Klassediagrammer (I)

Modellering og Implementering

Michael R. Hansen and Hubert Baumeister

Informatics and Mathematical Modelling
Technical University of Denmark

Spring 2008

Oversigt

- beskrivelser af klasser
- ensrettede associationer (unidirectional associatons)
- dobbeltrettede associationer (bidirectional assocations)
- aggregering (shared aggregation and composite aggregation)
- implementering

Næste uge: nedarvning (subclassing / inheritance), implementering, brug af tabeller (map datatype)

Klassediagrammer

Formål:

- at modellere begreber
- at identificere væsentlige objekter
  - deres tilstand og grænseflade
- at give et overblik over systemets sammensætning
- at give et overblik over systemets implementering

Klassebeskrivelse

En klasse beskriver en samling objekter med fælles karakteristika mht

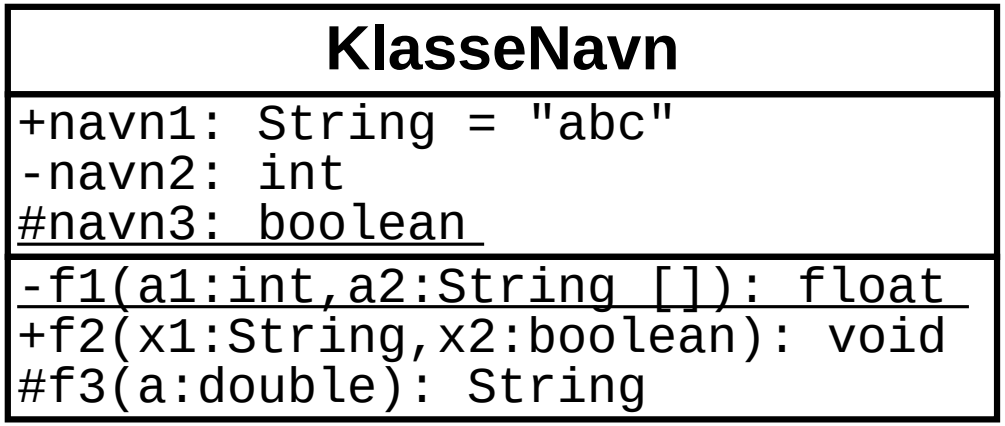
- tilstand (attributter)
- opførsel (operationer)
- relationer til andre klasser (associationer)

Beskrivelse af en klasse

’-’ : private

’+’ : public

’#’: protected



Klassens navn

Attributter

Operationer

’navn3’ og ’f1’ er statiske størrelser

private : kun synlighed i selve klassen

protected : synlig også i subklasser

public : synlig også for andre klasser

package : (∼) visible for other classes in the same package

- Attributes and operations that are underlined are static
  - Attributes can be accessed without that an instance of that class exists
  - Operations can be called without that an instance of that class exists

En klasses roller

- Grænsefladen (interface) for en klasse har to formål
  - at beskrive den funktionalitet som stilles til rådighed for omverdenen
  - at beskrive den funktionalitet som skal implementeres
- Private attributter indkapsler data
- Private operationer er hjælpefunktioner for andre operationer

Detaljeringsgrad ved en klassebeskrivelse afhænger af formål

Begrebsmodellering : typisk lav detaljeringsgrad

⋮

Implementering : typisk høj detaljeringsgrad

Klassebeskrivelse og programskelet (I)

Til hver klassebeskrivelse svarer et programskelet, f.eks.

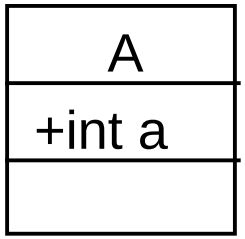
```
public class KlasseNavn
{
   public String navn1 = "abc"
   private int navn2
   protected static boolean navn3

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

Udviklingsværktøjer udnytter denne sammenhæng:

- Diagram → Programskelet — programmeludvikling
- Program → Diagram — reverse engineering

Klassebeskrivelse og programskelet (II): Implementing public attributes

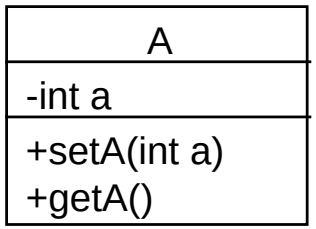
Attributes are implemented as fields




```
public class A {
   public int a;
}
```

In Java one uses setter and getter methods to access public fields instead of directly accessing the field

This corresponds to the following class diagram




```
public class A {
   private int a;

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

However, a lot of tools support the generation of above Java code from the original diagram

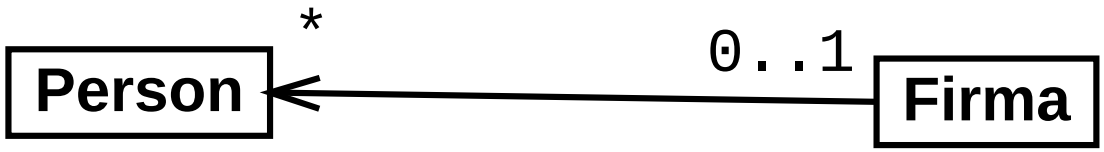
Associationer mellem klasser

- En *association* mellem to klasser betyder, at objekter tilhørende de to klasser har et *kendskab* til hinanden.
- Associationer kan være såvel ensrettede som dobbeltrettede

→ kendskabet kan altså være ensidigt eller gensidigt.

