Software Engineering I (02161)

Week 3

Assoc. Prof. Hubert Baumeister

Informatics and Mathematical Modelling
Technical University of Denmark

Spring 2011

Summary

- Requirements analysis is about finding out what the software should be able to do
- Two types of requirements functional and non-functional requirements
- Process: Discover, Document, Validate
- Use cases
    - Used for both finding and documenting the requirements
    - What are the functions the user can perform with the software?
        - Use case diagram: Graphical overview over the functionality of the software
        - Detailed use cases: Detailed (textual) description of what the software should do
- User Stories (used in XP)
    - Focus on interaction scenarios relevant for the user; tell a ”story” about the system
- Glossary: Document domain knowledge and define a common language between customer and software developer

What are tests?

- Tests are a method of discovering errors
  - program errors
  - design errors
  - requirement errors
- Tests compare the behaviour of the program with what is expected of the program
- Tests execute the program in many different situations (in principle all possible situations) and check if the program reacts correctly to tests
  - How to get to all possible situations?
  - How to describe what the expected behaviour of the program is?

Two purposes of tests

- Validation testing
  - Tests that the user requirements are satisfied
  - Have we built the right system?
- Defect testing
  - Tests that the system has no defects
  - Have we built the system right?

Types of tests in the Development Process

1. Developer tests
   - Tests done by the developer
   - Main purpose is to find defects (defect testing)

   a) Unit tests
      - Tests the functionality of a single unit (e.g. class) of the system

   b) Component tests

   c) System tests
2. Release tests
   - Done by the developer
   - Validation test: Have we build the right system?

   a) Scenario testing (based on Use case scenarios or user stories)

   b) Performance testing (but also reliability, security, . . .)
3. User tests
   - Done by the customer
   - Used to decide whether the contract is satisfied or not

   a) Acceptance tests

Systematic tests

- Goal: solving the problem of finding sufficient test cases to cover all possible situations
  - Not too few because that could miss some defects
  - Not too many, as tests are expensive (defining and, if manually, executing)

→ Solution: partition based testing

- The basic idea is to find a partition of all possible ways the program can run (e.g. of the external interactions, like user input)
- Do it for one is the same as doing it for all elements of the class

Two alternatives to get to these partitions

- Look at the problem and based on the problem domain look at the partitions. The assumption here is that the program logic won't be more complicated than the problem description
    - Focuses on errors implementing the problem definition
    - Called: Black box– or functional tests
- Look into the program itself and see how it works. Make sure that each statement is executed by at least one test case.
    - Focus on errors in the program implementation
    - Called White box– or structural tests

Example of a black box test (I): min, max computation

Problem: Find the minimum and the maximum of a list of integers

- Definition of the input partitions

Input data set	Input property
A	No numbers
B	One number
C1	Two numbers, equal
C2	Two numbers, increasing
C3	Two numbers, decreasing
D1	Three numbers, increasing
D2	Three numbers, decreasing
D3	Three numbers, greatest in the middle
D4	Three numbers, smallest in the middle

- Definition of the test values and expected results

Input data set	Contents	Expected output
A	(no numbers)	Error message
B	17	17 17
C1	27 27	27 27
C2	35 36	35 36
C3	46 45	45 46
D1	53 55 57	53 57
D2	67 65 63	63 67
D3	73 77 75	73 77
D4	89 83 85	83 89

Example of a white box test: min, max computation

- The same problem as above; now the input partitions are derived from the program (white box)

```
public static void main ( String[] args )
{
  int mi, ma;
  if (args.length == 0)                                /* 1 */
    System.out.println("No numbers");
  else
    {
      mi = ma = Integer.parseInt(args[0]);
      for (int i = 1; i < args.length; i++)            /* 2 */
        {
          int obs = Integer.parseInt(args[i]);
          if (obs > ma) ma = obs;                      /* 3 */
          else if (mi < obs) mi = obs;                 /* 4 */
        }
      System.out.println("Minimum = " + mi + "; maximum = " + ma);
    }
}
```

