Software Engineering I (02161)
Week 1

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2016

DTU

Contents

The course

- ▶ 5 ECTS course 02161: Software Engineering 1
- ▶ Target group: Bachelor in Software Technology and IT and Communication in the second semester
- ▶ Learning objectives
  - ▶ To have an overview over the field software engineering and what is required in software engineering besides programming
  - ▶ To be able to take part in bigger software development projects
  - ▶ To be able to communicate with other software designers about requirements, architecture, design
  - → To be able to conduct a *smaller* project from an *informal* and *open description* of the problem

Who are we?

- ▶ 119 students with different backgrounds (bachelor 115)
  - ▶ Bachelor Softwaretek.: 71
  - ▶ Bachelor It og Kom.: 39
  - ▶ Bacheolor other: 5
  - ▶ Other: 4
- ▶ Teacher
  - ▶ Hubert Baumeister, Assoc. Prof. at DTU Compute (huba@dtu.dk; office 303B.058)
- ▶ 3 Teaching assistants
  - ▶ Jonas Holger Hansen
  - ▶ Maja Lund
  - ▶ Mathias Kirkeskov Madsen

Contents
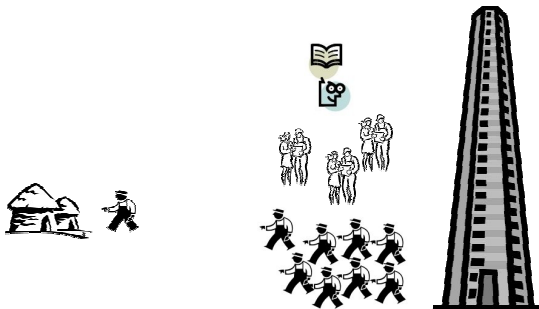
Building software

Tools and techniques for building software, in particular *large* software

What is software?

- ► Software is everywhere
  - ► Stand-alone application (e.g. Word, Excel), Mobile applications, Interactive transaction-based applications (e.g. flight booking), Embedded control systems (e.g., control software the Metro, mobile phones), Batch processing systems (e.g. salary payment systems, tax systems), Entertainment systems (e.g. Games), System for modelling and simulation (e.g. weather forecasts), Data collection and analysing software (e.g. physical data collection via sensors, but also data-mining Google searches), System of systems (e.g. cloud, system of interacting software systems), ...
- ► Types of software
  - ► Mass production, Customised software, Mixture of both
- → Not one tool, method, or theory
  - ► Though there are general principles applicable to all domains

Software attributes

- Maintainability
  - Can be evolved through several releases and changes in requirements and usage
- Dependability and security
  - Includes: reliability (robustness), security, and safety
- Efficiency
  - Don't waste system resources such as memory or processor cycles
  - Responsiveness, processing time, memory utilisation
- Acceptability
  - To the user of the system
  - understandable, usable, and compatible with the other systems the user uses

What belongs to software?

- Computer program(s), but also
- Validation (e.g. tests)
- Documentation (User–, System–)
- Configuration files
- …

Software *Engineering*

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
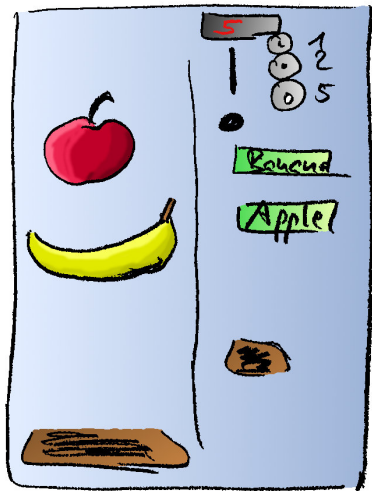
- ▶ Understand and document what kind of the software the customer wants
- ▶ Determine how the software is to be built
- ▶ Build the software
- ▶ Document and being able to talk about the software
- ▶ Validate that the software solves the customers problem
- $\rightarrow$ Each activity has a set of techniques and methods

Two approaches to software development

Waterfall

Agile

Example Vending Machine

Design and implement a control software for a vending machine

Vending Machine: Requirements documentation

- *Understand* and *document* what kind of the software the customer wants
  - → Glossary
  - → Use case diagram
  - → Detailed use case

Glossary

- ▶ 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.
- ▶ Buy fruit: Buy fruit is the process, by which the user inputs coins into the vending machine and selects a fruit by pressing a button. If enough coins have been provided the selected fruit is dispensed.
- ▶ Cancel: The *user* can cancel the process by pressing the button cancel. In this case the coins he has inserted will be returned.

