Software Engineering I (02161)

Week 6

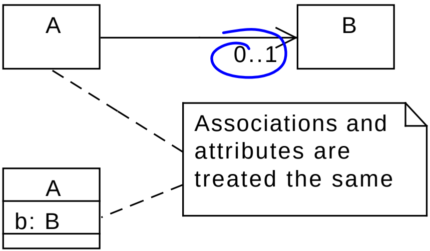
Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2018

Contents

Implementing Associations: Cardinality 0..1

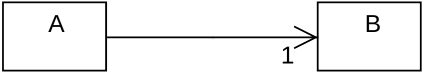


Associations and attributes are treated the same

- Field can be null

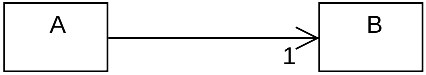
```
public class A {

   private B b;

   public B getB() { return b; }

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

Implementing Associations: Cardinality 1



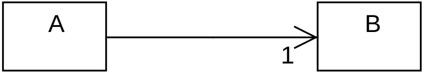
```
private B b = new B();

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

static public A constructA(B b) {
    if (b== null) { return null; }
    A a = new A();
    a.setB(b);
    return a; }
```

Implementing Associations: Cardinality 1



- Field may not be null

Implementing Associations: Cardinality 1



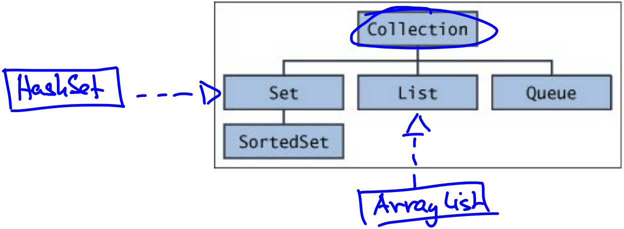
- Field may not be null

```
public class A {

   private B b = new B(); // 1st way of doing it

   public A(B b) { this.b = b;} // 2nd way

   public B getB() {              // 3rd way
     if (b == null) {b = computeB();}
     return b;
   }

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

" lazy initialization "

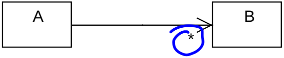
Interface *Collection<E>*

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



Interfaces

Implementing Associations: Cardinality *

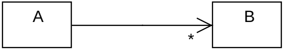


Default: Unordered, no duplicates

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

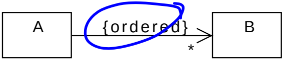
List<B> bs

Implementing Associations: Cardinality *



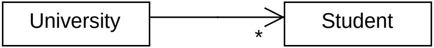
Default: Unordered, no duplicates

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



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

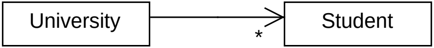
Encapsulation problem: getStudents



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

```
class University {
  private Set<Student>
          students = new
              HashSet();
  public Set<student>
   getStudents() {
      return students;
  }
```

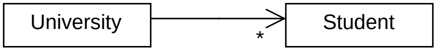
Encapsulation problem: getStudents



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

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

violation of encapsulation !

Encapsulation problem: getStudents

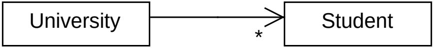


```
University dtu = new University("DTU");
..
Set<Student> students = dtu.getStudents();

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

Solution: getStudents returns an unmodifiable set

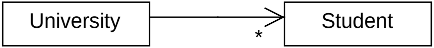
```
public void Set<Student> getStudents() {
   return Collections.unmodifiableSet(students);
}
```

Encapsulation problem: setStudents



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

Encapsulation problem: setStudents

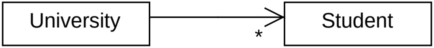


```
University dtu = new University("DTU");
..
Set<Student> students = new HashSet<Student>();
dtu.setStudents(students);

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

Solution: no setStudents or setStudents copies the set

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

Solution: How to change the association?