Choice	Input data set	Input property
1 true	A	No numbers
1 false	B	At least one number
2 zero times	B	Exactly one number
2 once	C	Exactly two numbers
2 more than once	E	At least three numbers
3 true	C	Number $>$ current maximum
3 false	D	Number $\leq$ current maximum
4 true	E number 3	Number $\leq$ current maximum and $>$ current minimum
4 false	E number 2	Number $\leq$ current maximum and $\leq$ current minimum

Example of a white box test (II): Test cases

Choice	Input data set	Input property
1 true	A	No numbers
1 false	B	At least one number
2 zero times	B	Exactly one number
2 once	C	Exactly two numbers
2 more than once	E	At least three numbers
3 true	C	Number $>$ current maximum
3 false	D	Number $\leq$ current maximum
4 true	E number 3	Number $\leq$ current maximum and $>$ current minimum
4 false	E number 2	Number $\leq$ current maximum and $\leq$ current minimum

Input data set	Contents	Expected output
A	(no numbers)	'No numbers'
B	17	17 17
C	27 29	27 29
D	39 37	37 39
E	49 47 48	47 49

Summary of the method

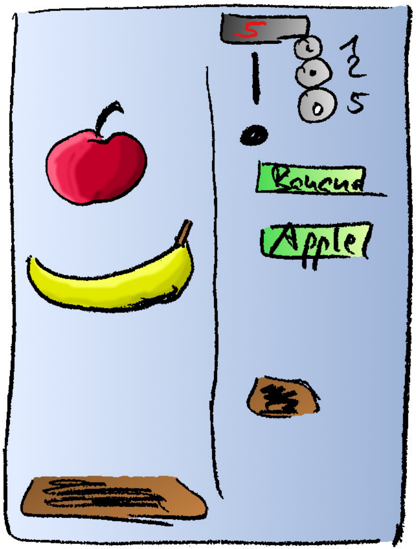
- Create a test plan consisting of two tables

1. table of possible input partitions
   - Name of the input partition
   - Property associated to that input partition
2. table of actual test cases (input / expected output)
   - Name of input partition
   - Concrete input values / input actions
   - Expected output values / reactions of the system

- Only difference between black box and white box: how the partitions are determined
  - black box: from the problem specification
  - white box: from the program code
- Technique can be applied to acceptance tests as well as unit tests

→ It is important to learn how to in all possible ways a program can be (mis)used

Example Vending Machine



- Actions
  - Input coins
  - Press button for bananas or apples
  - Press cancel
- Displays
  - current amount of money input
- Effects
  - Return money
  - Dispense banana or apple

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

Use Case: Buy Fruit (cont.)

alternative scenarios (cont.)

b1 User inputs less coins than necessary
b2 user selects a fruit
b3 No fruit is dispensed
b4 User adds the missing coins
b5 Fruit is dispensed
c1 User selects fruit
c2 User adds sufficient or more coins
c3 vending machine dispenses fruit and rest money
d1 user enters coins
d2 user selects cancel
d3 money gets returned

Use Case: Buy Fruit (cont.)

alternative scenarios (cont.)

- e1 user enters correct coins
- e2 user selects fruit but vending machine does not have the fruit anymore
- e3 nothing happens
- e4 user selects cancel
- e5 the money gets returned
- f1 user enters correct coins
- f2 user selects a fruit but vending machine does not have the fruit anymore
- f3 user selects another fruit
- f4 if money is correct fruit with rest money is dispensed; if money is not sufficient, the user can add more coins

