Software Engineering I (02161)

Week 1

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2013

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

  $\rightarrow$ To be able to conduct a *smaller* project from an *informal* and *open description* of the problem

Who are we?

- 117 students with different backgrounds
  - Bachelor Softwaretek.: 60
  - Bachelor It og Kom.: 44
  - Other bachelor: 9
  - Other: 4
- Teacher
  - Hubert Baumeister, Assoc. Prof. at DTU ~~Informatik~~ Compute (hub@imm.dtu.dk; office 322.010 (*will be changing during the course :-(*)
- 3 Teaching assistants
  - Thomas Feld
  - Patrik Reppien
  - NN

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)
  - 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)
  - ...

What is software?

- Software: Not only the computer program(s) but also
  - Documentation (User–, System–)
  - Configuration files, . . .
- Types of software
  - Mass production: The maker of the software owns the system specification
  - Customised software: The customer owns the system specification
  - Mixture: Customised software based on mass production software

→ Not one tool, method, or theory

  - Though there are general principles applicable to all domains

Attributes of Software

- Maintainability
- Dependability and security
- Efficiency
- Acceptability

Software *Engineering*

tools & methods

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.

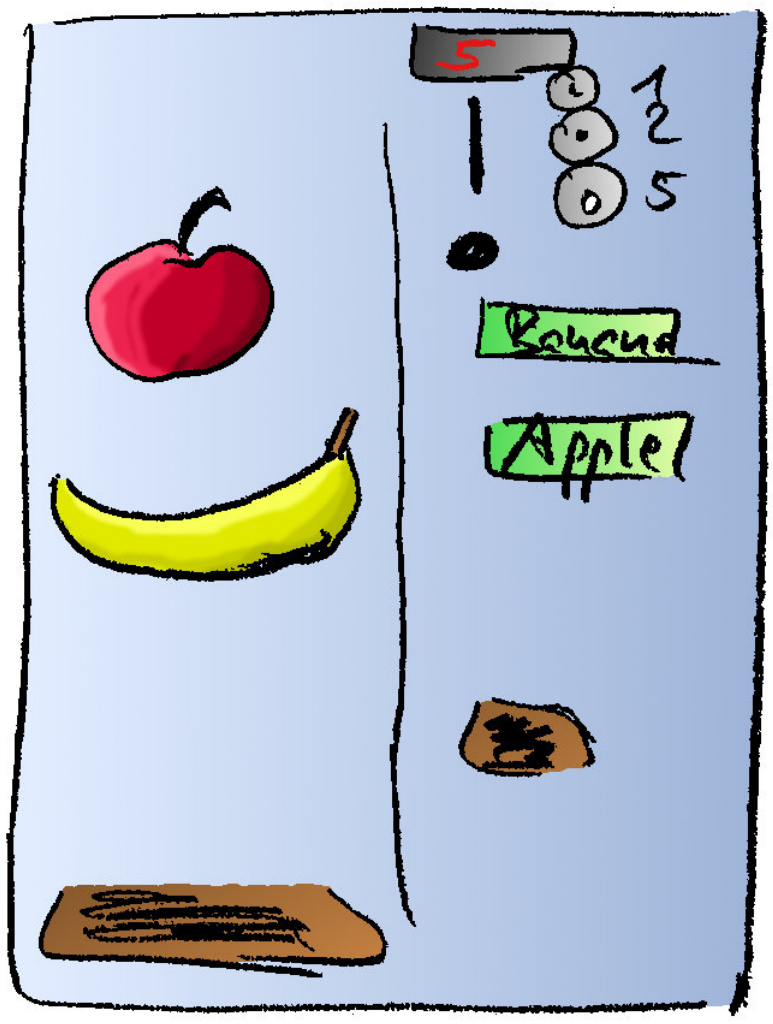
Basic Activities in Software Development

