Software Engineering I (02161)

Week 5

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2013

Contents

Refactoring

- Restructure the program without changing its functionality
- Goal: improved design
- Necessary step in agile processes and test-driven development (TDD)
- Requires: sufficient (automated) tests

goal code coverage

⇒ safety net

Refactoring

- Book: *Refactoring: Improving the Design of Existing Code*, Martin Fowler, 1999
- Set of refactorings
  - e.g. **renameMethod**, **extractMethod**, **encapsulateField**, **encapsulateCollection**, ...
  - → complete list `http://www.refactoring.com/catalog/index.html`
- Set of code smells
  - e.g. Duplicate Code, Long Method, Large Class, Long Parameter List, ...
  - → `http://c2.com/cgi/wiki?CodeSmell`, or `http://www.codinghorror.com/blog/2006/05/code-smells.html`
  - How to write unmaintainable code `http://thc.org/root/phun/unmaintain.html`
- Decompose large refactorings into several small refactorings
  - Each step: compiles and passes all tests
- IDE's have tool support for some refactorings

Example refactoring: RenameMethod

- Motivation
  - Sometimes a method name does not express precisely what the method is doing
  - This can hinder the understanding of the code; thus give the method a more intention revealing name
- Mechanics

System runs

1) Create a method with the new name
2) Copy the old body into the new method
3) In the old body replace the body by a call to the new method; compile and test
4) Find all the references to the old method and replace it with the new name; compile and test
5) Remove the old method; compile and test

→ Supported directly in some IDE's

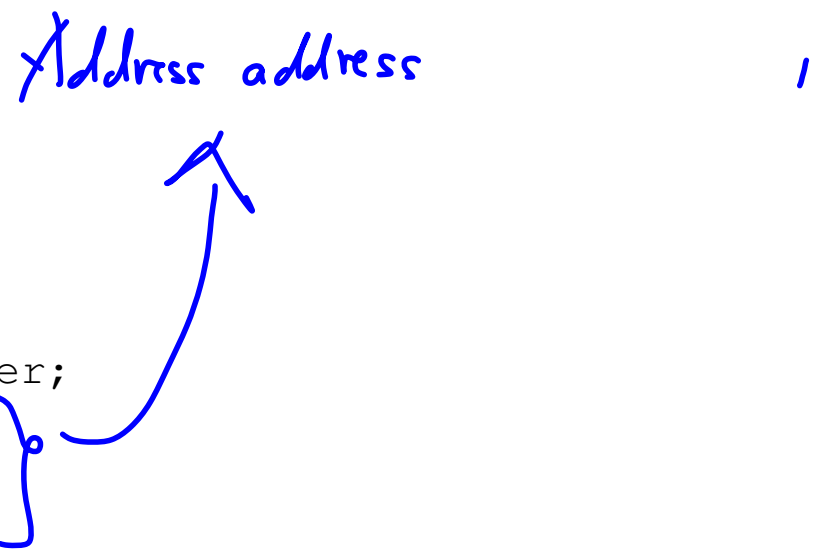
Code smells

If it stinks, change it Refactoring, Martin Fowler, 1999

- Duplicate Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Parallel Inheritance
- Lazy Class
- Speculative Generalisation
- Temporary Field
- Message Chains
- MiddleMan
- Inappropriate Intimacy
- Alternative Classes With Different Interfaces
- Incomplete Library
- Data Class
- Refused Bequest
- Comments

Code Smell: Data Clumps

```
public class Person {
   private String name;
   private Calendar birthdate;
   private Company company;
   private String street;
   private String city;
   private String zip;
   ...
}

public class Company {
   private String name;
   private String vat_number;
   private String street;
   private String city;
   private String zip;
   ...
}
```



Code Smell: Switch Statement

```
public class User {
   public double computeFine() {
      double fine = 0;
      for (Medium m : borrowedMedia) {
         if (m.overdue) {
            switch (m.getType()) {
               case Medium.BOOK : fine = fine + 10; break;
               case Medium.DVD: fine = fine + 30; break;
               case Medium.CD: fine = fine + 20; break;
               default: fine = fine + 5; break;
            }
         }
      }
      return fine;
   }
}
```

