Software Engineering I (02161)

Week 4

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2016

Recap

- Test in general
  - validation and defect test
  - unit-, component-, system-, integration-, acceptance tests
- Test driven development
  - mechanics: repeat: failing test (red), create production code (green), refactor

  → JUnit

  → Mock Objects (Mockito)
- Dates in Java (GregorianCalendar)
- Today:
  1. Systematic tests
     a) White box test
     b) Black box test
  2. Code coverage

Contents

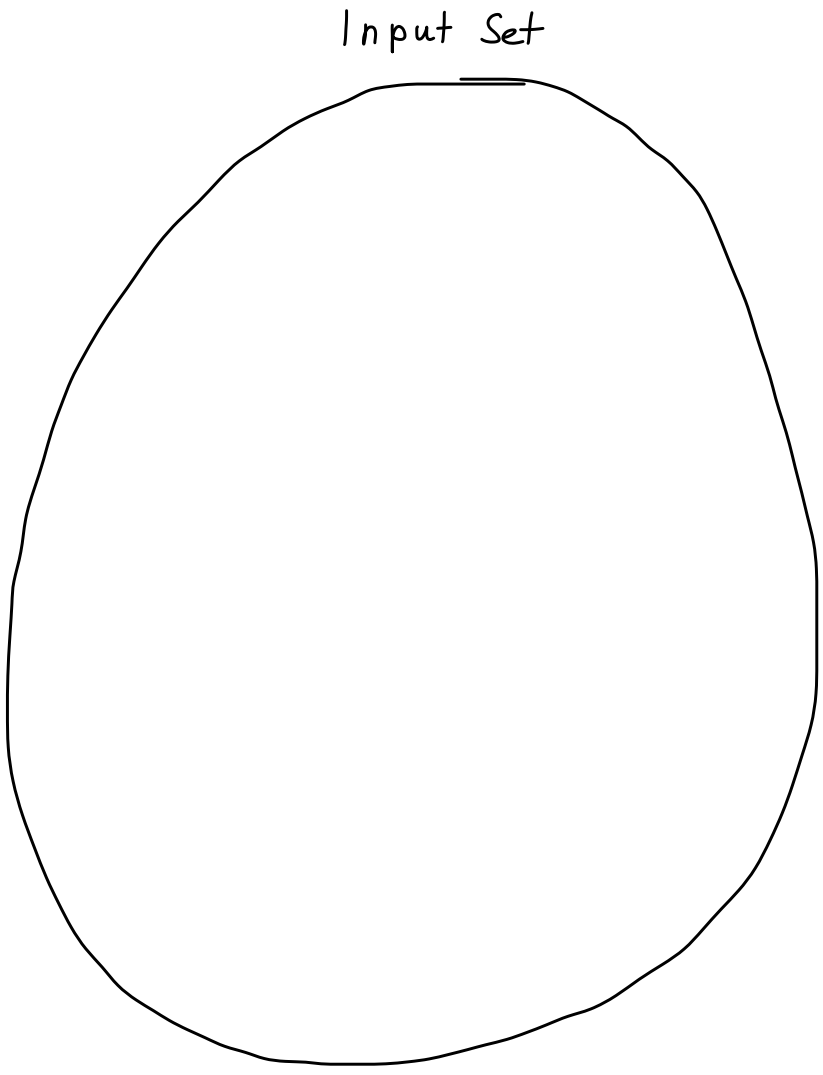
Systematic testing

- Tests are expensive
- Impractical to test all input values
- Not too few because one could miss some defects

