

# Software Engineering I (02161)

## Week 3

Assoc. Prof. Hubert Baumeister

DTU Compute  
Technical University of Denmark

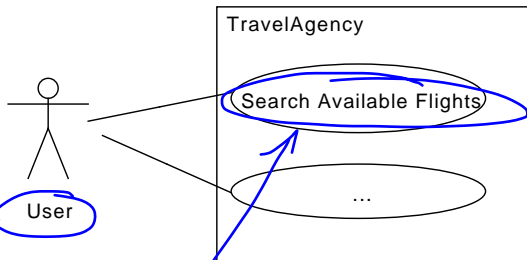
Spring 2013

# Recap

- ▶ Requirements Engineering
  - ▶ user- / system requirements
  - ▶ functional- / non-functional requirements
  - ▶ process: requirements elicitation, -specification, -validation
- ▶ Use Cases
  - ▶ "A set of scenarios with a common goal"
  - ▶ Two different views
    - ▶ use case diagrams
    - ▶ detailed use cases descriptions
- ▶ Glossary: Defines a common language

# Use Case: Diagram vs. Detailed Description

- ▶ Use Case: Search Available Flights
- ▶ Use Case Diagram



- ▶ Detailed Use Case Description

**name:** Search Available Flights

**description:** the user checks for available flights

**actor:** User

**main scenario:** ...

**alternative scenario:** ...

**note:** ...

# Contents

Software Testing

Acceptance tests

JUnit

Test Driven Development

How calendars and dates work in Java

Mock objects

Next week: Exam Project Groups

# Purpose of tests

- ▶ Goal: finding bugs
- ▶ Types of errors: requirement-, design-, implementation errors
- ▶ Types of testing:
  - ▶ validation testing test for functionality
  - ▶ defect testing finding defects

# Validation testing vs defect testing

## Tests

1. Start city is Copenhagen, destination city is Paris. The date is 1.3.2012. Check that the list of available flight contains SAS 1234 and AF 4245 *Validation test*
2. Start city is Copenhagen, the name of the destination city contains the Ctrl-L character. *Defect test*

## Questions

- a) Both tests are validation tests
- b) 1 is a validation test and 2 is a defect test.
- c) 1 is a defect test and 2 is a validation test
- d) Both tests are defect tests

# Types of tests

## 1. Developer tests

- a) Unit tests ← Classes, methods
- b) Component tests Set of classes
- c) System tests Integration of components

## 2. Release tests

Validation test + Defect testing

- a) Scenario based testing ← Use Case Scenarios
- b) Performance testing

## 3. User tests

- a) Acceptance tests

→ validation of req  
→ defects

# Contents

Software Testing

Acceptance tests

JUnit

Test Driven Development

How calendars and dates work in Java

Mock objects

Next week: Exam Project Groups



# Acceptance Tests

- ▶ Tests defined by / with the help of the user
- ▶ Traditionally
  - ▶ manual tests
  - ▶ by the customer
  - ▶ *after* the software is delivered
  - ▶ based on use cases / user stories
- ▶ Agile software development
  - ▶ automatic tests: JUnit, Fit, ...
  - ▶ created *before* the user story is implemented

# Example of an acceptance tests using JUnit

## ► Use case

name: Login Admin

actor: Admin

precondition: Admin is not logged in

main scenario

1. Admin enters password

2. System responds true

alternative scenarios: ...

postcondition: Admin is logged in

## ► Automatic test for the main scenario

```
@Test
public void testLoginAdmin() {
    LibraryApp libApp = new LibraryApp();


    assertFalse(libApp.adminLoggedIn());

    boolean login = libApp.adminLogin("adminadmin");

    assertTrue(login);
    assertTrue(libApp.adminLoggedIn());
}
```

*Main Scenario*

# Example of an acceptance tests using Fit



Test

Edit

Properties

Refactor

Where Used

Search

Files

Versions

We want to incinerate 2 tons of waste without any remains, and we are using 2 units per ton of the process "1MJ Electricity production (DK)".