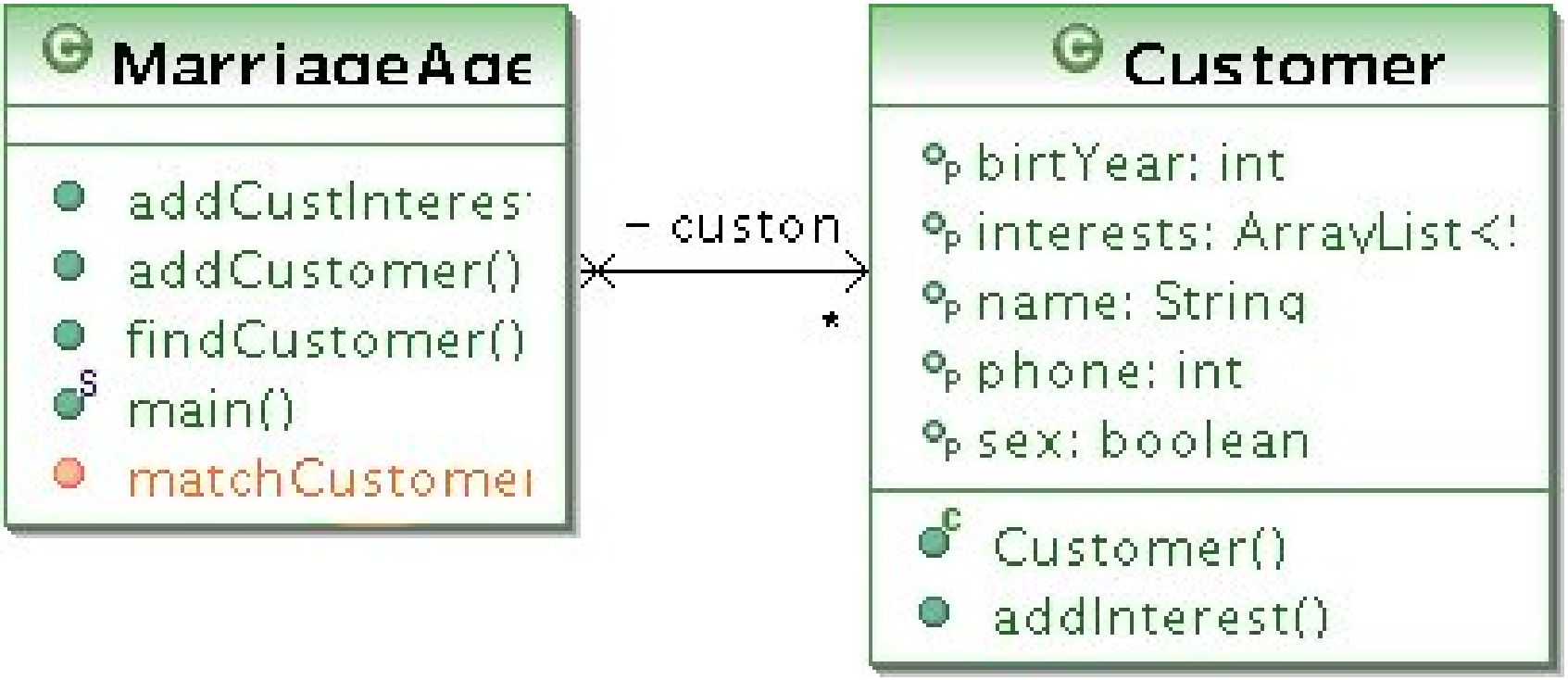
fine = fine + m.fine();

Better design

```
public class User {
   public double computeFine() {
      double fine = 0;
      for (Medium m : borrowedMedia) {
         if (m.overdue) { fine = fine + m.getFine();}
      }
      return fine;
   }
}

public class Medium {
   public double getFine() { return 5; }
}

public class Book extends Medium {
   public double getFine() { return 10; }
}

public class DVD extends Medium {
   public double getFine() { return 30; }
}

public class CD extends Medium {
   public double getFine() { return 20; }
}
```

Contents

MarriageAgency class diagram



- Refactoring example in detail
  - → `http://www2.imm.dtu.dk/courses/02161/2013/slides/refactoring_example.pdf`

Contents

UML

- Unified Modelling Language (UML)
- Set of graphical notations: class diagrams, state machines, sequence diagrams, activity diagrams, . . .
- Developed in the 90's
- ISO standard

Class Diagram

- Structure diagram of object oriented systems
- Possible level of details

Domain Modelling : typically low level of detail

⋮

Implementation : typically high level of detail

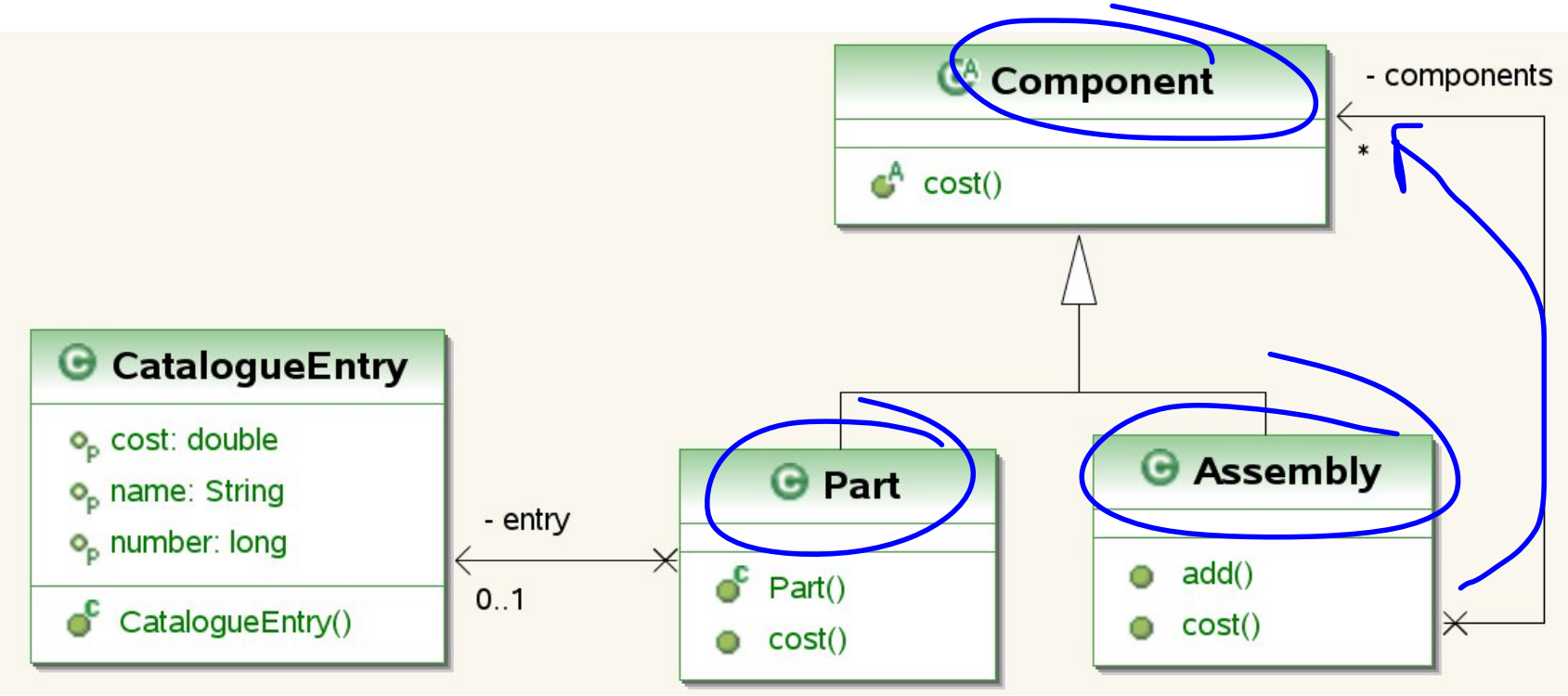
Why a graphical notation?

```
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() {}
}
```