Ensrettede associationer I

Eksempel: Personer og deres arbejdsgivere



- enhver person er tilknyttet en arbejdsgiver (firma)
- ethvert firma har 0, 1, eller flere ('*') ansatte (personer)

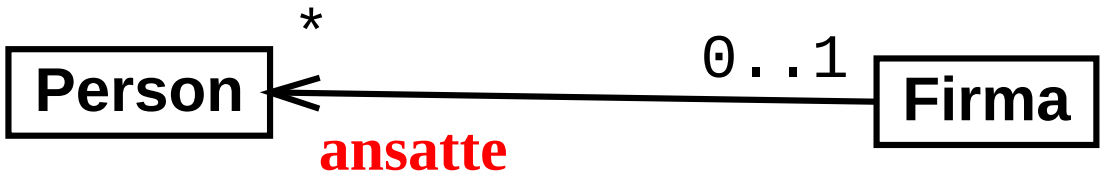
Pilen betyder at et firma har kendskab til alle ansatte (personer)

$\longrightarrow$ et firmaobjekt indeholder referencer til objekter for alle ansatte

- Omvendt behøver et personobjekt ikke at reference til sin arbejdsgivers objekt.
- Et $\times$ på en pil markerer ikke-navigerbarhed.
- *Navigerbarhed* i pilens retning.
- 0 og $*$ kaldes *multipliciteter* eller *kardinaliteter*

Ensrettede associationer: II

Eksempel: Ansatte og deres arbejdsgivere

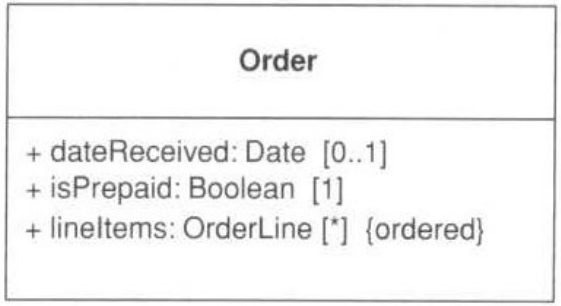


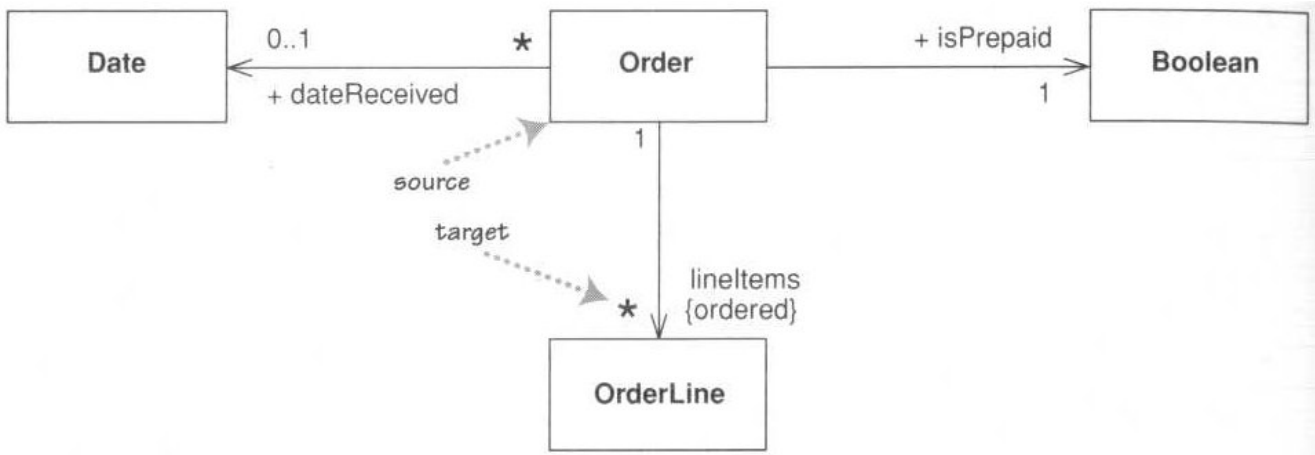
- et *rollenavn* (her **ansatte**) beskriver objekter (her personer) ved en ende af en association, set ud fra objekter tilhørende klassen i den anden ende (her firmaer).
- i en implementering er et rollenavn typisk en variabel. F.eks.

```
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?

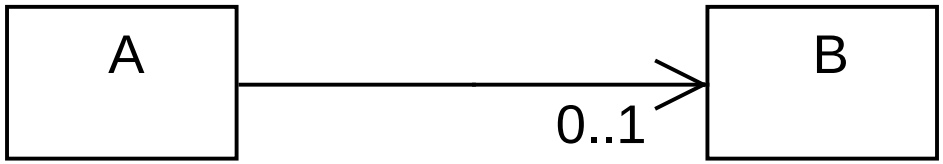
- Associations
  - When the target class of an association is shown in the diagram
  - The target class of an association is a major class of the model
    - e.g. Part, Assembly, Component, . . .
- Attributes
  - When the target class of an associations is not shown in the diagram
  - With datatypes / 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?

Kardinaliteter

- Ved associationers endepunkter (og andre steder) skal ofte angives hvor mange gange en størrelse kan forekomme. F.eks. hvor mange ansatte firmaer kan have.
- Disse angivelser kaldes *multipliciteter* eller *kardinaliteter*.
- Typisk forekommende angivelser af kardinalitet:

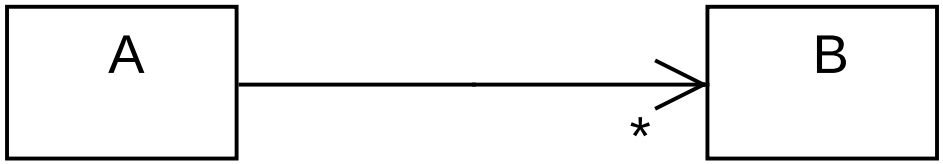
angivelse	**betydning**
`0`	0
`1`	1
m..*n*	intervallet af heltal fra *m* til *n*
*	$0, 1, 2, \ldots$
m..*	$m, m+1, m+2, m+3, \ldots$

Implementing Associations: Cardinality 0..1