  . . .

Use case diagram
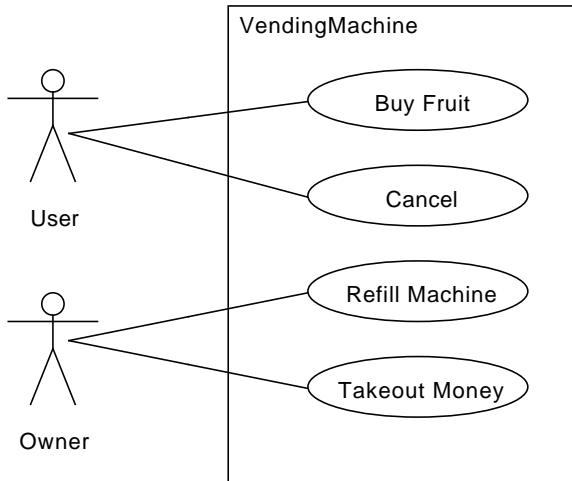
Detailed Use Case: Buy Fruit

name: Buy fruit

description: Entering coins and buying a fruit

actor: user

main scenario:

1. Input coins until the price for the fruit to be selected is reached
2. Select a fruit
3. Vending machine dispenses fruit

alternative scenarios:

a1. User inputs more coins than necessary
a2. select a fruit
a3. Vending machine dispenses fruit
a4. Vending machine returns excessive coins

. . .

Vending Machine: Specify success criteria

- *Prepare* for the validation
  - $\rightarrow$ Create *tests* together with the customer that show when system fulfils the customers requirements
  - $\rightarrow$ *Acceptance tests*
  - Test driven development
    - $\rightarrow$ create tests *before* the implementation
  - Otherwise: after the implementation

Functional Test for Buy Fruit Use Case: JUnit Tests

```java
@Test
public void testBuyFruitExactMoney() {
    VendingMachine m = new VendingMachine(10, 10);
    m.input(1);
    m.input(2);
    assertEquals(3, m.getCurrentMoney());
    m.selectFruit(Fruit.APPLE);
    assertEquals(Fruit.APPLE, m.getDispensedItem());
}

@Test
public void testBuyFruitOverpaid() {
    VendingMachine m = new VendingMachine(10, 10);
    m.input(5);
    assertEquals(5, m.getCurrentMoney());
    m.selectFruit(Fruit.APPLE);
    assertEquals(Fruit.APPLE, m.getDispensedItem());
    assertEquals(2, m.getRest());
}

// more tests
// at least one for each main/alternative scenario
```

Vending Machine: Design and implementation

- ► Determine *how* the software is to be built
  - → Class diagrams to show the structure of the system
  - → State machines and sequence diagrams to show how the system behaves
- ► *Build* the software
  - → Implement the state machine using the state design pattern

High-level Class diagram

VendingMachine
dispensedItem: Fruit
currentMoney: int
totalMoney: int
restMoney: int
input(money: int)
select(f: fruit)
cancel()

«enumeration» Fruit
APPLE
BANANA

*
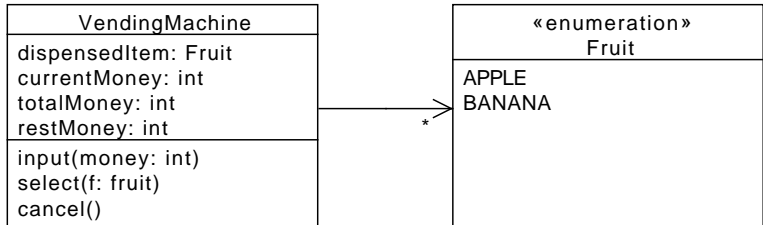
Application logic as state machine

state machine VendingMachine LifeCycle [VendingMachine LifeCycle]

input(m : Integer) / currentMoney := currentMoney + m

cancel() / dispense(nil)