→ Partition based tests

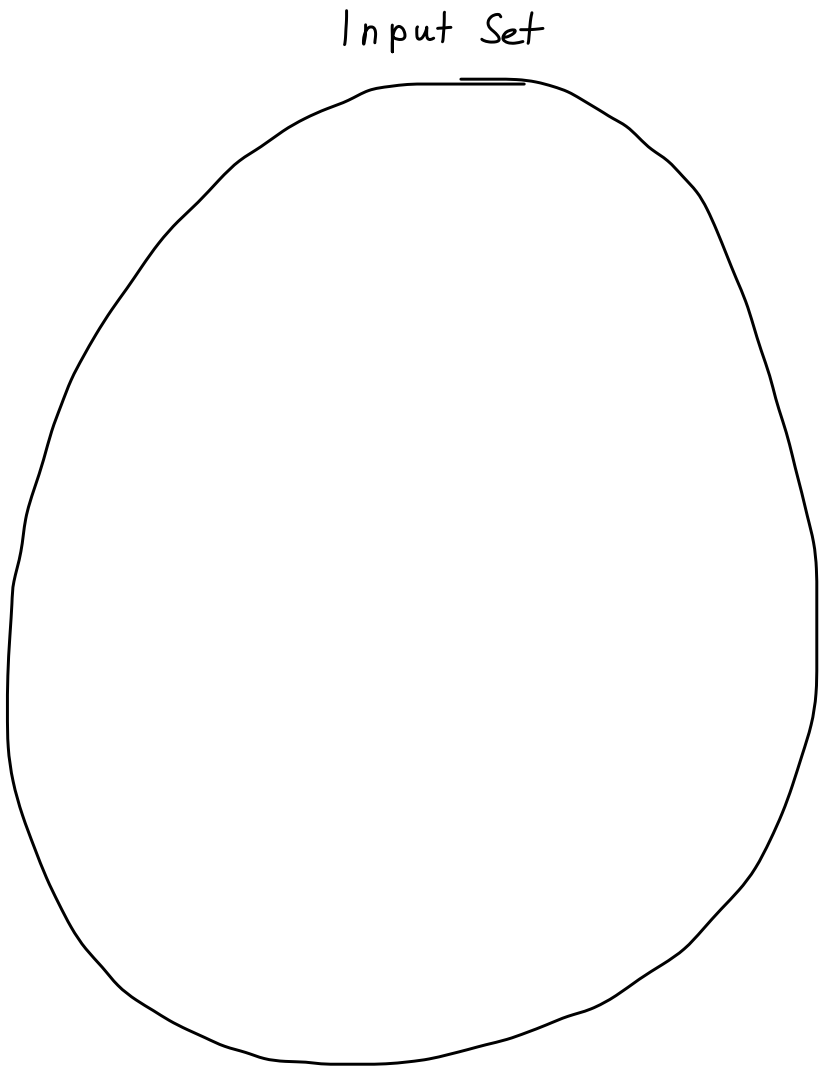
- Paper by Peter Sestoft (availalbe from the course home page)

Partition based tests



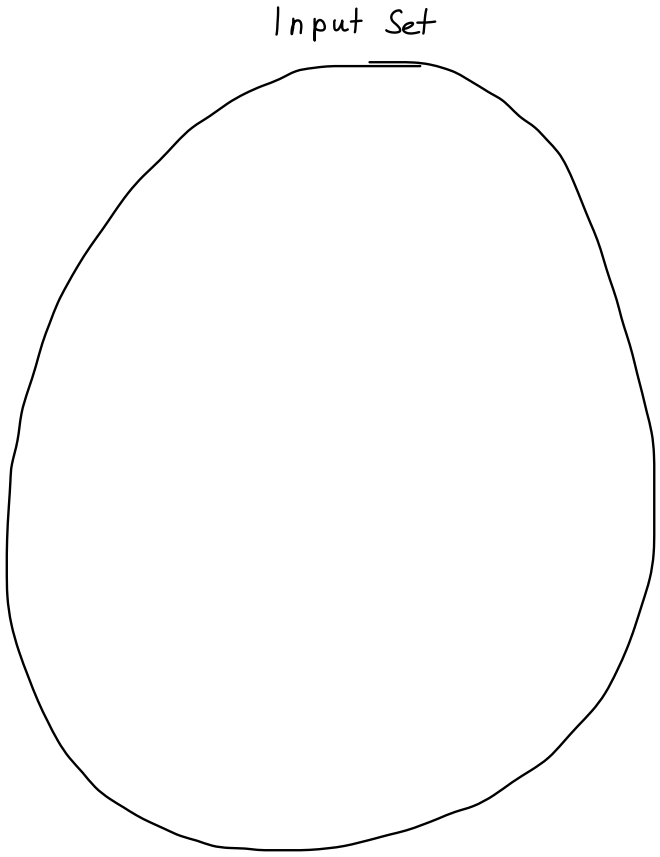
- Tests test expected behaviour
- SUT (System under test)

