Software Engineering 2 – Advanced Testing with Java 1

Hochschule für Technik Stuttgart

Marcus Deininger

SS 2021

Topics

Part 1 [this part]

- Introduction
  - Definitions
  - Test Types: Black Box-/Glass Box-Testing
  - Test Workflow / Test Management
- Tools
  - Overview
  - Junit: Quick Introduction / Retake
  - eclEmma: Measuring Coverage
  - Java-Reflection: Access the Inaccessible

Topics

Part 2 [next week]

- Mockito
  - Test-Doubles
  - Using Mockito
- AspectJ
  - Aspect Oriented Programming
  - AspectJ for Testing
- Selenium
  - Testing Web-Applications

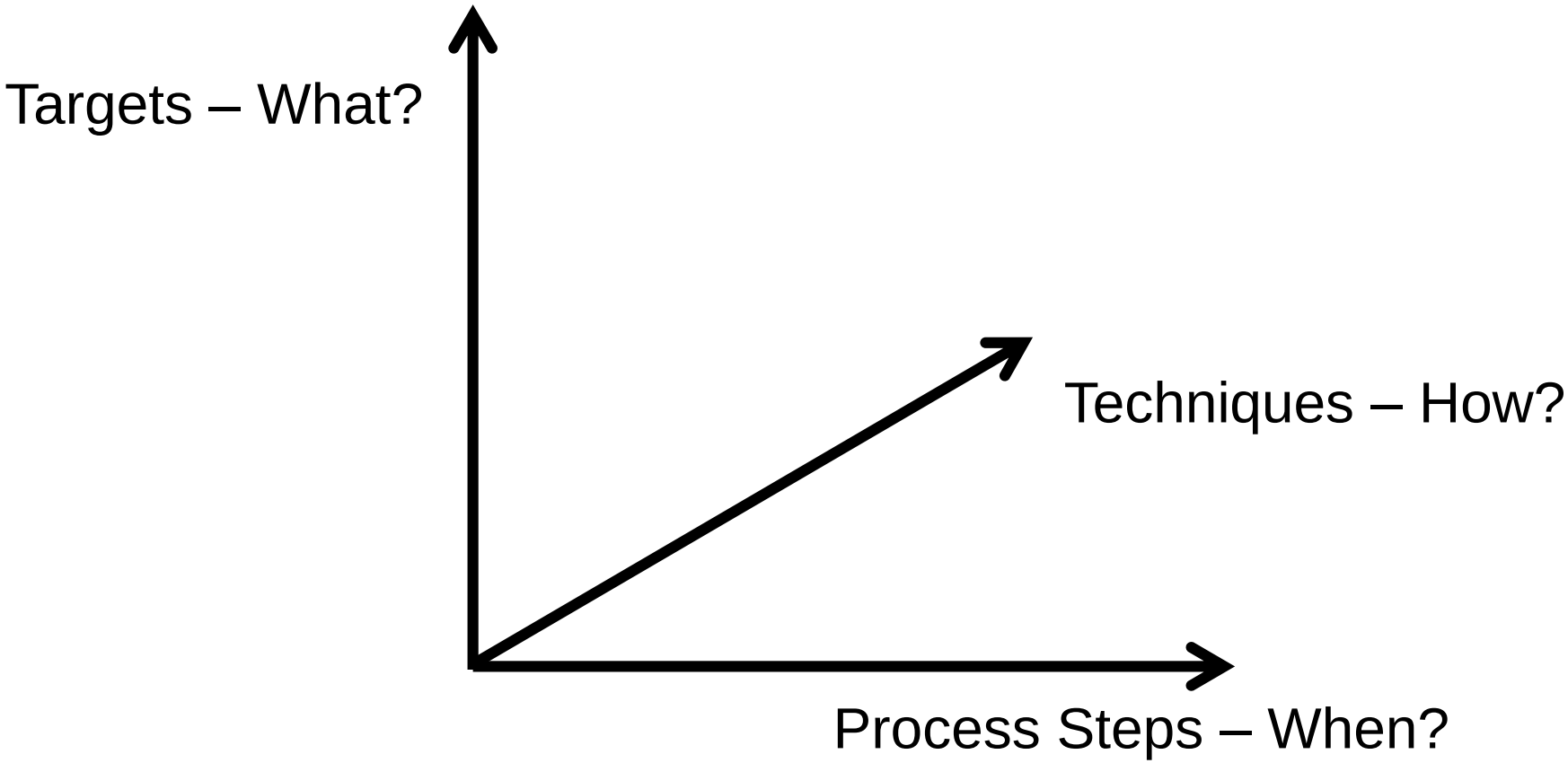
Definitions

Test

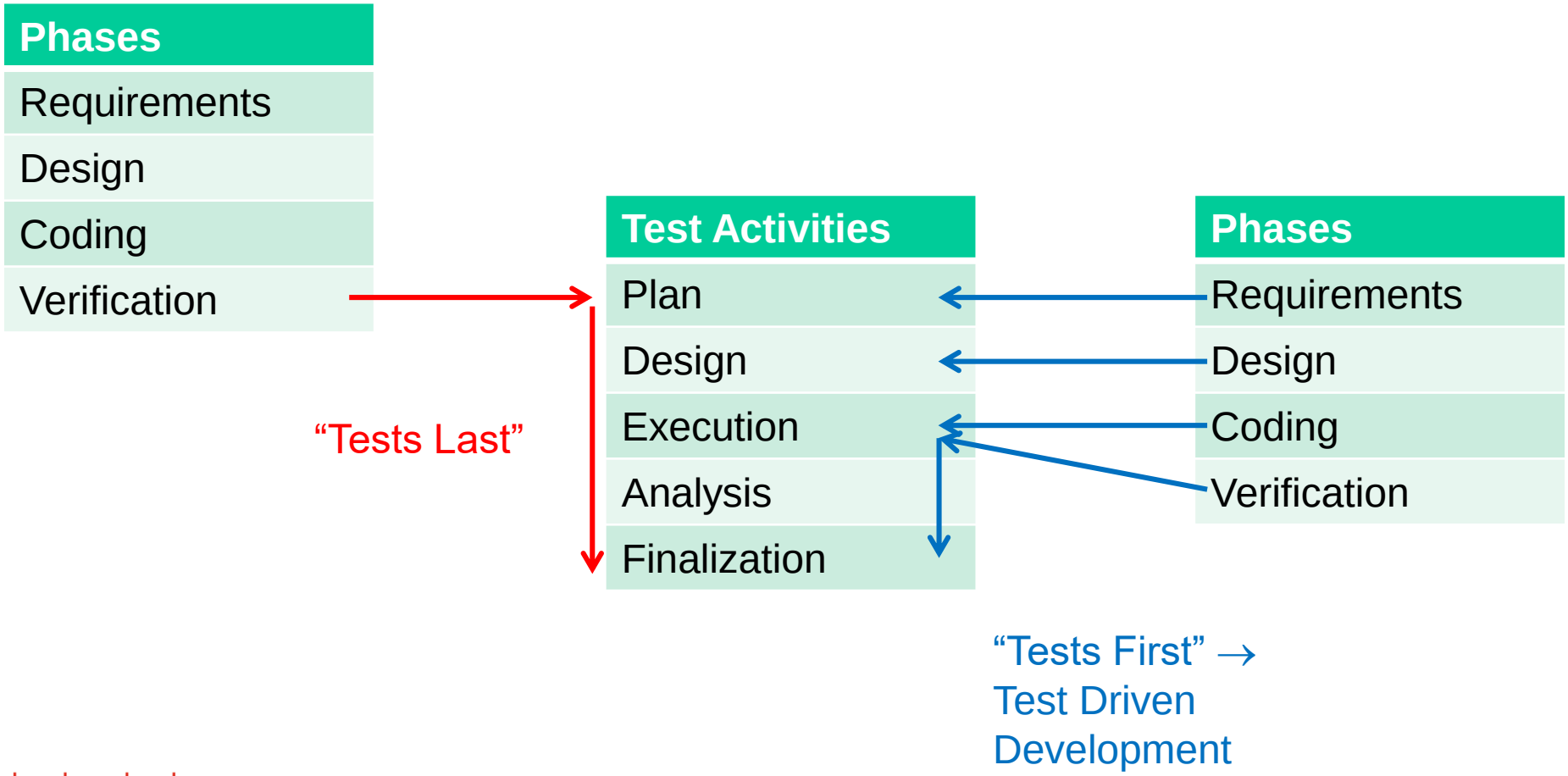
An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.