Functional Test: for Buy Fruit Use Case: Input Data Sets

Input data set	Input property
A	Exact coins; enough fruits; first coins, then fruit selection
B	Exact coins; enough fruits; first fruit selection, then coins
C	Exact coins; not enough fruits; first coins, then fruit selection, then cancel
D	Exact coins; not enough fruits; first fruit selection, then coins, then cancel
E	More coins; enough fruits; first coins, then fruit selection
F	More coins; enough fruits; first fruit selection, then coins
G	More coins; not enough fruits; first coins, then fruit selection, then cancel
H	More coins; not enough fruits; first fruit selection, then coins, then cancel
I	Less coins; enough fruits; first coins, then fruit selection
J	Less coins; enough fruits; first fruit selection, then coins
K	Less coins; not enough fruits; first coins, then fruit selection, then cancel
L	Less coins; not enough fruits; first fruit selection, then coins, then cancel

Functional Test for Buy Fruit Use Case: Test Cases

Input data set	Contents	Expected Output
A	1,2; apple	apple dispensed
B	Apple; 1,2	apple dispensed
C	1,2; apple; cancel	no fruit dispensed; returned DKK 3
D	Apple; 1,2; cancel	no fruit dispensed; returned DKK 3
E	5, apple	apple dispensed; returned DKK 2
F	Apple; 5	apple dispensed; returned DKK 2
G	5, apple; cancel	no fruit dispensed; returned DKK 5
H	Apple; 5; cancel	no fruit dispensed; returned DKK 5
I	5; banana	no fruit dispensed; current money shows 5
J	Banana; 5,1	no fruit dispensed; current money shows 6
K	5,1; banana; cancel	no fruit dispensed; returned DKK 6
L	Banana; 5,1;cancel	no fruit dispensed; returned DKK 6

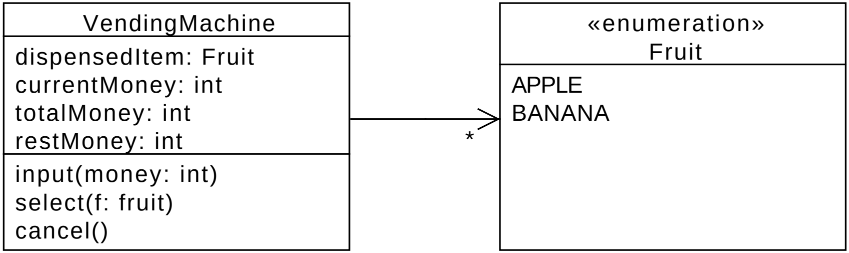
Manual vs Automated Tests

- The test cases can be tested manually
  - Open the application
  - Input the data according to the input data set description
  - Check that the output is the expected one
  - Very expensive
  - Can't be done too often
    - difficult to do regressiontests
- But also automatically
  - There are tools that execute the user interface automatically
    - But they have a lot of problems: e.g. don't tolerate additional GUI elements or changed GUI elements
  - Easier: test the application automatically
    - Advantage: Easy regression tests
    - Tests that no old defects are introduced again
    - Difficulties testing GUI:

      → Solution: Layered architecture and test the application layer

      → cf. lecture on architecture

Application Layer



Functional Test for Buy Fruit Use Case: JUnit Tests

```
public void testInputDataSetA() {
    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());
}

public void testInputDataSetB() {
    VendingMachine m = new VendingMachine(10, 10);
    m.selectFruit(Fruit.APPLE);
    m.input(1);
    m.input(2);
    assertEquals(0, m.getCurrentMoney());
    assertEquals(Fruit.APPLE, m.getDispensedItem());
}
```

Functional Test: JUnit Tests (cont.)

```
public void testInputDataSetC() {
    VendingMachine m = new VendingMachine(0, 0);
    m.input(1);
    m.input(2);
    assertEquals(3, m.getCurrentMoney());
    m.selectFruit(Fruit.APPLE);
    assertEquals(null, m.getDispensedItem());
    m.cancel();
    assertEquals(null, m.getDispensedItem());
    assertEquals(3, m.getRest());
}

public void testInputDataSetD() {
    VendingMachine m = new VendingMachine(0, 0);
    m.selectFruit(Fruit.APPLE);
    m.input(1);
    m.input(2);
    assertEquals(3, m.getCurrentMoney());
    m.cancel();
    assertEquals(null, m.getDispensedItem());
    assertEquals(3, m.getRest());
}
```

...