select(f : Fruit) [not(f = selectedFruit) and not(enoughMoney(f))] / selectedFruit := f
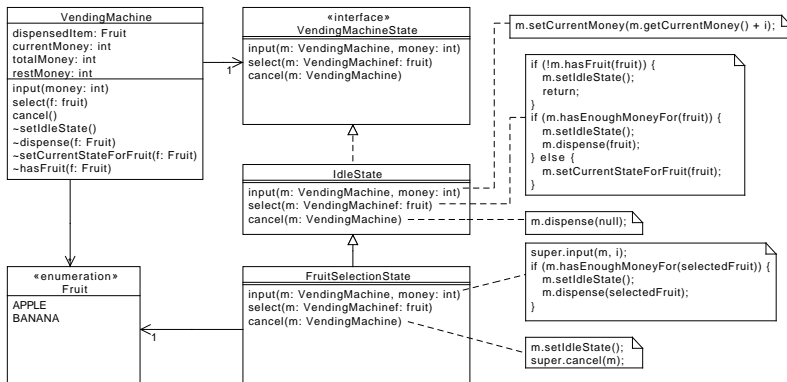
Idle

select(f : Fruit) [not(enoughMoney(fruit) and hasFruit(f)] / selectedFruit := fruit

Fruit selected

select(f : Fruit) [enoughMoney(f)and hasFruit(f)] / dispense(f)

select(f : Fruit) [not(hasFruit(f))] / dispense(nil)

select(f : Fruit) [enoughMoney(f)] / dispense(f)

input(m : Integer) [enoughMoney(selectedFruit)] / currentMoney := currentMoney + 1; dispense(f)

input(m : Integer) [not(enoughMoney(selectedFruit))] / currrentMoney := currentMoney + 1

cancel() / dispense(nil)

Design of the system as class diagram

Uses the state design pattern

VendingMachine
dispensedItem: Fruit
currentMoney: int
totalMoney: int
restMoney: int

input(money: int)
select(f: fruit)
cancel()
~setIdleState()
~dispense(f: Fruit)
~setCurrentStateForFruit(f: Fruit)
~hasFruit(f: Fruit)

«interface»
VendingMachineState
input(m: VendingMachine, money: int)
select(m: VendingMachinef: fruit)
cancel(m: VendingMachine)

«enumeration»
Fruit
APPLE
BANANA

IdleState
input(m: VendingMachine, money: int)
select(m: VendingMachinef: fruit)
cancel(m: VendingMachine)

FruitSelectionState
input(m: VendingMachine, money: int)
select(m: VendingMachinef: fruit)
cancel(m: VendingMachine)

m.setCurrentMoney(m.getCurrentMoney() + i);

```
if (!m.hasFruit(fruit)) {
    m.setIdleState();
    return;
}
if (m.hasEnoughMoneyFor(fruit)) {
    m.setIdleState();
    m.dispense(fruit);
} else {
    m.setCurrentStateForFruit(fruit);
}
```

m.dispense(null);

```
super.input(m, i);
if (m.hasEnoughMoneyFor(selectedFruit)) {
    m.setIdleState();
    m.dispense(selectedFruit);
}
```
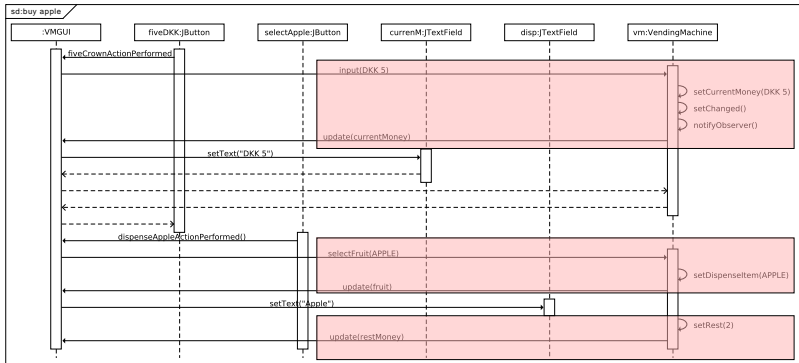
m.setIdleState();
super.cancel(m);

Vending Machine: Visualization of the Execution

- ▸ Designing the system
- ▸ Documentation the system
- → Use *Interaction Diagrams*, aka. *Sequence Diagrams*

Interaction Diagram: Swing GUI

Contents

Course content

1. Requirements Engineering
2. Software Testing (JUnit, Test Driven Development, Systematic Tests, Code Coverage)
3. System Modelling (mainly based on UML)
4. Architecture (e.g layered architecture)
5. Design (among others Design Patterns and Design by Contract)
6. Software Development Process (focus on agile processes)
7. Project Management (project planning)

