Software Engineering I (02161)

Week 3: Implementing Class Diagrams

Hubert Baumeister

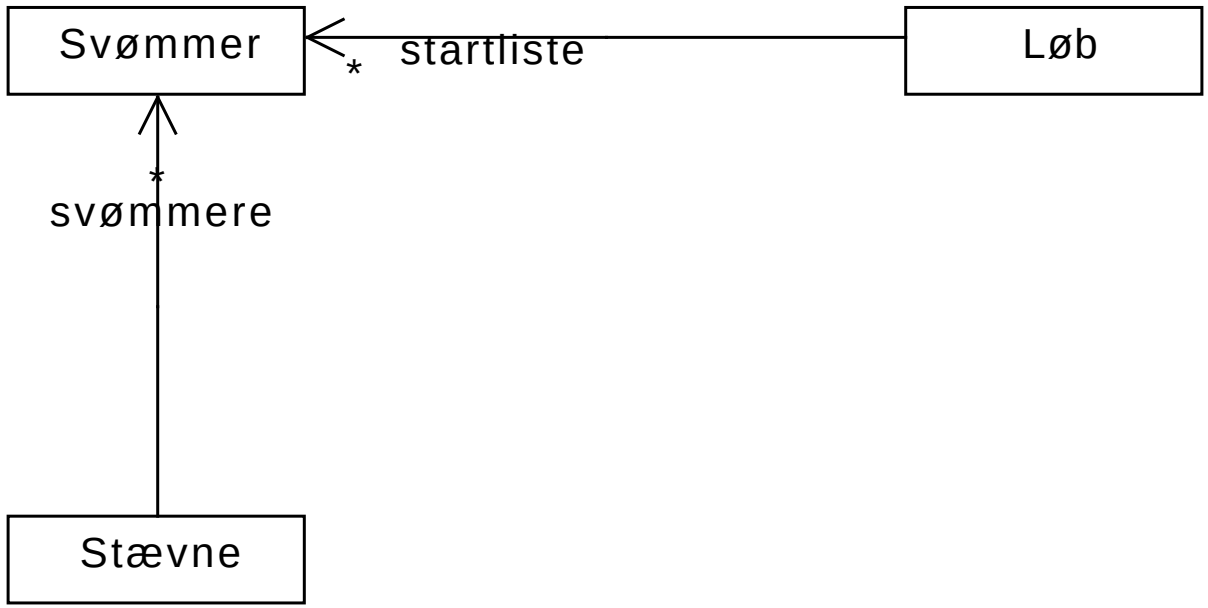
Informatics and Mathematical Modelling
Technical University of Denmark

Spring 2010

Recap

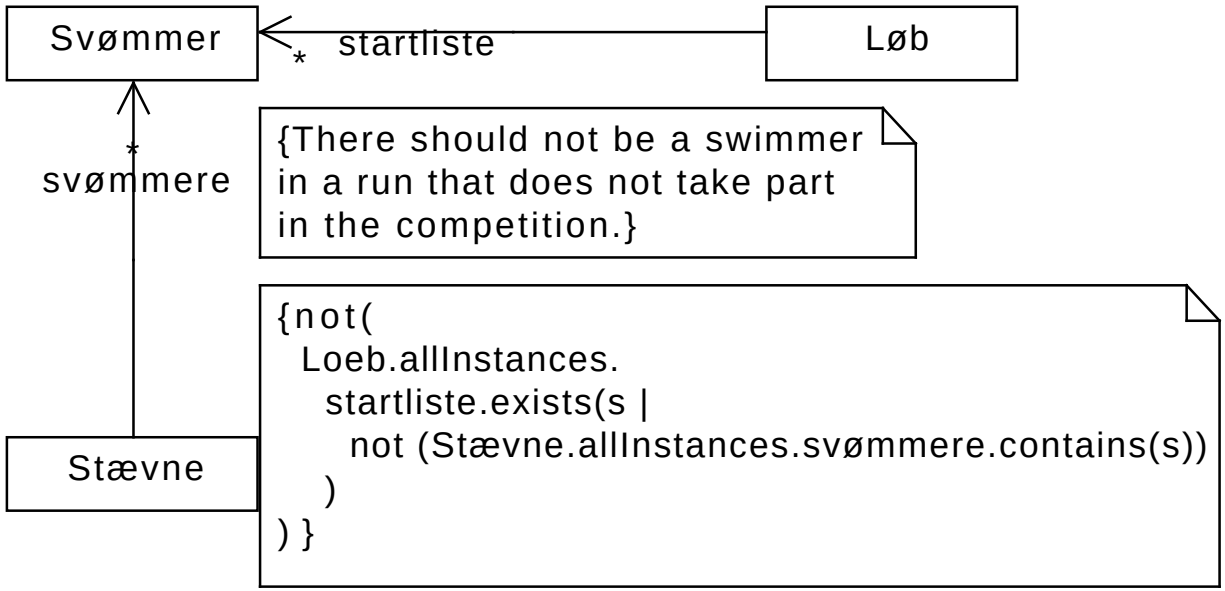
- From Requirements to Design
- Introduction to Class Diagrams
  - Classes
    - Attributes
    - Operations
  - Associations
    - uni- / bi-directional
    - aggregation and composition