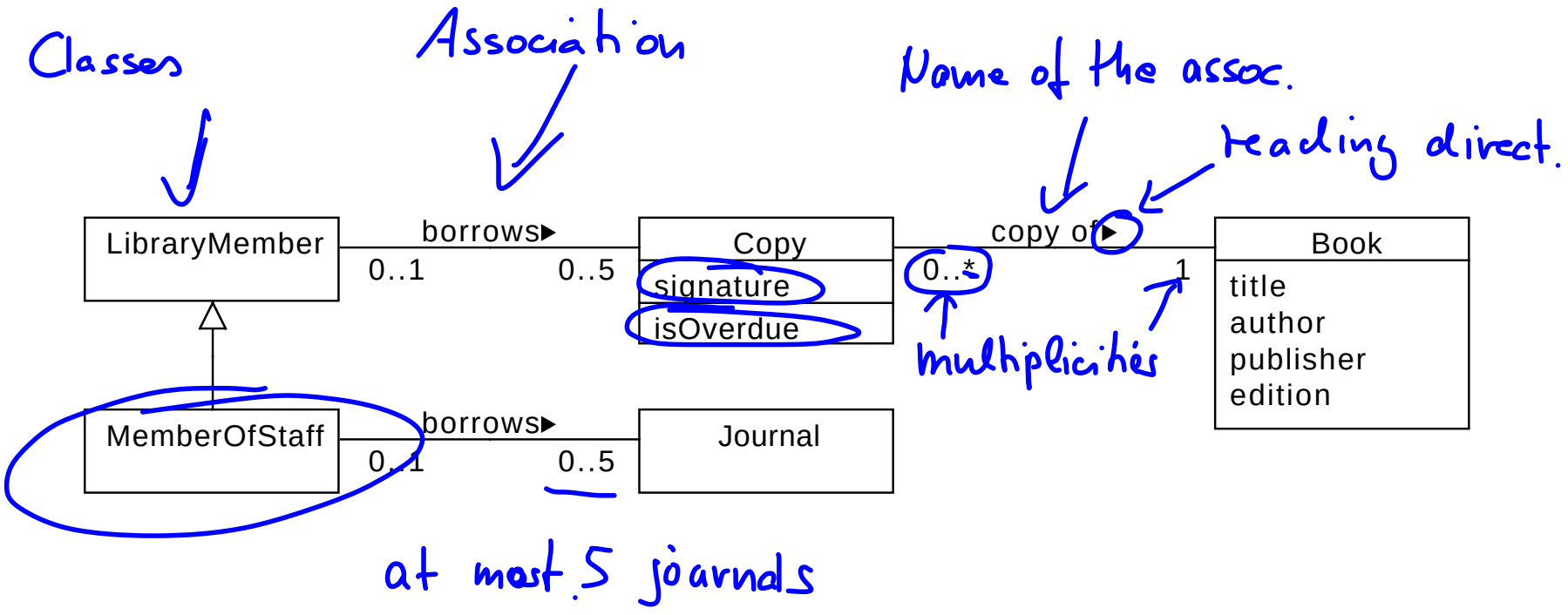
```
public abstract class Component {
 public abstract double cost();
}

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

Why a graphical notation?

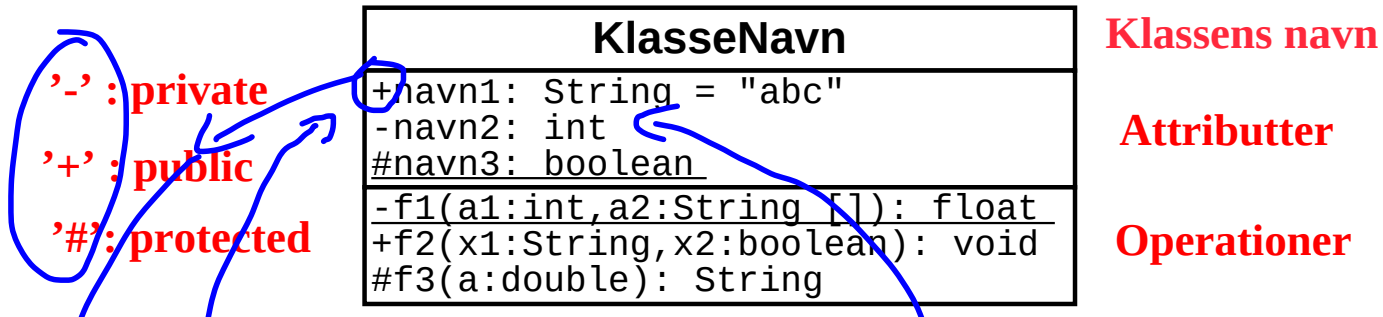
Composite pattern



Class Diagram Example



General correspondence between Classes and Programs

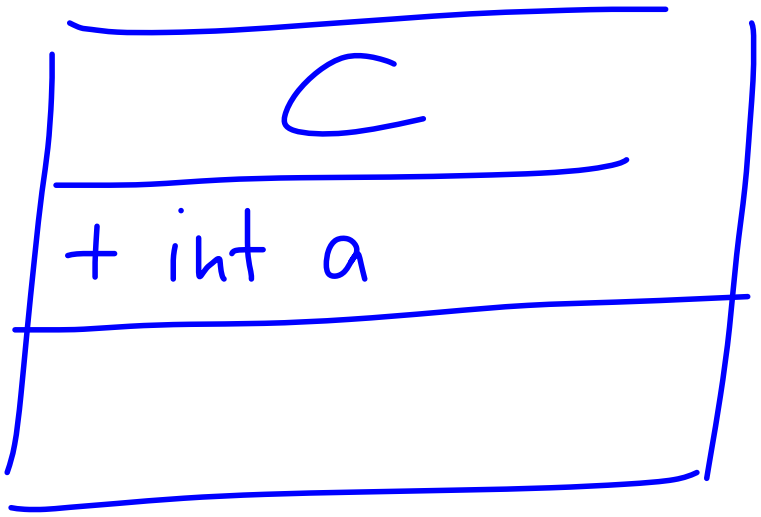