Course activities

- Lectures every Monday 13:00 — approx 15:00 (Lecture plan is on the course Web page)
- Exercises (databar 003, 015, 019 in building 341)
  - Teaching assistants will be present : 15:00 — 17:00
  - Expected work at home: *5 hours* (lecture preparation; exercises, . . .)
- Programming asssginments and non-programming assignments
  - not mandatory
  - But hand-in recommended to get feedback
  - Preparation for the examination project

Examination

- ▶ Exam project in groups (2—4)
  - ▶ Software, Report, Demonstration
  - → Focus on that you have learned the techniques and methods
  - ▶ *no* written examination
- ▶ Week 05: Project introduction and forming of project groups; participation mandatory
- ▶ Week 07: Submission of use cases and design
- ▶ Week 08: Peer review of use cases and design; start of implementation phase
- ▶ Week 13: Demonstration of the projects (each project 15 min)

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: Software Engineering 9 from Ian Sommerville and UML Destilled by Martin Fowler
  - Suplementary literature on the course Web page

Contents

Programming Assignments

- ▶ Implementation of a library software
- ▶ Guided development based on agile software development principles
    - ▶ User-story driven: The development is done based on user stories that are implemented one by one
    - ▶ Test-driven: Each user-story is implemented by first writing the test for it and then writing the code
- ▶ All programming assignments are available directly

Layered Architecture

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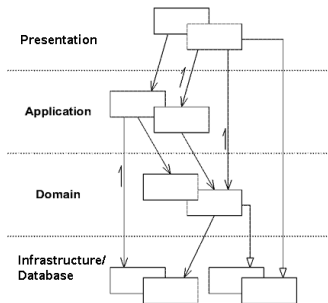
2004

1. Development of the application + domain layer (assignments 1 – 4)
2. Presentation layer: Command line GUI (assignment 5)
3. Simple persistency layer (assignment 6)

First week's exercise

- ▶ Using Test-Driven Development to develop the application + domain layer
- ▶ Basic idea: First define the tests that the software has to pass, then develop the software to pass the tests
  - ▶ Writing tests before the code is a *design* activity, as it requires to *define* the interface of the code and how to use the code, before the code is written
- ▶ Test are automatic using the JUnit framework
- ▶ First Week's exercise: Tests are given, you implement just enough code to make the tests pass
- $\rightarrow$ Video on the home page of the course
- ▶ This is done by uncommenting each test one after the other
- ▶ First implement the code to make one test run, only then uncomment the next test and make that test run

Contents

JUnit

- ▶ JUnit is designed by Kent Beck in Erich Gamma to allow one to write automated tests and execute them conveniently
- ▶ JUnit can be used standalone, but is usually integrated in the IDE (in our case Eclipse)
- ▶ We are going to use JUnit version 4.x which indicates tests to be run automatically using the @org.junit.Test annotation (or just @Test if org.junit.Test is imported)

Example of a JUnit Test

The following tests one scenario of the login functionality:

1. First check that the adminstrator is not logged in
2. login the adminstrator
3. Check that the login operation returns the correct return value (in this case true)
4. Check with the system, that the user is logged in

```
@Test
public void testLogin() {
  LibraryApp libApp = new LibraryApp();
  assertFalse(libApp.adminLoggedIn());
  boolean login = libApp.adminLogin("adminadmin");
  assertTrue(login);
  assertTrue(libApp.adminLoggedIn());
}
```

$\rightarrow$ The Web site of the course has a link to a video showing you how you should work on the programming assignments

Contents

User-defined Exceptions

- ▶ Purpose: To notify the caller about some exceptional or error state of the method

```
public void addBook(Book book)
    throws OperationNotAllowedException {
  if (!adminLoggedIn())
    throw new OperationNotAllowedException(...);
  ...
}
```

- ▶ Creating a user defined exception

```
public class OperationNotAllowedException extends Exception {
  public OperationNotAllowedException(String errorMsg) {
    super(errorMsg);
  }
}
```

- ▶ Throwing a user-defined exception

```
throw new OperationNotAllowedException("some error message");
```

Checked vs. unchecked Exceptions

- ► Checked Exception

  ```
  public class MyCheckedException extends Exception {...}
  ```
- → Methods which throw MyCheckedException must have throws MyCheckedException in the signature, e.g.

  ```
  public void m() throws MyCheckedException {...}
  ```
- ► Unchecked Exception

  ```
  public class MyUncheckedException extends Error {...}
  ```
- → Methods don't need the throw clause