WasteProcessFixture			
Amount	ExternalProcess	WasteProperty	edit?
-2 MJ/kg	1MJ Electricity production (DK)	Wet weight	true

ExternalProcessFixture			
Name	Flow produced	Amount	create?
1MJ Electricity production (DK)	"CO2, kg, to air"	20 kg/MJ	true

WasteFixture		
WasteProperty	Value	create?
Wet weight	2 kg	true

The result is a list of [ElementaryFlow](#) produced by the [WasteProcess](#).

LciFixture	
ElementaryFlow	Amount
CO2, kg, to air"	80 kg

green

Fit: <http://fit.c2.com> and Fitness: <http://www.fitnessse.org>

# Contents

Software Testing

Acceptance tests

JUnit

Test Driven Development

How calendars and dates work in Java

Mock objects

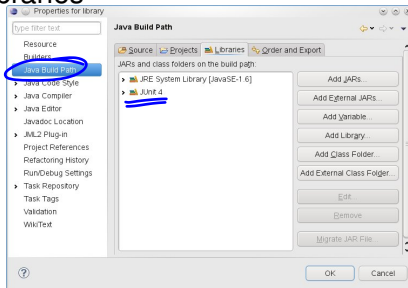
Next week: Exam Project Groups

# JUnit

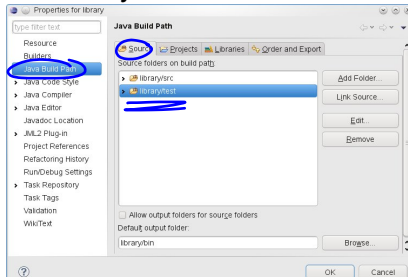
- ▶ Framework for automated tests in Java
- ▶ Developed by Kent Beck and Erich Gamma
- ▶ Unit-, component-, and *acceptance* tests
- ▶ <http://www.junit.org> *SUnit (Smalltalk = S)*
- ▶ xUnit

# JUnit and Eclipse

## ▶ JUnit 4.x libraries



## ▶ New source directory for tests



## JUnit 4.x structure

```
import org.junit Test;  
import static org.junit.Assert.*;  
  
public class C {  
    @Test  
    public void m1() {...}  
    @Test  
    public void m2() throws Exception {...}  
    ...  
}
```

- ▶ Independent tests
- ▶ No try-catch blocks (exception: checking for exceptions)

## JUnit 4.x structure (Before and After)

```
...
public class C {
    @After
    public void n2() {...}
    @Before
    public void n1() {...}
    @Test
    public void m1() {...}
    @Test
    public void m2() {...}
    ...
}
```

*Handwritten blue annotations:*

- A blue oval around `@After`.
- A blue oval around `@Before`.
- Handwritten blue text `n1; m1; n2; n1; m2; n2` next to the `n2()` method.

*Handwritten blue text:* @BeforeClass

*Handwritten blue text:* @AfterClass



# Structure of test cases

## Acceptance tests

- ▶ Test class = one use case
- ▶ Test method = one scenario
- ▶ Use inheritance to share sample data between use cases

```
public class SampleDataSetup {  
    @Before()  
    public void setUp() { .. }  
    @After()  
    public void tearDown { .. }  
    ... }
```

```
public class TestBorrowBook extends SampleDataSetup {..}
```



# JUnit assertions

## General assertion

```
import static org.junit.Assert.*;
```

```
assertTrue(bexp)  
assertTrue(msg, bexp)
```

without static: Assert.assertTrue(...)

## Specialised assertions for readability

1. assertFalse(bexp)    assertTrue(! bexp)
2. fail()    assertTrue(false)
3. assertEquals(exp, act)    assertTrue(exp.equals(act))
4. assertNull(obj)    assertTrue(obj == null)
5. assertNotNull(obj)    assertTrue(obj != null)
- ...

