Software Engineering I (02161)

Week 1

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2018

Contents

Course Introduction

Introduction to Software Engineering

Practical Information

Programming Assignment

The course

- 5 ECTS course 02161: Software Engineering 1
- Target group: Bachelor in Software Technology and Network Technology and IT in the second semester
- Learning objectives
  - Overview over software engineering: What is there more than programming?
  - Learn software engineering techniques
  - Communicate requirements, architecture, and design
  - Do a *smaller* project from an *informal* and *open description* of the problem

Who are we?

- 117 students with different backgrounds
  - Bachelor Software Technology: 73 (62%)
  - Bachelor Network Technology and IT: 22 (19%)
  - Other: 22 (19%)
- Teacher
  - Hubert Baumeister, Assoc. Prof. at DTU Compute (huba@dtu.dk; office 303B.058)
- 3 Teaching assistants
  - Sarah Dam
  - Theis Kierkegaard Hauge
  - Marcus Pagh

Course Language

- The course language is Danish; slides, notes, and other material mostly in English
- If everybody agrees to that, it can be given in English

Contents

Course Introduction

Introduction to Software Engineering
- Introduction
- Development Example

Practical Information

Programming Assignment

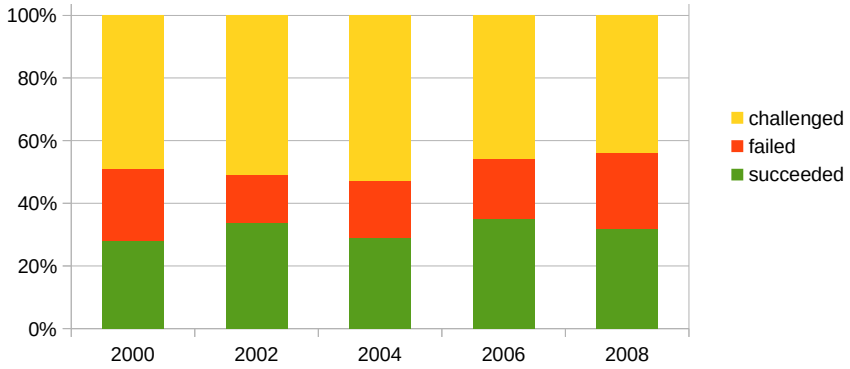
What is software?

- Software is everywhere
  - Desktop applications
  - Web applications
  - Embedded systems (IOT)
  - Large batch systems
  - Operatiing systems
  - Big-data
  - Artifical intelligence
  - ...
- Types of software
  - Mass production, Customised software, Mixture of both

Types of software (Lehmann)

- s-type: mathematical function, sorting: complete specfication
- p-type: real world problems, e.g., chess: modelling the real world
- e-type: embeded into socia-technical systems.
  Requirements change as the environment changes.
  System changes the environment: e.g., operating system
  - Continuing Change
  - Increasing Complexity
  - Conservation of Organisational Stability (invariant work rate)
  - Continuing Growth
  - Declining Quality

Success rate for software projects 2000—2008



CHAOS Summary 2009 Report

- Succeeded: 32%
- Failed: 20%
- Challenged: 48% (over time, over budget, . . .)

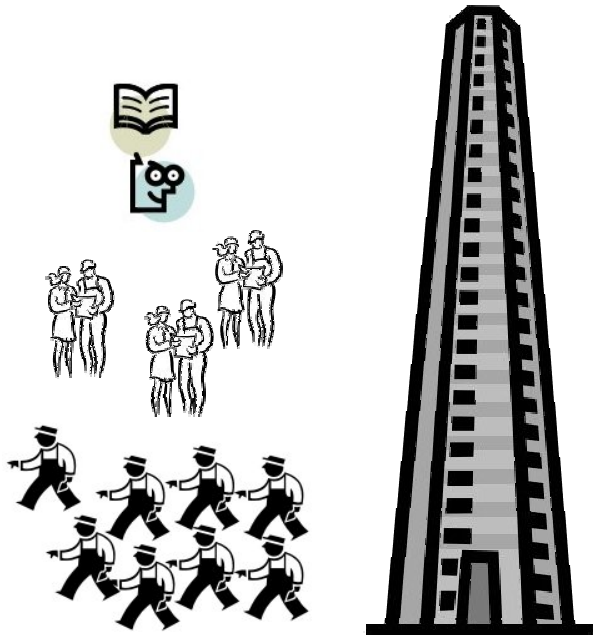
Failed or challenged

- Amanda
- Rejsekortet
- Obamacare Website
- German road toll system
- Denver airport baggage system
- . . .

Scaling software development

Small hut

- one person
- no special knowledge



Skyscraper

- not possible with one person
- special knowledge: static, electricity, water, waste, elevator, ...

Small software — large software: bug fixing

Small program

- find the defect
- fix the defect
- adjust documentation

Large software

- report defect
- collect defect reports
- analyse problem
- identify bug
- define a bug fixing strategy
- fix the bug
- testing: bug fixed; no new bugs
- accept the fixed version
- integrate parallel changes
- update release documentation
- release the new system

Software attributes

- Maintainability
  - Readable code (clean code, self documenting code, good documentation)
  - Reduce complexity (Design Pattern, low coupling/high cohesion)