```
public class KlasseNavn
{
    private String navn1 = "abc";
    private int navn2;
    protected static boolean navn3;

    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) {...}
}
```

underline

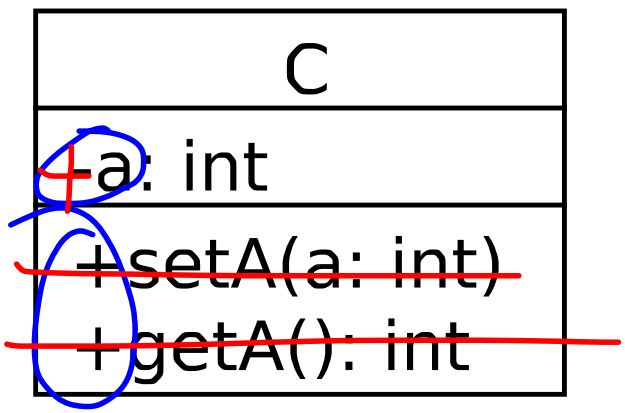
Class Diagram and Program Code

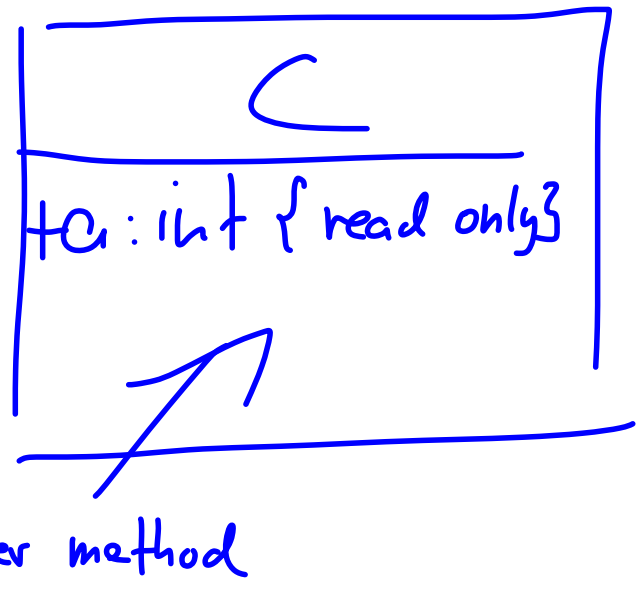
```
public class C {
  private int a;