- *Understand* and *document* what kind of the software the customer wants
- Determine *how* the software is to be built → Design
- *Build* the software → Implementation
- *Document* and being able to *talk about* the software
- *Validate* that the software solves the customers problem

→ Test

Modelling

Example Vending Machine



Design and implement a control software for a vending machine

Vending Machine: Requirements documentation

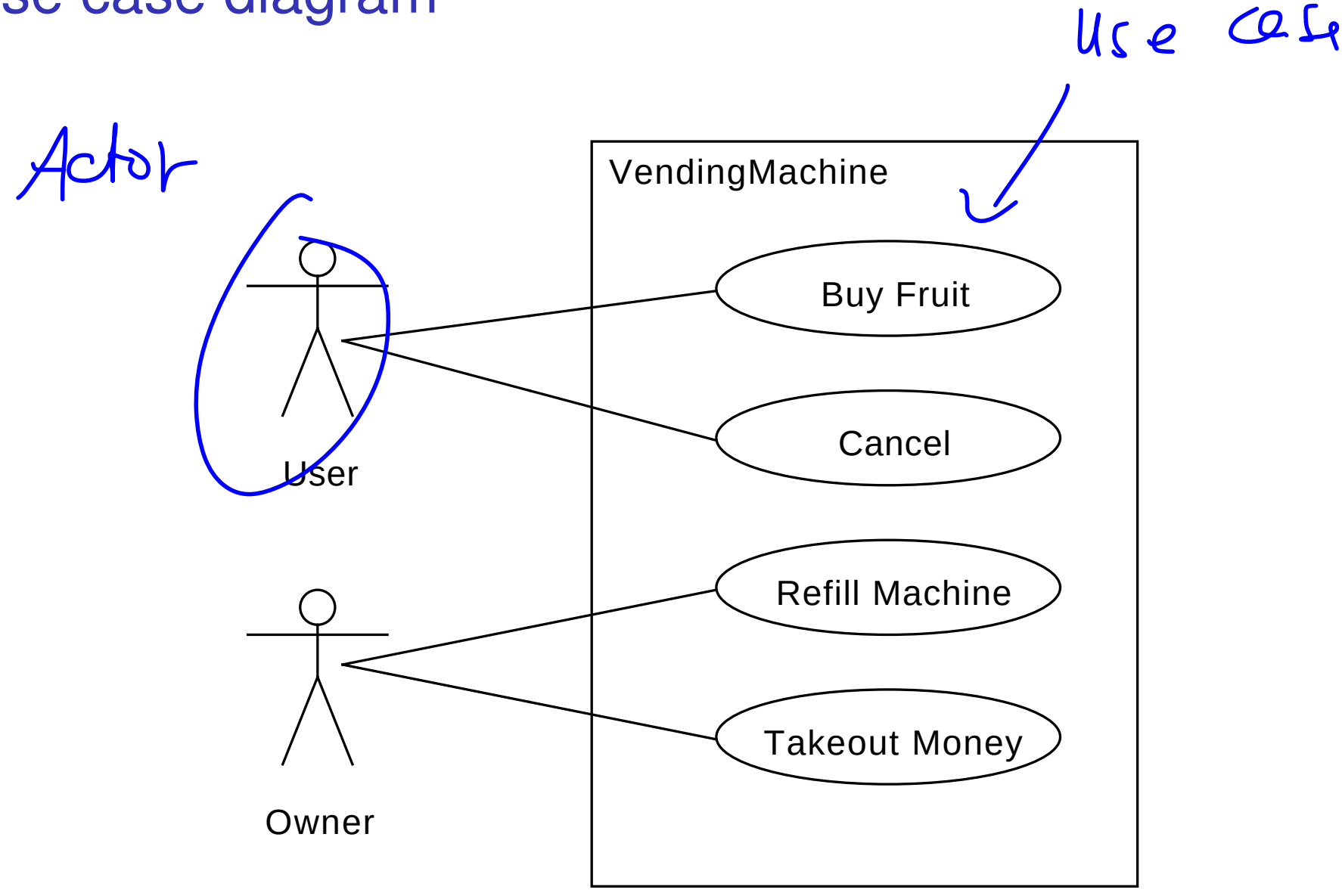
- *Understand* and *document* what kind of the software the customer wants
  - → Glossary
  - → Use case diagram
  - → Detailed use case