Partition based tests: Black box



- Expected behaviour: isEven(n)
- SUT implementation of isEven(n)

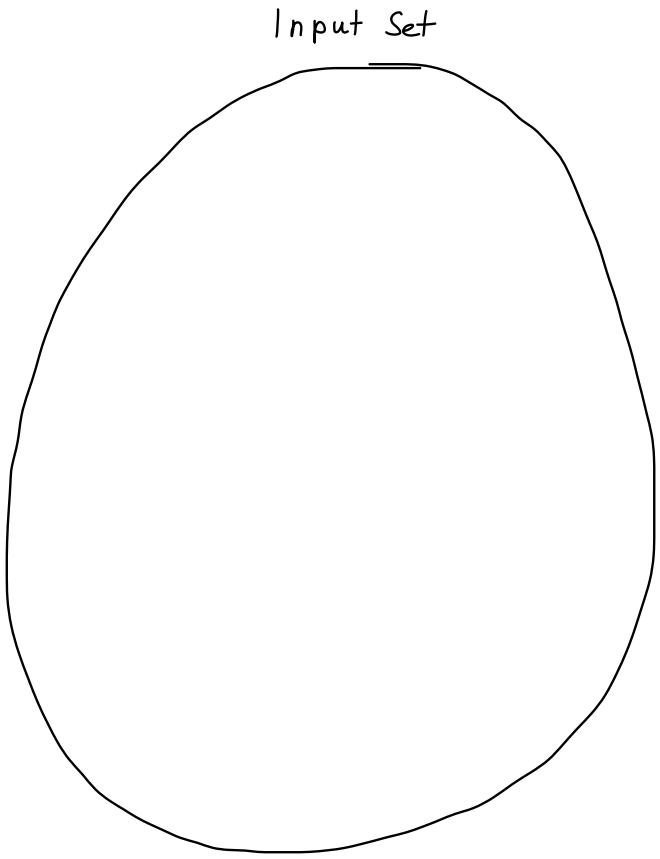
Partition based tests: White Box



- Expected behaviour: isEven(n)
- SUT implementation of isEven(n)

```
public boolean isEven(int n) {
    if (n % 2 == 0) {
        return true;
    } else {
        return false;
    }
}
```

Partition based tests: White Box



- Expected behaviour: isEven(n)
- SUT implementation of isEven(n)

```
public boolean isEven(int n) {
    if  (n == 101) return true;
    if (n % 2 == 0) {
        return true;
    } else {
        return false;
    }
}
```

How to find the right partitions?

1. white box test / structural test
2. black box test / functional test

White Box tests

- Find the minimum and the maximum of a list of integers

```
public class MinMax {
    int min, max;
    public void minmax(int[] args) throws Error {
        if (args.length == 0)                          // 1
            throw new Error("No numbers");
        else {
            min = max = args[0];
            for (int i = 1; i < args.length; i++) { // 2
                int obs = args[i];
                if (obs > max)                         // 3
                    max = obs;
                else if (min < obs)                    // 4
                    min = obs;
            }
        }
    }
    public int getMin() { return min; }
    public int getMax() { return max; }
}
```

- Path from method entry to exit $\rightarrow$ partition
- Input property $=$ conjunction of all conditions on path

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 ≤ current maximum
4 true	E number 3	Number ≤ current maximum and > current minimum
4 false	E number 2	Number ≤ current maximum and ≤ 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

JUnit Tests

```
public class WhiteBoxTest {
  MinMax sut = new MinMax();

  @Test(expected = Error.class)
  public void testInputDataSetA() {
    int[] ar = {};
    sut.minmax(ar);
  }

  @Test
  public void testInputDataSetB() {
    int[] ar = {17};
    sut.minmax(ar);
    assertEquals(17,sut.getMin());
    assertEquals(17,sut.getMax());
  }

  @Test
  public void testInputDataSetC() {
    int[] ar = {27, 29};
    sut.minmax(ar);
    assertEquals(27,sut.getMin());
    assertEquals(29,sut.getMax());
  }
```

