Software Engineering I (02161)

Week 3

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2018

Contents

Systematic tests

Code coverage

Refactoring

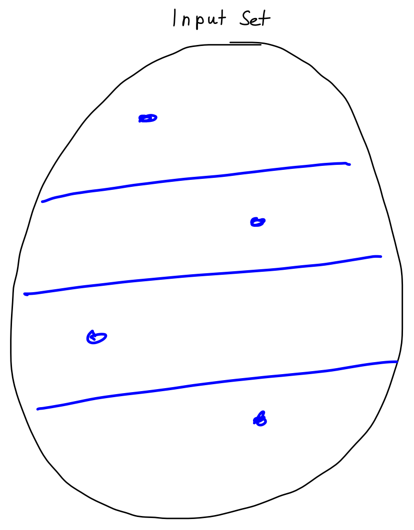
Systematic testing

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

$\rightarrow$ Partition based tests

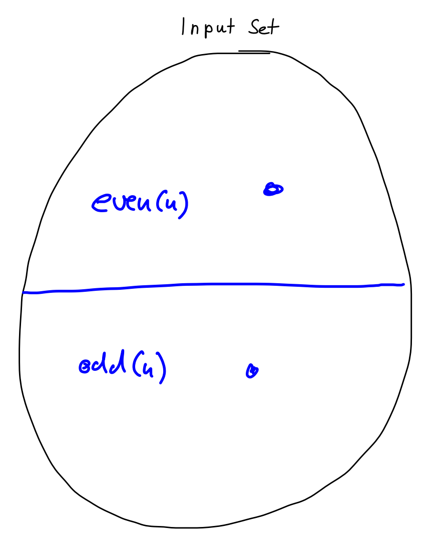
- Paper by Peter Sestoft (availalbe from the course home page)

Partition based tests



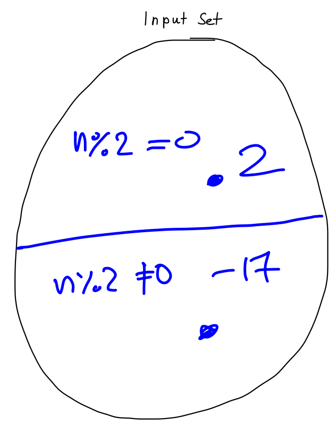
- Tests test expected behaviour
- SUT (System under test)
- Partition: All elements behave the same under a test
- Choice of partitions depends on tests

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

test cases 2, -17

Input Set

$n \% 2 = 0 \wedge n \neq 101$

$n = 101$

$n \% 2 \neq 0 \wedge n \neq 101$

- 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	Input data set	Input Property
1 true	A	No numbers
not 1, 2(0)	B	one number
not 1, 2(1), 3 true	C	two numbers and a(1) > max = a(0)
not 1, 2(1), not 3, not 4	D	two numbers and a(1) <= max = a(0)
not 1, 2(2), 3, not 3, 4 true	E	three numbers and a(1) > a(0) and a(2) <= a(1) and a(0) < a(2)

- Partition: Path from method entry to exit
- Input property = conjunction of all conditions on path

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

Path	Input data set	Input Property
1 true	A	No numbers
not 1, 2(0)	B	one number
not 1, 2(1), 3 true	C	two numbers and $a(1) > max = a(0)$
not 1, 2(1), not 3, not 4	D	two numbers and $a(1) <= max = a(0)$
not 1, 2(2), 3, not 3, 4 true	E	three numbers and $a(1) > a(0)$ and $a(2) <= a(1)$ and $a(0) < a(2)$

Input data set	Contents	Expected Output
A	no numbers	Exception
B	17	$min = 17$, $max = 17$
C	27 29	$min = 27$, $max = 29$
D	39 37	$min = 37$, $max = 39$
E	47 49 48	$min = 47$, $max = 49$

will fail

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());
}

@Test
public void testInputDataSetF() {
  int[] ar = {47, 49, 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

- What are the partitions?

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

What do you think and why?