Software attributes

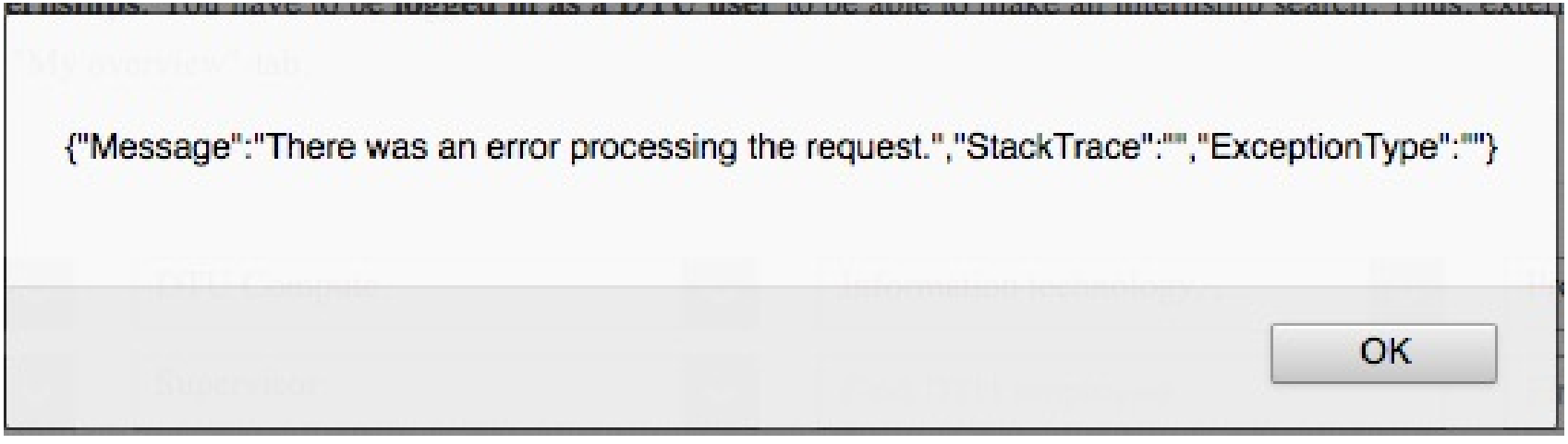
- Maintainability
  - Readable code (clean code, self documenting code, good documentation)
  - Reduce complexity (Design Pattern, low coupling/high cohesion)
- Dependability and security
  - Includes: reliability (robustness), privacy, and safety
  - Example: Apple root access on macOS

Software attributes

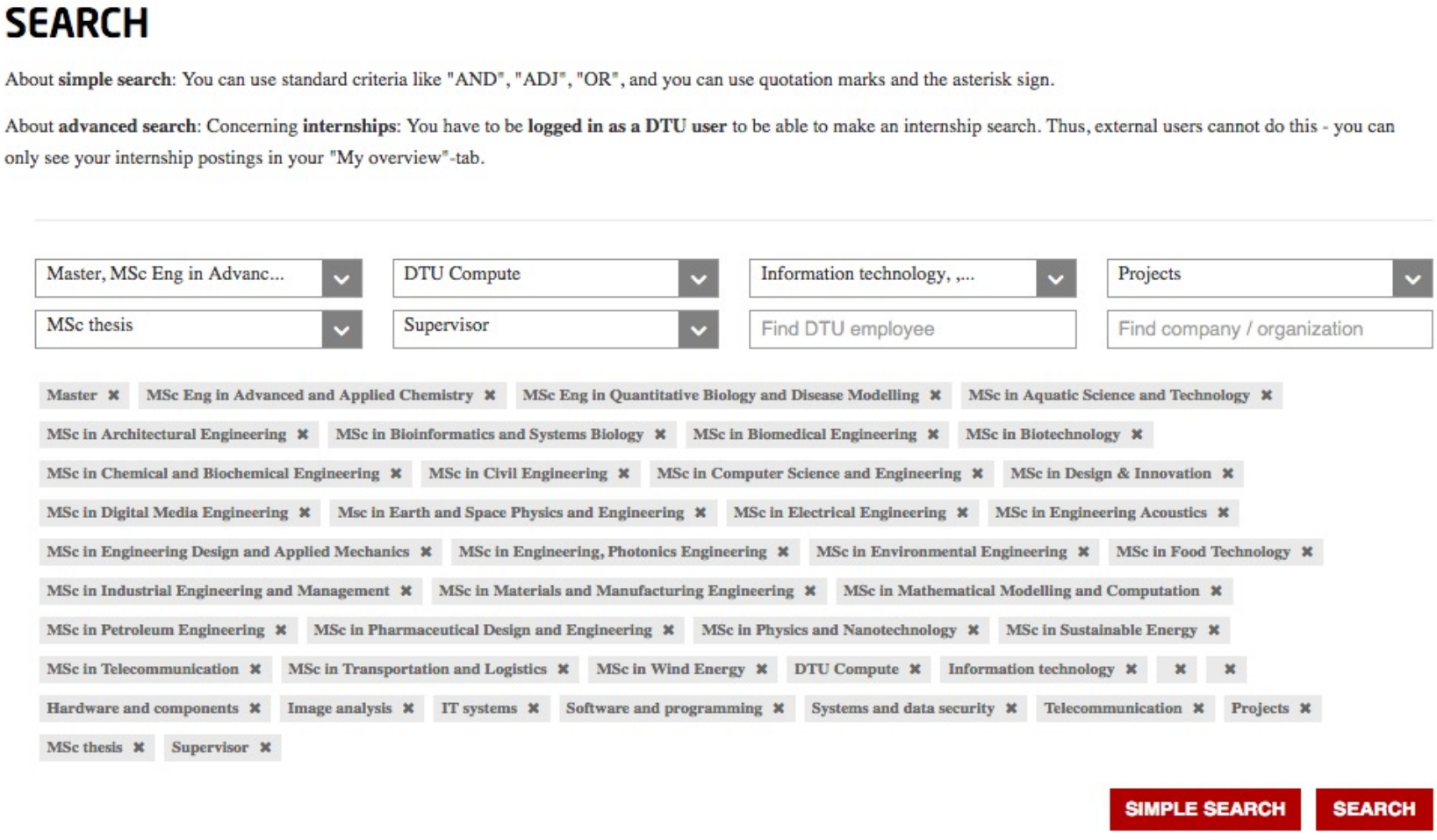
- Maintainability
  - Readable code (clean code, self documenting code, good documentation)
  - Reduce complexity (Design Pattern, low coupling/high cohesion)
- Dependability and security
  - Includes: reliability (robustness), privacy, and safety
  - Example: Apple root access on macOS
- Efficiency
  - Don’t waste system resources
  - Responsiveness, processing time, memory utilisation

