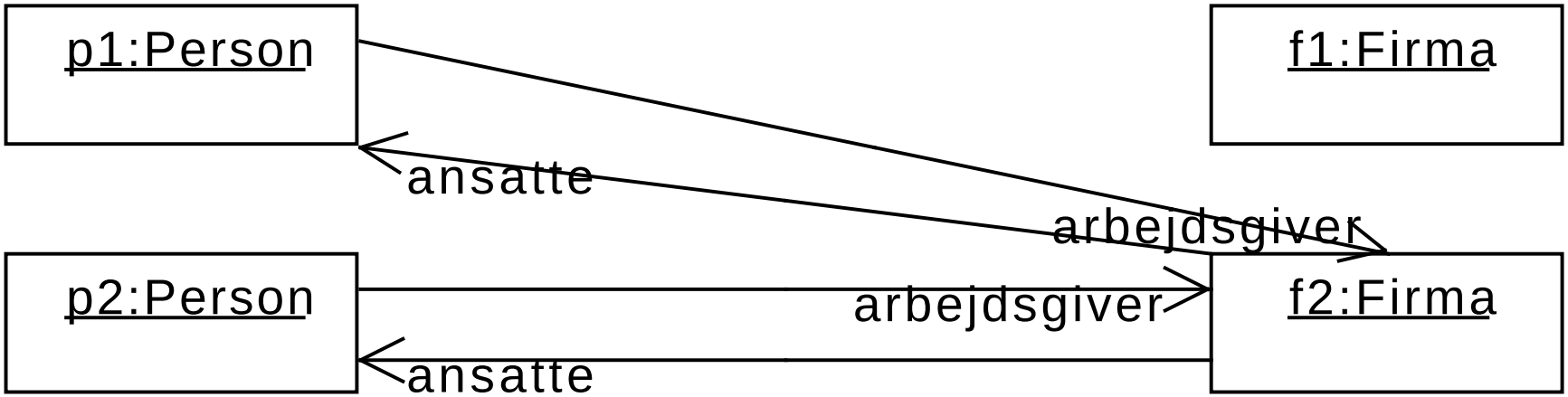


Referential Integrity: setArbejdsgiver



- setArbejdsgiver needs to ensure that the company is removed from the old employer and added to the new employer

Referential Integrity: addAnsatte



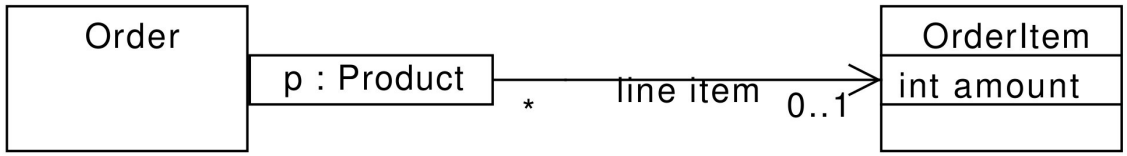
- addAnsatte needs to ensure that the old employer for the person is removed and set to the new new employer

Summary bi-directional associations

- Use bi-directional associations only when necessary
- Don’t rely on that the clients will do the bookkeeping for you

Qualified Assocations I

- A qualified association is an association, where an object is associated to another object via a qualifier (a third object)
- The interpretation is that to get to the order item, one has to provide an order and a qualifier, i.e. a product.



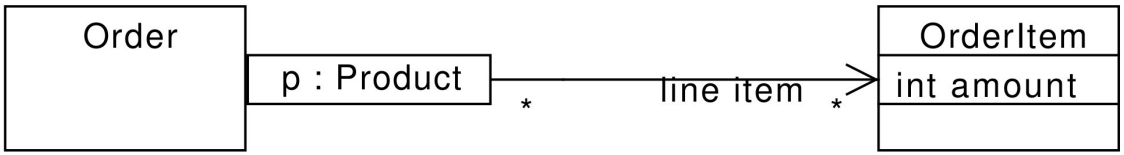
- An order is associated via a product to an order item called list item
  - This associations implies (with its multiplicity) that it is not possible to have two order items for the same product
- An order has for each product at most one list item

  → This is usually implemented by a map or dictionary mapping products to order items

```
public class Order {
   private Map<Product,OrderItem>
      listItem = new HashMap<Product,OrderItem>()
   ...
}
```

Qualified Associations II

- If the multiplicity is *, then several order items may be associated to a product



