Software Engineering

QA + Testing II

Software & Systems Engineering | Prof. Dr. Andreas Vogelsang | 06.12.2023

@andivogelsang
vogelsang@cs.uni-koeln.de

Learning Goals for Today

- Know what White-box testing is and how it is applied
- Know what code coverage metrics are and what they say about test quality
- Know what Black-box testing is and how it is applied
- Know the most common methods for Black-box testing
- Know how to apply JUnit as a framework for testing in Java

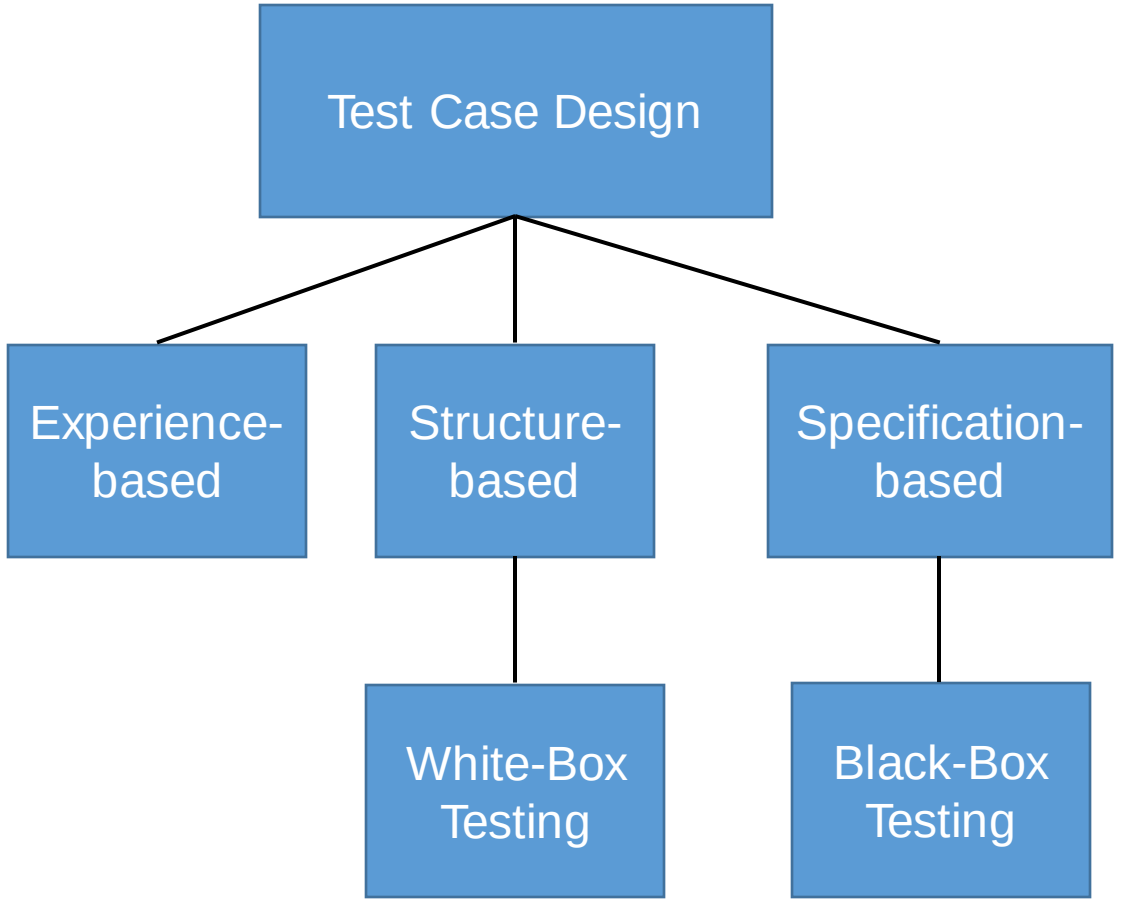
Recall: Test Case Design

Goal [Ludewig and Lichter]

Detect a large number of failures with a low number of test cases.