Software attributes

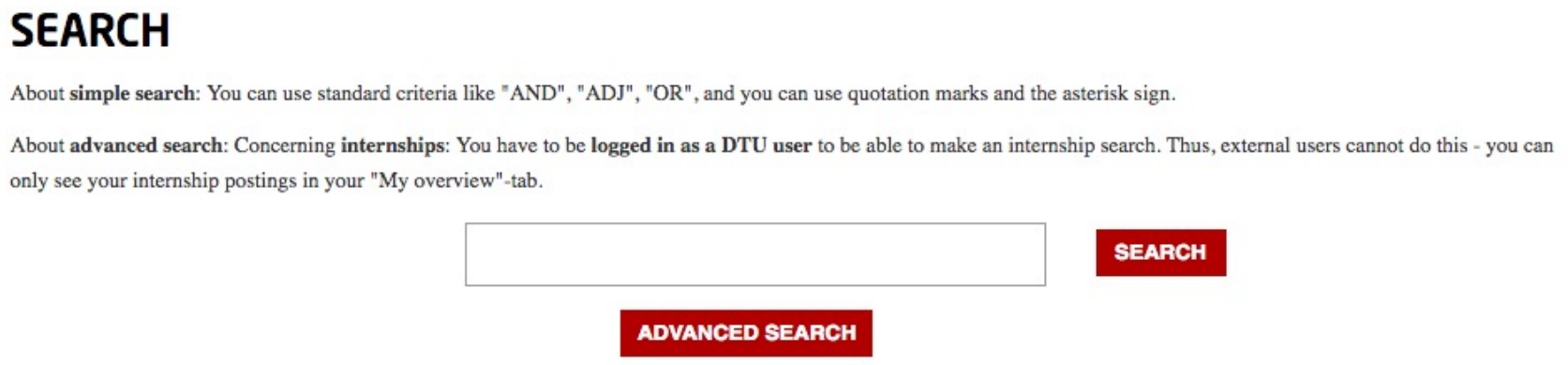
- Maintainability
  - Readable code (clean code, self documenting code, good documentation)
  - Reduce complexity (Design Pattern, low coupling/high cohesion)
- Dependability and security
  - Includes: reliability (robustness), privacy, and safety
  - Example: Apple root access on macOS
- Efficiency
  - Don’t waste system resources
  - Responsiveness, processing time, memory utilisation
- Acceptability / user friendliness



Acceptability / user friendliness



Acceptability / user friendliness



What belongs to software?

- 10.000 LOC program, no special knowledge needed: How much time?

What belongs to software?

- 10.000 LOC program, no special knowledge needed: How much time?
- Industry estimate: 24 month: around 23 LOC per work day.

What belongs to software?

- 10.000 LOC program, no special knowledge needed: How much time?
- Industry estimate: 24 month: around 23 LOC per work day.
- Software development is more than programming
  - Validation (e.g. tests)
  - Documentation (User–, System–)
  - Configuration files
  - ...

Program vs product

Student software	industrial strength software
Developer is user	Client is user
Bugs are tolerable	Bugs not tolerated
No/minor documentation	Lots of documentation
SW not in critical use	Supports business functions
Reliability/Robustness not crucial	Reliability/Robustness crucial
No investment	Heavy investment ($5 - $25 / LOC)
Portability not so important	Portability is economic advantage

Factor 3—20 from program to product

Software *Engineering*

Young disciplin: 1968 Nato conference

Software Engineering Definition (Sommerville 2010)

Software engineering is an *engineering discipline* that is concerned with *all aspects* of **software production** from the early stages of system specification through to maintaining the system after it has gone into use.

- An *engineer*
  - applies appropriate theories, methods, and tools
- *All aspects* of software production:
  - Not only writing the software but also
  - Software project management and creation of tools, methods and theories

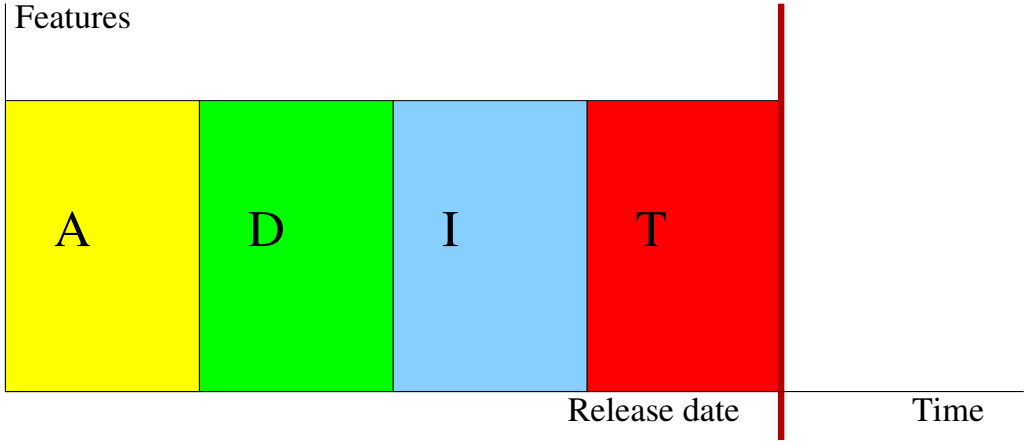
Basic Activities in Software Development