- Then the map has to return a collection for each product

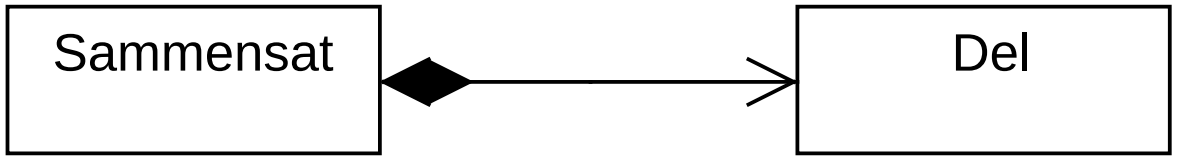
```
public class Order {
   private Map<Product,Collection<OrderItem>>
      listItems = new HashMap<Product,Collection<OrderItem>>()
   ...
}
```

Map<K,V> Interface

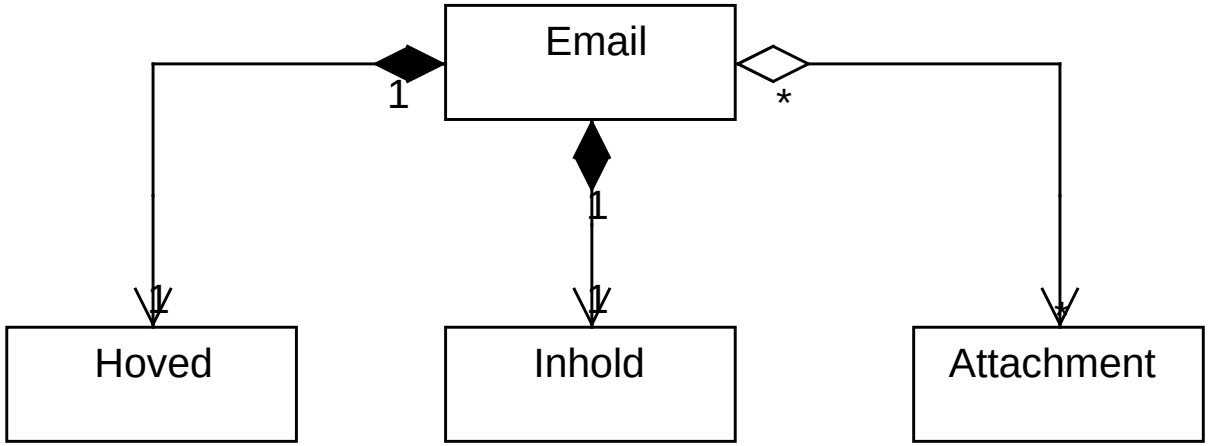
- Map<K,V> is an interface describing maps from objects of class K (keys) to objects of class V (values)
- A commonly used implementation of the map interface is HashMap<K,V> (implementation based on hash tables → Algorithm and Datastructures course)
- Most important operations
  - `m.containsKey(aK)` is true if there is a value to `aK` associated in `m`
  - `m.put(aK,aV)`: assigns the object `aV` to `aK`
  - `m.get(aK)`: retrieves the value stored under the key `aK`
  - Test 1 for maps: `m.put(aK,aV);`
    `assertTrue(m.containsKey(aK));`
    `assertSame(aV,m.get(aK));`
  - Test 2 for maps: `assertFalse(m.containsKey(aK));`
    `assertNull(m.get(aK));`
    - i.e. If the key is not in the map, `null` is returned

Composite Aggregation (I)

A special relation between ”part-of” between objects



Example: An email consists of a header, a content and a collection of attachments



Composite Aggregation (II)

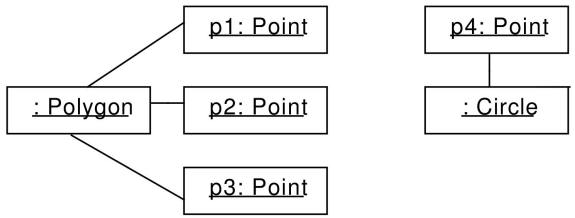
- The basic two properties of a composite aggregation are:
  - A part can only be part of one object
  - The of the part object is tied to the life of the containing object

→ This results in requirements to the implementation

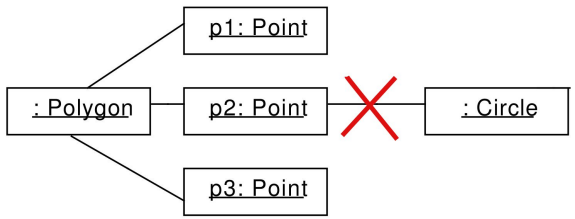
Composite Aggregation (III)