Exercise 2



- How to express the constraint
  - There should not be a swimmer in a run 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

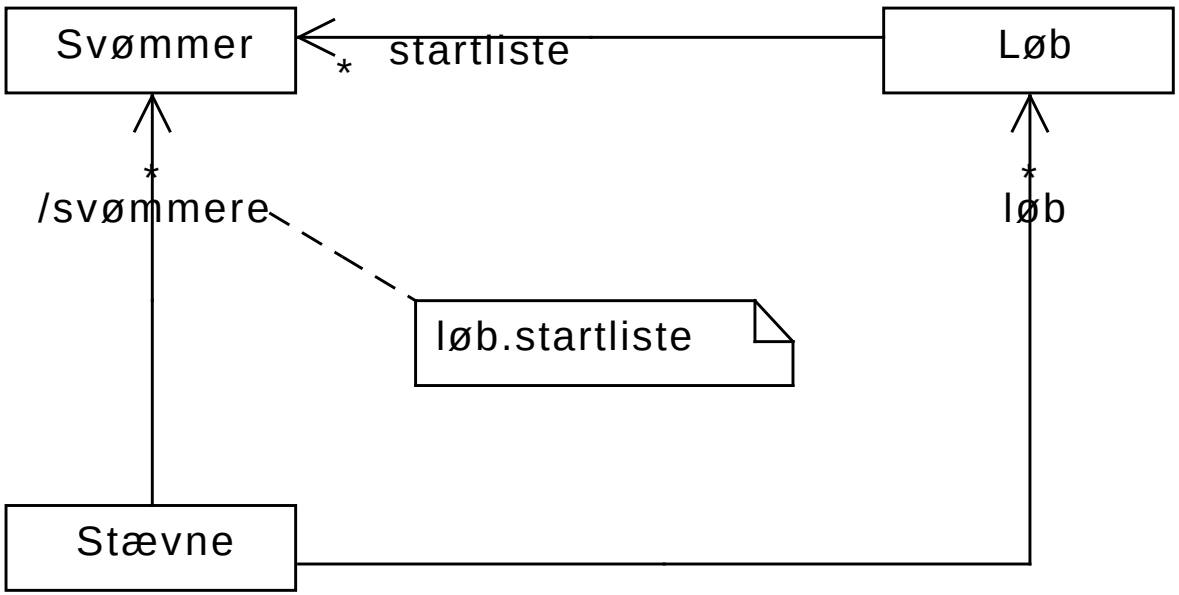
Notes in UML diagrams



- 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 runs in a competition



- Svømmere 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 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

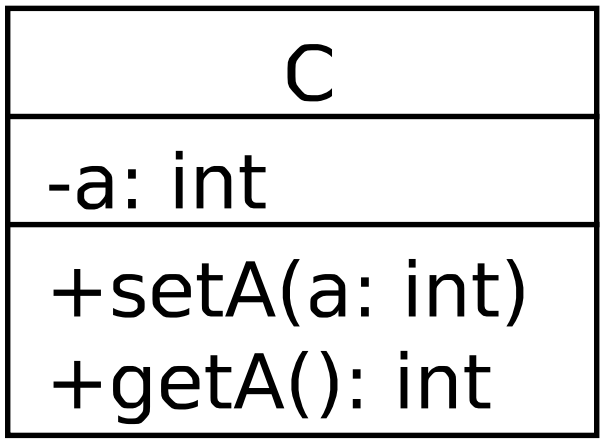
→ It is important to learn who these abstract, object-oriented concepts embodied in class diagrams are implemented in Java programs