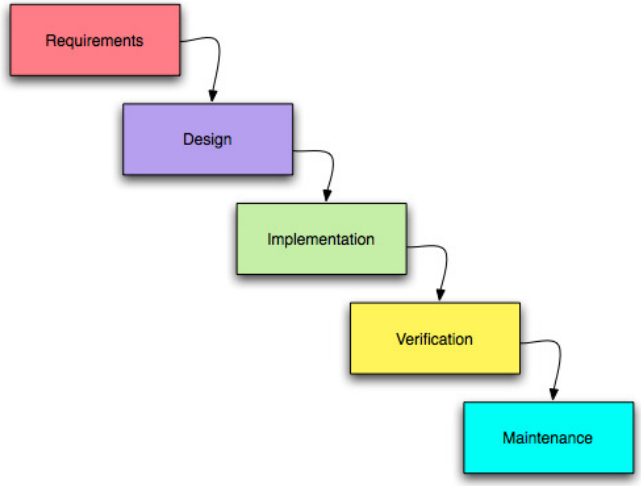
- Understand and document what kind of the software the customer wants (*Requirements Analysis*)
- Determine how the software is to be built (*Design*)
- Build the software (*Implementation*)
- Validate that the software solves the customers problem (*Test*)

$\rightarrow$ Each activity has a set of techniques and methods

Software Development Process Types

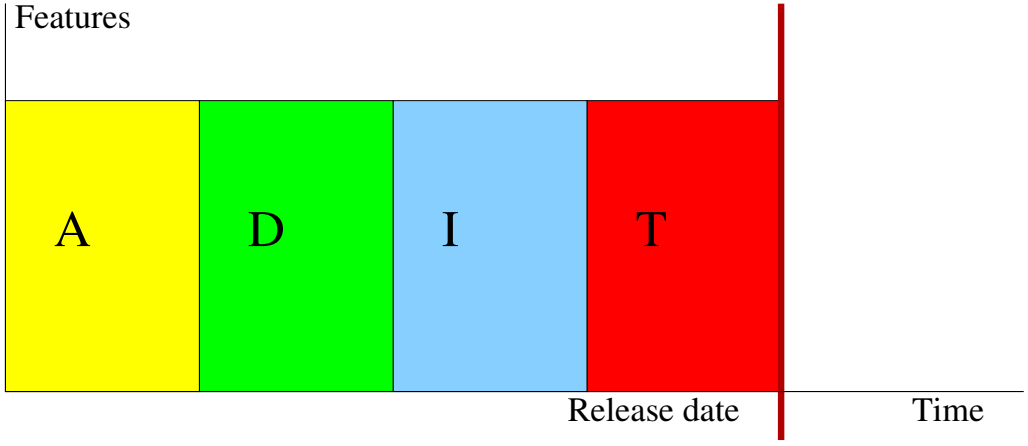
Plan-driven development (Waterfall, RUP, . . .)

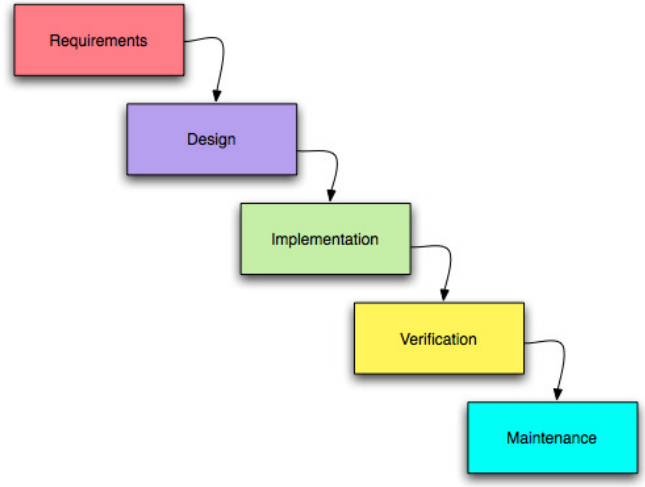




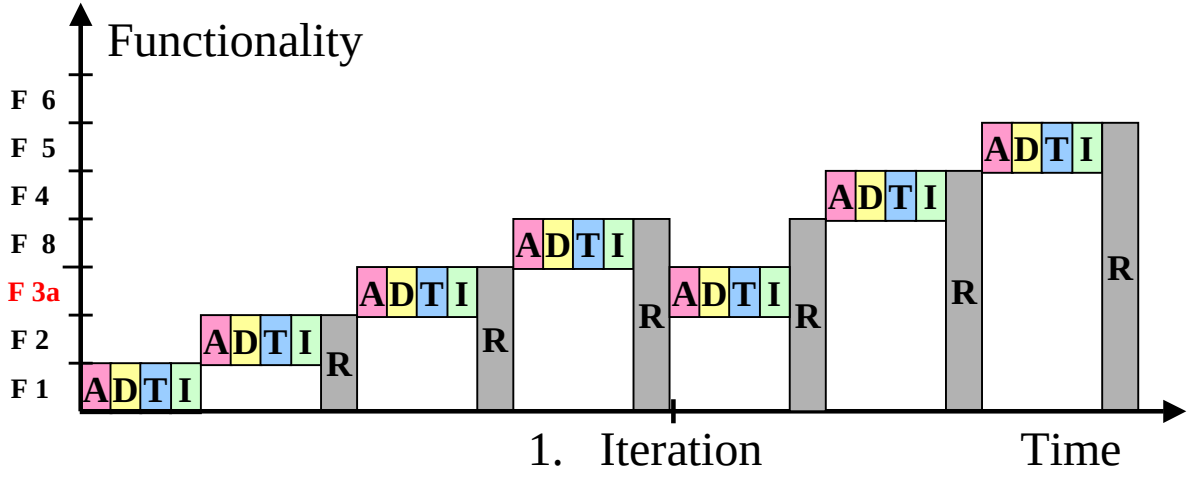
Software Development Process Types

Plan-driven development (Waterfall, RUP, . . .)