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:

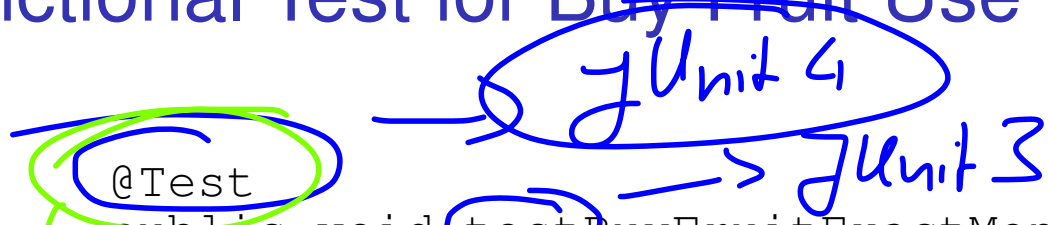
a1. User inputs more coins than necessary
a2. select a fruit
a3. Vending machine dispenses fruit
a4. Vending machine returns excessive coins

...

Vending Machine: Specify success criteria

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

Functional Test for Buy Fruit Use Case: JUnit Tests

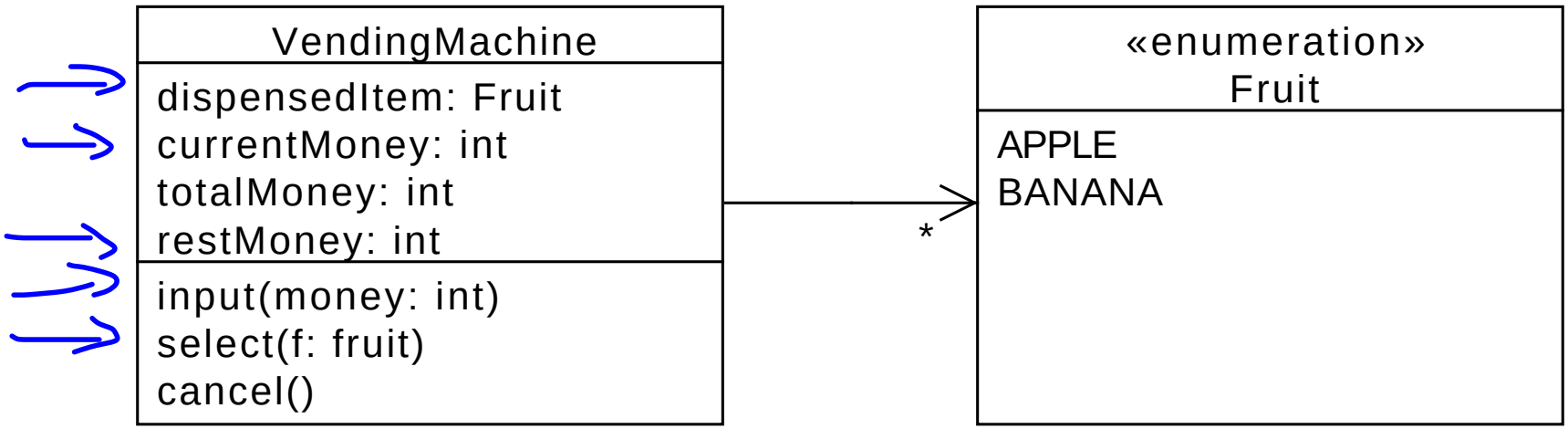



```
@Test
public void testBuyFruitExactMoney() {
    VendingMachine m = new VendingMachine(10, 10);
    m.input(1);
    m.input(2);
    assertEquals(3, m.getCurrentMoney());
    m.selectFruit(Fruit.APPLE);
    assertEquals(Fruit.APPLE, m.getDispensedItem());
}

@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
```