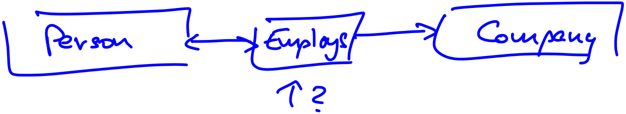


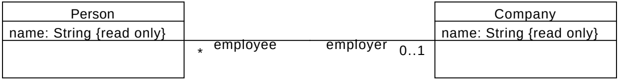
```
public class University {

   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);}
}
```

registerStudent

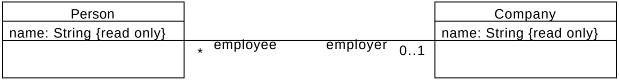
domain specific
method

Bi-directional associations

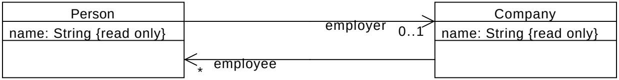




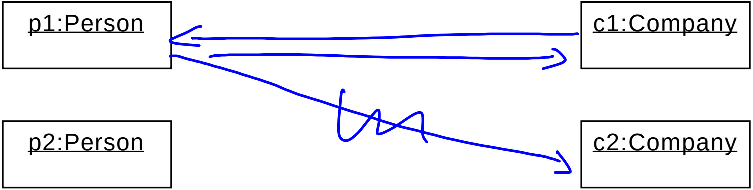
Bi-directional associations



Implemented as two *uni-directional* associations



Referential Integrity



Referential Integrity

p1:Person

c1:Company

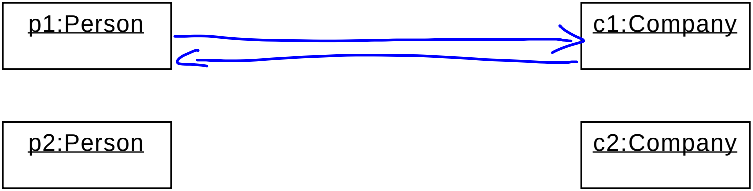
p2:Person

c2:Company

Referential Integrity:

$$
\begin{aligned}
&\forall c : \mathit{Company} : \forall p : \mathit{Person} \\
&\qquad p \in c.\mathit{employee} \implies p.\mathit{company} = c \\
&\qquad\qquad \wedge \\
&\qquad p \in p.\mathit{company}.\mathit{employees}
\end{aligned}
$$

Referential Integrity: setEmployer



p1:Person

c1:Company

p2:Person

c2:Company

Referential Integrity: setEmployer

p1:Person

c1:Company

p2:Person

c2:Company

In a client

```
Person p = new Person();
Company c = new Company();
p.setEmployer(c);
c.addEmployee(p);
```

DRY

Don't repeat yourself.

Referential Integrity: setEmployer

p1:Person

c1:Company

p2:Person

c2:Company

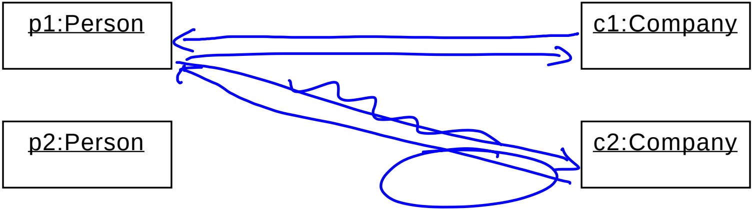
In a client

```
   Person p = new Person();
   Company c = new Company();
   p.setEmployer(c);
   c.addEmployee(p);
```

better: In Person

```
public void setEmployer(Company c) {
   employer = c;
   c.addEmployee(this);
}
```

Referential Integrity: addEmployee



```
public void addEmployee(Person p) {
   employees.add(p);
   p.setEmployer(this);
}
```

Referential Integrity: implementation

```
public void setEmployer(Company c) {
   employer = c;
   c.addEmployee(this);
}
```

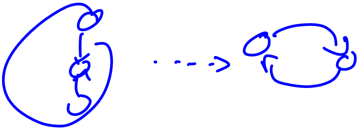
```
public void addEmployee(Person p) {
   employees.add(p);
   p.setEmployer(this);
}
```