An ideal test case is… [Ludewig and Lichter]

- **representative**: represents a large number of feasible test cases
- **failure sensitive**: has a high probability to detect a failure
- **non-redundant**: does not check what other test cases already check



White-Box Testing

White-Box Testing

White-Box Testing [Ludewig and Lichter]

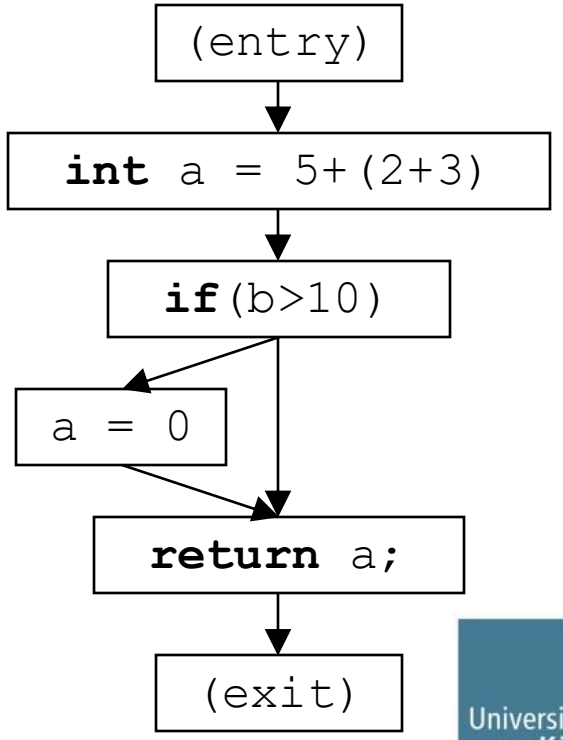
- inner structure of test object is used
- idea: coverage of structural elements within the test object
- code translated into control flow graph
- specific test case (concrete inputs) derived from logical test case (conditions) derived from path in control flow graph

Control Flow Graph

Control Flow Graph

- Graph-based representation of the control flow within a program
  - Contains **all possible paths**
- Every node represents a **basic block** of the program. A basic block is composed of a set of statements that is executed together no matter what happens
- Edges represent **possible ways** the control flow can take
- **Intra-procedural** (within a function) vs. **inter-procedural** (across several functions)

```
int a = 5 + (2 + 3)
if (b > 10) {
   a = 0;
 }
return a;
```



Coverage Criteria (Überdeckungskriterien)

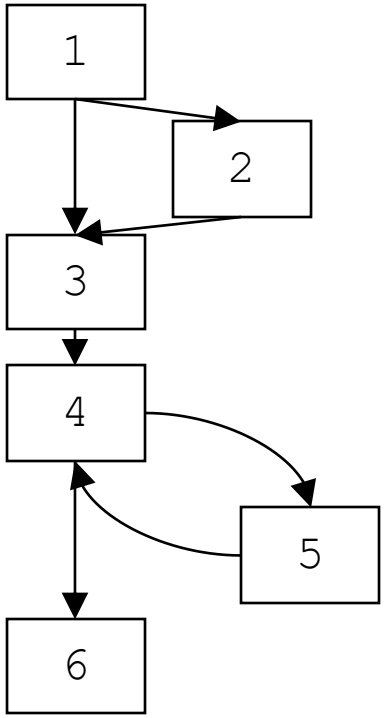
CC BY SA

Coverage Criteria

- **Function coverage** (Funktionsüberdeckung): each function of a program is executed at least once during testing
- **Statement coverage** (Anweisungsüberdeckung): each statement (LOC) of a program is executed at least once during testing
- **Branch coverage** (Zweigüberdeckung): statement coverage plus for each branching statement all branches have been exercised
- **Condition coverage** (Bedingungsüberdeckung): Every condition in a decision in the program has taken all possible outcomes at least once.
- **Multiple condition coverage** (Mehrfachbedingungsüberdeckung): all combinations of conditions inside each decision are tested
- **Path coverage** (Pfadüberdeckung): each possible path through a program is executed at least once during testing.