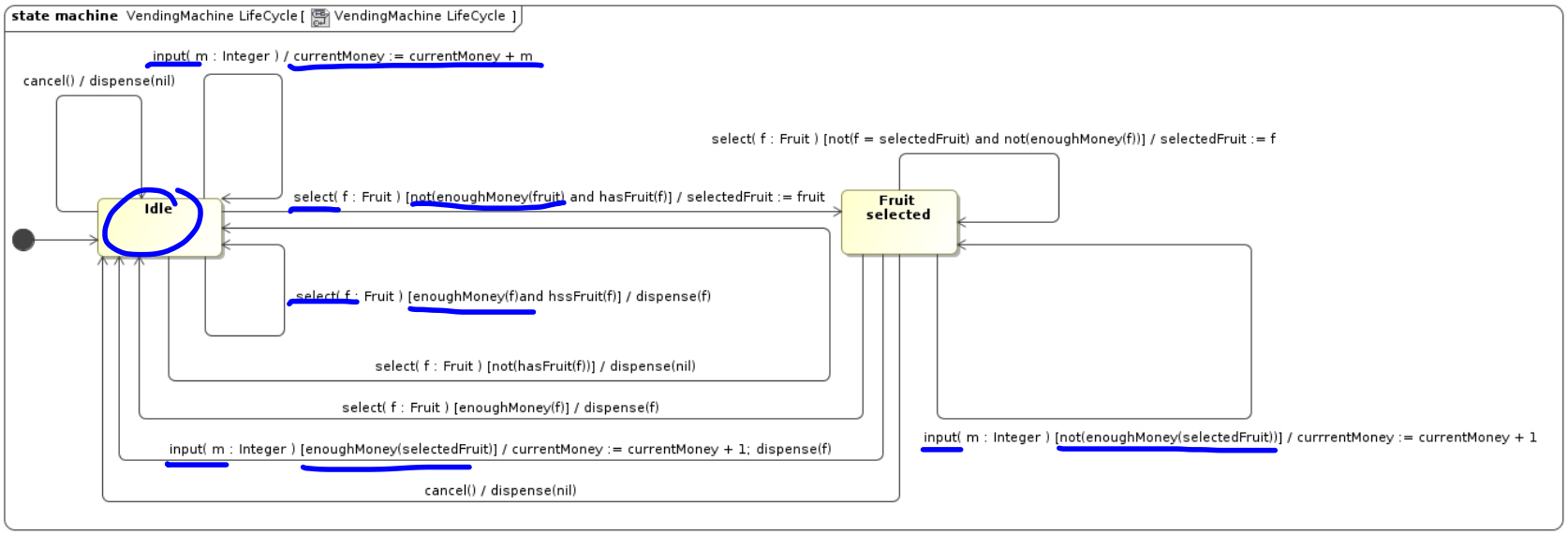
Vending Machine: Design and implementation