```
public class A {

   private B b;

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

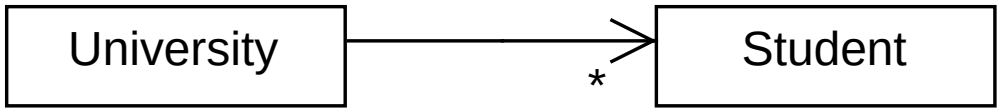
Implementing Associations: Cardinality *




```
public class A {

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

   public Collection<B> getB() { return bs;}
   public void setB(Collection<B> bs) { this.bs = bs; }
}
```

- If the multiplicity is >1, one adds a plural s to the role name: b $\rightarrow$ bs
- 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!

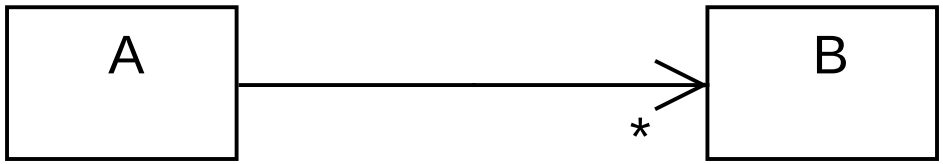
Encapsulation problem




```
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.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); }
}
```

- addB, removeB, . . . control the access to the association
- The methods should have more intention revealing names, like registerStudent for addStudent

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

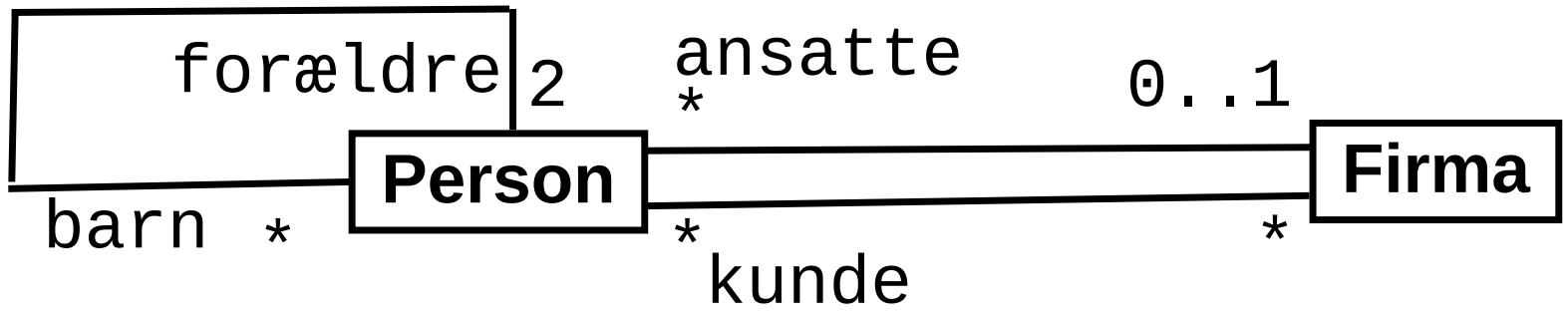
Example of iterating over a collection

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

Collection cannot be instantiated directly

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

Dobbeltrettede associationer



Når associationer ingen pile har, kan de enten forstås

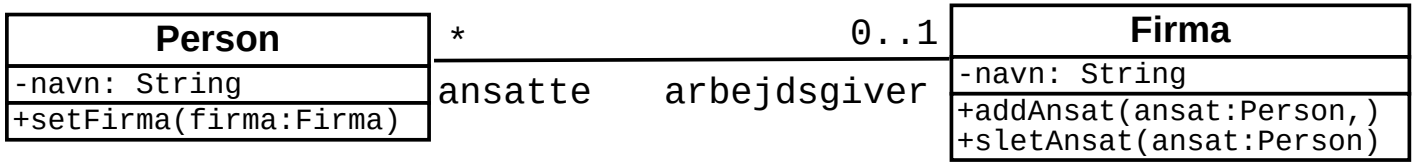
- som *dobbeltrettede*, dvs. navigerbare i begge retninger, hvor man har besluttet at ikke viser navigerbarhed, f.eks.
  - hvert personobjekt har en reference til sin arbejdsgiver
  - hvert firmaobjekt har referencer til sine ansatte
- eller som en underspecifikation af navigerbarhed.

Eksemplet viser i øvrigt

- to forskellige associationer mellem personer og firmaer
- en selv-association

Dobbeltrettede associationer (II)

Eksempel:



Bemærk:

- Ændring af et person-objekts arbejdsgiver giver anledning til ændringer i op til to firmaer-objekter liste over ansatte.
- Ændring af et firma-objekts liste over ansatte afleder ændring af et person-objekts arbejdsgiver.