Statement Coverage – Example

```
public static int modulo(int a, int b) {
  if (b < 0) {                  // 1 ✓
    b *= -1;                    // 2 ✓
  }
  int m = a;                    // 3 ✓
  while (m < 0 || m > b) {      // 4 ✓
    m += (m < 0 ? b : -b);      // 5 ✓
  }
  return m;                     // 6 ✓
}
```



Test Case:

1. `modulo(5, -3) == 2`

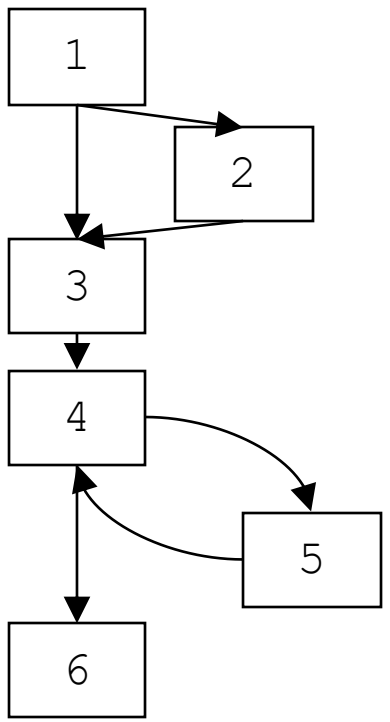
$$Statement\ Coverage = \frac{lines\ covered}{lines\ total}$$

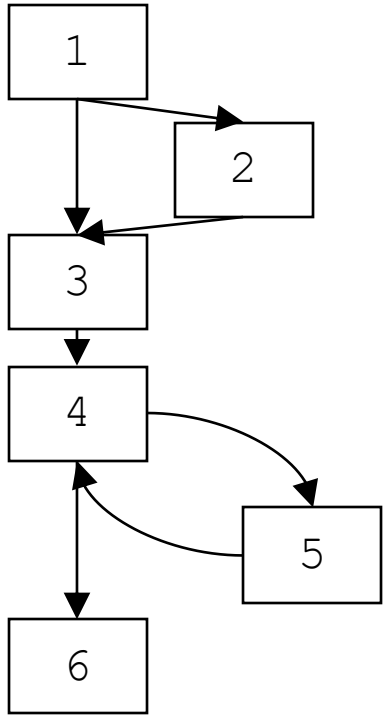
Statement Coverage:

6/6 = 100%

Branch Coverage – Example

```
public static int modulo(int a, int b) {
  if (b < 0) {                  // 1
    b *= -1;                    // 2
  }
  int m = a;                    // 3
  while (m < 0 || m > b) {      // 4
    m += (m < 0 ? b : -b);      // 5
  }
  return m;                     // 6
}
```





Test Cases:

1. `modulo(5, -3) == 2`
2. `modulo(0, 5) == 0`

$$Branch\ Coverage = \frac{decision\ outcomes\ covered}{decision\ outcomes\ total}$$

Branch Coverage:

Only test case 1: 3/4 = 75%

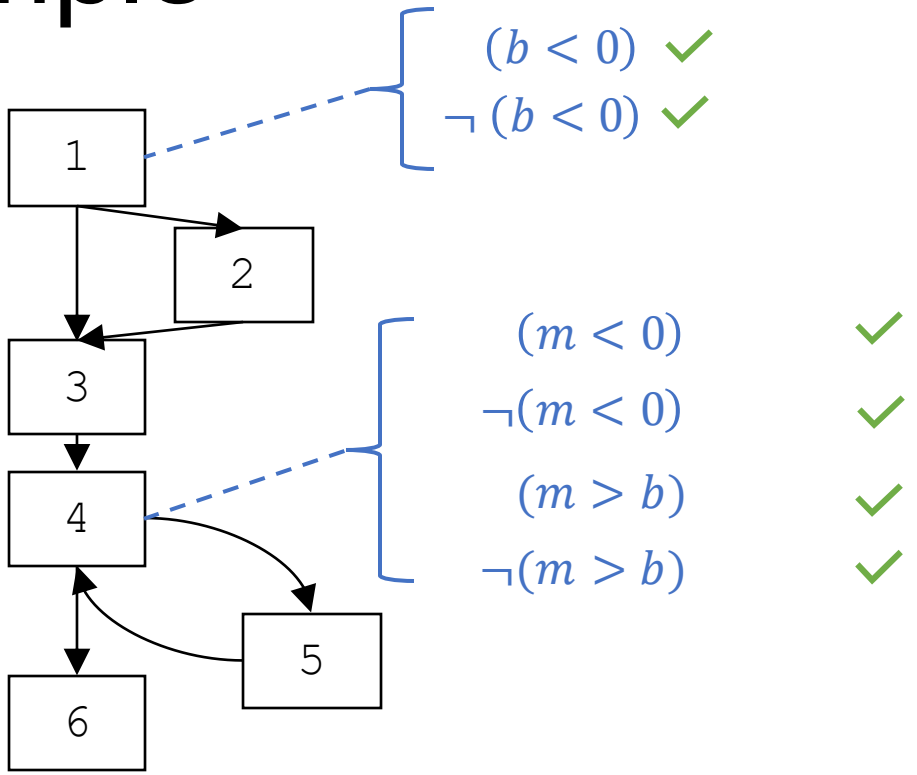
Both test cases: 4/4 = 100%

Ternary Operator in Stmt 5

- could also be treated as branching statement
- how to adapt the control flow graph then?
- are the two test cases still sufficient?

Condition Coverage – Example

```
public static int modulo(int a, int b) {
  if (b < 0) {                    // 1
    b *= -1;                      // 2
  }
  int m = a;                      // 3
  while (m < 0 || m > b) {        // 4
    m += (m < 0 ? b : -b);        // 5
  }
  return m;                       // 6
}
```



Test Cases:

1. `modulo(5, -3) == 2`
2. `modulo (0, 5) == 0`
3. `modulo(-2, 5) == 3`

$$Condition\ Coverage = \frac{condition\ outcomes\ covered}{condition\ outcomes\ total}$$

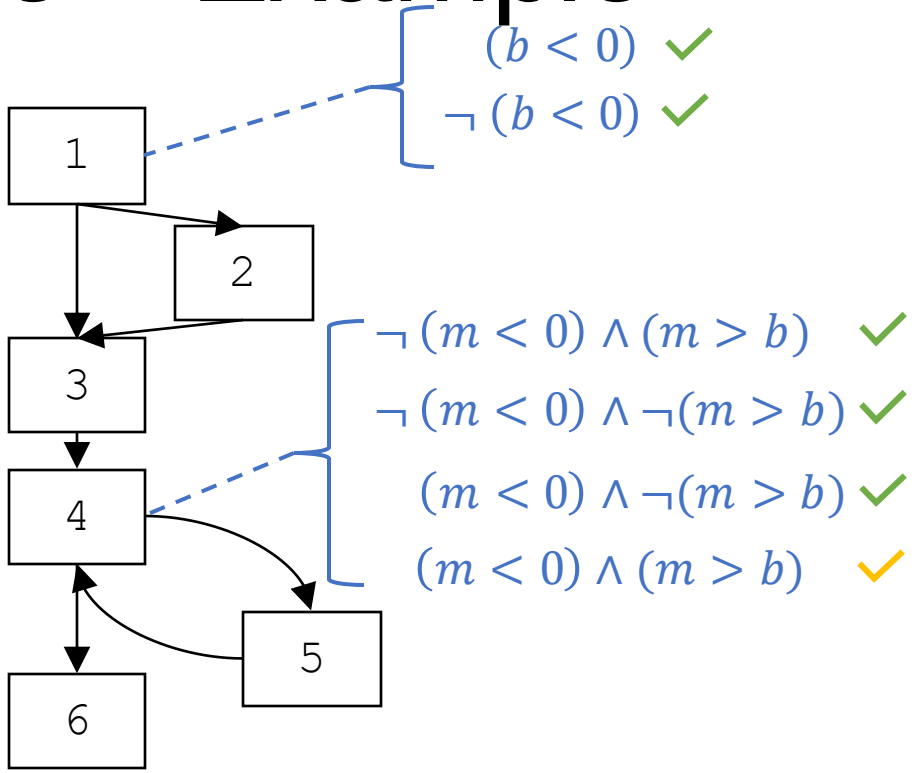
Condition Coverage:

Only test cases 1 and 2: 5/6 = 83%

Test cases 1,2, and 3: 5/6 = 100%

Multiple Condition Coverage – Example

```
public static int modulo(int a, int b) {
  if (b < 0) {                    // 1
    b *= -1;                      // 2
  }
  int m = a;                      // 3
  while (m < 0 || m > b) {        // 4
    m += (m < 0 ? b : -b);        // 5
  }
  return m;                       // 6
}
```



Test Cases:

1. `modulo(5, -3) == 2`
2. `modulo (0, 5) == 0`
3. `modulo(-2, 5) == 3`

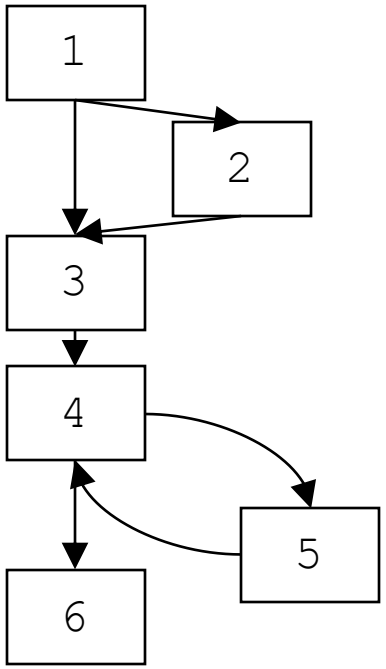
$$\textit{Multiple Condition Coverage} = \frac{\textit{condition outcomes covered}}{\textit{all possible condition outcomes}}$$

Multiple Condition Coverage:

Only test cases 1 and 2: 4/6 = 66%

Test cases 1,2, and 3: 5/6 = 83%

Path Coverage – Example



Paths of Test Cases:

1. `modulo( 5,-3) == 2:` $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 4 \rightarrow 6$
2. `modulo( 0, 5) == 0:` $1 \rightarrow 3 \rightarrow 4 \rightarrow 6$
3. `modulo(-2, 5) == 3:` $1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 4 \rightarrow 6$
4. `modulo(10, 4) == 1:` $1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 4 \rightarrow 5 \rightarrow 4 \rightarrow 6$
5. …

$$Path\ Coverage = \frac{paths\ covered}{paths\ total}$$

Path Coverage Limitations

Path Coverage for programs with loops often impossible to reach (and even calculate)

Coverage Criteria (Überdeckungskriterien)

Coverage Criteria

- **Function coverage** (Funktionsüberdeckung): each function of a program is executed at least once during testing
- **Statement coverage** (Anweisungsüberdeckung): each statement (LOC) of a program is executed at least once during testing
- **Branch coverage** (Zweigüberdeckung): statement coverage plus for each branching statement all branches have been exercised
- **Condition coverage** (Bedingungsüberdeckung): Every condition in a decision in the program has taken all possible outcomes at least once.
- **Multiple condition coverage** (Mehrfachbedingungsüberdeckung): all combinations of conditions inside each decision are tested
- **Path coverage** (Pfadüberdeckung): each possible path through a program is executed at least once during testing.