Acceptance Tests

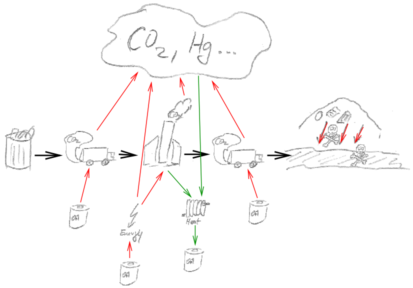
- These are tests defined by the customer.
- If the software passes these tests, then the software is said to be accepted and can be delivered to the customer
- Traditional
  - usually manual tests administered by the customer after the software has been build
  - Tests can be ad hoc or planned (e.g. following a systematic approach)
- Agile/XP
  - Acceptance tests are created before the user story is implemented and is usually automatic
  - These tests can be expressed in JUnit (done in this course) but also using more customer friendly notations, like fit (`fit.c2.com`)

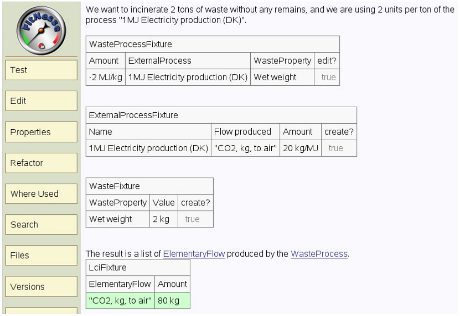
Example of XP acceptance tests using JUnit

Adding a book to the library

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

   // Check first that the administrator is not logged in.
   assertFalse(libApp.adminLoggedIn());

   // Step 1)
   boolean login = libApp.adminLogin("adminadmin");

   // Step 2) Check that the method returned true and check that admin is
   assertTrue(login);
   assertTrue(libApp.adminLoggedIn());
}
```

Example of XP acceptance tests using Fit





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

Test Driven Development

- Use tests to drive the development (write tests before the code)
- However: beware of waterfall: Instead, incrementally develop tests and functionality
  - $\rightarrow$ repeat: 1) choose new functionality 2) Define test 3) Implement functionality 4) Refactor code
- Functionality can be a user story (i.e. use case scenario), but also functionality of unit (e.g. classes): I.e. driver can be
  - acceptance tests
  - unit tests
- Developed out of XP test-first practice

TDD cycle

- TDD Cycle
  - red: Create a test (acceptance or unit)
    - → Important: test should fail
    - Tests can contain new classes, operations, etc.
  - green: Implement the code until the test is green
    - First fix compiler errors
    - Then fix the program logic
  - refactor the software to the best possible design
- Repeat the cycle with new tests coming from
  - behaviour (black box)
  - implementation (white box)
  - bugs (regression tests)
  - Experiments: How does the software behave if . . . ?
- Done if
  - a) all the intended functionality is tested and implemented
  - b) no additional loose ends remain (e.g. possible error and exceptional situations; what if scenarios ...)

Refactoring

- Refactoring is to restructure the software without changing its functionality
- Goal: Improve the design of the software
- With agile software development this contributes to the design activity
- This is step is necessary to keep the software simple and adaptable
- Ideally, refactoring corresponds to small transformations on the code that change the structure but not the functionality
  - e.g. rename method/class/variable; extract a method (create a new method from some lines of code), create new classes, merge two classes, create superclasses, pull-up/push-down fields in the class hierarchy

Refactoring and tests

- Sufficient (automated) tests are prerequisite to refactoring
  - Without tests it is not ensured the refactoring does not destroy existing functionality → regression tests

→ Software becomes soft

- Very helpful: Tool support (e.g. Eclipse refactoring menu: rename, extract method, encapsulate field, . . .)
- Refactoring can change tests: e.g. if the interfaces / design of the system changes

Refactoring: History

- Refactoring: PhD thesis by William Opdyke (1992): Analysis of the concept and first refactoring transformations
- First tool: Smalltalk refactoring browser
- Martin Fowler: Refactoring: Improving the Design of Existing Code (1999)

TDD, Refactoring and design

- How to incorporate design into agile methods / XP?

1) XP usually starts out with a rough understanding of the architecture of the system
2) Discussion on how a user story can be realised is design
3) Writing a test uses the system according to a design → writing tests is a step of designing the system
4) After the functionality is implemented and the tests succeed, the question is, how can the software be improved to better represent the programs intentions → refactoring

JUnit

- Framework for automated tests
- Developed by Kent Beck and Erich Gamma (based on Kent Beck's SUnit (Smalltalk Unit))
- Intended for unit tests, but can also be used for acceptance tests
- Info `http://www.junit.org`
- Base for similar frameworks for other languages *x* (*x*Unit)

JUnit 4.x structure

- Make sure that Eclipse uses the JUnit library (cf. slides of first lecture)

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

public class C {
   @Test
   public void m1() {..}
   @Test
   public void m2() {..}
   ...
}
```