  → Improves your object-oriented design skills

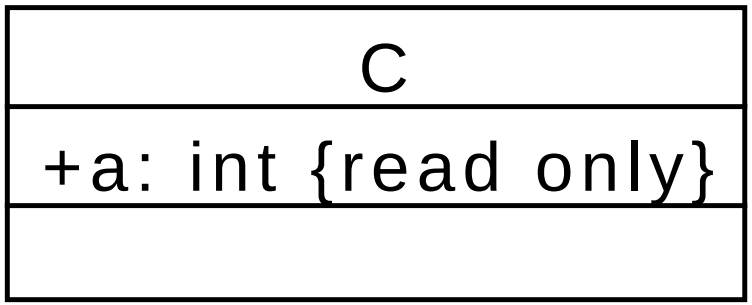
Example

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; }
}
```



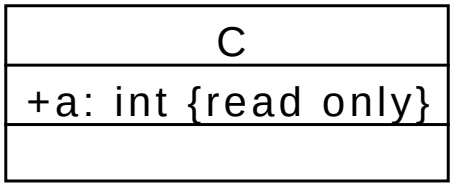
Class With Read-Only Attributes I



- Alternative 1: No setter method; value is set on construction time

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

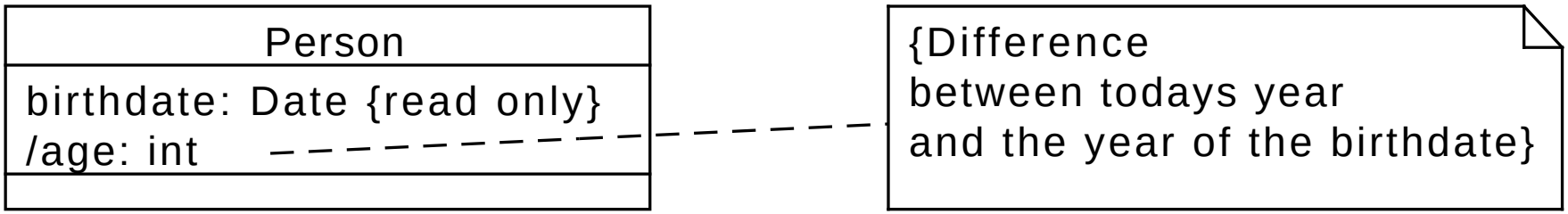
Class With Read-Only Attributes II



- Alternative 2: No setter method; value is computed on demand:

```
public class C {
  private int a;
  public int getA() {
    if (a == null) {
      a = computeA();
    }
    return a;
  }
  ...
}
```

Class With Derived Attributes



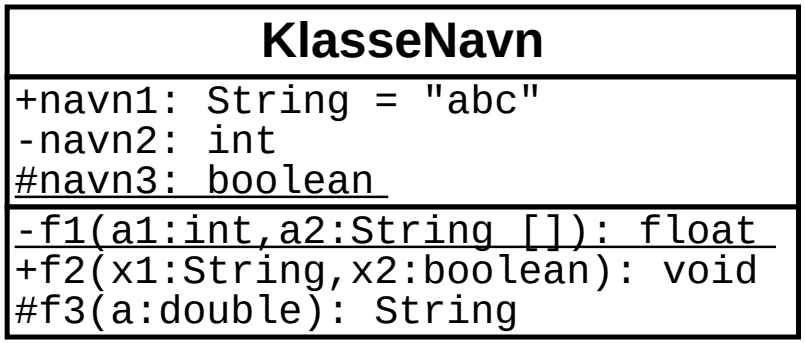