ISO/IEC/IEEE 24765-2010 –
Standard Glossary of Software Engineering Terminology

Dimensions



Process Steps

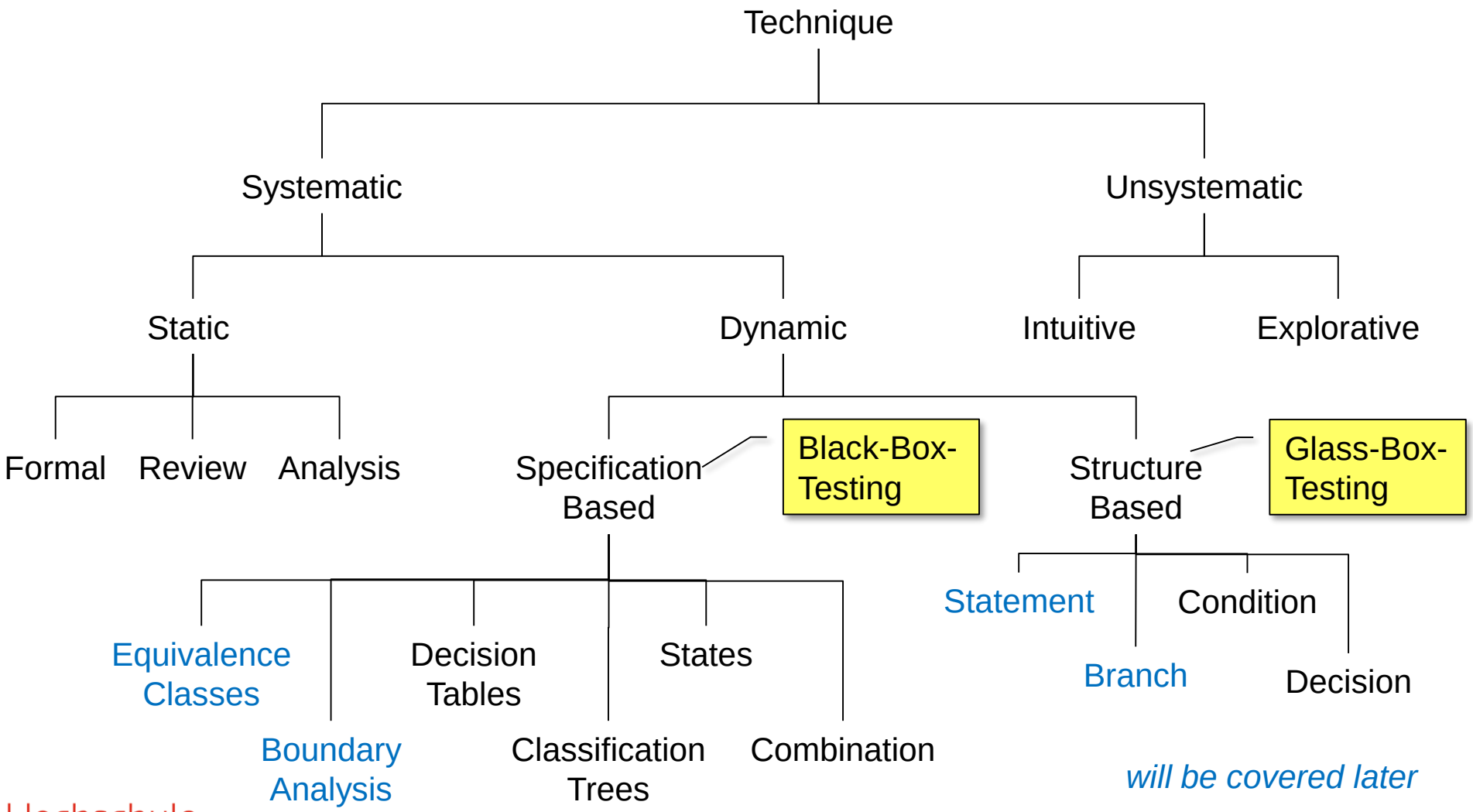


Test Targets

Type	tests	finds
Unit test / class test	individual classes	Class errors
Integration test	several classes / packages	Errors in common usage
System test	Overall system	Defects in accordance with acceptance criteria of the specification
Acceptance test	Overall system with customer	*(Actually) nothing more.*

If serious errors occur during the acceptance test, the system test was performed carelessly!

Test Techniques



Definitions

Black Box Testing

Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.