- A part can only be part of one object

- Allowed

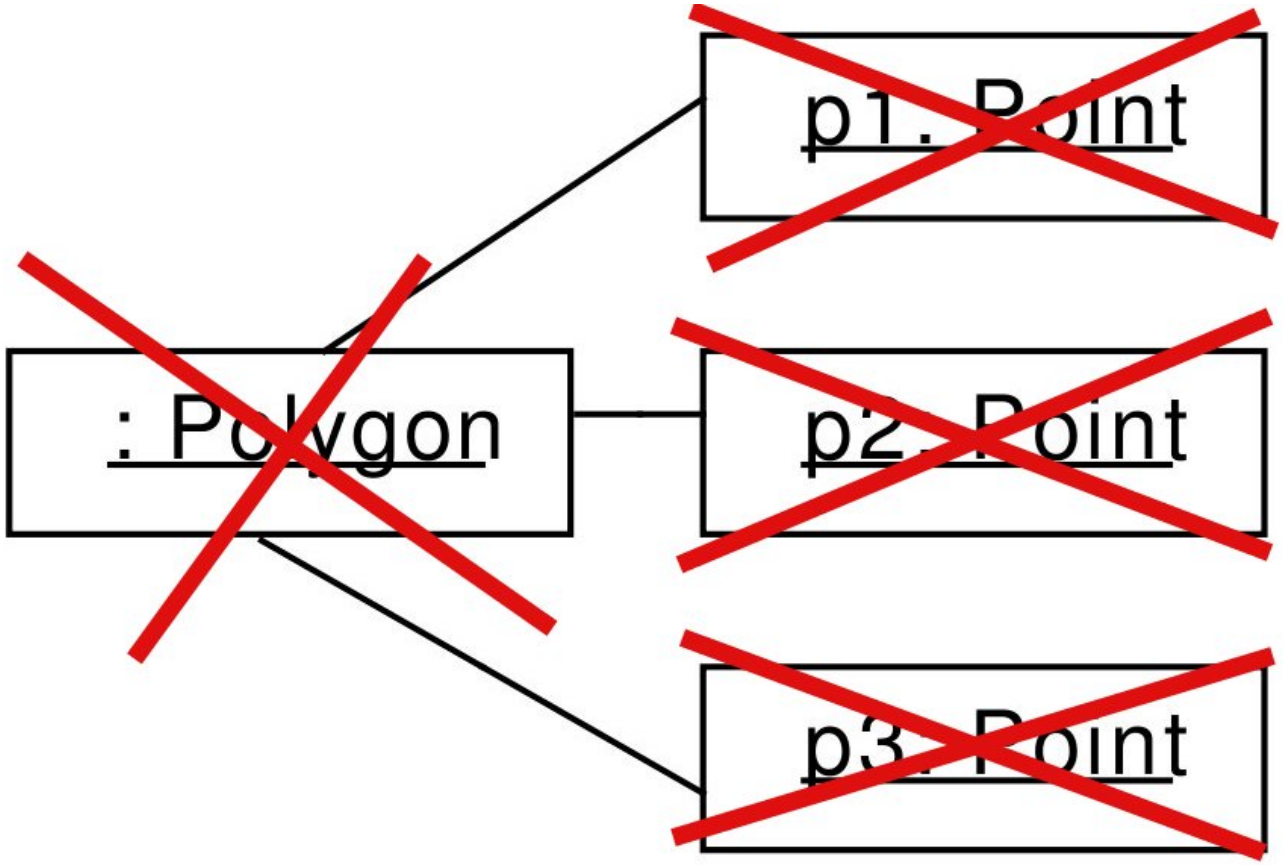


- Not Allowed



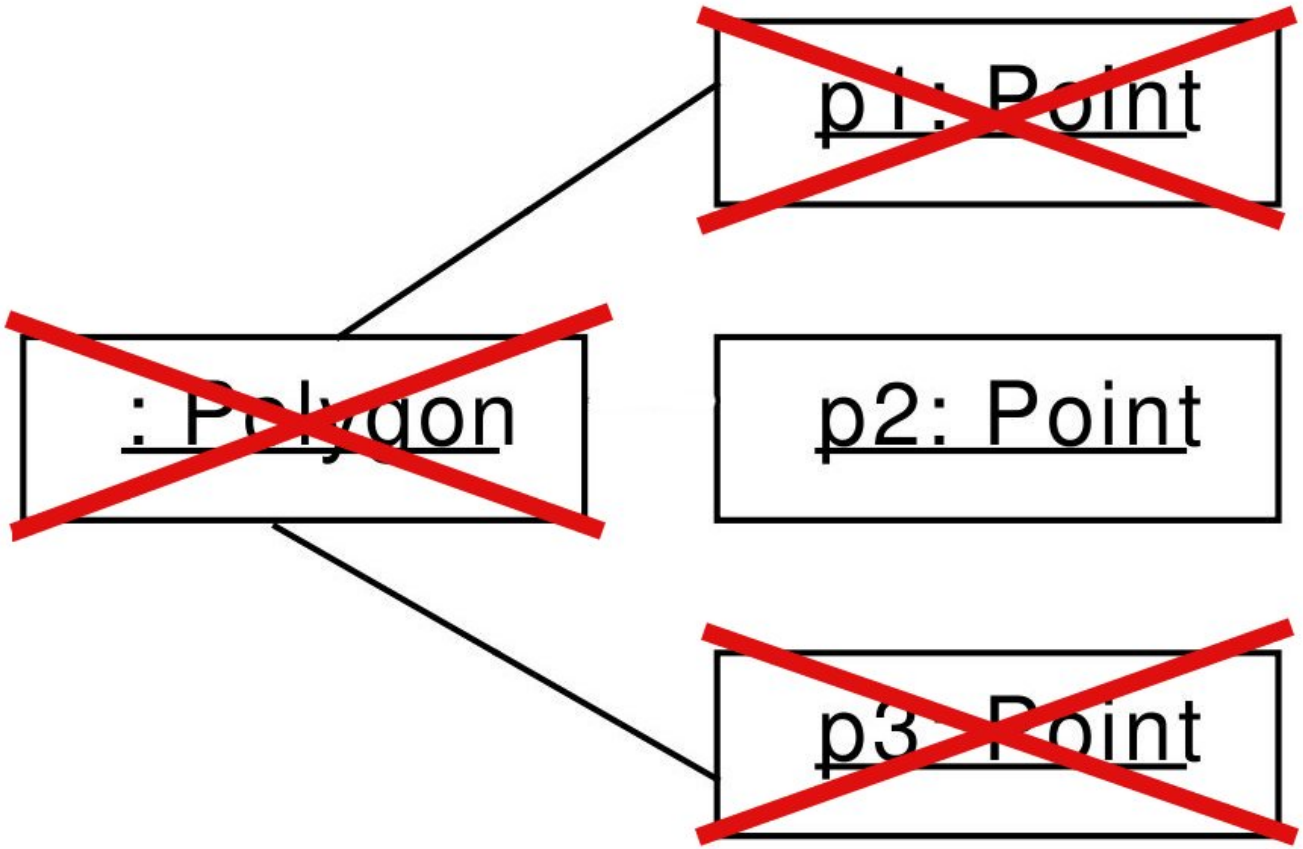
Composite Aggregation (IV)

- The life of the part object is tied to the life of the containing object
- If the containing object dies, so does the part object

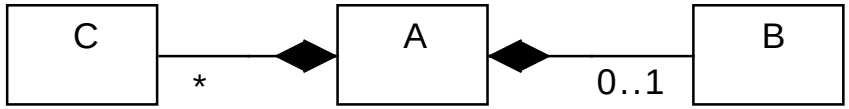


Composite Aggregation (V)

- But: A part can be removed before the composite object is destroyed



Implementing Composition



- Constraints to observe:
  1. a part can only be part of one composite
  2. parts die when the composite dies
     - → Problem of dangling references in programming languages where one can destroy objects (e.g. C++)
     - → Problem of objects not being garbage collected in languages like Java
- Idea: Ensure the constraints for all possible clients
  - → don't provide access to the parts!! If you have to, return a clone of the part
  - → No setB() or addC() method