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

High-level Class diagram

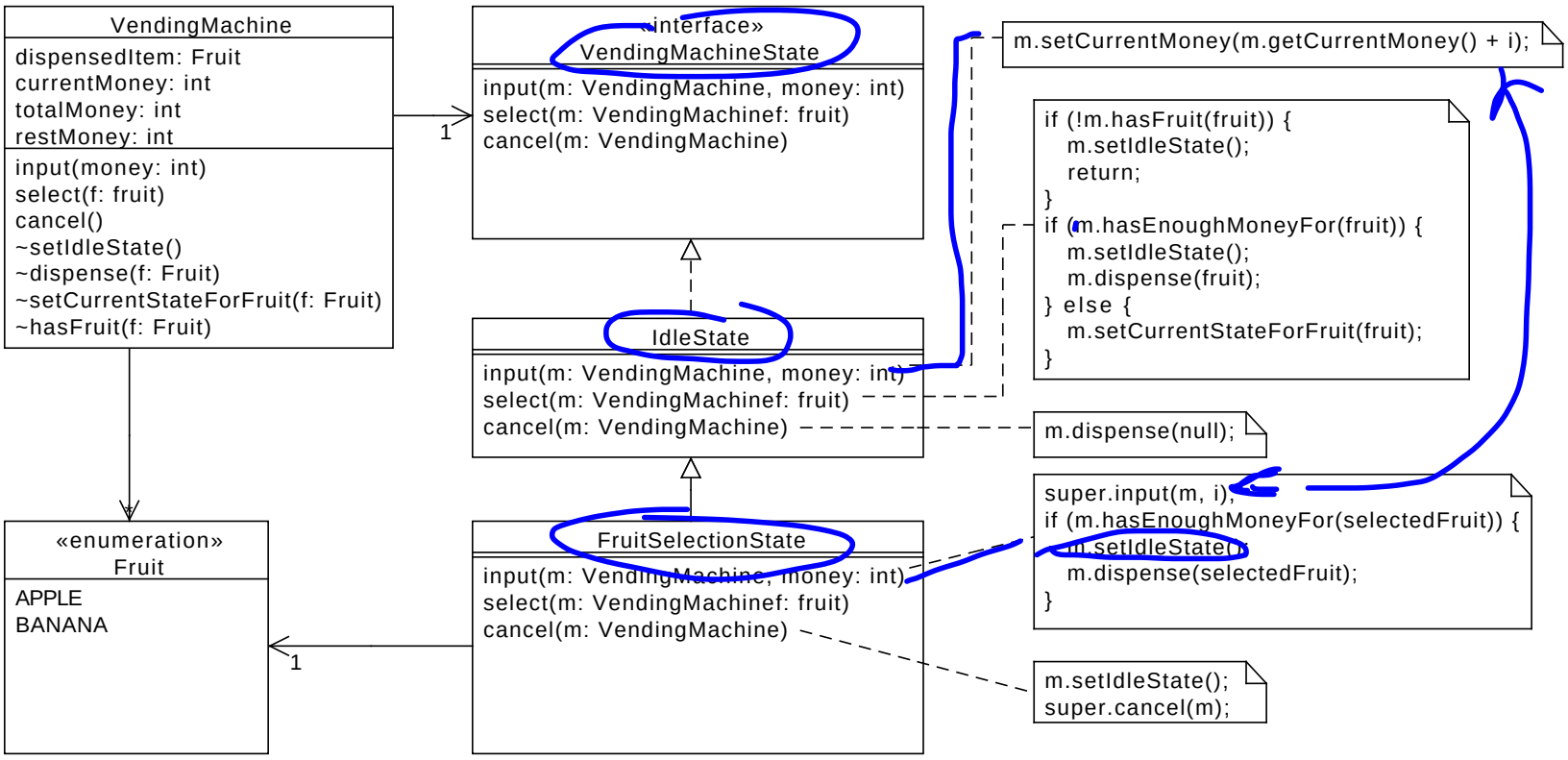


Application logic as state machine



Design of the system as class diagram

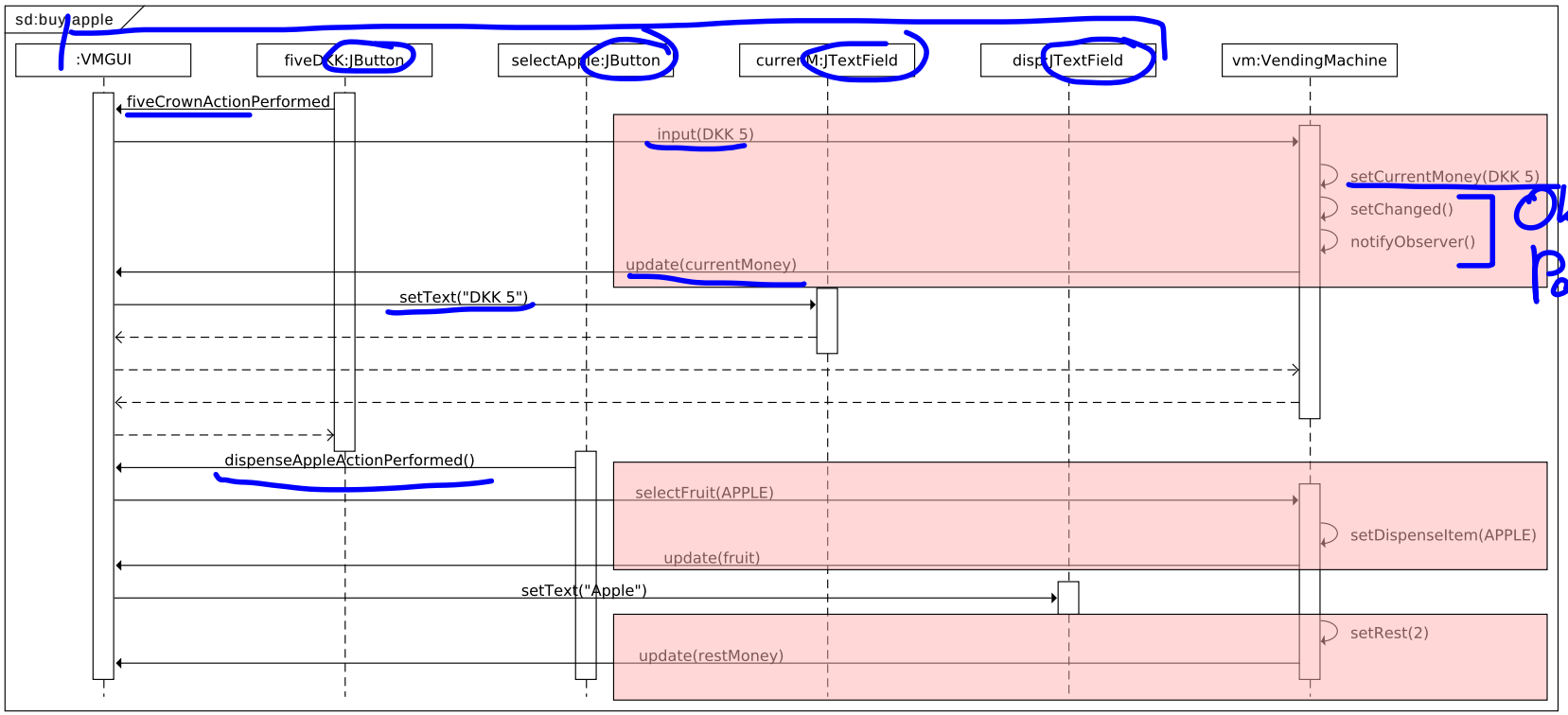
Uses the state design pattern



Vending Machine: Visualization of the Execution

- Documentation of how the implementation of the Vending Machine works:
  - $\rightarrow$ Use *Interaction Diagrams*, aka. *Sequence Diagrams*

Interaction Diagram: Swing GUI



Contents

Course content

0. Introduction
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)

Approach to teaching

- Providing a general overview of what makes up software engineering
- Teach a concrete method of doing a project (i.e. agile software development with test-driven development)
  - e.g. test driven development, user stories, agile project planning, ...