Referential Integrity: implementation

```
public void setEmployer(Company c) {
   employer = c;
   c.addEmployee(this);
}

~~public~~ void addEmployee(Person p) {
   employees.add(p);
   p.setEmployer(this);
}
```

```
public void setEmployer(Company c) {
   if (employer == c) { return; }
   employer = c;
   c.addEmployee(this);
}

public void addEmployee(Person p) {
   if (employees.contains(p) {
      return;
   }
   employees.add(p);
   p.setEmployer(this);
}
```

Referential Integrity: implementation



```
public void setEmployer(Company c) {
   employer = c;
   c.addEmployee(this);
}

public void addEmployee(Person p) {
   employees.add(p);
   p.setEmployer(this);
}
```

```
public void setEmployer(Company c) {
   if (employer == c) { return; }
   employer = c;
   c.addEmployee(this);
}

public void addEmployee(Person p) {
   if (employees.contains(p) {
      return;
   }
   employees.add(p);
   p.setEmployer(this);
}
```

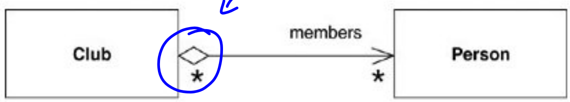
Summary

- Avoid bi-directional associations if possible
- Don’t rely on that the clients will do the bookkeeping for you

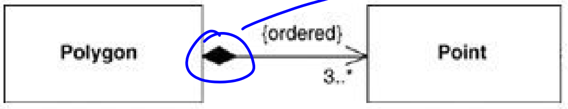
Part of relationship

Special type of associations application specific semantics

- aggregation



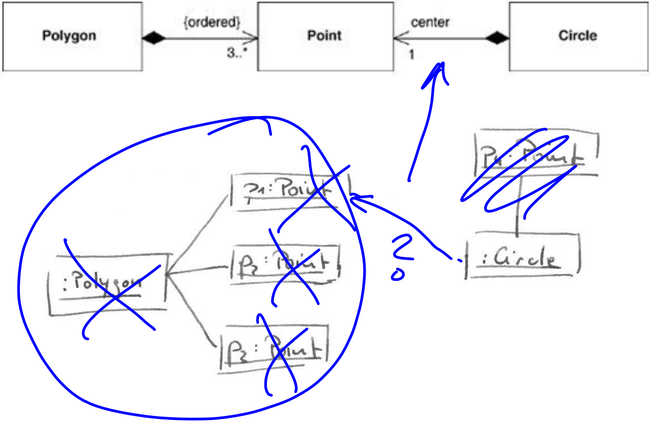
- composition (composite aggregation) → fixed: meaning



- Use part_of instead of has_a
  - → A car has an engine = an engine is part of the car
  - → But Peter has a house != the house is part of Peter

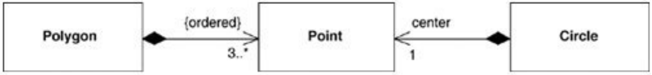
Composition

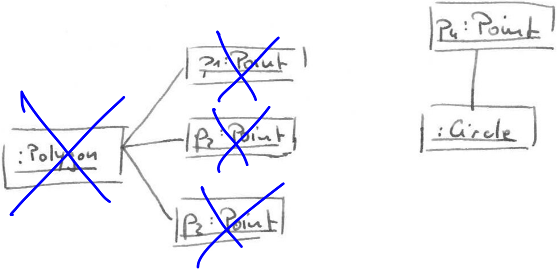
1. *A part can only be part of one object*
2. *The life of the part object is tied to the life of the containing object*



Composition

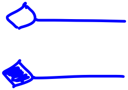
1. *A part can only be part of one object*
2. *The life of the part object is tied to the life of the containing object*