  public int getA() { return a; }
  public void setA(int a) { this.a = a; }
}
```



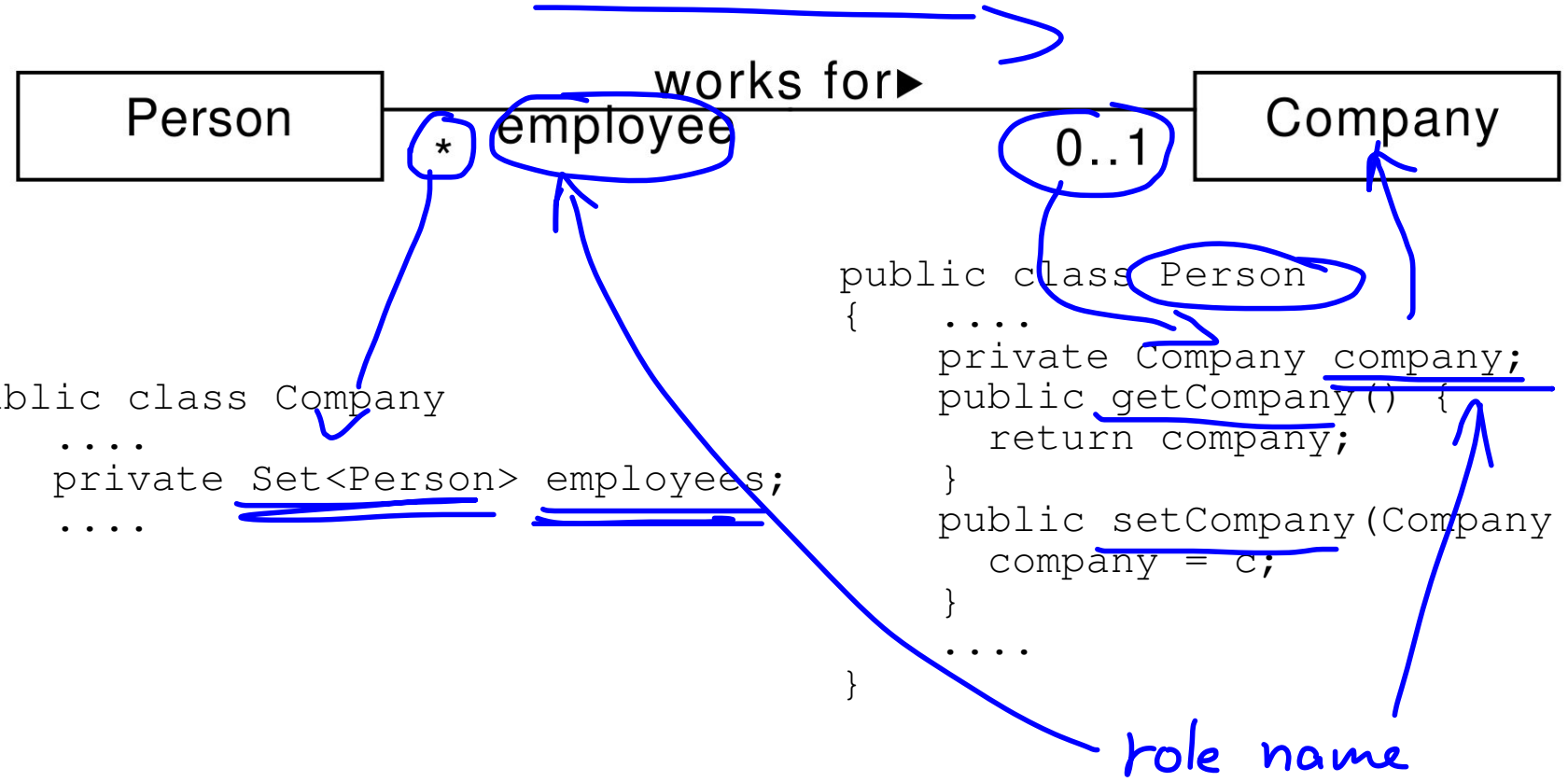
Class Diagram and Program Code

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

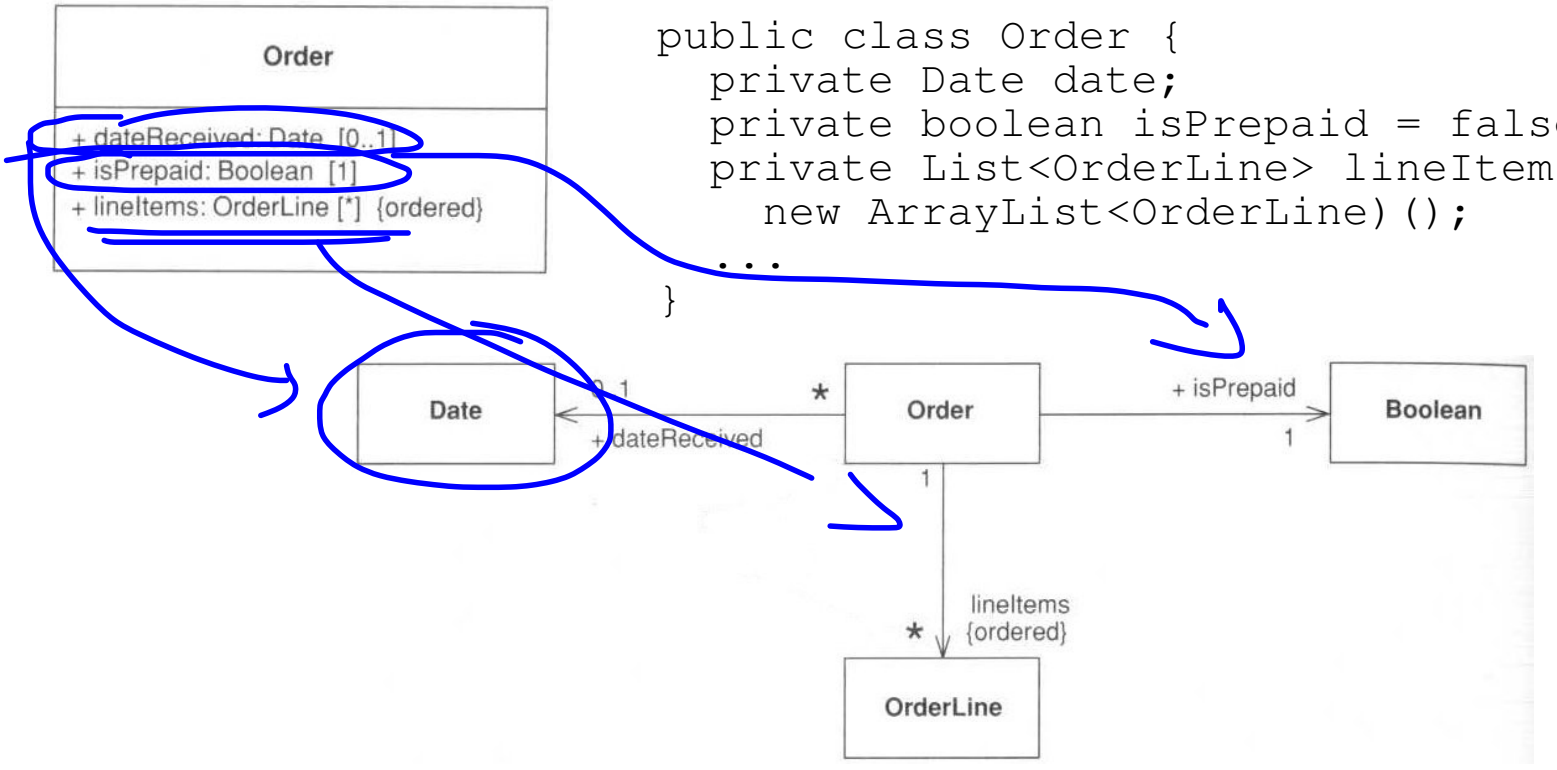




Associatons between classes

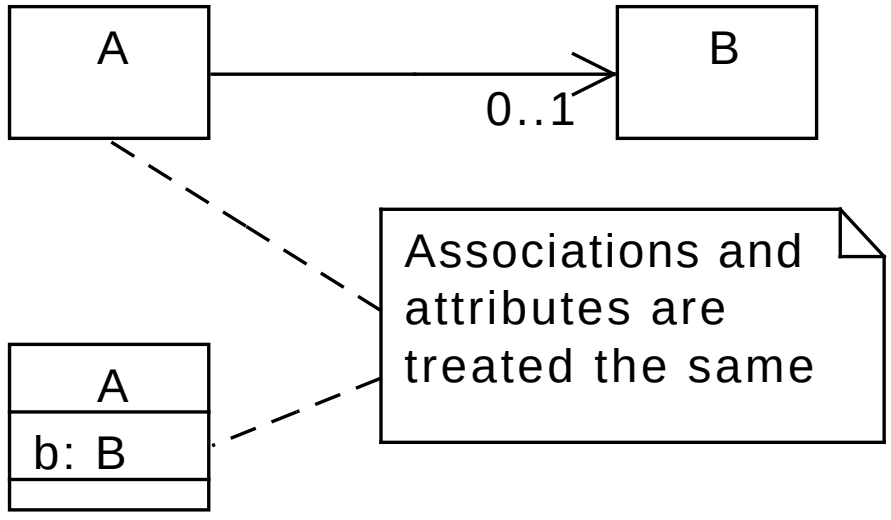


Attributes and Associations