JUnit Tests (cont.)

```
  @Test
  public void testInputDataSetD() {
    int[] ar = {39, 37};
    sut.minmax(ar);
    assertEquals(37,sut.getMin());
    assertEquals(39,sut.getMax());
  }

  @Test
  public void testInputDataSetE() {
    int[] ar = {49, 47, 48};
    sut.minmax(ar);
    assertEquals(47,sut.getMin());
    assertEquals(49,sut.getMax());
  }

}
```

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

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

White box vs. Black box testing

- White box test
  - finds defects in the implementation
  - can’t find problems with the functionality

White box vs. Black box testing

- White box test
  - finds defects in the implementation
  - can't find problems with the functionality
- Ex.: Functionality: For numbers *n* below 100 return `even(n)`, for numbers 100 or above return `odd(n)`.

```
public boolean isEvenBelow100andOddAbove(int n) {
  if (n % 2 == 0) {
    return true;
  } else {
    return false;
  }
}
```

White box vs. Black box testing

- Black box test
  - finds problems with the functionality
  - can’t find defects in the implementation

White box vs. Black box testing

- Black box test
  - finds problems with the functionality
  - can't find defects in the implementation
- Ex.: Functionality: For a number *n* return `even(n)`

```
public boolean isEven(int n) {
  if (n == 100) {
       return false;
  }
  if (n % 2 == 0) {
    return true;
  }
  return false;
}
```

TDD vs. White box and Black box testing

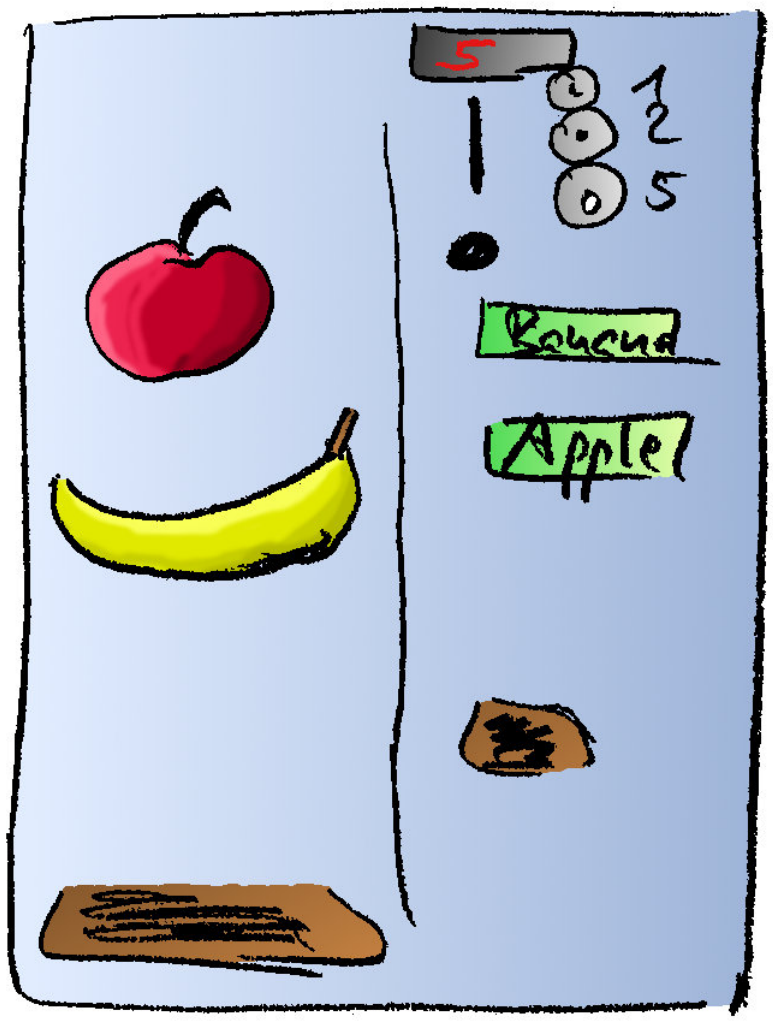
- TDD: Black box + white box testing
- TDD starts with tests for the functionality
- Any production code needs to have a failing test first

Summary

Test plan: Two tables

- Table for the input partitions
- Table for the test data (input / expected output)