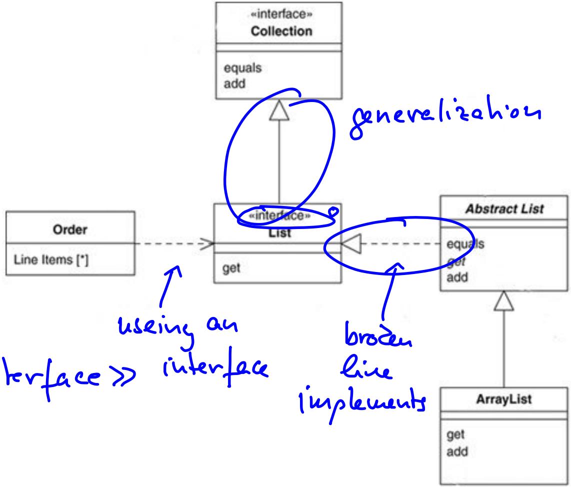
Composition: Implementation issues

- Important concept with C++: No automatic garbage collection
    - Destructor has to destroy parts
    - Rule of thumb: Don’t expose the parts to the outside
- Not as relevant in Java: Java has automatic garbage collection
- Rule of thumb: Don’t use composition unless you need its semantics



Contents

Interfaces



generelization

useing an interface

<<interface>>

broken line implements

Contents

Implementing Associations

Interfaces

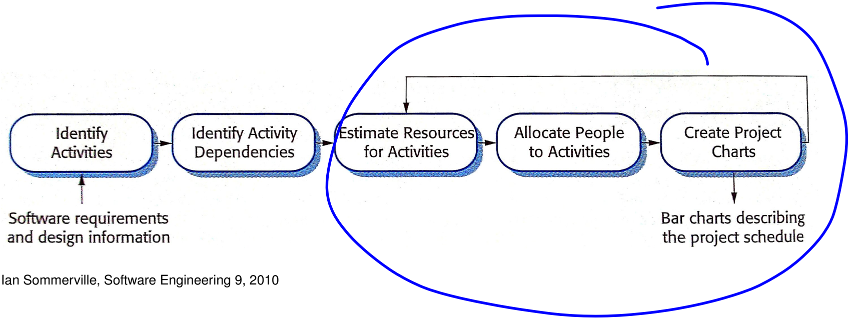
Project planning

Project

Project Planning

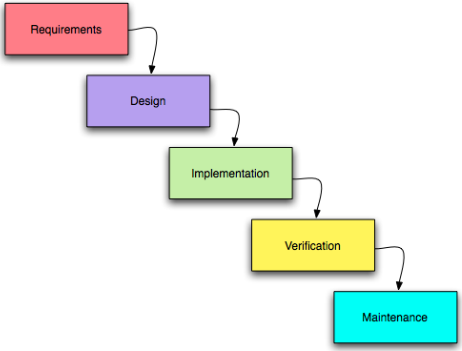
- Project plan
  - Defines how work is done
  - Estimates resources (time, person/months): price
- Project planning
  - Proposal stage: Price, Time to finish
  - During the project: Progress tracking, Adapt to changes

Traditional Project scheduling



Ian Sommerville, Software Engineering 9, 2010

Traditional Processes



Requirements

Design

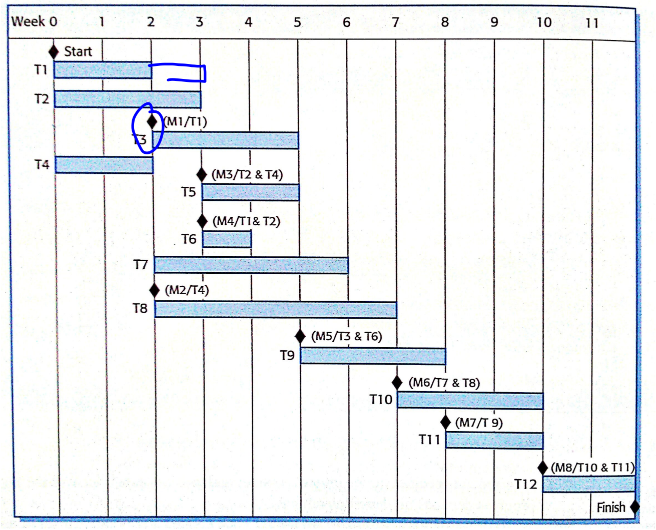
Implementation

Verification

Maintenance

- milestones/deliverables: system specification, design specification, . . .
- Typical tasks: Work focused on system components