# JUnit: testing for exceptions

- ▶ Test that method `m()` throws an exception `MyException`

```
@Test
public void testMThrowsException() {
    ...
    try {
        m();
        fail(); // If we reach here, then the test fails because
                // no exception was thrown
    } catch(MyException e) {
        // Do something to test that e has the correct values
    }
}
```

- ▶ **Alternative**

```
@Test(expected=MyException.class)
public void testMThrowsException() {...}
```

# Contents

Software Testing

Acceptance tests

JUnit

Test Driven Development

- Test Driven Development

- Example of Test-Driven Development

- Refactoring

How calendars and dates work in Java

Mock objects

Next week: Exam Project Groups

# Test-Driven Development

- ▶ Test *before* the implementation
- ▶ Tests = expectations on software
- ▶ All kind of tests: unit-, component-, system tests

# TDD cycle

- ▶ Repeat

- red: Create a failing test

- green: Make the test pass *as simple as possible*

- refactor: clean up your code → *software design*

- ▶ Until: no more ideas for tests

- ▶ Important: One test at a time

# Ideas for tests

1. Use case scenarios (missing functions): Acceptance tests
  2. Possibility for defects (missing code): Defect tests *Regression test*
  3. You want to write more code than is necessary to pass the test
  4. Complex behaviour of classes: Unit tests *Class based*
  5. Code experiments: "How does the system behave, if ..."
- Make a list of new test ideas

# TDD example: Borrow Book

## ► Use case

**name:** borrow book

**description:** the user borrows a book

**actor:** user

**main scenario:**

1. the user borrows a book

**alternative scenario**

1. the user wants to borrow a book, but has already 10 books borrowed
2. the system presents an error message



# Create a test for the main scenario

## ▶ test data:

- ▶ a user with CPR "1234651234" and book with signature "Som001"
- ▶ Test case
  - ▶ Retrieve the user with CPR number "1234651234"
  - ▶ Retrieve the book by the signature "Som001"
  - ▶ The user borrows the book
  - ▶ The book is in the list of books borrowed by that user

## Create a test for the main scenario

```
@Test
public void testBorrowBook() throws Exception {
    String cprNumber = "1234651234";
    User user = libApp.userByCprNumber(cprNumber);
    assertEquals(cprNumber, user.getCprNumber());
    String signature = "Som001";
    Book book = libApp.bookBySignature(signature);
    assertEquals(signature, book.getSignature());
    List<Book> borrowedBooks = user.getBorrowedBooks();
    assertFalse(borrowedBooks.contains(book));
    user.borrowBook(book);
    borrowedBooks = user.getBorrowedBooks();
    assertEquals(1, borrowedBooks.size());
    assertTrue(borrowedBooks.contains(book));
}
```

Size = 0

# Implement the main scenario

```
public void borrowBook(Book book) {  
    borrowedBooks.add(book);  
}
```

# Create a test for the alternative scenario

- ▶ test data:

- ▶ a user with CPR "1234651234", book with signature "Som001", and 10 books with signatures "book1", ..., "book10"

- ▶ Test case

- ▶ Retrieve the user with CPR number "1234651234"
- ▶ Retrieve and borrow the books with signature "book1", ..., "book10"
- ▶ Retrieve and borrow the book by the signature "Som001"
- ▶ Check that a TooManyBooksException is thrown

# Implementation of the alternative scenario

```
public void borrowBook(Book book) throws TooManyBooksException  
if (book == null) return; → needs a test first  
    if (borrowedBooks.size() >= 10) {  
        throw new TooManyBooksException();  
    }  
    borrowedBooks.add(book);  
}
```

# More test cases

- ▶ What happens if `book == null` in `borrowBook`?