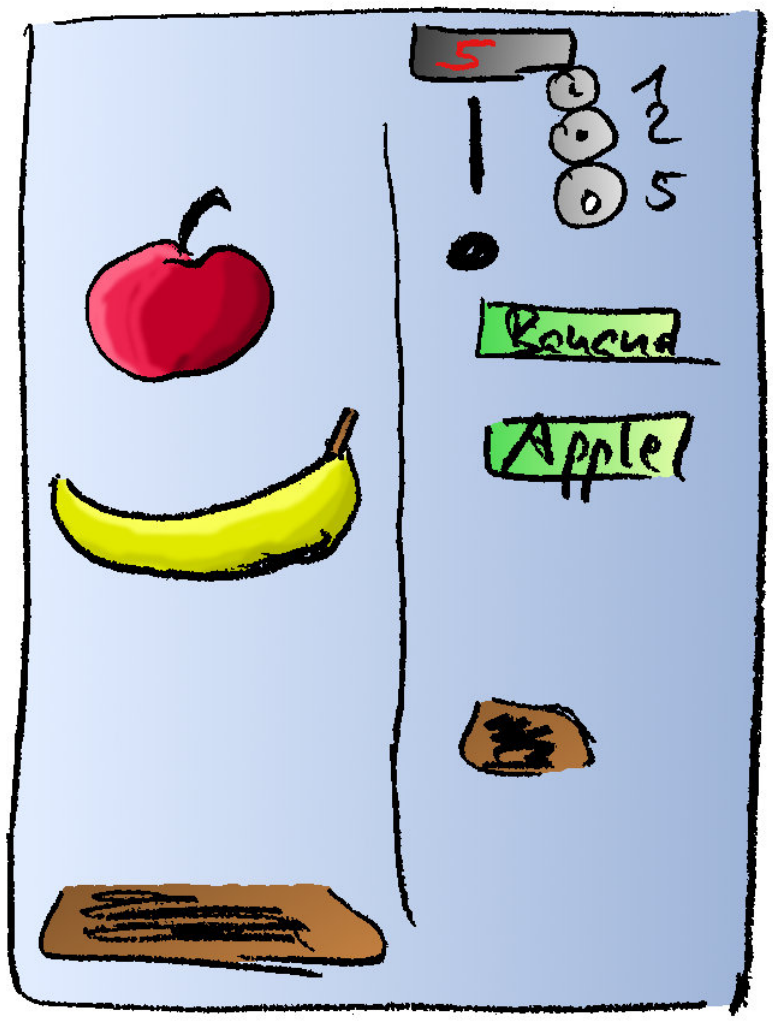




Agile development (Extreme Programming, Scrum, . . .)



Example Vending Machine Controller



Controller software for a vending machine

Requirements: Glossary

Purpose:

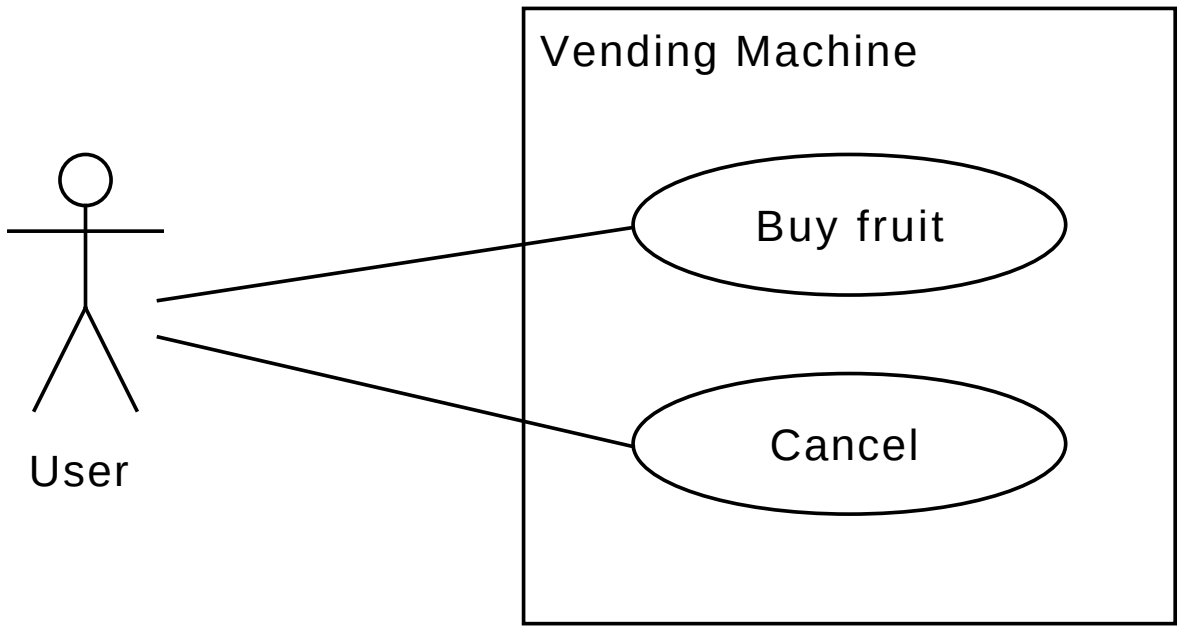
- Understand the problem domain
- Common language

Example

- Vending machine: The vending machine allows users to *buy fruit*.
- User: The user of the *vending machine* buys fruit by inserting coins into the machine.
- Owner: The owner owns the *vending machine*. He is required to refill the machine and can remove the money from the machine.
- Display: The display shows how much money the *user* has inserted.

  ...