→ referential integrity

Person

```
private String navn;
private Firma arbejdsgiver;

public Person(String nv){navn = nv;}

public void setFirma(Firma f){ ... }
```

Firma

```
private String navn;
private ArrayList<Person> ansatte = ....
public Firma(String nv){navn = nv; }

public void addAnsat(Person p} { ... }

public void sletAnsat(Person p) { ... }
```

Implementing bidirectional associations

- Bidirectional associations are implemented as two unidirectional associations
- Person:

```
Firma arbejdsgiver = null;
```

- Firma:

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

→ Konsistensproblem

```
public void testAddAnsatte() {
   Firma firma = new Firma();
   Person hans = new Person();

   firma.addAnsatte(hans);

   // hans.setArbejdgiver(firma) needs to be tested too!

   assertTrue(firma.containsAnsatte(hans));
   assertEquals(firma,hans.getArbejdsgiver());
}
```

- sletAnsatte and setArbejdsgiver(null) have similar problems

Konsistens via Hjælpefunktioner

Tre ekstra operationer bruges i pakken til at opretholde konsistens

- Person:

Implemenation 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:

Implemenation 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);}
```

Exercise

- Is the implementation correct?
- What happens if a person changes companies?

Eksempel: Svømmere og klubber

Problem description

En svømmer kan være medlem af en klub, men behøver det ikke hvis hun f.eks. svømmer på et elitecenter. En svømmer kan højst stille op til stævner for een klub. En svømmer beskrives ved navn, bedste tider, osv. En klub har en række medlemmer, og beskrives ved et navn.

- Man skal kunne oprette svømmere og klubber
- Man skal kunne tilføje, slette og ændre medlemskab

Hvordan gribes opgaven an?

- What is the user relevant functionality of the system? Can one do example scenarios?
- For each scenario
  - identificer væsentlige begreber
  - lav en klassemodellering
  - Write a test for the scenario
  - implementer

Step 1: Identify functionality

Functionality #1: Add a swimmer to a club

- Create a club with name Holte
- Create a swimmer with name Ole
- Add swimmer Ole to club Holte
- Check that Ole is member of Holte

Functionality #2: Remove a swimmer from a club

- Create a club with name Virum
- Create a swimmer with name Per
- Add swimmer Per to club Virum
- Check that Per is member of Virum
- Remove swimmer Per from club Virum
- Check that Per is not member of Virum anymore

Step 2: Identify relevant notions

Problem Description

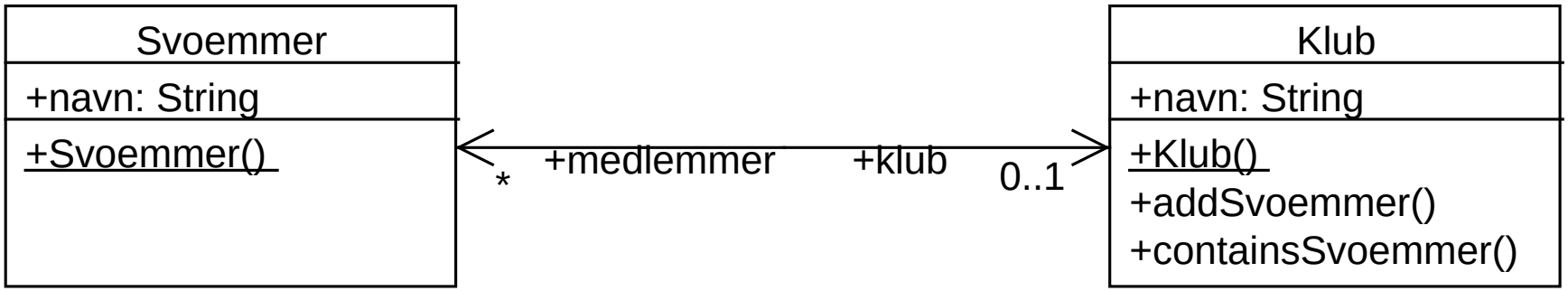
En svømmer kan være medlem af en klub, men behøver det ikke hvis hun f.eks. svømmer på et elitecenter. En svømmer kan højst stille op til stævner for en klub. En svømmer beskrives ved navn, bedste tider, osv. En klub har en række medlemmer, og beskrives ved et navn

Add swimmer scenario

- Create a club with name Holte
- Create a swimmer with name Ole
- Add swimmer Ole to club Holte
- Check that Ole is member of Holte

- Nouns → Classes
- Relationship between nouns → Attributes and Associations
- Actions → Methods

Step 3: Class modelling



Step 4: Write a test case for the scenario

```
public void testAddSvoemmer1() {
    Klub holte = new Klub("Holte");
    Svoemmer ole = new Svoemmer("Ole");

    assertFalse(holte.containsSvoemmer(ole));

    holte.addSvoemmer(ole);

    assertTrue(holte.containsSvoemmer(ole));
    assertSame(holte,ole.getKlub());
}

public void testAddSvoemmer2() {
    Klub holte = new Klub("Holte");
    Svoemmer ole = new Svoemmer("Ole");

    assertFalse(holte.containsSvoemmer(ole));

    ole.setKlub(holte);

    assertSame(holte,ole.getKlub());
    assertTrue(holte.containsSvoemmer(ole));
}
```

Step 5: Implement Svoemmer

```
package imm.swe1.svoemmer;

public class Svoemmer {
   private String navn;
   private Klub klub;

   public Svoemmer(String navn) { this.navn = navn; }

   public String getNavn() { return navn; }

   public Klub getKlub() { return klub; }

   public void setKlub(Klub klub) {
      if (this.klub == klub) return;
      this.klub = klub;
      klub.addSvoemmer(this);
   }

   protected void setK(Klub klub) { this.klub = klub; }
}
```

Step 5: Implement Klub