- Test methods m1, m2, should be independent from each other
- Eclipse has a tool to run these tests and nicely view the result
  - shows passed, failed, and errors

  → pass on exceptions (don't use try/catch to hide 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() {..}
   ...
}
```

- Methods with Before and After annotations are always executed before each test method
  - n1(); m1(); n2(); n1(); m2(); n2(), ...
  - Contain code for setting up the test case (e.g. test data) and destroying it (e.g. clearing the database)

JUnit assertions

- General assertion

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

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

- Specialised assertions for readability

```
assertFalse(bexp) // The same as assertTrue(!bexp)
fail() // The same as assertTrue(false)
assertEquals(exp,act); // assertTrue(exp.equals(act))
assertNull(obj) // assertTrue(obj == null)
assertNotNull(obj) // assertTrue(obj != null)
...
```

- The first argument to all assert statements can be a message explaining the assertion
    - e.g. fail(msg), assertEquals(msg,exp,act), ...

JUnit: testing for exceptions

- Test that method m() throws an exception MyException

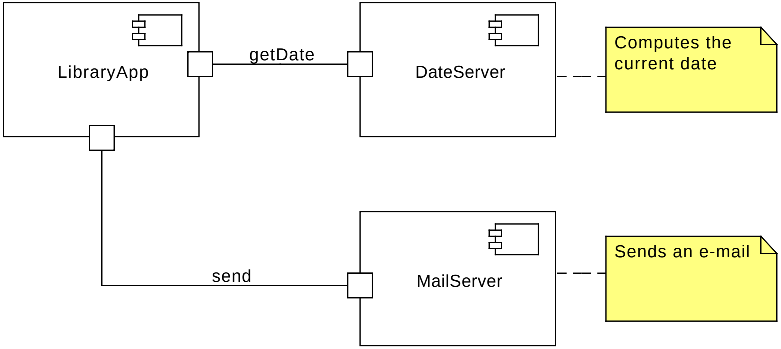
```
@Test
public void testMThrowsException() {
   ...
   try {
      m();
      fail(); // If we reach here, then the test fails because
              // no exception was thrown
   } catch(MyException e) {
      // Do something to test that e has the correct values
   }
}
```

Mock objects

- Are a framework for testing systems independent from other systems
  - Diagram of a system which depends on other systems (Library System and E-Mail sending application and DateServer)
- Basic idea is that all the messages to the subsystems get
  a) recorded, so that one can verify that the system is used correctly
  b) can return some previously defined values (called stubbing)
- A stub only refers to b), while a mock refers to a) and b)
- The framework we use is Mockito
  - `http://code.google.com/p/mockito/`
  - Documentation `http://docs.mockito.googlecode.com/hg/org/mockito/Mockito.html`
- Don't forget to test the system that is mocked!!

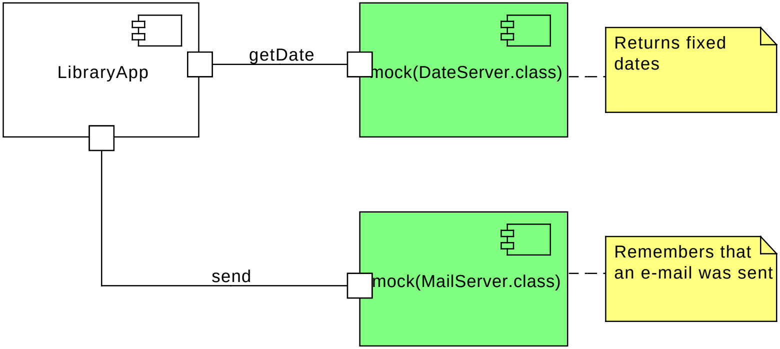
Mock Examples

- How to test that a book is overdue?
  - Test depends on that time passes between borrowing a book and checking if it is overdue
- How to test that an e-mail is sent to those who have overdue books?



Mock Examples

- How to test that a book is overdue?
  - Test depends on that time passes between borrowing a book and checking if it is overdue
- How to test that an e-mail is sent to those who have overdue books?



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());
}
```