ISO/IEC/IEEE 24765-2010 –
Standard Glossary of Software Engineering Terminology

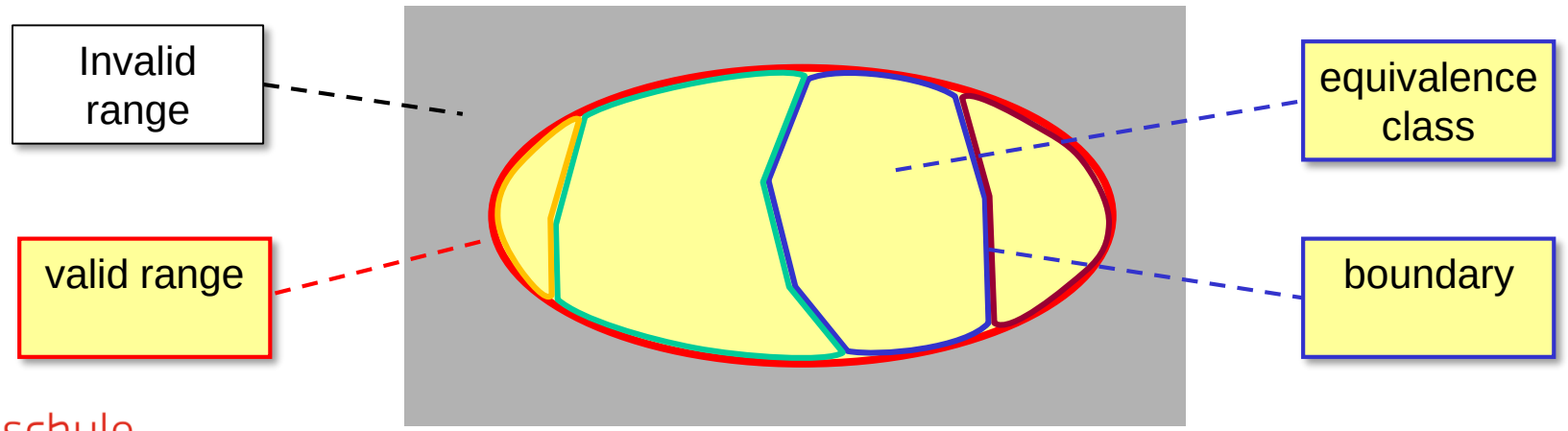
Glass Box Testing

Testing that takes into account the internal mechanism of a system or component. Types include branch testing, path testing, statement testing. Syn: structural testing; white-box testing

IEEE 610.12-1990 -
Standard Glossary of Software Engineering Terminology

Equivalence Classes / Boundaries

- For each input parameter
  - identify its domain → valid / invalid input values
  - group elements with expected equivalent behavior (→ "equivalence classes") – *this may not be true, as software is not static, but without knowing the code you can only guess*



Equivalence Classes / Boundaries

- For each equivalence class
  - select representatives from within each class
  - calculate expected output for each representative (from specification)
- For each boundary between equivalence class
  - select one boundary value
  - one value just below boundary
  - one value just above boundary
  - calculate expected output for each value (from specification)

Glass-Box Test - Statement and Branch Coverage

Statement coverage: number of statements in the program that have been processed (in relation to all statements)

- Target: 100% Statement coverage
- But: alone not sufficient
- Difficult for exception handling

Branch coverage: number of branches in the program (in relation to all possible branches)

- Goal: 100% branch coverage (*discrepancy to statement coverage if there is no alternative path*)
- (Realistic) minimum condition of the glass box test

Glass Box Test - Statements and Branches - 1

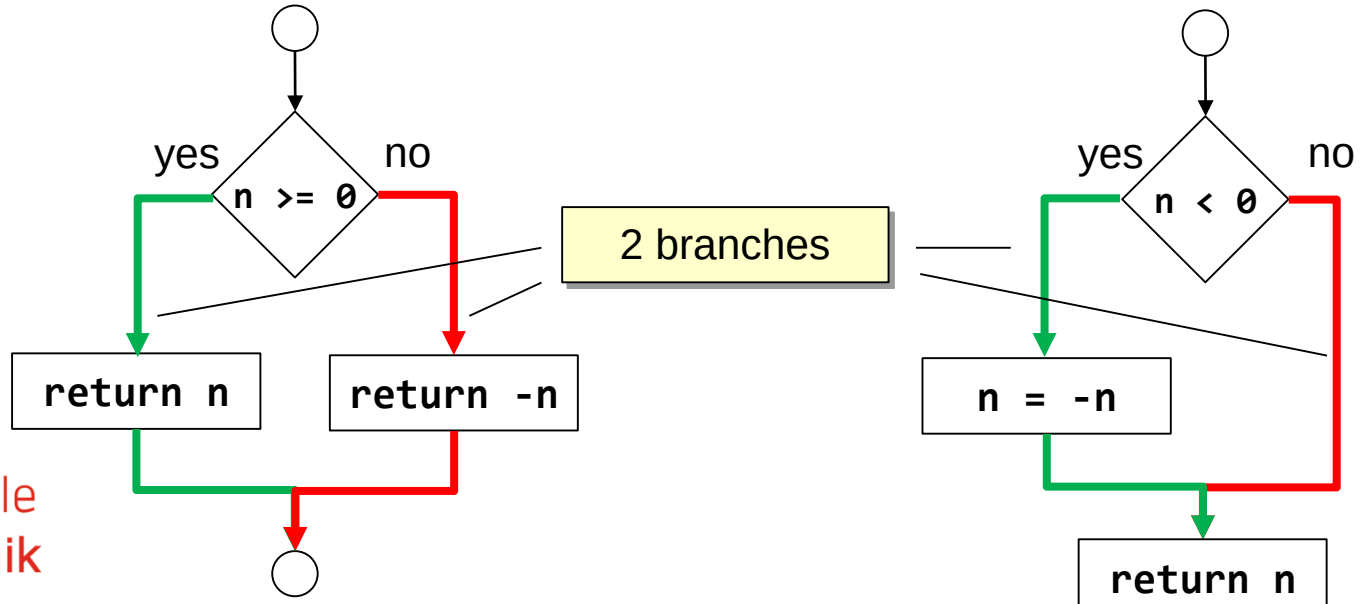
Program 1

```
public static int abs(int n){
        if(n >= 0)
                return n;
        else
                return -n;
}
```