Schedule Representation: Gantt Chart / Bar chart



Ian Sommerville, Software Engineering 9, 2010

Traditional: Algorithmic cost modelling: COCOMO

- Constructive Cost Model (COCOMO) Bary Boehm et al., 1981, ...
  - based on empirical studies
- LOC (lines of code) estimation
  - e.g. function point analysis based on requirements: complexity of functions and data
- Effort: in person months: $PM = a * LOC^b$
  - a: type of software: $2.4 \leq a \leq 3.6$
  - b: cost drivers like platform difficulty, team experience, ...: $1 \leq b \leq 1.5$
- Project duration: $TDEV = 3 * PM^{0.33+0.2*(b-1.01)}$
- Staffing: $STAFF = PM/TDEV$

Traditional: Algorithmic cost modelling: COCOMO

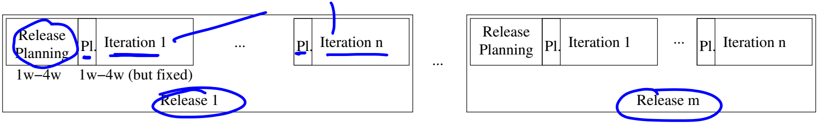
- Constructive Cost Model (COCOMO) Bary Boehm et al., 1981, . . .
  - based on empirical studies
- LOC (lines of code) estimation
  - e.g. function point analysis based on requirements: complexity of functions and data
- Effort: in person months: $PM = a * LOC^b$
  - a: type of software: $2.4 \leq a \leq 3.6$
  - b: cost drivers like platform difficulty, team experience, . . . : $1 \leq b \leq 1.5$
- Project duration: $TDEV = 3 * PM^{0.33+0.2*(b-1.01)}$
- Staffing: $STAFF = PM/TDEV$
- Brooks law: ”adding human resources to a late software project makes it later”. (*The Mythical Man Month* Fred Brooks 1975)

Planning Agile Projects

- *fixed* general structure

→ quarterly cycle / weekly cycle practices in XP / sprints in Scrum

collection of user stories



Release Planning

Pl.

Iteration 1

...

Pl.

Iteration n

1w–4w

1w–4w (but fixed)

Release 1

3m–6m

...

Release Planning

Pl.

Iteration 1

...

Pl.

Iteration n

Release m

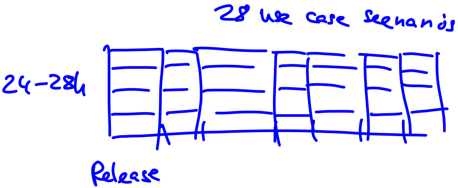
- *time boxing*
  - fixed: release dates and iterations
  - adjustable: scope
- Planning: Which user story in which iteration / release

Planning game

- Customer defines:
    - user stories
    - priorities (e.g. MoSCoW)
- Developer define:
    - costs, risks
    - suggest user stories
- Customer decides: is the user story worth its costs?
    → split a user story
    → change a user story
- Result: Release / Iteration plan

Scrum/XP: User story estimation (based on ideal time)

- Estimation
  - Estimate *ideal_time* (e.g. person hours / week) to finish a user story
  - real_time = ideal_time * load_factor (e.g. load_factor = 2)
  - Add user stories to an iteration based on real_time and priority



6-7h per student per week

4 students

24-28h per group per week

7 · 24-28

168-196h ~~ideal~~ time available

3h per use case scenario

* load factor (2)

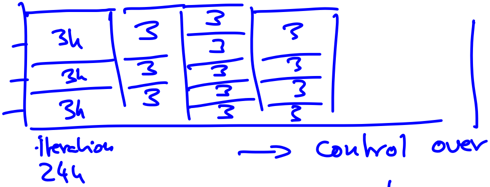
6h

Scrum/XP: User story estimation (based on ideal time)

- Monitoring
  - New *load factor*: total_iteration_time / user_story_time finished