```
public class Klub {
   private String navn ;
   private Collection<Svoemmer> medlemmer =
                                  new java.util.ArrayList<Svoemmer>()

   public Klub(String navn) { this.navn = navn; }

   public String getNavn() { return navn; }

   public void addSvoemmer(Svoemmer svoemmer) {
      if (containsSvoemmer(svoemmer)) return;
      medlemmer.add(svoemmer);
      svoemmer.setK(this);
   }

   public boolean containsSvoemmer(Svoemmer svoemmer) {
      return medlemmer.contains(svoemmer);
   }
}
```

Step 2: Identify relevant notions

Problem description

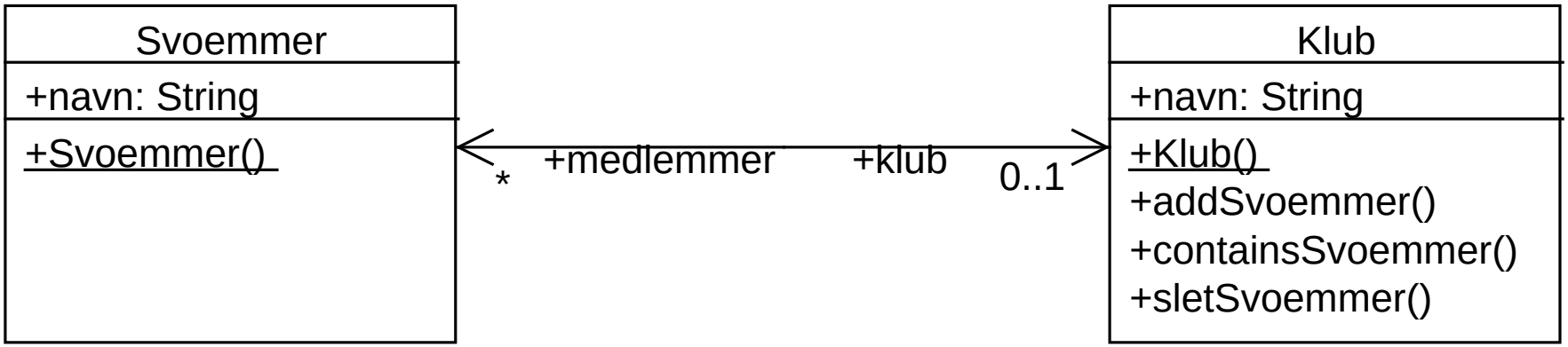
En svømmer kan være medlem af en klub, men behøver det ikke hvis hun f.eks. svømmer på et elitecenter. En svømmer kan højst stille op til stævner for en klub. En svømmer beskrives ved navn, bedste tider, osv. En klub har en række medlemmer, og beskrives ved et navn.

Remove svimmer scenario

- Create a club with name Virum
- Create a swimmer with name Per
- Add swimmer Per to club Virum
- Check that Per is member of Virum
- Remove swimmer Per from club Virum
- Check that Per is not member of Virum anymore

- Nouns → Classes
- Relationship between nouns → Attributes and Associations
- Actions → Methods

Step 3: Class modelling



Step 4: Write a test case for the scenario

```
public void testSletSvoemmer1() {
    Klub virum = new Klub("Virum");
    Svoemmer per = new Svoemmer("Per");
    virum.addSvoemmer(per);

    assertTrue(virum.containsSvoemmer(per));

    virum.sletSvoemmer(per);

    assertFalse(virum.containsSvoemmer(per));
    assertNull(per.getKlub());
}

public void testSletSvoemmer2() {
    Klub virum = new Klub("Virum");
    Svoemmer per = new Svoemmer("Per");
    virum.addSvoemmer(per);

    assertTrue(virum.containsSvoemmer(per));

    per.setKlub(null);

    assertNull(per.getKlub());
    assertFalse(virum.containsSvoemmer(per));
}
```

Step 5: Implement Svoemmer

Additions to class Svoemmer