User-defined Exceptions: Example

- Catching an user-defined exception

```
try {
    libApp.addBook(book1);
} catch (OperationNotAllowedException e) {
 // Error handling code
}
```

Compiler error: Unreachable catch block

- ▶ Test code
  ```
  try {
    libApp.addBook(book1);
    fail();
  } catch (OperationNotAllowedException e) { .. }
  ```
- ▶ Code added by Eclipse
  ```
  public void addBook(Book book) { }
  ```
- ▶ Compiler error: "Unreachable catch block for OperationNotAllowedException. This exception is never thrown from the try statement body"

Compiler error: Unreachable catch block

- Test code
```
try {
  libApp.addBook(book1);
  fail();
} catch (OperationNotAllowedException e) { .. }
```

- Code added by Eclipse
```
public void addBook(Book book) { }
```

- Compiler error: "Unreachable catch block for OperationNotAllowedException. This exception is never thrown from the try statement body"

- Solution
```
public void addBook(Book book)
      throws OperationNotAllowedException { }
```

- Problem only occurs with *checked exceptions*

Testing and exceptions

- ▶ Test for the presence of an exception

```java
@Test
public void testSomething() {
  ...
  try {
    // Some code that is expected to
    // throw OperationNotAllowedException
    assertFalse(libApp.adminLoggedIn());
    libApp.addBook(b);
    fail("Expected OperationNotAllowedException to be thrown");
  } catch (OperationNotAllowedException e) {
    // Check, e.g., that the error message is correctly set
    assertEquals(expected, e.getMessage());
  }
}
```

- ▶ Alternative test

```java
@Test(expected=OperationNotAllowedException.class)
public void testSomething() {...}
```

- ▶ No try-catch if you don't test for an exception: JUnit knows best how to handle not expected exceptions

Lists (Collections)

- ▶ Interface: java.util.List<T>
  - → `http://docs.oracle.com/javase/1.4.2/docs/api/java/util/List.html`
- ▶ Classes implementing the List interface:
  - ▶ java.util.ArrayList<T>, java.util.Vector<T> (among others)
- → Use java.util.List<T> in all methods and as the type of the instance variable
- → Information hiding
  - ▶ decoupling implementation from usage

Creating a List

- Instance variable containing a list
  ```
  List<Book> books = new ArrayList<Book>();
  ```
- Alternative (not so good)
  ```
  ArrayList<Book> books = new ArrayList<Book>();
  ```

Iterating over a list

- ► Variant a)

```
for (int i = 0; i < books.size(); i++) {
    Book book = books.get(i);
    // do something with book
}
```

Iterating over a list

- Variant a)
```
for (int i = 0; i < books.size(); i++) {
    Book book = books.get(i);
    // do something with book
}
```
- Variant b)
```
for (Iterator it = books.iterator(); it.hasNext(); ) {
    Book book = it.next();
    // do something with book
}
```

Iterating over a list

- Variant a)
  ```
  for (int i = 0; i < books.size(); i++) {
      Book book = books.get(i);
      // do something with book
  }
  ```
- Variant b)
  ```
  for (Iterator it = books.iterator(); it.hasNext(); ) {
      Book book = it.next();
      // do something with book
  }
  ```
- Variant c) *recommended* way
  ```
  for (Book book : books) {
      // do something with book
  }
  ```

Iterating over a list

- ▶ Variant a)
  ```java
  for (int i = 0; i < books.size(); i++) {
      Book book = books.get(i);
      // do something with book
  }
  ```
- ▶ Variant b)
  ```java
  for (Iterator it = books.iterator(); it.hasNext(); ) {
      Book book = it.next();
      // do something with book
  }
  ```
- ▶ Variant c) *recommended* way
  ```java
  for (Book book : books) {
      // do something with book
  }
  ```
- ▶ Variant d) using Streams in Java 8
  ```java
  books.stream().forEach( b -> { /* do something */ });
  ```

Pre Java 8 vs Java 8

Finding an element:

- ▶ Using foreach in Java 7

```java
public Book findBook(String name) {
    for (Book book : books) {
        if (book.getName().equals(name)) {
            return book;
        }
    }
}
```

- ▶ Using streams in Java 8

```java
public Book findBook(String name) {
    Optional r = books
        .stream()
        .filter(b -> b.getName().equals(name))
        .findFirst();
    return r.isPresent() ? r.get() : null;
}
```