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

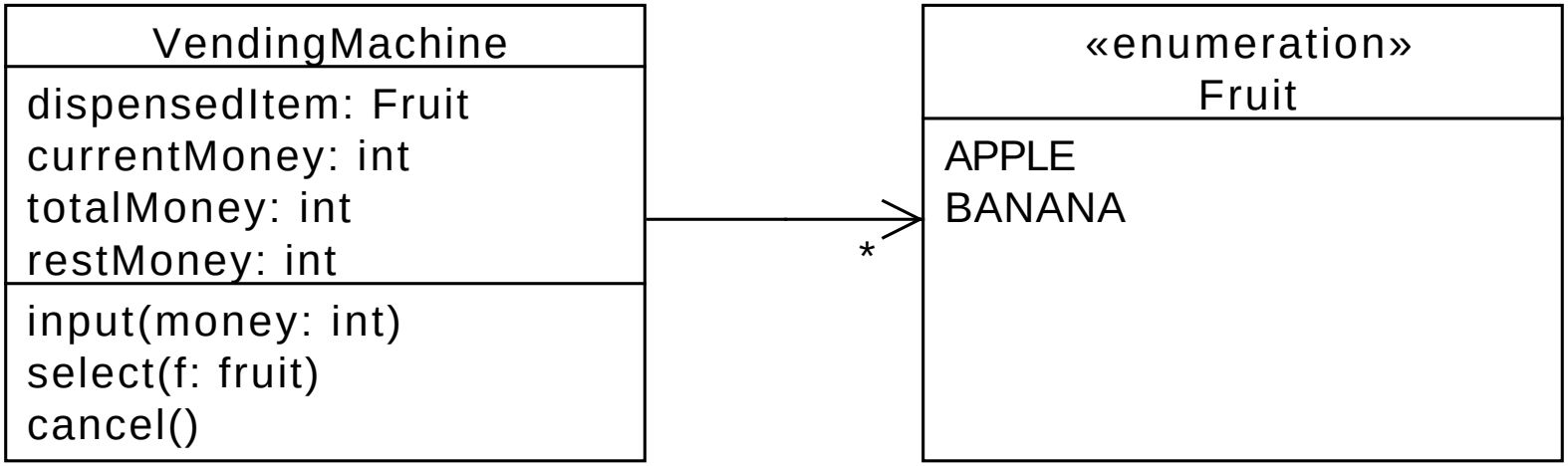
Functional Test for Buy Fruit Use Case: Test Cases

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

- Manual test-plans
  - Table of input / expected output
  - Run the application
  - Check for desired outcome
- Automatic tests
  a. Test the GUI directly
  b. Testing ”under the GUI”
     → Layered 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());
}

...
```

Contents

Code coverage

- How good are the tests?

→ When the tests have covered all the code

- Code coverage
  - statement coverage
  - decision coverage
  - condition coverage
  - path coverage
  - ...

Code coverage: statement, decision, condition