```
public class Order {
  private Date date;
  private boolean isPrepaid = false;
  private List<OrderLine> lineItems =
    new ArrayList<OrderLine)();
  ...
}
```

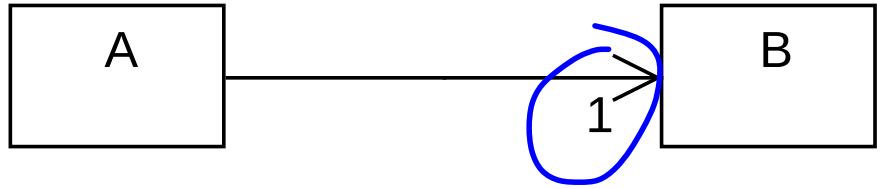
Implementing Associations: Cardinality 0..1



- Field can be null

```
public class A {

   private B b;

   public B getB() {
      return b;
   }

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

Implementing Associations: Cardinality 1



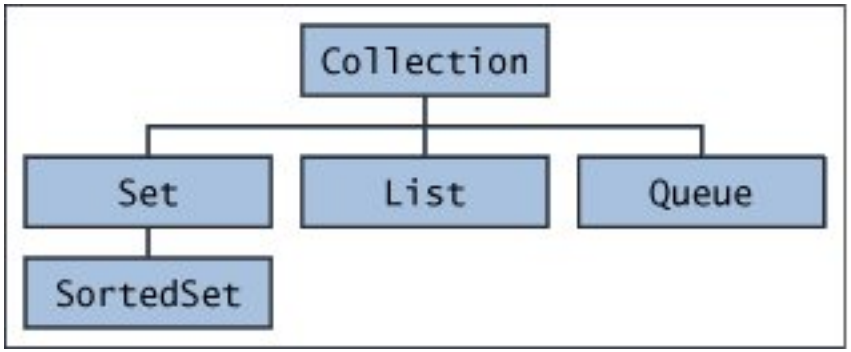
- Field may not be null

```
public class A {

   private B b = new B();

   public A(B b) { this.b = b;}

   public B getB() {
     if (b == null) {b = computeB();}
     return b;
   }

   public void setB(B b) { if (b != null) {this.b = b;} }
}
```