```
public void setKlub(Klub klub) {
   if (this.klub == klub) return;
   Klub oldKlub = this.klub;
   this.klub = klub;
   if (oldKlub != null) {
      oldKlub.sletS(this);
   }
   if (this.klub != null) {
      this.klub.addSvoemmer(this);
   }
}
```

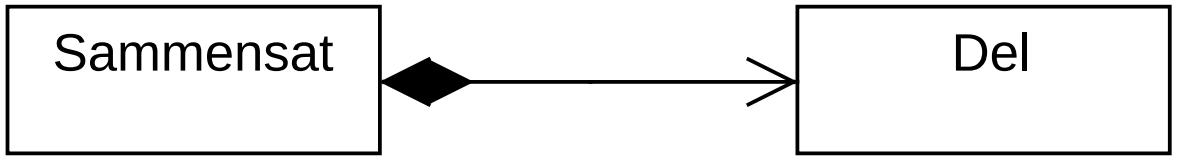
Step 5: Implement Klub

Additions to class Klub

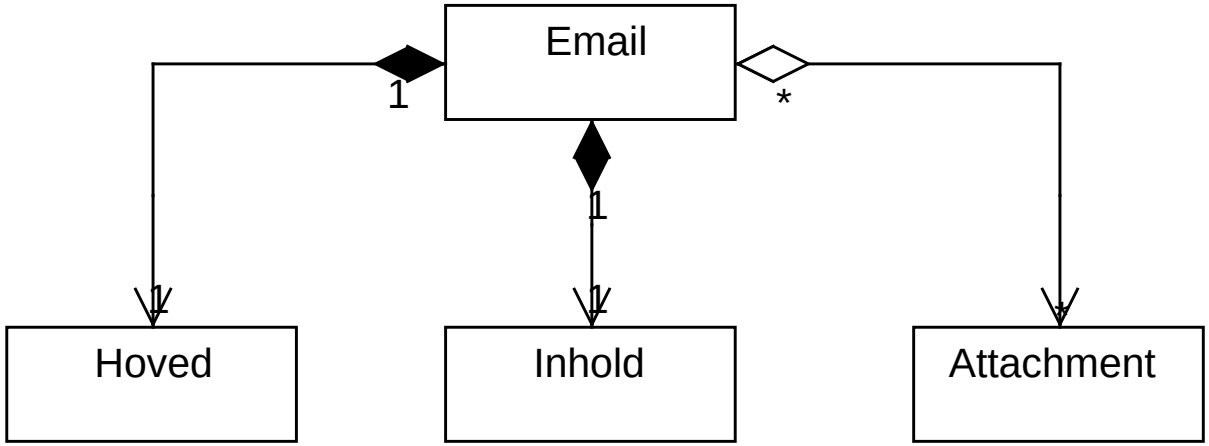
```
public void sletSvoemmer(Svoemmer svoemmer) {
   if (!medlemmer.contains(svoemmer))
      return;
   medlemmer.remove(svoemmer);
   svoemmer.setK(null);