- ▶ Test Case:

- ▶ Retrieve the user with CPR number "1234651234"
- ▶ Call the `borrowBook` operation with the null value
- ▶ Check that the number of borrowed books has not changed

# Final implementation

```
public void borrowBook(Book book) throws TooManyBooksException {
    if (book == null) return;
    if (borrowedBooks.size() >= 10) {
        throw new TooManyBooksException();
    }
    borrowedBooks.add(book);
}
```

## Another example

- ▶ Creating a program to generate the n-th Fibonacci number
- Codemanship's Test-driven Development in Java by Jason Gorman
- <http://youtu.be/nt2KKUSSJsY>
- ▶ Note: Uses JUnitMax to run JUnit tests automatically whenever the test files change ([junitmax.com](http://junitmax.com))



# Refactoring and TDD

- ▶ Third step in TDD
- ▶ restructure the system without changing its functionality
- ▶ Goal: improve the design of the system, e.g. remove code duplication (DRY principle)
- ▶ Necessary step for TDD
- ▶ Requires good test suite



# Design and TDD

## 1. System level:

- ▶ Rough system design: component diagram, class diagram,  
...

## 2. Use case level:

- ▶ design a possible solution, e.g. using class diagrams,  
sequence diagrams
- ▶ Important:
  - ▶ Implement the design using TDD
  - ▶ Add new tests / use case scenarios as you go along
  - Adapt the design as needed (Refactoring)

# TDD: Advantages

- ▶ Test benefits
  - ▶ Good code coverage: Only write production code to make a failing test pass
  - ▶ Regression test suite: Make sure old functionality works after adding new features or doing bug fixes
- ▶ Design benefits
  - ▶ Helps design the system: defines usage of the system before the system is implemented
  - ▶ Testable system

# Contents

Software Testing

Acceptance tests

JUnit

Test Driven Development

How calendars and dates work in Java

Mock objects

Next week: Exam Project Groups

# How to use Date and calendar (I)

- ▶ Date class deprecated
- ▶ Calendar and GregorianCalendar classes
- ▶ An instance of Calendar is created by

```
new GregorianCalendar() // current date and time
new GregorianCalendar(2011, Calendar.JANUARY, 10)
```
- ▶ Note that the month is 0 based (and not 1 based). Thus 1 = February.
- ▶ Best is to use the constants offered by Calendar, i.e. Calendar.JANUARY

# How to use Date and calendar (I)

- ▶ One can assign a new calendar with the date of another by

```
newCal.setTime(oldCal.getTime())
```

- ▶ One can add years, months, days to a Calendar by using add: e.g.

```
cal.add(Calendar.DAY_OF_YEAR, 28)
```

- ▶ Note that the system roles over to the new year if the date is, e.g. 24.12.2010

- ▶ One can compare two dates represented as calendars using before and after, e.g.

```
currentDate.after(dueDate)
```

# Contents

Software Testing

Acceptance tests

JUnit

Test Driven Development

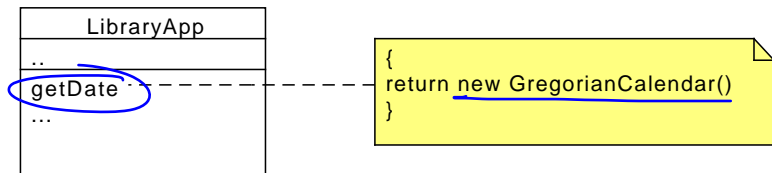
How calendars and dates work in Java

Mock objects

Next week: Exam Project Groups

# Problems

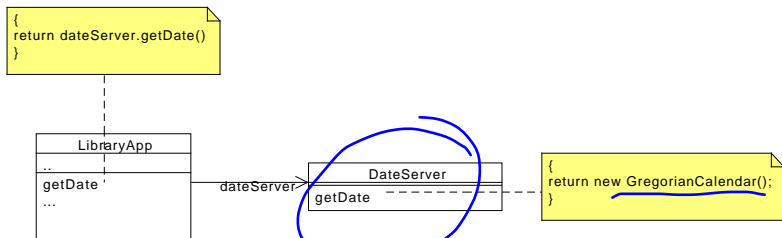
- ▶ How to test that a book is overdue?