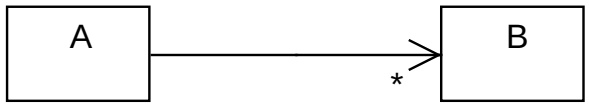
lazy initialization

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

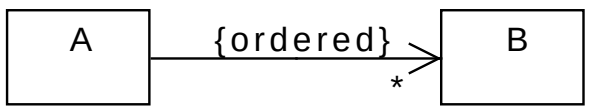


Implementing Associations: Cardinality *



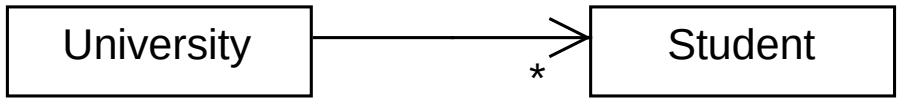
Default: Unordered, no duplicates

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



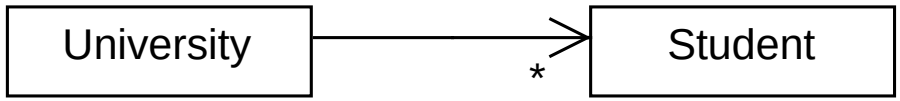

```
public class A {
   private List<B> bs = new ArrayList<B>();
   ...
}
```

Encapsulation problem: getStudents




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

Encapsulation problem: getStudents

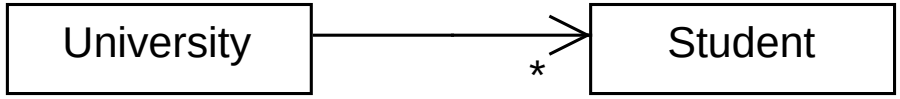



```
University dtu = new University("DTU");
..
Student hans = new Student("Hans");
Set<Student> students = dtu.getStudents();


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

Solution: getStudents returns an unmodifiable set

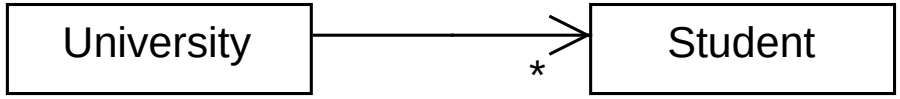
```
public void Set<Student> getStudents() {
   students = Collections.unmodifiableSet();
}
```

Encapsulation problem: setStudents




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

Encapsulation problem: setStudents

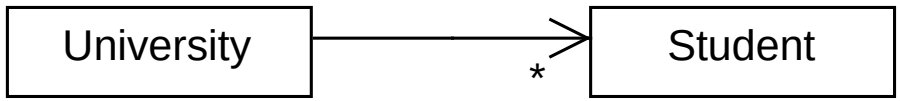



```
University dtu = new University("DTU");
..
Set<Student> students = new HashSet<Student>();
dtu.setStudents(students);


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

Solution: setStudents copies the set

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

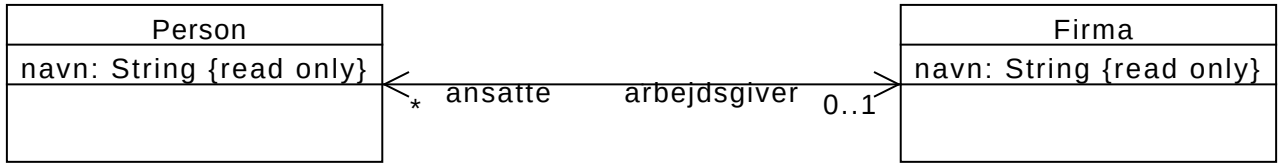
Solution: How to change the association?



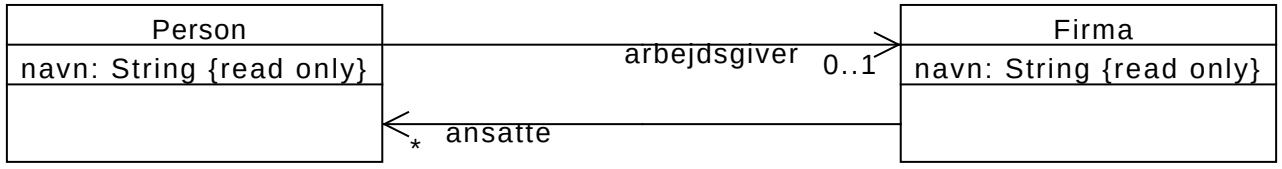

```
public class University {

   private Set<Student> bs = new HashSet<Student>();

   public void addStudent(Student s) {students.add(student);}
   public void containsStudent(Student s) {return students.contains(s)
   public void removeStudent(Student s) {students.remove(s);}
}
```