}

protected void sletS(Svoemmer svoemmer) {
   medlemmer.remove(svoemmer);
}
```

Composite Aggregation (I)

En speciel relation *del af* mellem objekter



Eksempel: En email består af et hoved, et indhold og en række attachments



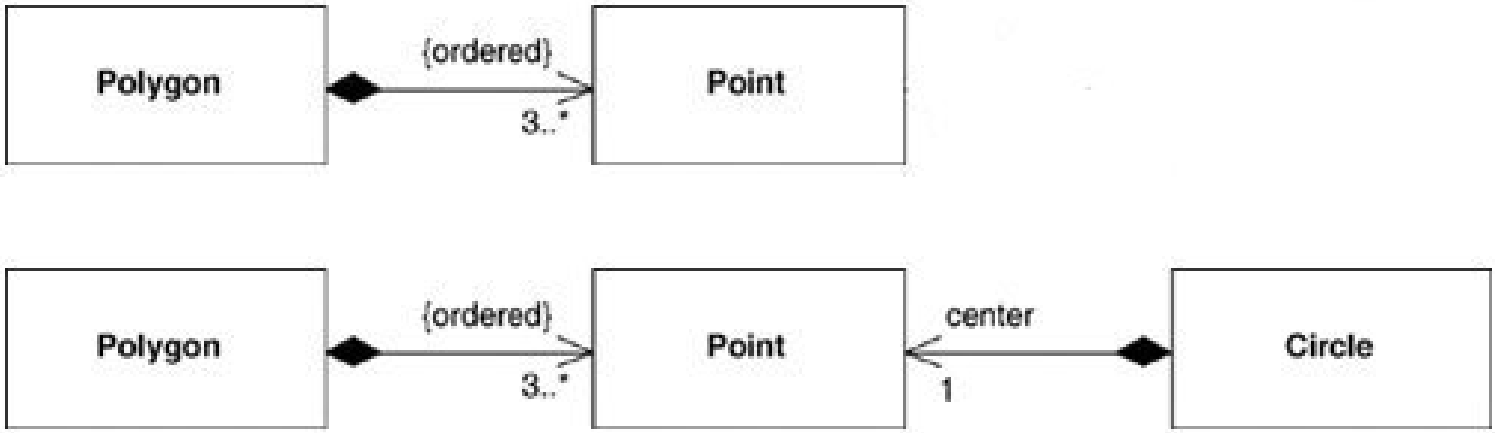
Composite Aggregation (II)

- 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

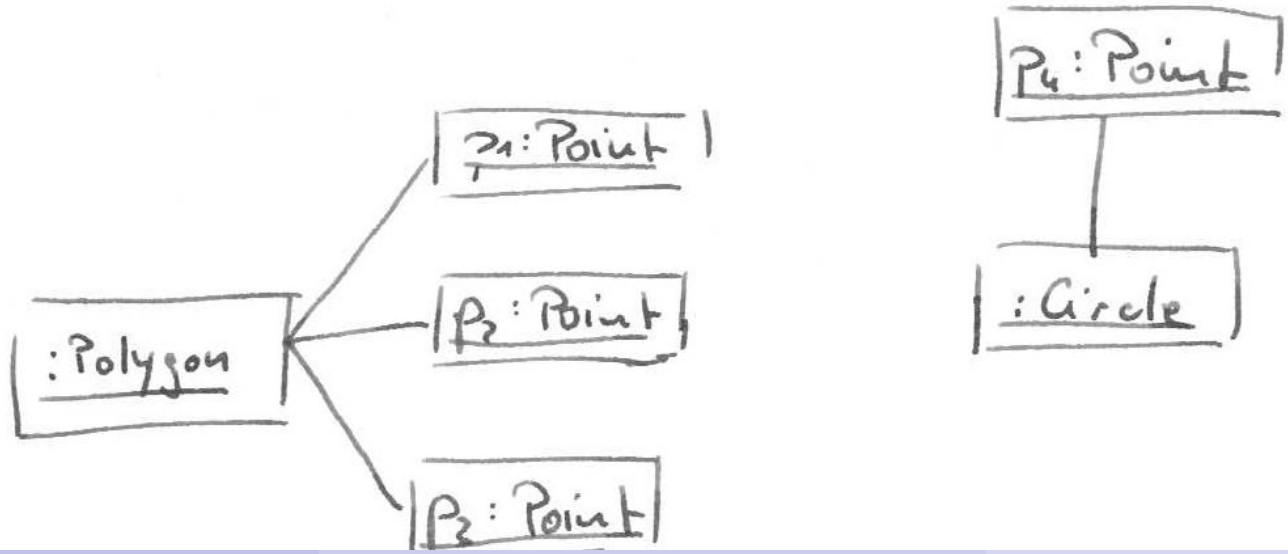
→ This results in requirements to the implementation

Composite Aggregation (III)

- A part can only be part of one object

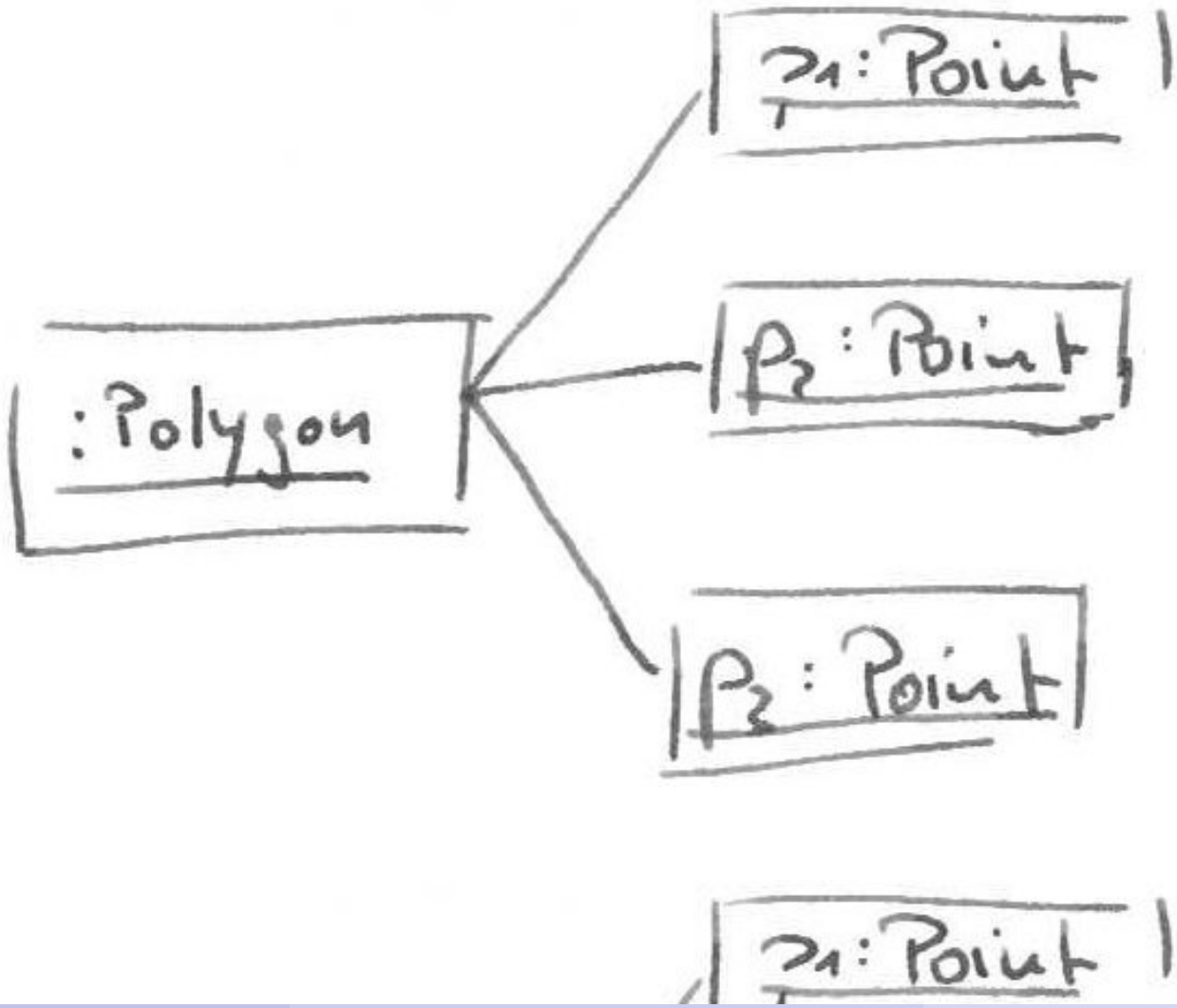


- Allowed



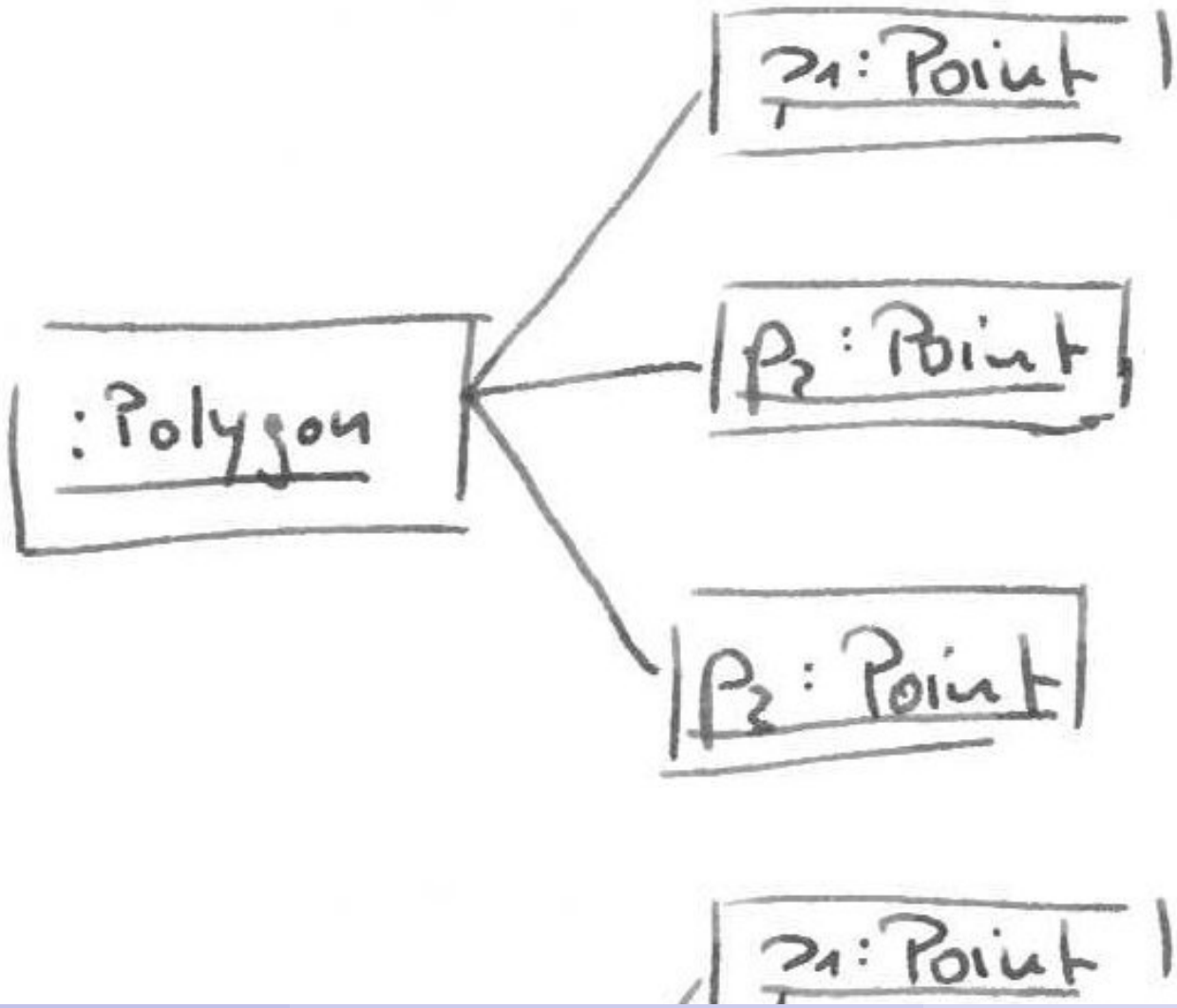