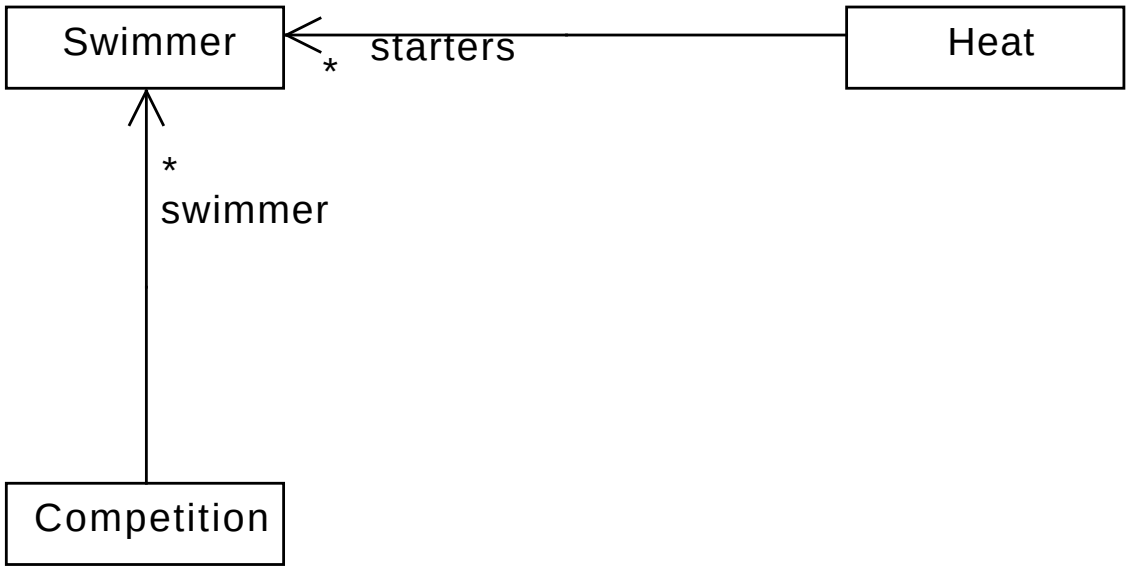
```
public class A {
  private B b = new B();
  private Set<C> cs = new Set<C>();
  public A() { cs.add(new C()); ... }
  public getB() { return b.clone(); }
  ...
}
```

Shared Aggregation

- Shared Aggregation
  - General ”part of” relationship
  - Notation: empty diamond

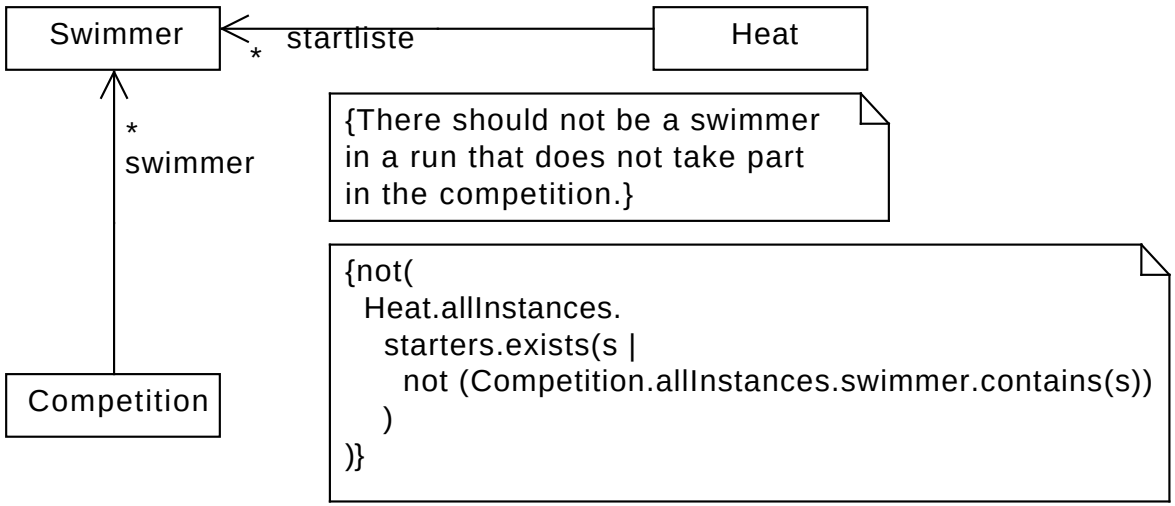
  - ”Precise semantics of shared aggregation varies by application area and modeller.” (from the UML 2.0 standard)

Notes in UML Diagrams I



- How to express the constraint
  - There should not be a swimmer in a heat that does not take part in the competition
- Note: It is not possible, in general, to express all constraints in a class diagram
  - → Use of notes to explicitly state the constraints