1. If there are white box tests, no black box tests are needed 2
2. if there are black box tests, no white box tests are needed 4
3. Both, black box and white box tests are needed all the time >11

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;
  }
}
```

n%2=0

n%2=0

white box

even(n)

odd(n)

n<100

even(n)

odd(n)

n≥100

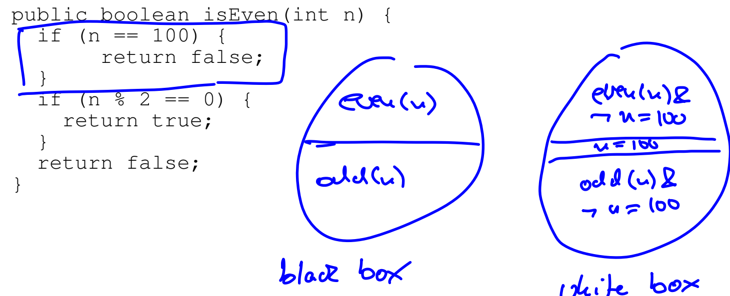
black box

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 (black box)
- Any production code that you want to write needs a failing test (white box)

→ Quality critera: 100% code coverage of tests (white box test critera). Beware that all required functionality is tested too.

Contents

Systematic tests

Code coverage

Refactoring

Code coverage

- How good are the tests?

$\rightarrow$ When the tests have covered all the code

- Code coverage
  - statement coverage
  - decision coverage
  - condition coverage
  - path coverage
  - …

Code coverage: statement, decision, condition

statement coverage
x > & y > 0

decision coverage
x > 0 & y > 0
~~x > 0 & y ≤ 0~~
¬(x > 0 & y > 0)

condition coverage
X > 0 & y > 0
x ≤ 0 & y > 0
x > 0 & y ≤ 0
x ≤ 0 & y ≤ 0

```
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;
  }
}
```

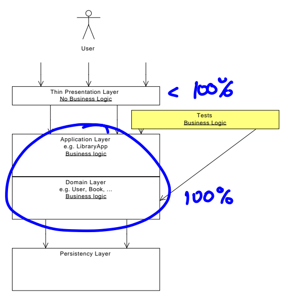
b1, b2 / ¬b1, ¬b2 } condition coverage — s_1, s_3 / s_2, s_4

¬b1, b2 — s_2, s_3

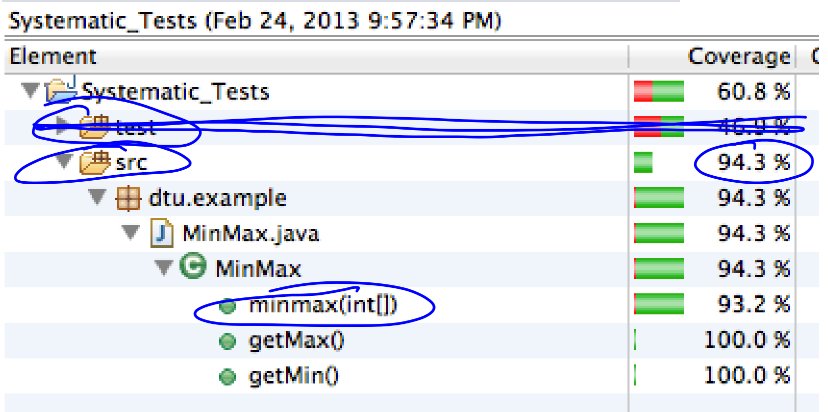
b1, ¬b2 — s_1, s_4

Coverage Tool

- Statement, decision, and condition coverage
- EclEmma (`http://eclemma.org`)
- Goal: 100% test coverage
  - → Sufficient white box tests
- TDD helps to achieve the goal



Coverage with EclEmma



Coverage with EclEmma

```
    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;
    }
}
```

Coverage with EclEmma

```
            min = max = args[0];
            for (int i = 1; i < args.length; i++) { // 2
                int obs = args[i];
                if (obs > max)                      // 3
                    max = obs;
1 of 2 branches missed.e if (min < obs)             // 4
                    min = obs;
            }
        }
```

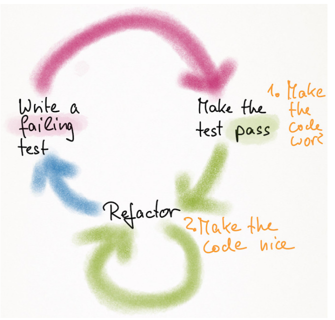
Contents

Systematic tests

Code coverage

Refactoring
- Refactoring
- Refactoring Example

Refactoring and TDD



- *restructure* the system **without** *changing* its functionality
- Goal: *improve* the design
- Requires tests

Refactoring

- PhD thesis by William Opdyke (1992)
- *Refactoring: Improving the Design of Existing Code*, Martin Fowler, 1999
- Set of refactorings
  - e.g. renameMethod, extractMethod, encapsulateField, encapsulateCollection, . . .
- Set of code smells
  - e.g. Duplicate Code, Long Method, Large Class, . . .

Example of Code smells

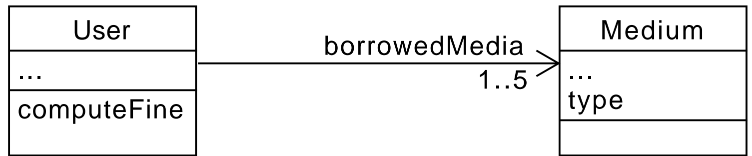
If it stinks, change it Refactoring, Martin Fowler, 1999

- Duplicate Code
- Long Method
- Large Class (God class)
- Switch Statements
- Comments
- ...

http://en.wikipedia.org/wiki/Code_smell

Code Smell: Switch Statement