- ▶ Solution: Create a *mock* object returning fixed dates
- ▶ Problem: Can't replace `GregorianCalendar` class

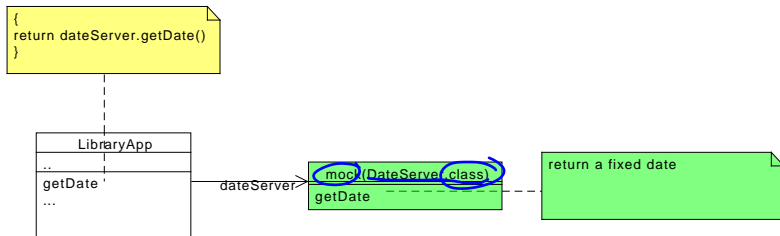


# Creating a DateServer class



# Creating a DateServer class

- The DateServer can be mocked



# How to use

- ▶ Import helper methods

```
import static org.mockito.Mockito.*;
```

- ▶ Create a mock object on a certain class

```
SomeClass mockObj = mock(SomeClass.class)
```

- ▶ return a predefined value for m1(args)

```
when(mockObj.m1(args)).thenReturn(someObj);
```

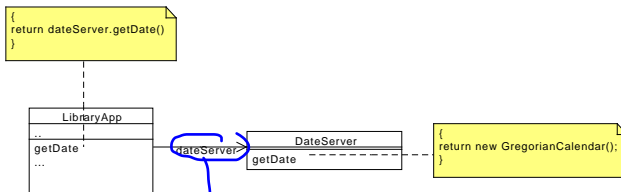
- ▶ verify that message m2(args) has been sent

```
verify(mockObj).m2(args);
```

# Mock Example 1: Overdue book

```
@Test
public void testOverdueBook() throws Exception {
    DateServer dateServer = mock(DateServer.class);
    libApp.setDateServer(dateServer);
    Calendar cal = new GregorianCalendar(2011, Calendar.JANUARY, 10);
    when(dateServer.getDate()).thenReturn(cal);
    ...
    user.borrowBook(book);
    newCal = new GregorianCalendar();
    newCal.setTime(cal.getTime());
    newCal.add(Calendar.DAY_OF_YEAR, MAX_DAYS_FOR_LOAN + 1);
    when(dateServer.getDate()).thenReturn(newCal);
    assertTrue(book.isOverdue());
}
```

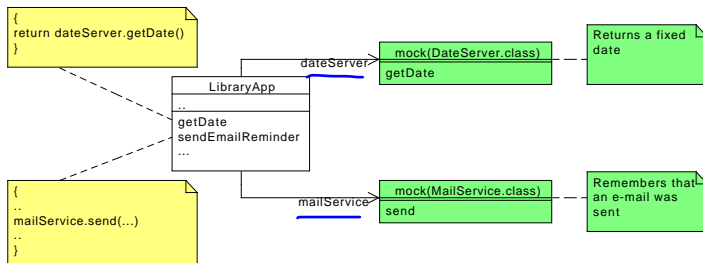
# LibraryApp Code



```
public class LibraryApp {
    private DateServer ds = new DateServer();
    public setDateServer(DateServer ds) { this.ds = ds; }
    ...
}
```

```
public class DateServer {
    public Calendar getDate() {
        return new GreogorianCalendar();
    }
}
```

# Testing for e-mails



```
@Test
public void testEmailReminder() throws Exception {
    DateServer dateServer = mock(DateServer.class);
    libApp.setDateServer(dateServer);

    MailService mailService = mock(MailService.class);
    libApp.setMailService(mailService);
    ...
    libApp.sendEmailReminder();
    verify(mailService).send("..", "..", "..");
}
```

# Verify

Check that no messages have been sent

```
verify(ms, never()).send(anyString(), anyString(), anyString());
```

**Mockito documentation:** <http://docs.mockito.googlecode.com/hg/org/mockito/Mockito.html>

# Contents

Software Testing

Acceptance tests

JUnit

Test Driven Development

How calendars and dates work in Java

Mock objects

Next week: Exam Project Groups



# Next Week

- ▶ Systematic tests and code coverage
- ▶ Exam project introduction
- ▶ Group forming: mandatory participation in the lecture
- ▶ Group forming: *next week*
  - ▶ Either you are **personally** present or someone can **speak for you**
  - ▶ If not, then there is no guarantee for participation in the exam project

group size: 2-4