```
int foo (int x, int y)
{
    int z = 0;
    if ((x>0) && (y>0)) {
        z = x;
    }
    return z;
}
```

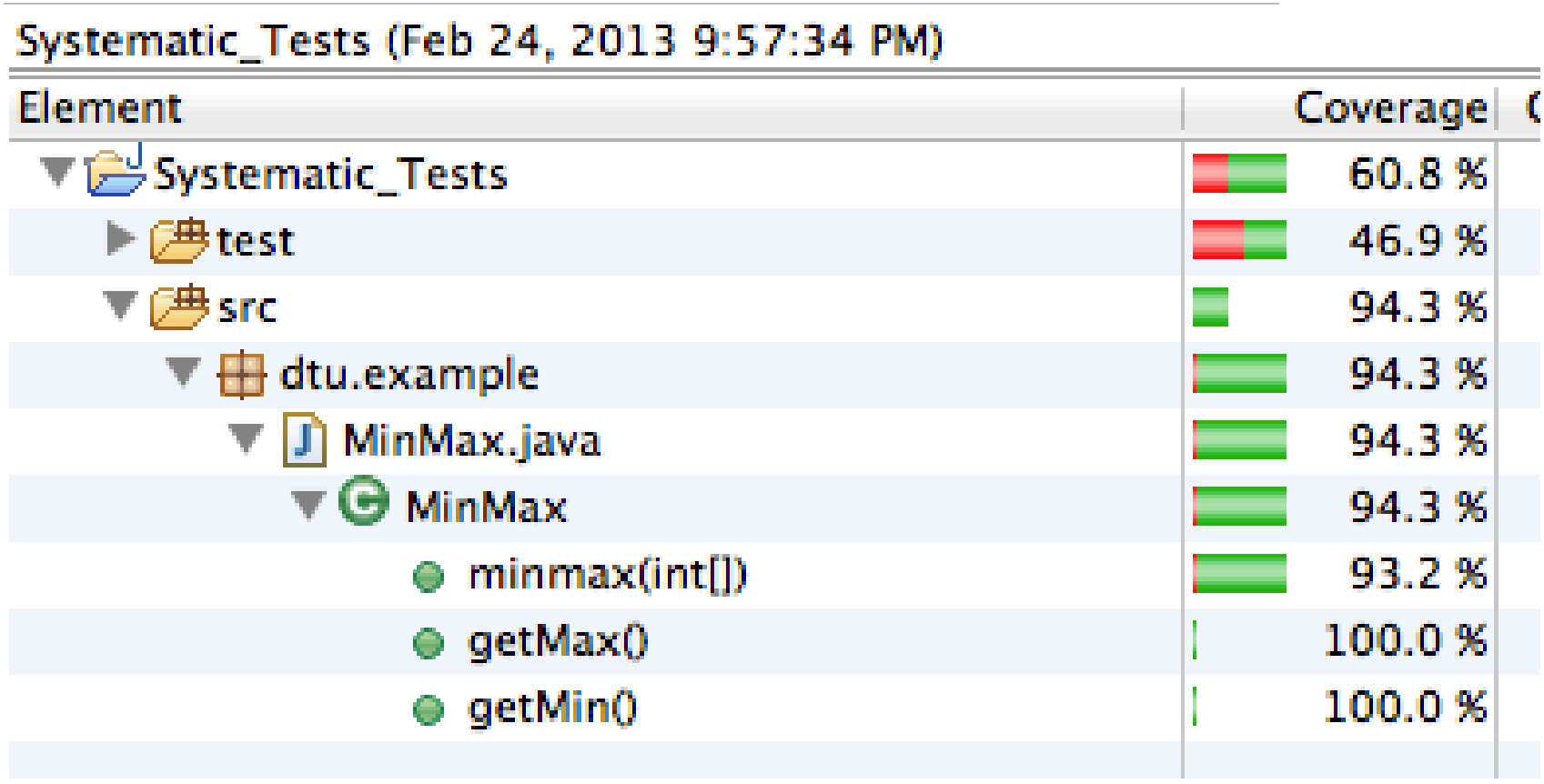
Code coverage: path