Requirements: Use case diagram



Requirements: Detailed Use Case: Buy Fruit

```
Feature: Buy fruit
   Description: A user buys a fruit from the vending machine
   Actors: User

Scenario: Buy a fruit with enough money
   Given the vending machine has fruits
   When the user enters enough money for a fruit
   And the user selects a fruit
   Then the fruit will be dispensed
   And the machine returns the rest money
   And the machine remembers its earnings

... (More scenarios)
```

Validation: Specify success criteria: *Acceptance tests*

Use detailed use cases directly (Cucumber)

```
Scenario: Buy a fruit with enough money
   Given the vending machine has fruits
   When the user enters enough money for a fruit
   And the user selects a fruit
   Then the fruit will be dispensed
```

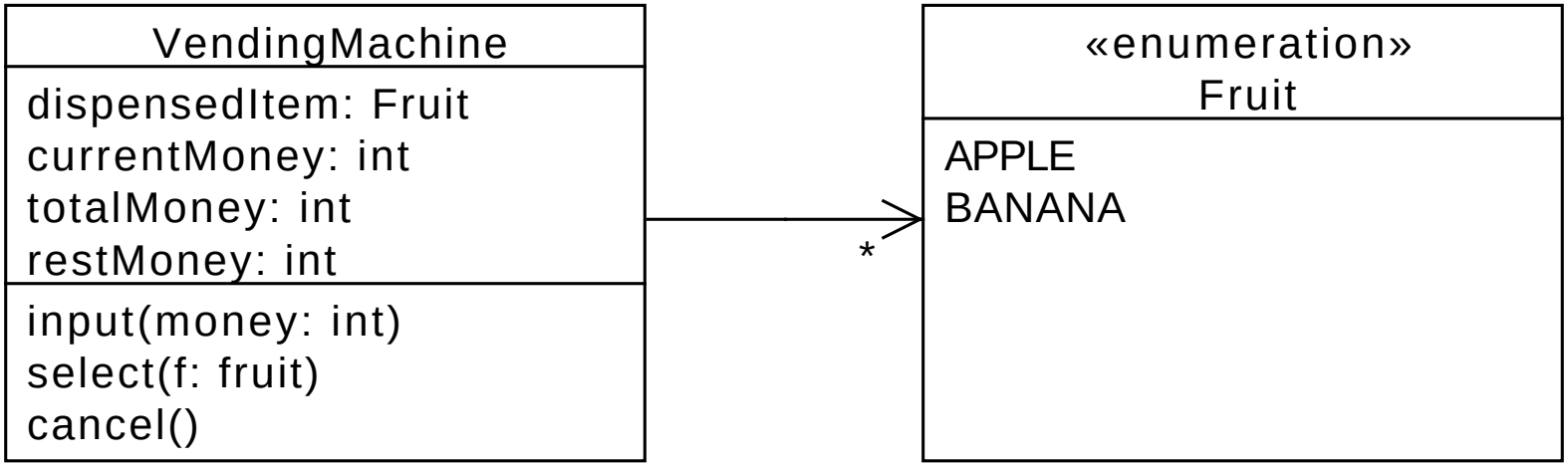
VendingMachineSteps.java

```
@Given("^the vending machine has fruits$")
public void theVendingMachineHasFruits() throws Exception {
   vendingMachine = new VendingMachine(2,2);
}
@When("^the user enters enough money for a fruit$")
public void theUserEntersEnoughMoneyForAFruit() throws Exception {
   vendingMachine.input(3);
}
@When("^the user selects a fruit$")
public void theUserSelectsTheFruit() throws Exception {
   vendingMachine.selectFruit(Fruit.APPLE);
}
@Then("^the fruit will be dispensed$")
public void theFruitWillBeDispensed() throws Exception {
   assertEquals(Fruit.APPLE, vendingMachine.getDispensedItem());
}
```

Vending Machine: Design and implementation

- Determine *how* the software is to be built
  - → Class diagrams
  - → Sequence diagrams
  - → State machines
- *Build* the software