```
public class Person {
  private Date birthdate;
  public Person(Date birthdate) {
    this.birthdate = birthdate;
  }
  public Date getBirthdate() { return birthdate; }
  public int getAge() {
    return new Date().getYear() - birthdate.getYear();
  }
}
```

General correspondence between Classes and Programs

'-' : private

'+' : public

'#': protected



Klassens navn

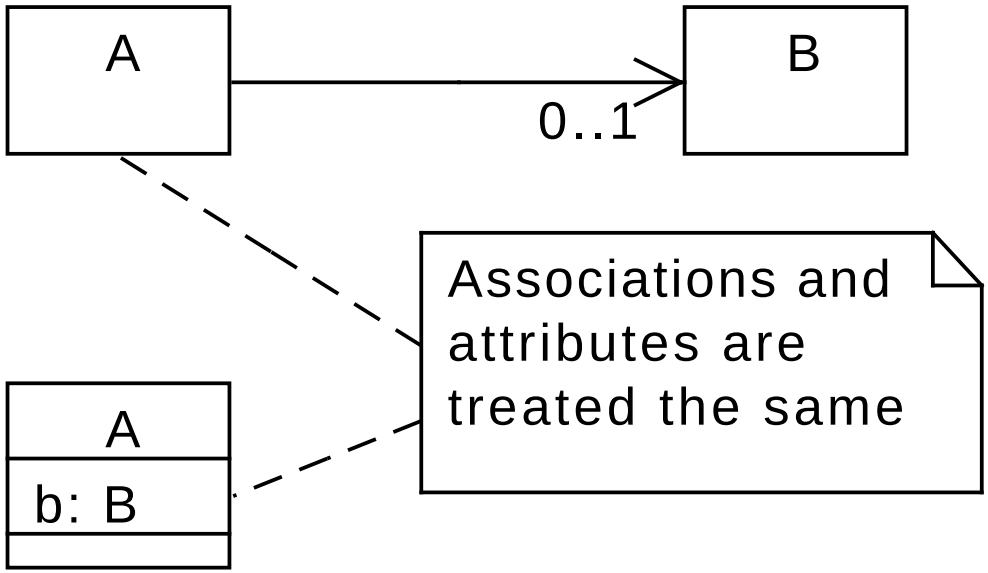
Attributter

Operationer

'navn3' og 'f1' er statiske størrelser

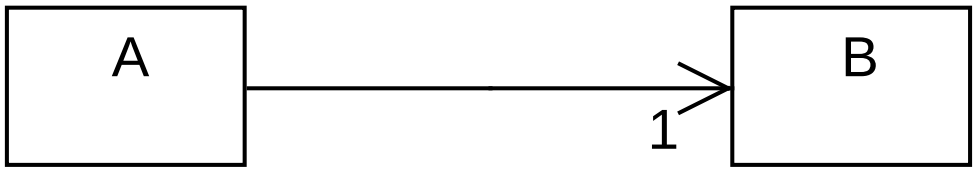
```
public class KlasseNavn
{
   private String navn1 = "abc";
   private int navn2;
   protected static boolean navn3;

   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) {...}
}
```

Implementing Associations: Cardinality 0..1




```
public class A {

   private B b;

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

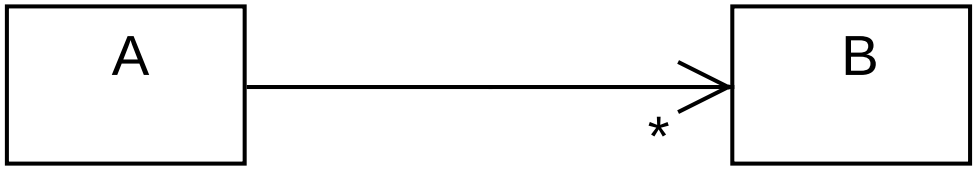
Implementing Associations: Cardinality 1



- 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 *



- Cannot be implemented anymore directly in Java

→ 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

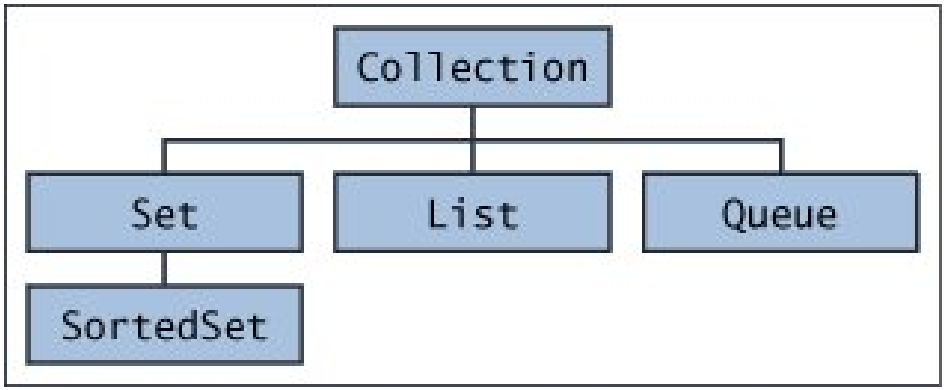
Interface *Collection<E>*

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

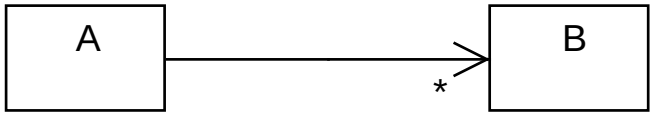


- Collection: Superclass of all collections
- Set: Order is irrelevant; no duplicates allowed
- List: Order is relevant; duplicates are allowed; allows positional access in addition to Collection operations
  - E get(int index);
  - E set(int index, E element);
  - void add(int index, E element);
  -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