```
int foo (boolean b1, boolean b2)
{
   if (b1) {
      s1;
   } else {
      s2;
   }
   if (b2) {
      s3;
   } else {
      s4;
   }
}
```

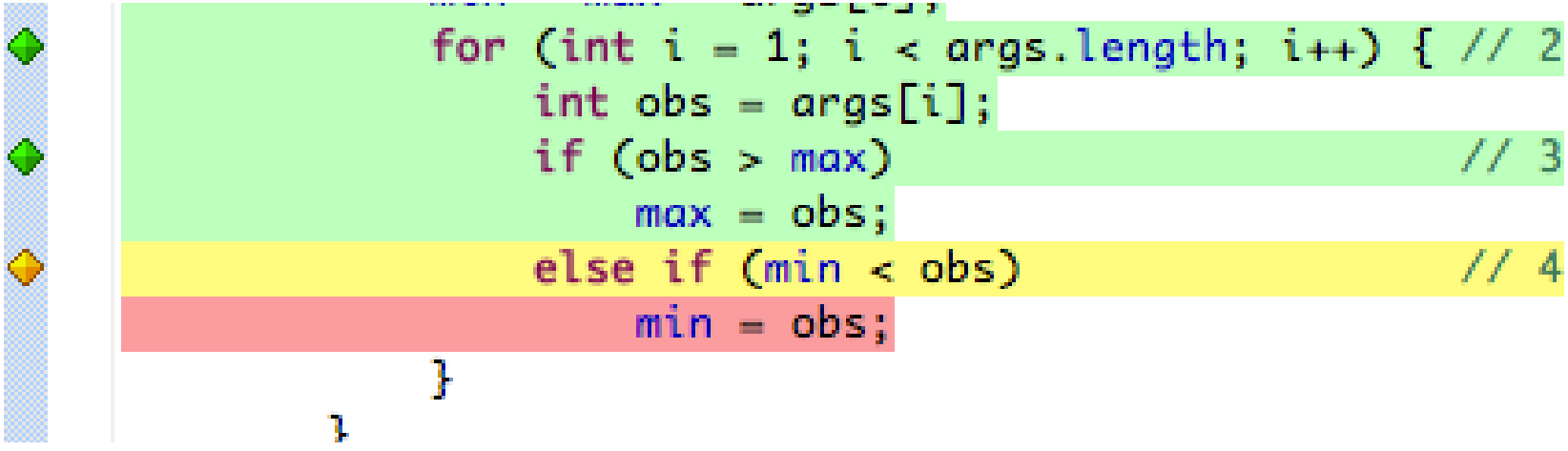
Coverage Tool

- Statement, decision, and condition coverage
- EclEmma (`http://eclemma.org`):

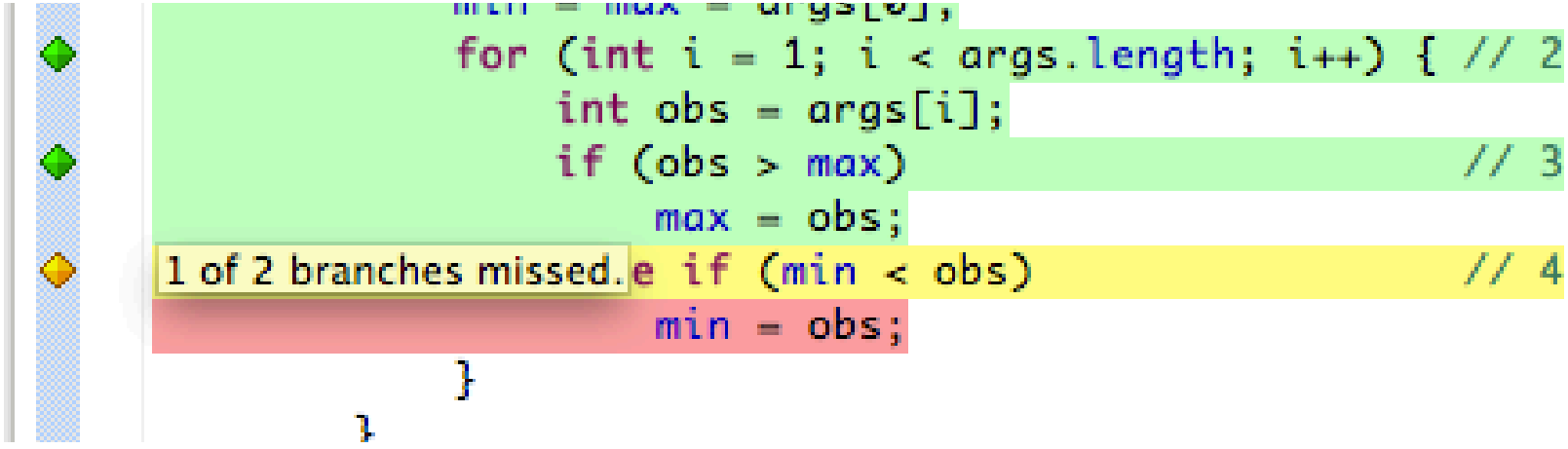
Coverage with EclEmma



Coverage with EclEmma



Coverage with EclEmma



Next Week

- From user requirements to design
- Documenting design
  - Class diagrams
  - Sequence diagrams
- Exam project introduction

Exam project

- Exam project
  - Week 05: Project introduction and forming of project groups; participation mandatory
  - Week 07: Submission of use cases and design
  - Week 08: Submission of peer review of use cases and design; start of implementation phase
  - Week 13: Demonstration of the projects (each project 15 min)
- Group forming
  - Group forming: **mandantory** participation in the lecture 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*