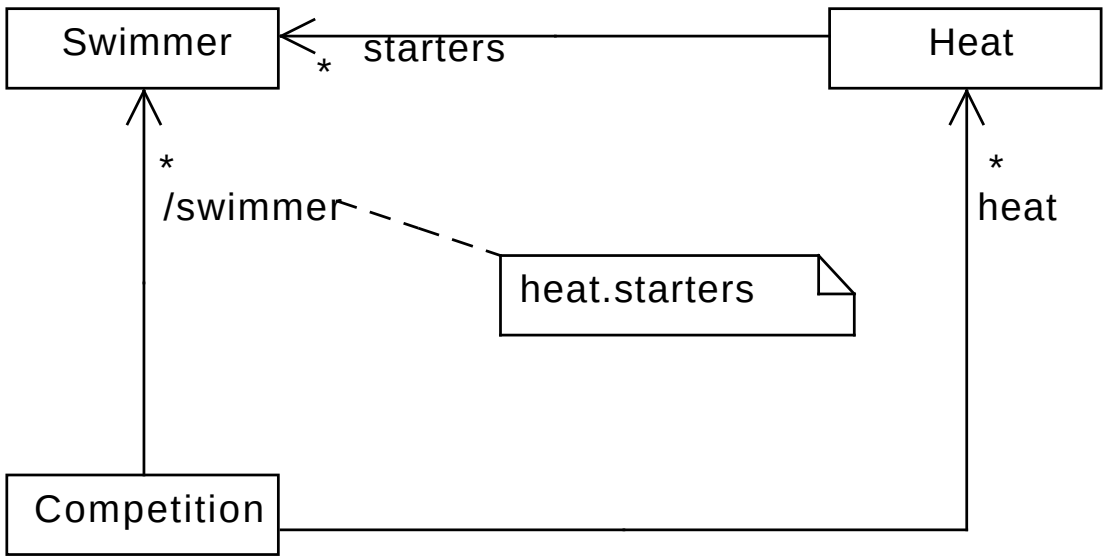
Notes in UML diagrams II



- Notes can be added to state the constraint
  - Informal: plain text describing the constraint
  - Formal: Using, e.g., OCL constraints (OCL = Object Constraint Language)
    - OCL is the default formal language for the UML

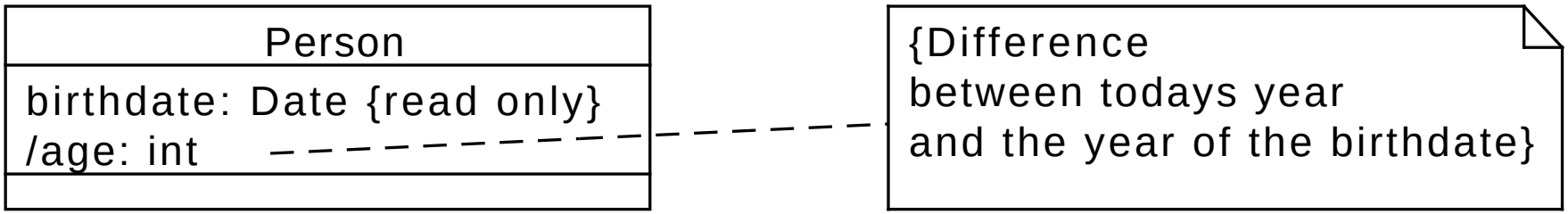
Derived Associations/Attributes

- Actually the diagram is missing an association from a competition to all the heats in a competition



- Swimmer is derived as it represents all swimmers that take part in a run in a competition
- Derivation is marked with a / together with a constraint telling how the attribute/association is derived

Class With Derived Attributes




```
public class Person {
  private Date birthdate;
  public Person(Date birthdate) {
    this.birthdate = birthdate;
  }
  public Date getBirthdate() { return birthdate; }
  public int getAge() {
    return new Date().getYear() - birthdate.getYear();
  }
}
```

Summary: Class Diagrams

- Class diagram: Visualize OO programs (i.e. based on OO programming languages)
  - → However, have more abstract language
- Classes: combines data and methods related to a common aspect (e.g. Person, Address, Company, . . .)
- Generalization between classes (corresponds to inheritance)
- Interfaces
- Association between classes
  - Unidirectonal
    - Associations vs. attributes
  - Multiplicities and how to implement them: 0..1, 1, *
  - Bi-directional
  - Qualifed assocations: Corresponds to the use of maps or dictionaries
  - Aggregation and Composition