Collection and their subinterfaces cannot be instantiated directly

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

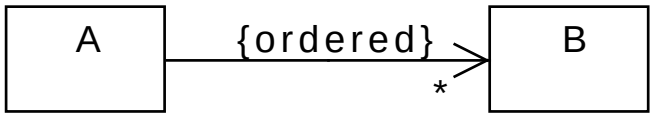
Implementing Associations: Cardinality *



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

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

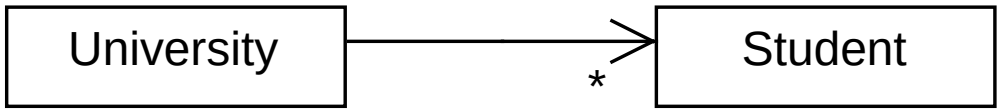
If one wants the collection to be ordered with duplicates one has to use {ordered}




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

Encapsulation problem

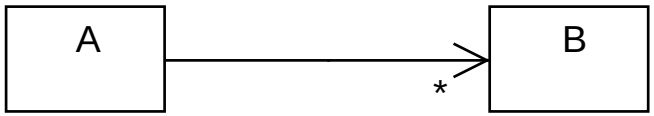
- Access to the implementation of the association using setB and getB poses encapsulation problems
  - A client of A can change the association without A knowing it!




```
University dtu = new University("DTU");
..
Student hans = new Student("Hans");

Collection<Student> students = dtu.getStudents();

students.add(hans);

students.remove(ole);
...
```

- Students can be added and removed, without the university knowing about it!

Implementing Associations: Cardinality * (II)

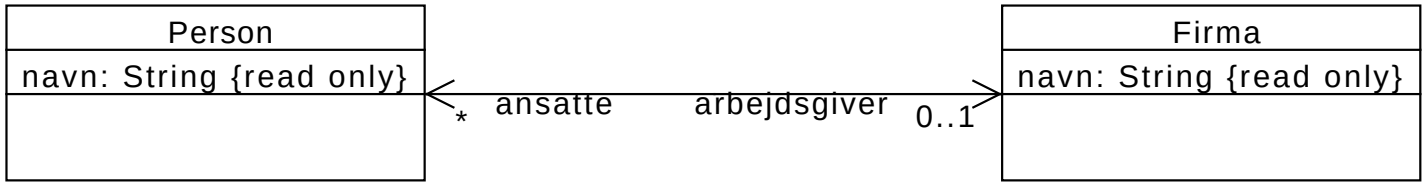



```
public class A {

    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); }
}
```

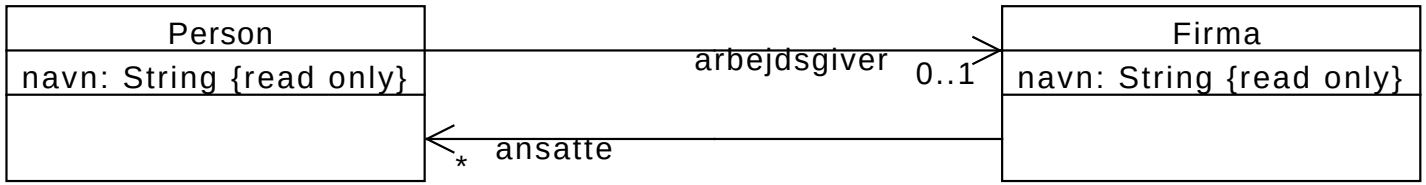
- addB, removeB, . . . control the access to the association
- The methods could have more intention revealing names, like registerStudent for addStudent
  - addB, removeB, . . . would normally not shown in class diagrams
  - intention revealing methods like registerStudent would be shown in class diagrams

Implementing bi-directional associations

Example:



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

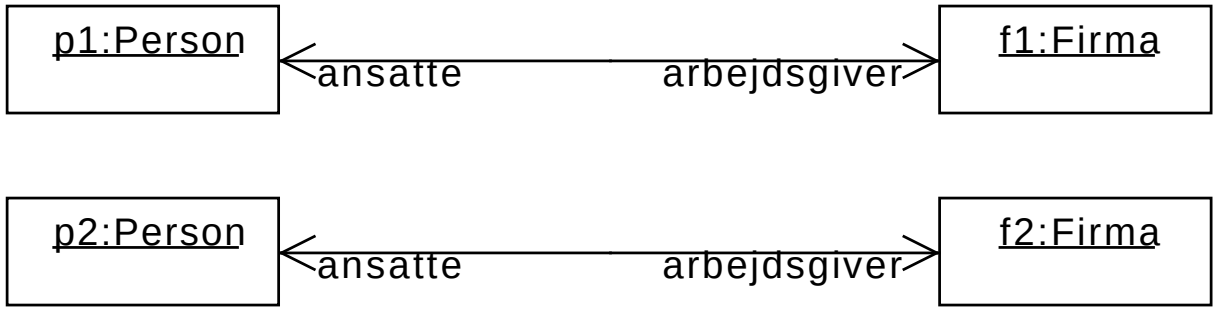


Note:

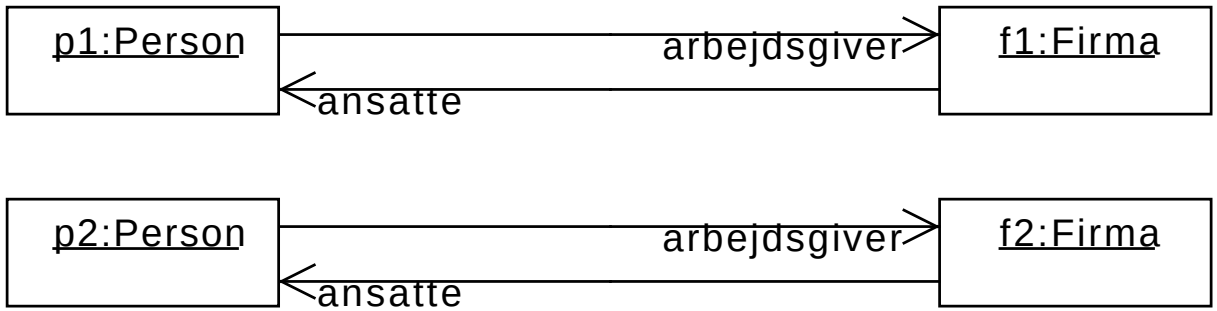
- Changes of a person objects employer gives rise to changes in up to two company objects list of employees
- Changes in the company's objects list of employees gives rise to a change in the person objects employer