Design: High-level Class diagram

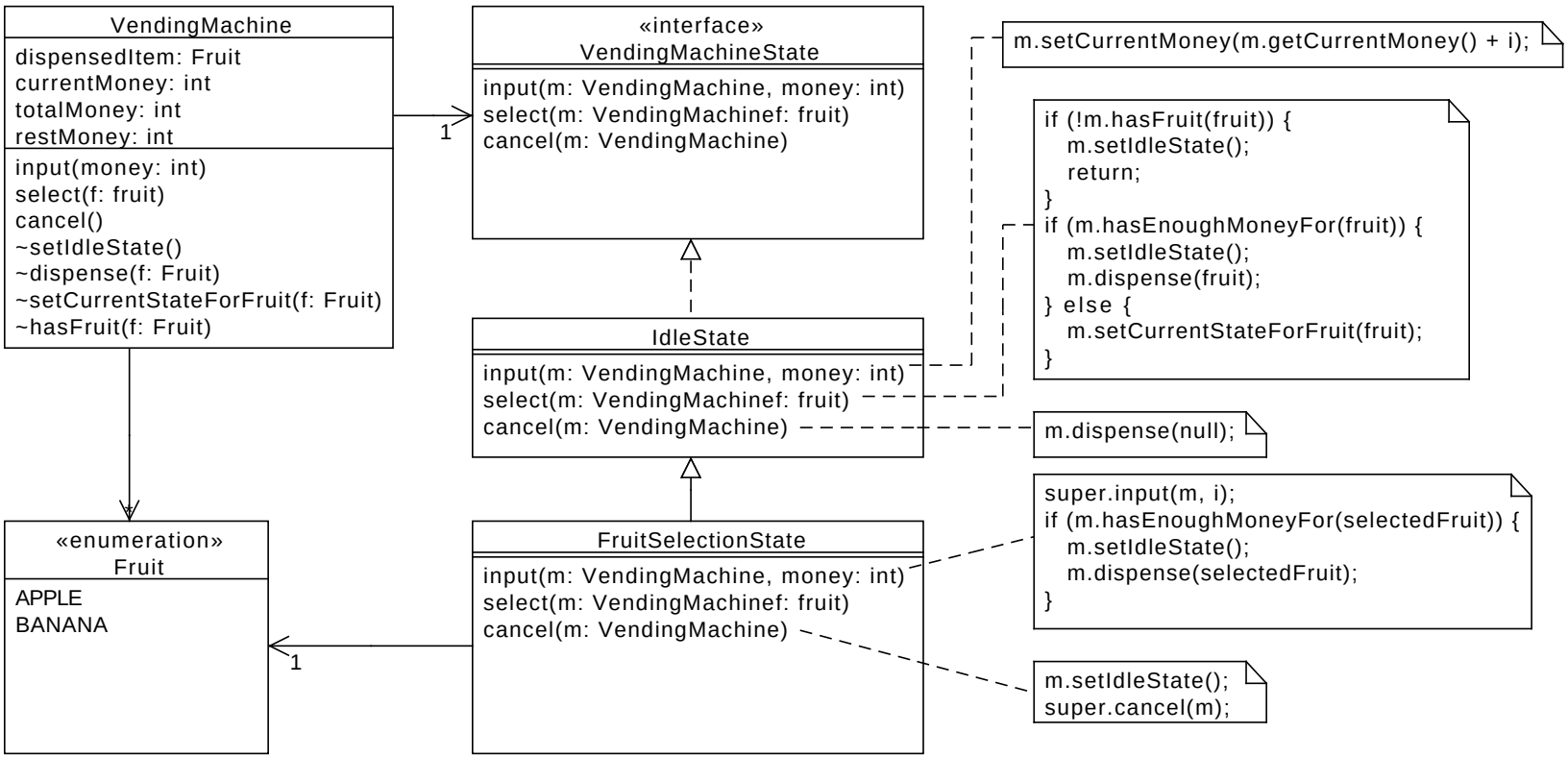


Application logic as a state machine

event	guard	state	state	state
		Idle (I)	*Banana selected and not enough money (B)*	*Apple selected and not enough money (A)*
banana	*enough money for banana*	dispense banana and rest money	dispense banana and rest money-> I	dispense banana and rest money-> I
banana	*not enough money for banana*	-> B		-> B
banana	*no bananas available*		-> I	-> I
apple	*enough money for apple*	dispense apple and rest money -> I	dispense apple and rest money -> I	dispense apple and rest money -> I
apple	*not enough money for apple*	-> A	-> A	
apple	*no apples available*		-> I	-> I
money	*enough money for banana*	add money to current money	dispense banana and rest money-> I	add money to current money
money	*enough money for apple*	add money to current money	add money to current money	dispense apple and rest money -> I
money	*not enough money for neither banana nor apple*	add money to current money	add money to current money	add money to current money
cancel		return current money	return current money -> I	return current money -> I

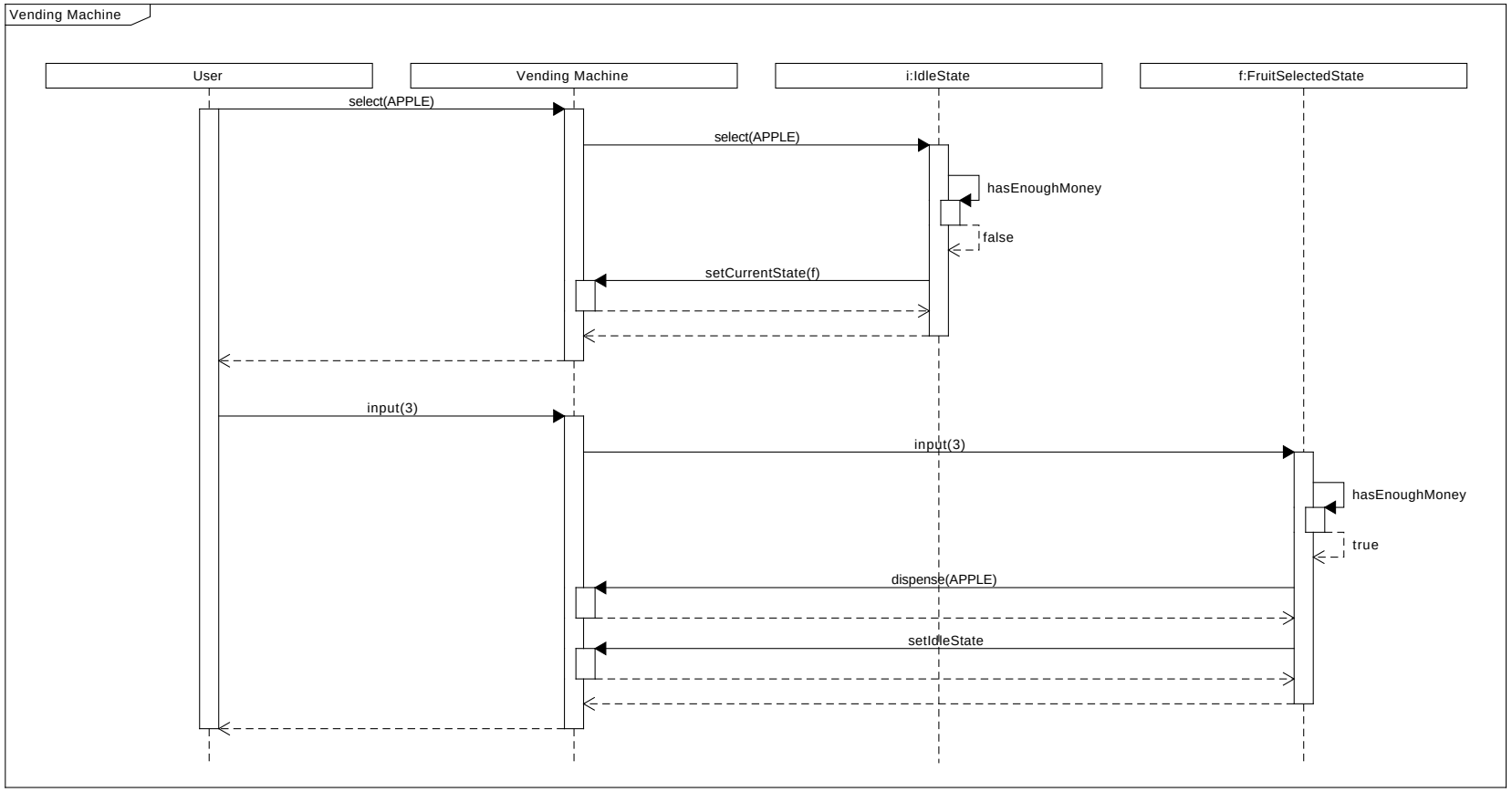
Design: Design of the system as class diagram