Program 2

```
public static int abs(int n){
        if(n < 0)
                n = -n;
        return n;
}
```

3 Statements



Glass Box Test - Statements and Branches - 2

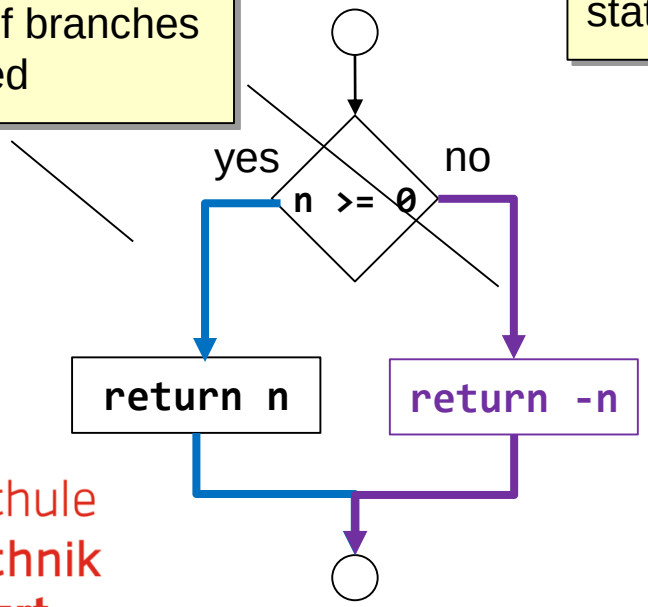
Program 1

Test of
n: 1 *and* n: -1

```
public static int abs(int n){
        if(n >= 0)
                return n;
        else
                return -n;
}
```

100% of the statements

100% of branches executed

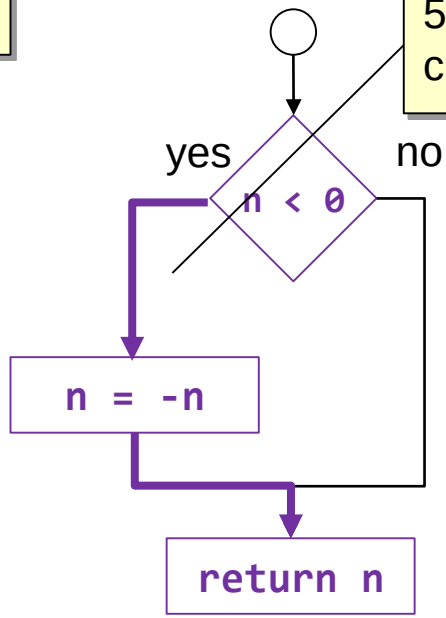


Program 2

Test of
n: -1

```
public static int abs(int n){
        if(n < 0)
                n = -n;
        return n;
}
```