Composite Aggregation (IV)

- 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



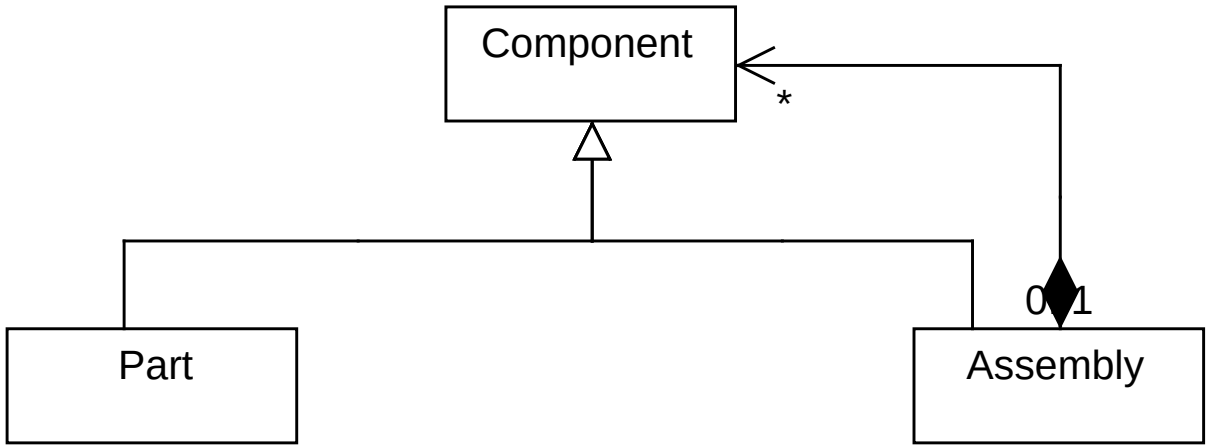
Composite Aggregation (V)

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



Composite Aggregation (VI): Styklister

- En kompleks komponent (assembly) kan ikke have sig selv som komponent.
- Styklister kan derfor modelleres bedre ved brug af aggregering



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)

Opsummering

- beskrivelser af klasser
- ensrettede associationer
- dobbeltrettede associationer
- aggregering
- komposition
- implementering