Use State pattern (a *design pattern*)



Design: Visualization of the Execution

- *Interaction Diagrams*, aka. *Sequence Diagrams*
  - used for designing the system
  - used for documenting the system
- Vending Machine



Contents

Course Introduction

Introduction to Software Engineering

Practical Information

Programming Assignment

Course content

1. Requirements Engineering (Use Cases, User Stories, Glossary)
2. Software Testing (JUnit, Cucumber, Test Driven Development, Systematic Tests, Code Coverage)
3. System Modelling (Class Diagrams, Sequence Diagrams, Activity Diagrams)
4. Architecture (hexagonal, layered, . . .)
5. Design (SOLID, Design Patterns, Design by Contract, . . .)
6. Software Development Processes (focus on agile processes)

Software and Tools

- Programming language: Java (latest version of Java 8)
- IDE: latest version of Eclipse: Simple Java IDE is sufficient
- Test framework: Cucumber and JUnit

...

Course activities

- Lectures every Monday 13:00 — 15:00
- Exercises after the lecture
  - Teaching assistants will be present : 15:00 — 17:00
  - Expected work at home: *5 hours*
  - Important: Ask questions: during the lecture, e-mail, CampusNet, . . .

Examination

- Exam project in groups of 4
  - Model, Software, Report, Demonstration
  - → Focus on that you have learned the techniques and methods
  - *no* written/oral examination
- Week 13: Demonstration of the projects (each project 10 min) and submission of final implementation

→ Grade is based on an evaluation of submissions as a whole (helheds vurdering)

Course material

- Course Web page:
  `http://www.imm.dtu.dk/courses/02161` contains
  - practical information: (e.g. lecture plan)
  - Course material (e.g. slides, exercises, notes)
  - *Check the course Web page regularly*
- CampusNet: Is being used to send messages;
  - *make sure that you receive all messages from CampusNet*
- Books:
  - Textbook: UML Distilled by Martin Fowler (online via DTU library), Software Engineering 9/10 from Ian Sommerville
  - Suplementary literature on the course Web page

Contents

Course Introduction

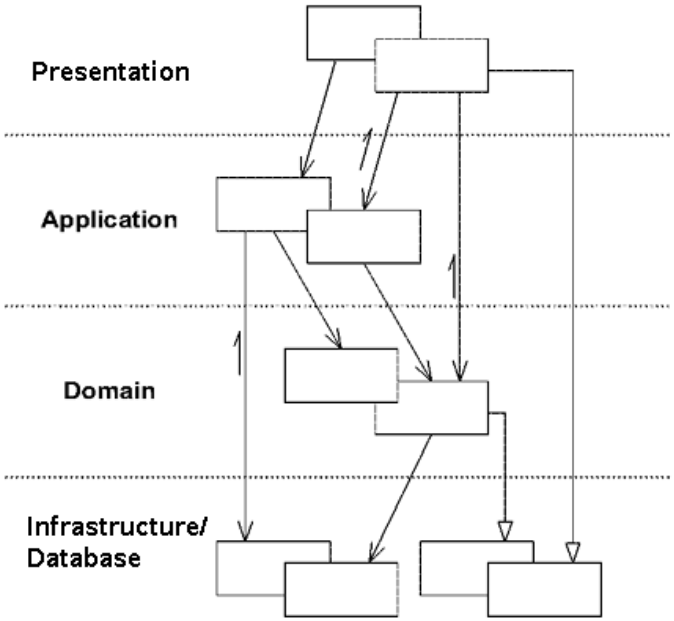
Introduction to Software Engineering

Practical Information

Programming Assignment

Programming Exercise

- Library software
- Guided development based on agile software development principles
  - Behaviour-driven-development (TDD/BDD)



Eric Evans, Domain Driven Design, Addison-Wesley, 2004

1. Development of the application + domain layer
2. Presentation layer
3. Simple persistency layer

First week's assignment

- Given use case scenarios for: Admin Login, Admin logout, Add book, and Search book

```
Scenario: Administrator can login
   Given that the administrator is not logged in
   And the password is "adminadmin"
   Then the administrator login succeeds
   And the adminstrator is logged in
```

- And step definitions

```
@Given("^that the administrator is not logged in$")
public void thatTheAdministratorIsNotLoggedIn() throws Exception
   assertFalse(libraryApp.adminLoggedIn());
}
@Then("^the administrator login succeeds$")
public void theAdministratorLoginSucceeds() throws Exception {
   assertTrue(libraryApp.adminLogin(password));
}
...
```

- Implement the production code
  - Test have to pass
- Tools: Cucumber and JUnit