50% of branches executed



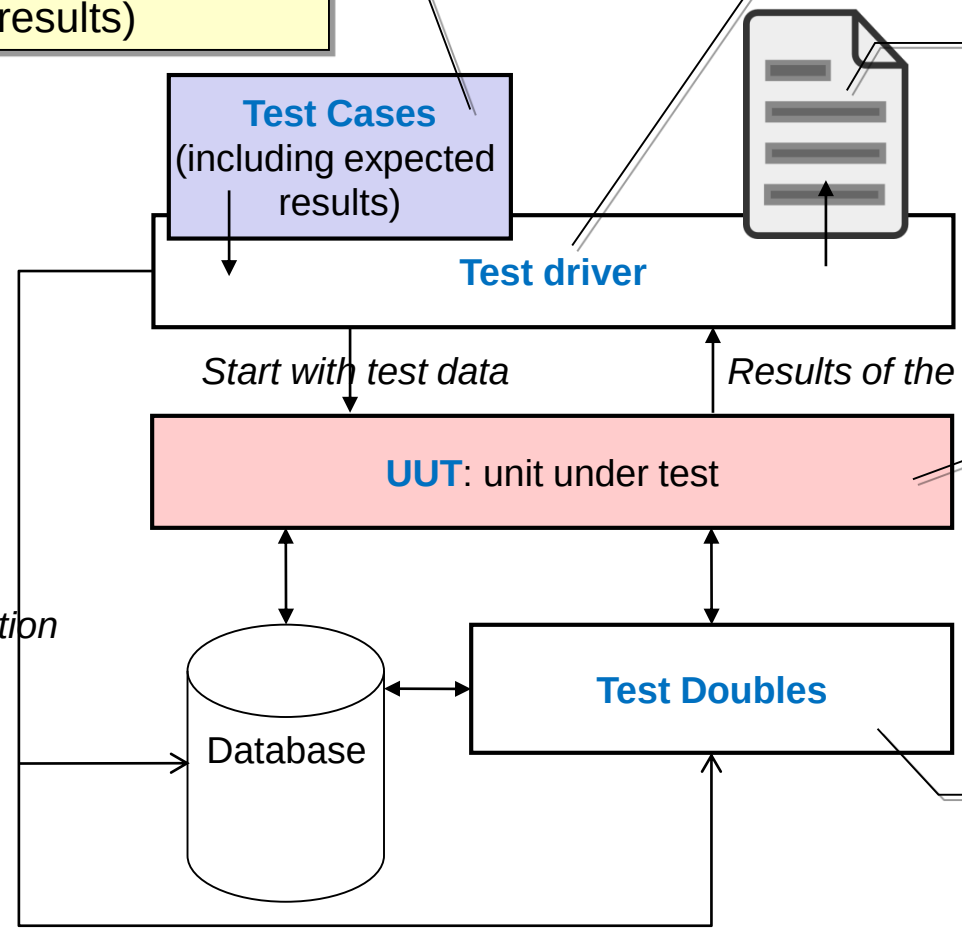
Hochschule für Technik Stuttgart

Test Workflow

The test department supplies the test cases (+ expected results)

Test driver calls the UUT with the test cases. (for example, JUnit)

Test progress **report**: conformity / deviation of actual results against expected results.



The developer provides the UUT

Developers/testers provide stubs that replace the real (too complex/not yet available) components

Test Procedures

1. Use an automated test environment
2. Start with Black-box testing
   - detect / correct errors (until a certain error-rate is reached)
   - finally measure coverage of test cases
3. In case of insufficient coverage
   - do Glass-box testing (until expected coverage is reached)
   - detect/correct more errors
4. After correcting errors
   - repeat tests (“regression test”)

Additionally: Testing of Object-Oriented Systems

Test inheritance

- top-down: super classes first, then the subclasses (Prerequisite: subclasses should be specialized from super classes!)

Reuse Super-Class-Tests

- for unchanged subclass behavior

New subclass tests

- Black-Box-tests for new subclass methods
- Glass-Box-test for overridden methods

Test Management – At Project Start

- Ideally implement test cases first (during the specification/design)
- “test-ware” is software too – not disposable code
- Test and development team should be independent (and by no means identical)
- Define realistic and measurable test objectives
- For each test provide end criteria (e.g. “95% of all test cases executed successfully”)

Test Management – During the Test Phase

- Always test against an expected result
- Complete testing is impossible → test as much as necessary, optimize test, also test the unexpected
- Check all test results
- During testing **always** expect problems
- After the end of a “regular” test : the number of undetected errors ≈ number discovered errors!

Tools for Testing – 1

Test Cases

- Support for defining equivalence classes (→ Decision trees)
- Generation of test cases from a formal specification

Test Automation

- Description of test cases in scripts (→ JUnit *)
- Any automatic repetition (→ regression testing)

** Will be presented later*

Tools for Testing – 2

Tools for test support

- Measurement of coverage (→ EclEmma*)
- native Java (→ Java-Reflection*)
- Analysis tools for the detection of possible errors (→ FindBugs*)
- Metrics to detect “critical” candidates (→ metrics*)
- Tools for building stubs and manipulation of behavior (→ Mockito*, AspectJ*)
- Tools for record/replay interaction (→ Selenium*)
- Runtime Analysis
- Load and Performance Testing

** Will be discussed later*

Tools for Testing – 3

Test Management

- Planning of tests
- Visualizing the actual test status
- Visualizing the test process