Mock Objects

- Need to import

  ```
  import static org.mockito.Mockito.*;
  ```
- Create a mock object on a certain class
  - The mock object can now be used at any place an object of the original class can be used

    ```
    SomeClass mockObj = mock(SomeClass.class)
    ```
- Define what to return when sending m1(args) to the mock object
  - Note that m1 has to be a method of class SomeClass

    ```
    when(mockObj.m1(args)).thenReturn(someObj);
    ```
- Check that method m2 with arguments args has been called on the mock object
  - Again, m2 needs to be a method of class SomeClass

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

Mock Example 2: Send E-Mail reminder

```
@Test
public void testEMailReminder() throws Exception {
   MailService ms = mock(MailService.class);
   libApp.setMailService(ms);
   DateServer dateServer = mock(DateServer.class);
   libApp.setDateServer(dateServer);
   ...
   user.borrowBook(book);

   Calendar newCal = new GregorianCalendar();
   newCal.setTime(cal.getTime());
   newCal.add(Calendar.DAY_OF_YEAR, 28+1);
   when(dateServer.getDate()).thenReturn(newCal);
   assertTrue(book.isOverdue());

   libApp.sendEMailReminder();
   String email = user.getEmail();
   String subject = "Overdue book(s)";
   String text = "You have 1 overdue book(s)";
   verify(ms).send(email,subject,text);
}
```

Calendar and Dates in Java

- Originally, there was only one class representing the Date
- Its implementation was based on how many milliseconds has passed since January 1, 1970 00:00:00 GMT
- Could ask for year, month, day in a month etc.
- However, did not consider Internationalisation
  - → the corresponding methods in date are therefore deprecated

→ Introduction of Calendar class with subclass GregorianCalendar for these type of questions

How to use Date and calendar (I)

- 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)`

Software Testing

- Two purposes: Validation testing and defect testing
- Systematic tests: Design your tests by looking at all possible input classes
  - Black box test: test cases are generated from the problem description
  - White box test: test cases are generated by looking into the implementation
- Manual vs automatic tests: automatic tests are preferred
- Acceptance tests: tests defined with the customer to define when a user story / use case is implemented: Best automatic
- Test driven development:
  - $\rightarrow$ repeat: 1) choose new functionality 2) Define test 3) Implement functionality 4) Refactor
- Refactoring: improve the desing of the code without adding new functionality
  - Essential for the design process in XP
- Java Date and Calendar classes