```
public class User {
   public double computeFine() {
      double fine = 0;
      for (Media m : borrowedMedia) {
         if (m.overdue) {
            switch (m.getType()) {
               case Media.BOOK : fine = fine + 10; break;
               case Media.DVD: fine = fine + 30; break;
               case Media.CD: fine = fine + 20; break;
               default fine = fine + 5; break;
            }
         }
      }
      return fine;
   }
}
```



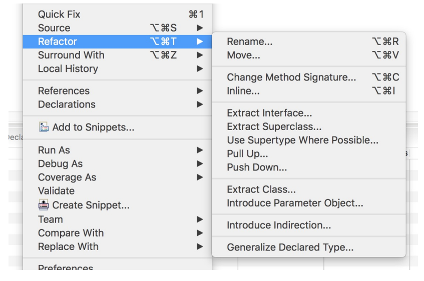
Better Design

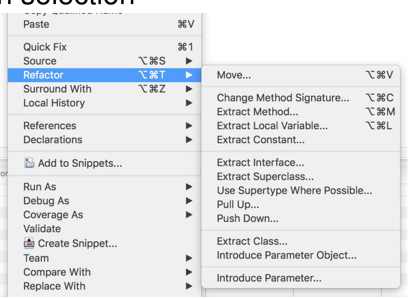
```
public class User {
   public double computeFine() {
      double fine = 0;
      for (Media m : borrowedMedia) {
         if (m.overdue) { fine = fine + m.getFine();}
      }
      return fine;
   }
}

abstract public class Media {
   abstract public double getFine();
}

public class Book extends Media {
   public double getFine() { return 10; }
}

public class DVD extends Media {
   public double getFine() { return 30; }
}

public class CD extends Media {
   public double getFine() { return 20; }
}
```

User
...
computeFine
borrowedMedia
1..5
Medium
...
getFine
Book
...
getFine
CD
...
getFine
DVD
...
getFine

Eclipse Refactoring Support

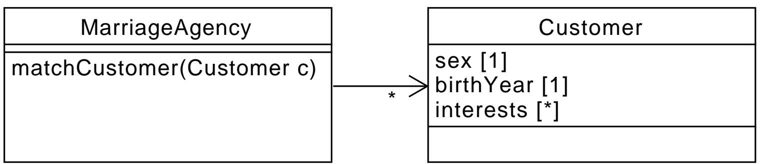
- Menu in the code editor
- Different menus depending on selection





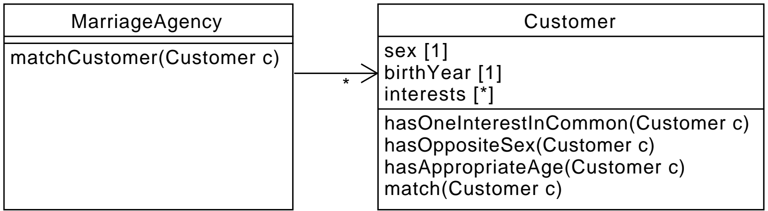
- Important
  - Very small steps even for complex refactorings: Strategy
  - Each step from green to green

MarriageAgency class diagram



- Refactoring example in detail
  → `http://www2.imm.dtu.dk/courses/02161/2018/slides/refactoring_example.pdf`
- Framework for running tests as soon as the code changes:
  → Infinitest `http://infinitest.github.io/`

Remark on refactoring



- Always green to green
- Decompose a large refactoring into small refactorings
  - → No failing tests for too long (e.g. hours, days, weeks, ...)
  - Ideally, refactorings can be interrupted