Commercial Tools

- HP Quality Center
- IBM Rational

Open Source Tools

- www.opensourcetesting.org

JUnit

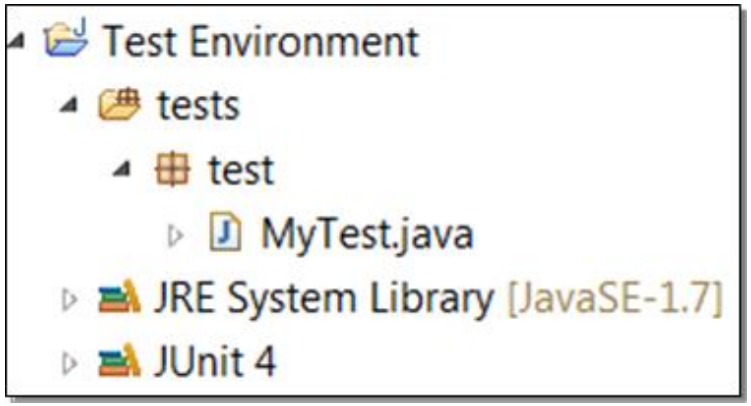
A quick repetition

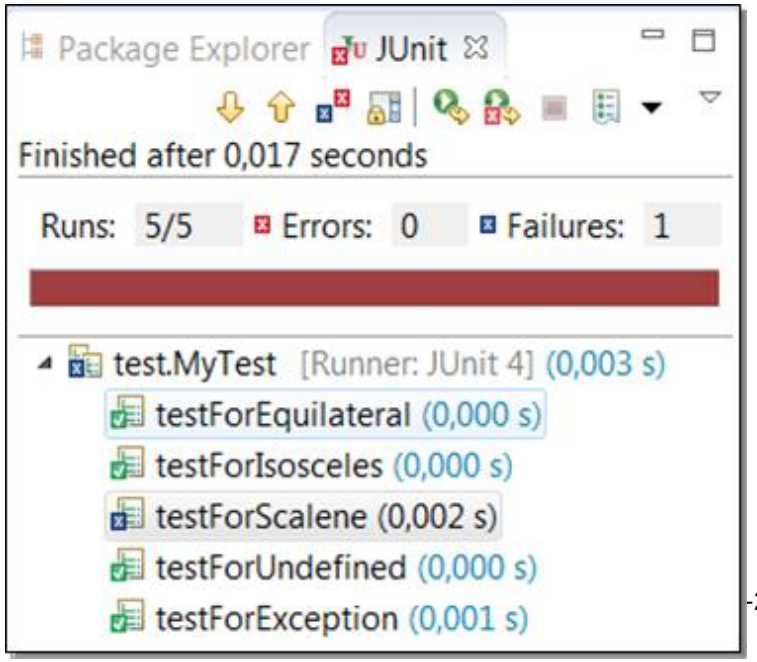
JUnit

- developed by Kent Beck and Erich Gamma
- tests methods at class level
- standard java test environment
- basis for numerous extensions
- web www.junit.org
- versions
  - JUnit 3.x works through inheritance (until Java 1.4)
  - JUnit 4.x is based on annotations (since Java 1.5)
  - JUnit 5.x is based on lambdas (since Java 1.8)
- Integration in all IDEs

JUnit – Usage in Eclipse

- create a test project
- add a new source folder “tests”
- add JUnit-Library
- implement test methods (→ see next slides)
- create a development project
- add development project to the test project
- run tests while development





JUnit – Test Case Structure

Static Import allows direct access to assert… - methods

A test case.

Check: expected value = actual value; if not the test case fails.

Check: Is an Exception thrown? if not the test case fails.

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

public class MyTest {

    @Test  // The "Test"-Annotation marks the methods
    public void test1() {

        type expected = … ; // expected value
        type actual = functionUnderTest(parameter);
        assertEquals(expected, actual);

    }

    @Test
    public void test2() {

        type n = … // should throw an exception
        assertThrows(RuntimeException.class,
                              () -> functionUnderTest(n));