Coverage Criteria Properties

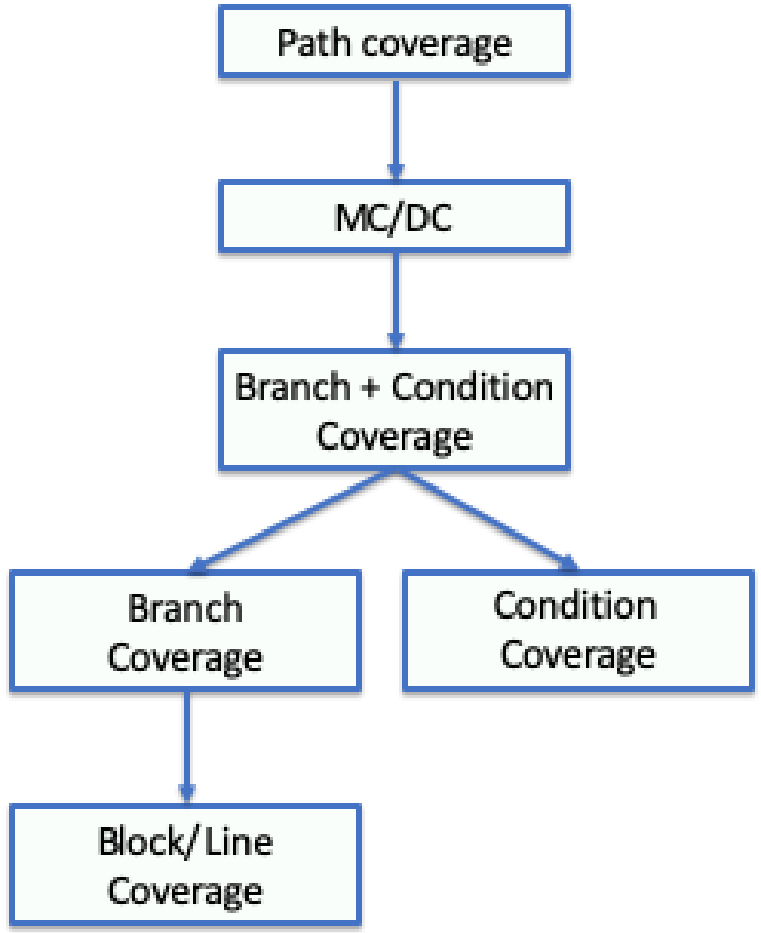
- 100% statement coverage not feasible in presence of dead code or some unreachable error handling
- 100% multiple condition coverage not feasible for certain dependencies between choices: `even() || odd()`
- 100% path coverage not feasible for programs with unbounded loops
- Branch coverage implies statement coverage (but not vice versa)
- Branch coverage and condition coverage do not imply each other

Universität zu Köln

Further Variants of Coverage Criteria

Coverage Criteria

- **Loop boundary adequacy criterion:** For every loop, one test case exercises the loop zero times, one test case exercises the loop once, one test case exercises the loop multiple times.
- **MC/DC (Modified Condition/Decision Coverage)**: Exercise each condition in a way that it can, independently of the other conditions, affect the outcome of the entire decision. In short, this means that every possible condition of each parameter must have influenced the outcome at least once



Black-box Testing

Black-Box Testing

Motivation

- source code not always available (e.g., outsourced components, obfuscated code)
- specific test cases derived from logical ones using arbitrary values
- specification not incorporated so far (only for expected results)
- invalid inputs not tested
- errors are not equally distributed

Black-Box Testing (Funktionstest)

- test-case design based on specification
- source code and its inner structure is ignored (assumed as a black-box)

Equivalence Class Testing

- **idea**: classify inputs and outputs into equivalence classes
- **assumption**: equivalent test cases detect the same faults, one test case is sufficient

Boundary Testing

- extension of equivalence class testing
- **goal**: use experience (e.g., off-by-one errors)
- for every equivalence class: consider smallest, typical, and largest value

In Practice

- Often combinations of white-box and black-box testing

Equivalence Class Testing

```
/**
* Computes the remainder of the
* Euclidean division of a by b. In
* contrast to the Java version a % b,
* the output will always be positive.
* Throws ArithmeticException when b is
* equal to 0.
*
* @param a dividend
* @param b divisor != 0
* @return remainder r with 0 <= r < b
*/
public static int modulo(int a, int b) {
```

Equivalence Classes

- Input a: $a < 0,\ a \geq 0$
- Input b: $b < 0,\ b = 0, b > 0$
- Output: $m = 0,\ m > 0, exception$