Course activities

- Reading assignments before the lecture: I will assume that you have read the assignments!!!
- Pre-flight tests checking that you have read the assignments
- Lectures every Monday 13:00 — approx 15:00 (Lecture plan is on the course Web page)
- Exercises in the E-databar (341.003, 015)
  - Teaching assistants will be present : 15:00 — 17:00
  - Expected work at home: *5 hours* (lecture preparation; exercises, . . .)
- 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
  - *no* written examination
- Week 04: Project introduction and forming of project groups → participation is mandatory!
- Week 07: Submission of project plans by the project groups
- Week 08: Start of the project
- Week 13: Demonstration of the projects (each project 15 min)

Course material

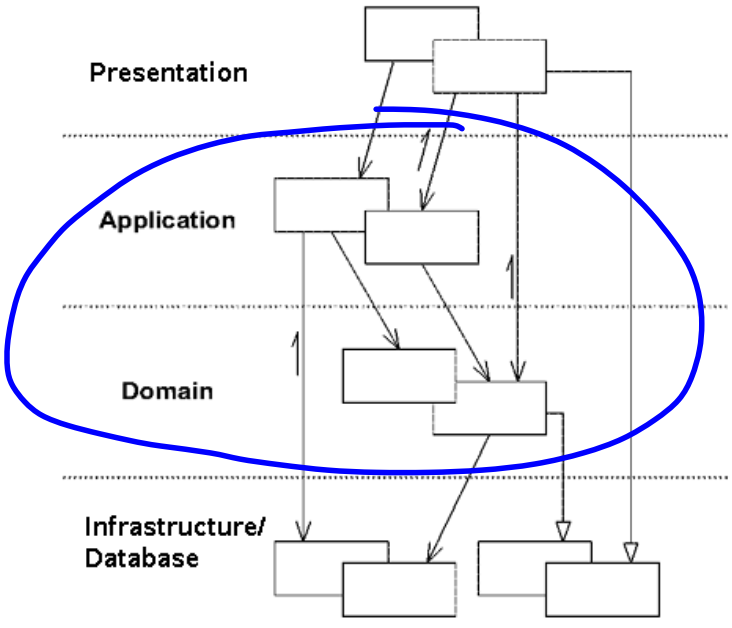
- Course Web page:
  `http://www.imm.dtu.dk/courses/02161` contains
  - practical information: (e.g. lecture plan)
  - Course material (e.g. slides, exercises, notes)
  - *Check the course Web page regularly*
- CampusNet: Is being used to send messages;
  - *make sure that you receive all messages from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

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

→ 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

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

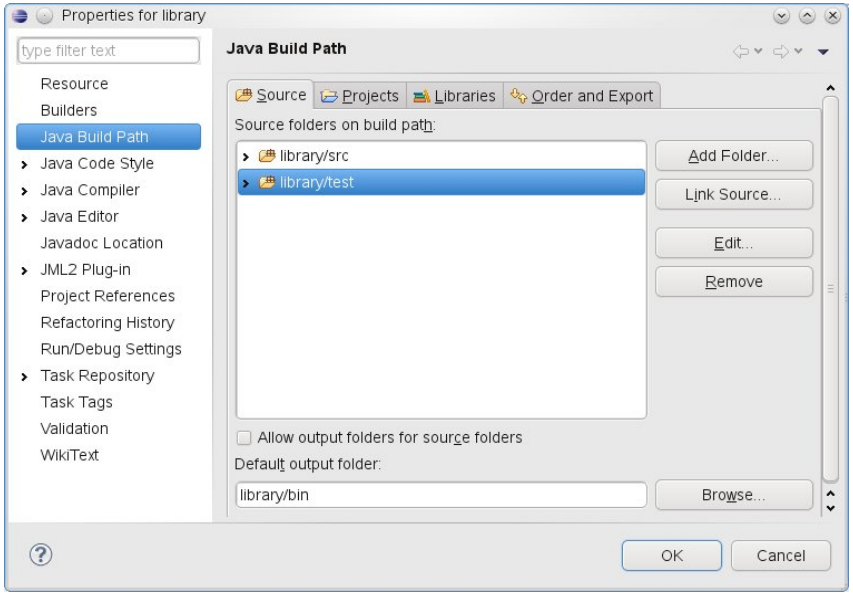
The following tests one scenario of the login functionality:

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());
}
```

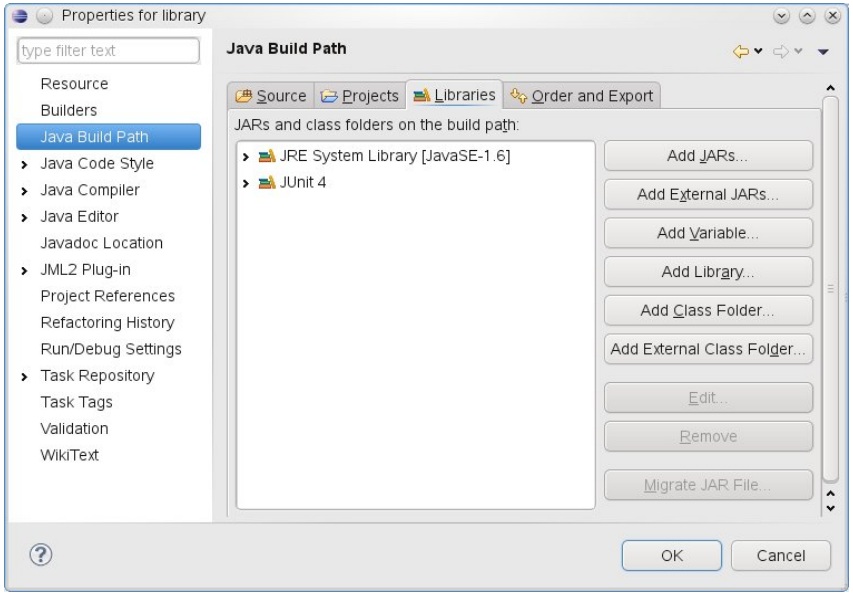
JUnit: Creating new Eclipse projects I

- With JUnit 4.x every class can have tests by just annotating the method with @Test
- However, I suggest to separate tests from the source code by putting them into their own source folder
  - This can be done either on creation time or by
  - Using the properties dialog (selecting Java Build Path and then Source)



JUnit: Creating new Eclipse projects II

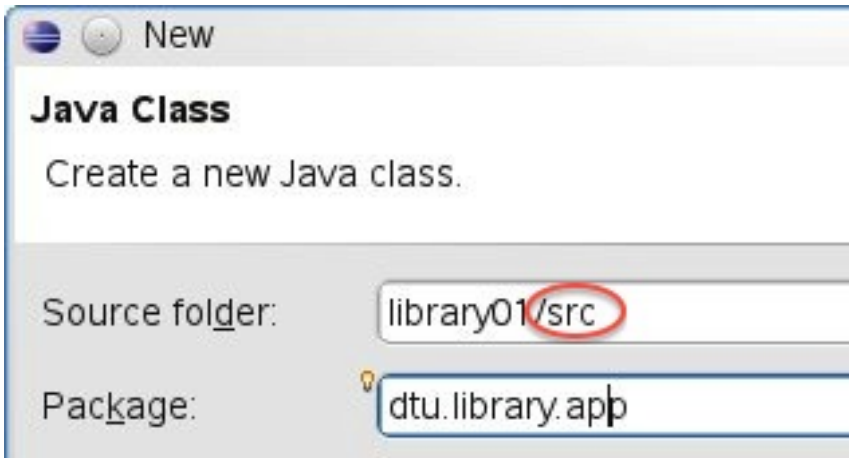
- In addition, the JUnit 4 libraries have to be available in the project. This can be done again in the properties dialog (selecting Java Build Path and then Libraries)



Eclipse code hint

- Eclipse helps with Test-Driven Development by offering help to fix the code, e.g. implementing missing classes and methods
- In the first test case, Eclipse does not know the class LibraryApp and proposes to create it if one clicks on the light bulb .

- Make sure that the source folder ends with src and not test



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 (not(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 {
    // Some block of code
} catch (OperationNotAllowedException e) {
 // Error handling code
}
```

- Test for the presence of an exception

```
@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

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