Even better: domain specific methods like *registerStudent*

Bi-directional associations



Implemented as two *uni-directional* associations



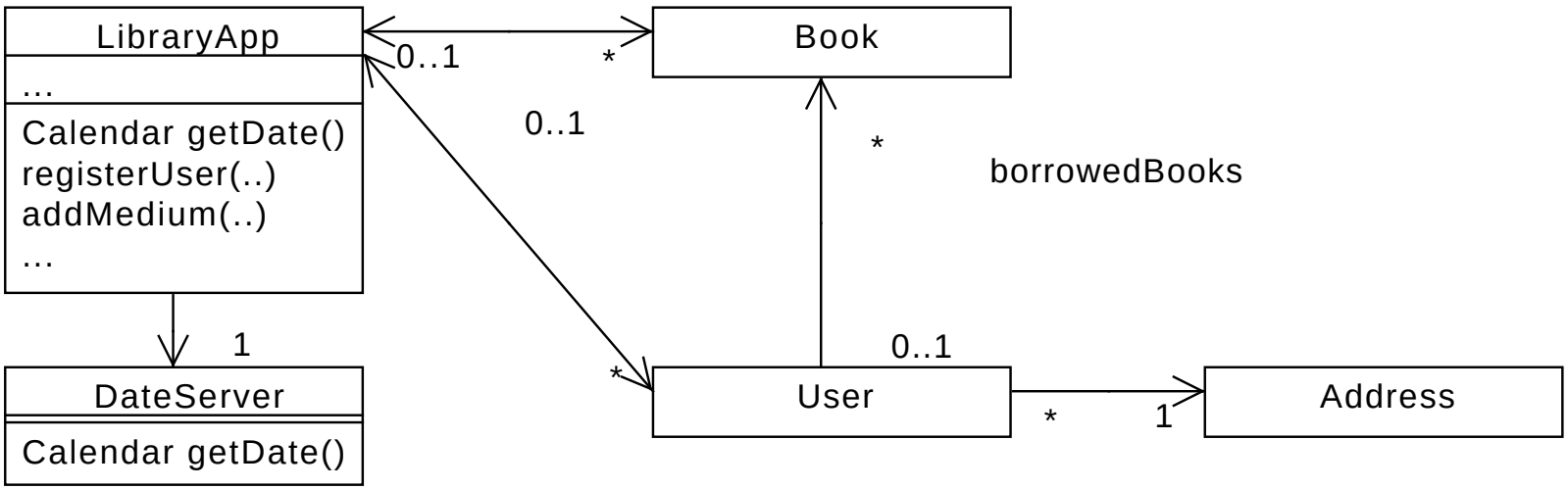
→ Problem of referential integrity

Referential Integrity

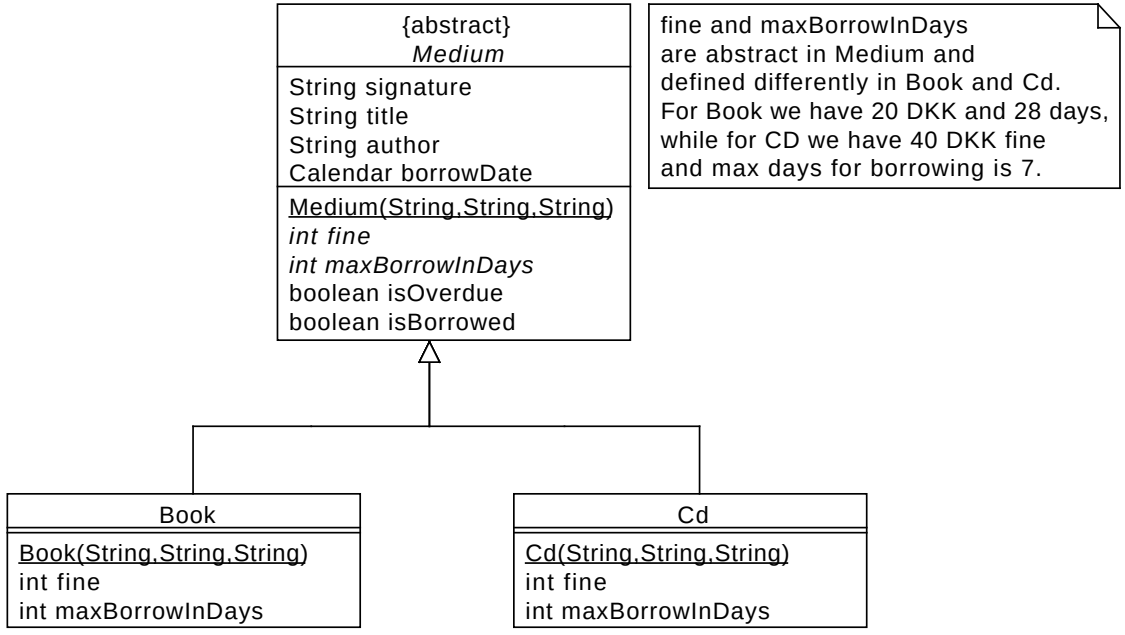
Referential Integrity: setArbejdsgiver

Referential Integrity: addAnsatte

Library application



Generalisation Example



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).*”

Contents