Test Cases	TC1	TC2	TC3
$a < 0$	X		
$a \geq 0$		X	X
$b < 0$	X		
$b > 0$		X	
$b = 0$			X
$m = 0$	X		
$m > 0$		X	
$exception$			X
input a	-3	1	2
input b	-3	2	0
expected output	0	1	exception
result	0 ✓	1 ✓	timeout ✗

Boundary Testing

Test Cases	TC1	TC2	TC3	TC4	TC5	TC6	TC7
$a < 0$	X			min	max		
$a \geq 0$		X	X			min	max
$b < 0$	X			max		min	
$b > 0$		X			max		min
$b = 0$			X				
$m = 0$	X			X		X	X
$m > 0$		min			max		
$exception$			X				
input a	-3	1	2	minInt	-1	0	maxInt
input b	-3	2	0	-1	maxInt	minInt	1
expected output	0	1	exception	0	maxInt-1	0	0
result	0	1	timeout	0	maxInt-1	timeout	1
	✓	✓	✗	✓	✓	✗	✗

Detected Faults

```
public static int modulo(int a, int b) {
  if (b < 0) {                  // 1
    b *= -1;                    // 2
  }
  int m = a;                    // 3
  while (m < 0 || m > b) {      // 4
    m += (m < 0 ? b : -b);      // 5
  }
  return m;                     // 6
}
```

TC3: infinite loop for b = 0, missing exception compared to JavaDoc

TC6: b remains negative as $-Integer.MIN\_VALUE == Integer.MIN\_VALUE$ and the loop condition is fulfilled for any integer

TC7: indicates that $m > b$ in the loop condition should be fixed to $m \geq b$

```
/**
* Computes the remainder of the
* Euclidean division of a by b. In
* contrast to the Java version a % b,
* the output will always be positive.
* Throws ArithmeticException when b is
* equal to 0.
*
* @param a dividend
* @param b divisor != 0
* @return remainder r with 0 <= r < b
*/
public static int modulo(int a, int b) {
  int m = a % b;
  return m < 0 ? m + b : m;
}
```

Passes all tests ✓

Reasons for Positive Test Cases

Reasons for Positive Test Cases

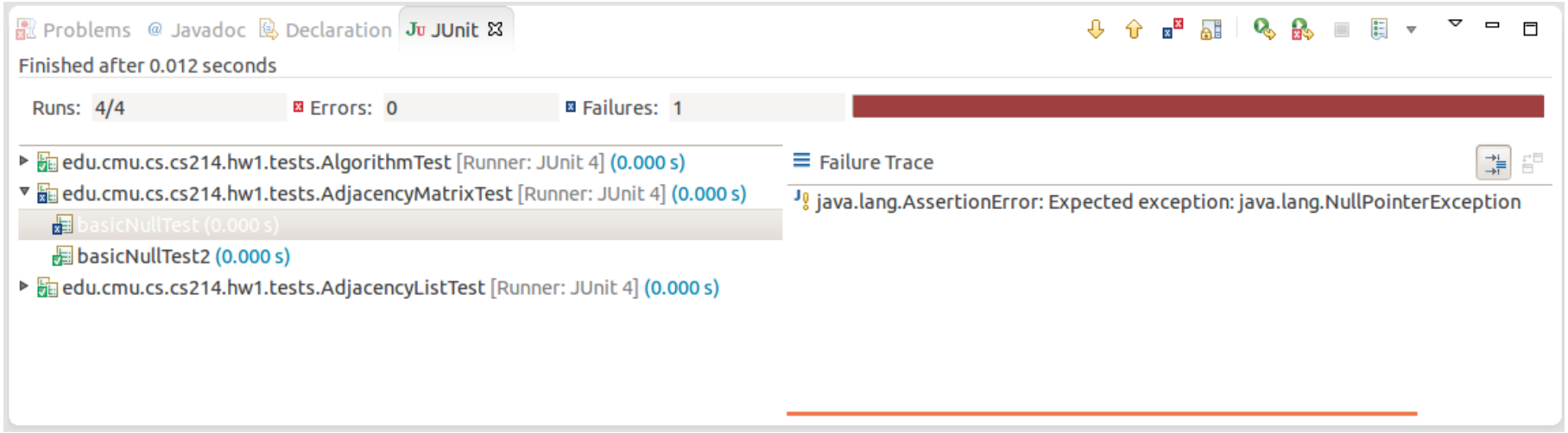
- actual fault
- wrong test case (input and expected results do not match)
- interaction with other programs/libraries
- fault in the compiler
- fault in the operating system / device drivers
- fault in the hardware or hardware defect
- not enough memory
- does not halt (cf. undecidability of the halting problem)
- bitflip due to cosmic ray
- ...