→ referential integrity

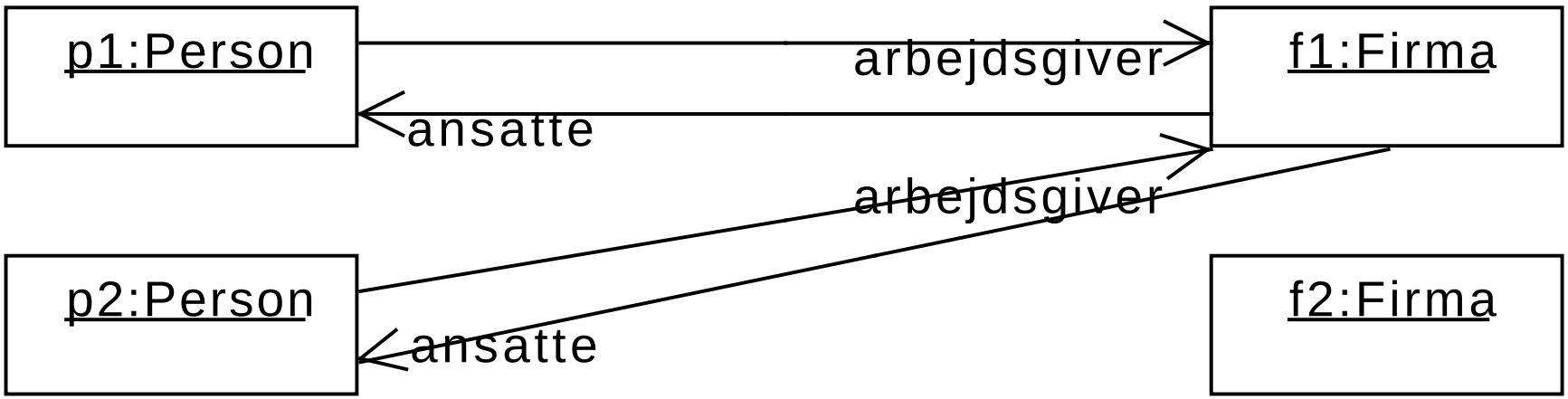
Referential Integrity



- 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

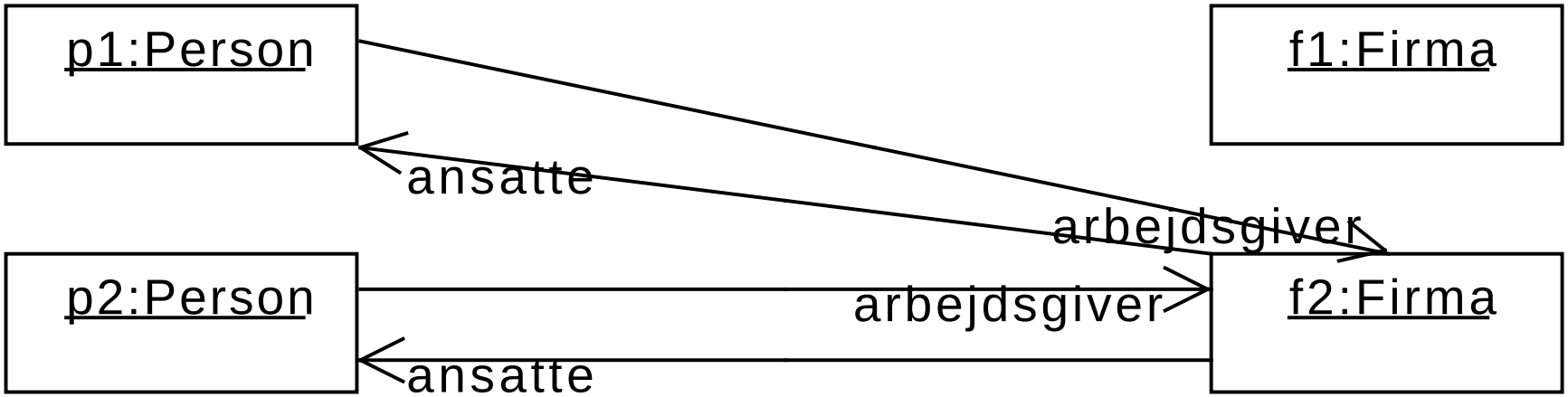


Referential Integrity: setArbejdsgiver



- 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

Referential Integrity via helper functions

Three extra operations are used to ensure consistency

- Person:

Implementation of role arbejdsgiver in Person

```
Firma arbejdsgiver = null;

protected void setF(Firma f){arbejdsgiver = f;}
public void setFirma(Firma f){
   if (arbejdsgiver != null) arbejdsgiver.sletP(this);
   if (f != null) f.addP(this);
   arbejdsgiver = f;}
```

- Firma:

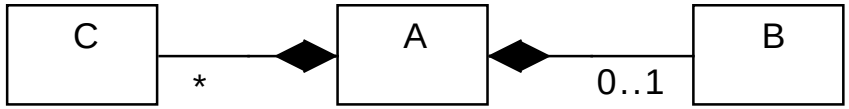
Implementation of role ansatte in Firma

```
Collection<Person>ansatte = new ArrayList<Person>();

public void addAnsat(Person p)
 {if (!ansatte.contains(p)) {ansatte.add(p); p.setF(this);} }

public void sletAnsat(Person p)
 {if (ansatte.contains(p)) {ansatte.remove(i); p.setF(null);}}

protected void addP(Person p){if (!ansatte.contains(p)) ansatte.add(p);}

protected void sletP(Person p)
 {if (ansatte.contains(p) ansatte.remove(p);}
```

Summary bi-directional associations

- Exercise
  - Is the implementation correct?
  - What happens if a company adds an employee that works already for another company?
- Summary
  - Use bi-directional associations only when necessary
  - Don’t rely on that the clients will do the bookkeeping for you

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(); }
  ...
}
```