  → What can be done in the next iteration
- Yesterdays weather focus only on previous iteration
- Focus on *few* stories and finish them (time boxing)

highest priority first



→ control over future user stories

load factor $= \frac{24}{\textcircled{12}} = 2$

$\textcircled{12} = 4 \cdot \underline{\underline{3}}$

new load factor $= \frac{24}{9} = \underline{\underline{2{,}7}}$

$9 = 3 \cdot 3$

now load factor $= \frac{24}{15} = 1{,}6$

Contents

Course 02161 Exam Project

- Week 6 – 8: Report 1
  - Requirements: Glossary, use case diagram, detailed use cases (i.e. cucumber scenarios)
  - Draft design: Class diagram + sequence diagrams
- Week 8 – 9: Report 2
  - Peer review of report 1
- Week 8—13:
  - Implementation
  - Systematic tests and design by contract
- Week 13: Report 3, Source code
  - 10 min demonstrations of the tests

Introduction to the project

- Problem:
  - Design and implement a project planning and time recording system
  - UI required, but not a graphical UI; storage of data in database or in a file is not required
- Deliver
  - Sa 17.3: report 1: requirement specification and design
  - Su 25.3: report 2: peer review of another groups report 1
  - Week 13:
    - report 3: systematic tests, design by contract
    - **Eclipse** project: source code, tests, running program (**ZIP** file that can be imported in Eclipse)
    - demonstration in front of TA's (**participation mandatory**; does not contribute to final grade)
- More detail on CampusNet

Organisational issues

- Group size: 4
- Reports can be written in Danish or English
- Program written in Java with Eclipse and tests use Cucumber and JUnit
- Each section, diagram, etc. needs to name the author who made the section, diagram, etc.
- **You can talk with other groups (or previous students that have taken the course) on the assignment, but *it is not allowed to copy from others parts of the report or the program*.**
  - *Any copying of text without naming the sources is viewed as cheating*
- In case of questions with the project description ask on Piazza or send email to `huba@dtu.dk`

Week 6+7: Requirements and Design

Recommended design process

1 Create glossary, use cases, and domain model
2 Identify use case scenarios and their priority
3 Create a set of initial classes based on the domain model → initial design
3 Take one user story
  - a) Design the system by executing the user story in your head
    - → e.g. using CRC cards (next week)
  - b) Extend the existing class diagram with classes, attributes, and methods → Refactor your design
  - c) Document the scenario using a sequence diagram
3 Repeat step 2 with the other use case scenarios

- Pareto principle: 20% of the work gives 80%
- Model does not have to be perfect: Guides implementation

Week 8: Peer Review the models of your colleagues

Criteria to check for

- Correct notation (use case diagram, class diagram, sequence diagrams)
- Consistency and completeness
    - use case names in use case diagrams and detailed use cases
    - glossary explains terminolgoy used in detailed use cases
    - sequence diagrams fit to the use case scenarios
    - use case diagram describes the complete behaviour of the system
    - …
- Readability
    - Do you understand the model?

Learning objectives of Week 6—8

- Learn to think abstractly about object-oriented programs
  - Programming language independent
- Learn how to communicate requirements and design
  - Requirements are read by the customer and the programmers
  - Talk with fellow programmers about design: class and sequence diagrams
- I don't expect you to create perfect models
  - I expect your final implementation will differ from your model
  - → Comparing your model with your final implementation: you learn about the relationship between modelling and programming

Week 9—13

Recommended implementation process

1 Choose a set of use case scenarios to implement

1 Select the use case scenario with the highest priority

  a) Create the Cucumber test for it

  b) Implement the use case scenario test-driven, creating additional tests (Cucumber as well as JUnit) as necessary

    - **guided** by your design

    → based on the classes, attributes, and methods of the model

    → implement **only** the classes, attributes, and methods needed to implement the user story

    → Criteria: ideally 100% code coverage of the business logic (i.e. application layer) based on the tests you have

3 Repeat step 2 with the use case scenario with the next highest priority

  Remember: priorities can change

Grading

- The project will be graded as a whole
  - → no separate grades for the models, report, and the implementation
- Evaluation criteria
  - In general: correct use and understanding of the techniques introduced in the course
  - Implementation: good architecture, understandable code and easy to read (e.g. short methods, self documenting method names and variable names, use of abstraction)
  - Rather focus on a subset of the functionality with good code quality than on having everything implemented but with bad code quality
  - ”Sufficient tests and quality of tests”