CC BY SA

Testing with JUnit

JUnit

- Popular Testing Framework for Java
- Easy to use (Tests are Java methods)
- Tool support available



Universität zu Köln

Tests in JUnit

- Tests are usually in a separate package (e.g. src/test/java)
- Tests are methods of a special test class

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class CalculatorTests {

    @Test
    public void TC1() {
        assertEquals(0,Calc.modulo(-3, -3), “should be 0”);
    }

    @Test
    public void TC7() {
        assertEquals(0, Calc.modulo(Integer.MAX_VALUE, 1));
    }
}
```

Annotations mark a test

The method *modulo* of class *Calc* is tested

Assert statements define the expected result (see also *assertTrue*, *assertNull*, *assertSame*,…)

Integration Tests

- Difference to unit tests?
- JUnit for integration tests?

Unit vs. Integration Test

```
public Double calculateArea(Type type, Double... r )
{
    switch (type)
    {
        case RECTANGLE:
            if(r.length >=2)
            return rectangleService.area(r[0],r[1]);
            else
                throw new RuntimeException("Missing required params");
        case SQUARE:
            if(r.length >=1)
                return squareService.area(r[0]);
            else
                throw new RuntimeException("Missing required param");

        case CIRCLE:
            if(r.length >=1)
                return circleService.area(r[0]);
            else
                throw new RuntimeException("Missing required param");
        default:
            throw new RuntimeException("Operation not supported");
    }
}
```

Unit test for this method:

- What shall be tested?
- What shall not be tested?

Universität zu Köln

Unit vs. Integration Test

In unit tests “external components“ are replaced by “mocks“

```
public Double calculateArea(Type type, Double... r )
{
    switch (type)
    {
        case RECTANGLE:
            if(r.length >=2)
            return rectangleService.area(r[0],r[1]);
            else
                throw new RuntimeException("Missing
        case SQUARE:
            if(r.length >=1)
                return squareService.area(r[0]);
            else
                throw new RuntimeException("Missing

        case CIRCLE:
            if(r.length >=1)
                return circleService.area(r[0]);
            else
                throw new RuntimeException("Missing
        default:
            throw new RuntimeException("Operation n
    }
}
```

```
    @Before
    public void init()
    {
        rectangleService = Mockito.mock(RectangleService.class);
        squareService = Mockito.mock(SquareService.class);
        circleService = Mockito.mock(CircleService.class);
        calculateArea = new CalculateArea(squareService,rectangleService,circleService);
    }

    @Test
    public void calculateRectangleAreaTest()
    {

        Mockito.when(rectangleService.area(5.0d,4.0d)).thenReturn(20d);
        Double calculatedArea = this.calculateArea.calculateArea(Type.RECTANGLE, 5.0d,
4.0d);
        Assert.assertEquals(new Double(20d),calculatedArea);

    }
```

Unit vs. Integration Test

- Mocks are replaced in integration tests by actual implementations.
- Focus on inter-component functionality
  - Database – Controller – UI

Universität zu Köln

Summary

White-box and Black-box testing

- In white box testing, the structure of a program is used to derive good test cases
- In black-box testing, the specification is leveraged to derive good test cases
- In practice, you usually start with black-box testing (based on the requirements) and then you augment your test suite with test cases that increase code coverage (white-box testing)
- JUnit is a great framework to integrate testing practices directly in Java programs