    }
}
```

lambda-Expression for supplying the method

Junit – Running the Test Cases

Each assertion checks, if the actual result matches the expected result.

Three possible outcomes

- `actual result == expected result` → success
- `actual result != expected result` → failure
- `an error occurs` → error

If assertions pass, the entire test run is successful, otherwise it is a failure

In an IDE, this is often indicated by a color code:

- green means passed
- red means failed

JUnit-Assertions

`assertEquals(expected, actual)` `assertEquals(expected, actual, delta)`	Compare two values for equality (with possible delta)
`assertTrue(actual)` `assertFalse(actual)`	Test whether expression is true or false
`assertNull(actual)` `assertNotNull(actual)`	Test the reference variable to null or not null
`assertSame (expected, actual)` `assertNotSame(expected, actual)`	Test (Non-)equality of the two reference variables
`fail (String)`	forced fail

JUnit-Assertions with Delta

Delta allows to accept to floating point values to be equal, if they differ “delta”. This helps to compensate numerical problems.

Example:

- `assertEquals(0.16d, 1/6.0d, 0.02)`

Please note that this can lead to problems, if delta is wrongly chosen → incorrect values may be recognized as correct

- `assertEquals(0.8d, 1/6.0d, 1.0)`

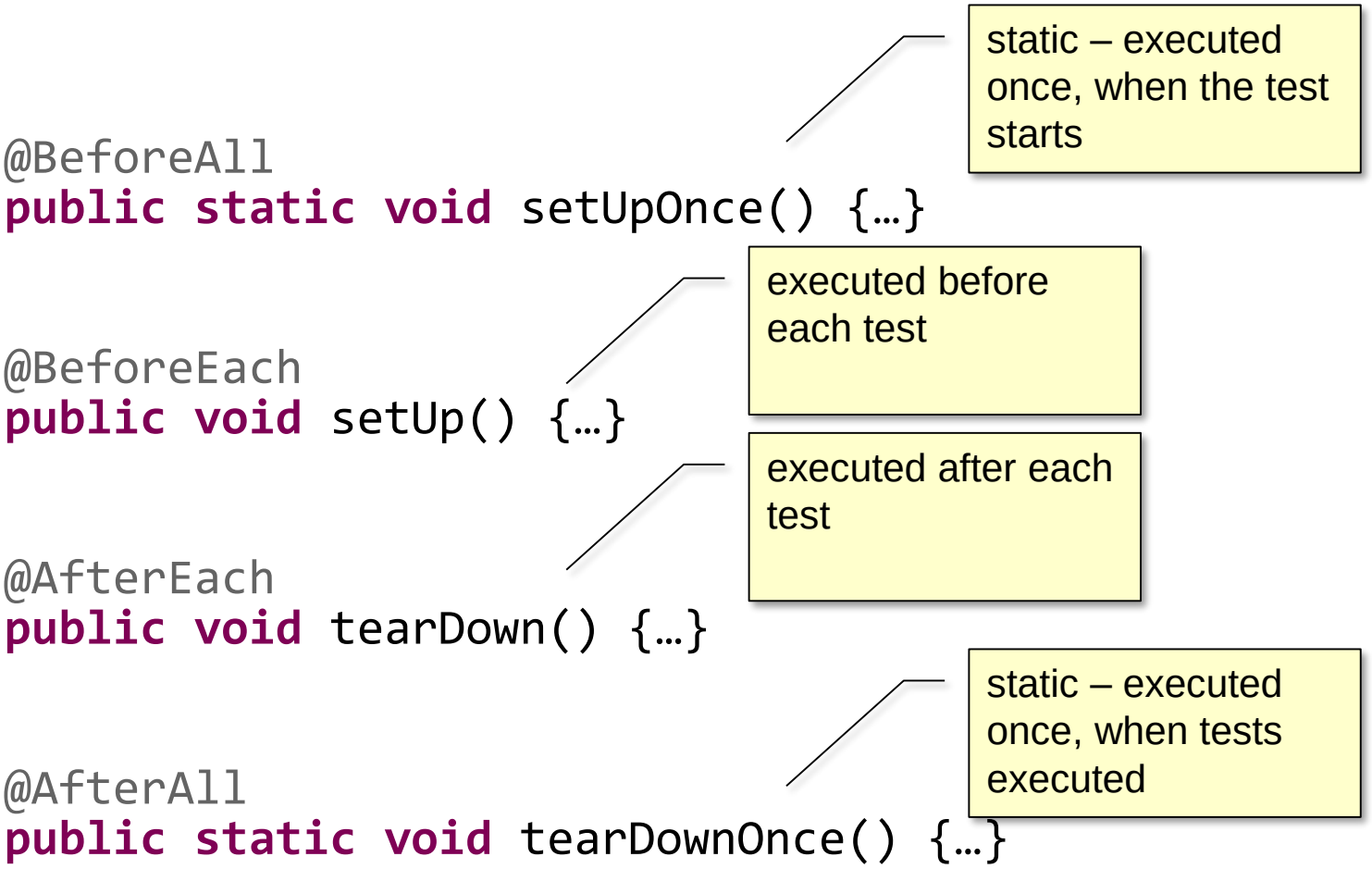
JUnit-Assertions for time-outs

Optionally specify a timeout to cause a test method to fail if it takes longer than that number of milliseconds.

Examples:

```
@Test
@Timeout(2)  // Seconds
public void testforTimeout(){
        procedure(); // fails if it takes longer
}
@Test
@Timeout(value = 2000, unit = TimeUnit.MILLISECONDS)
public void testforTimeout(){
        procedure(); // fails if it takes longer
}
```

Junit-Set-Up/-Tear-Down



JUnit-Control – Test Suites

Grouping several test classes in one test suite, which can be run at once.

```
@RunWith(JUnitPlatform.class)
@SelectClasses({Test1.class, Test2.class})
public class TestSuite {

}
```

These are the classes containing the test cases

or

```
@RunWith(JUnitPlatform.class)
@SelectPackages({package1, package2})
public class TestSuite {

}
```

These are the packages containing the test classes (with suffix “Test”

EclEmma

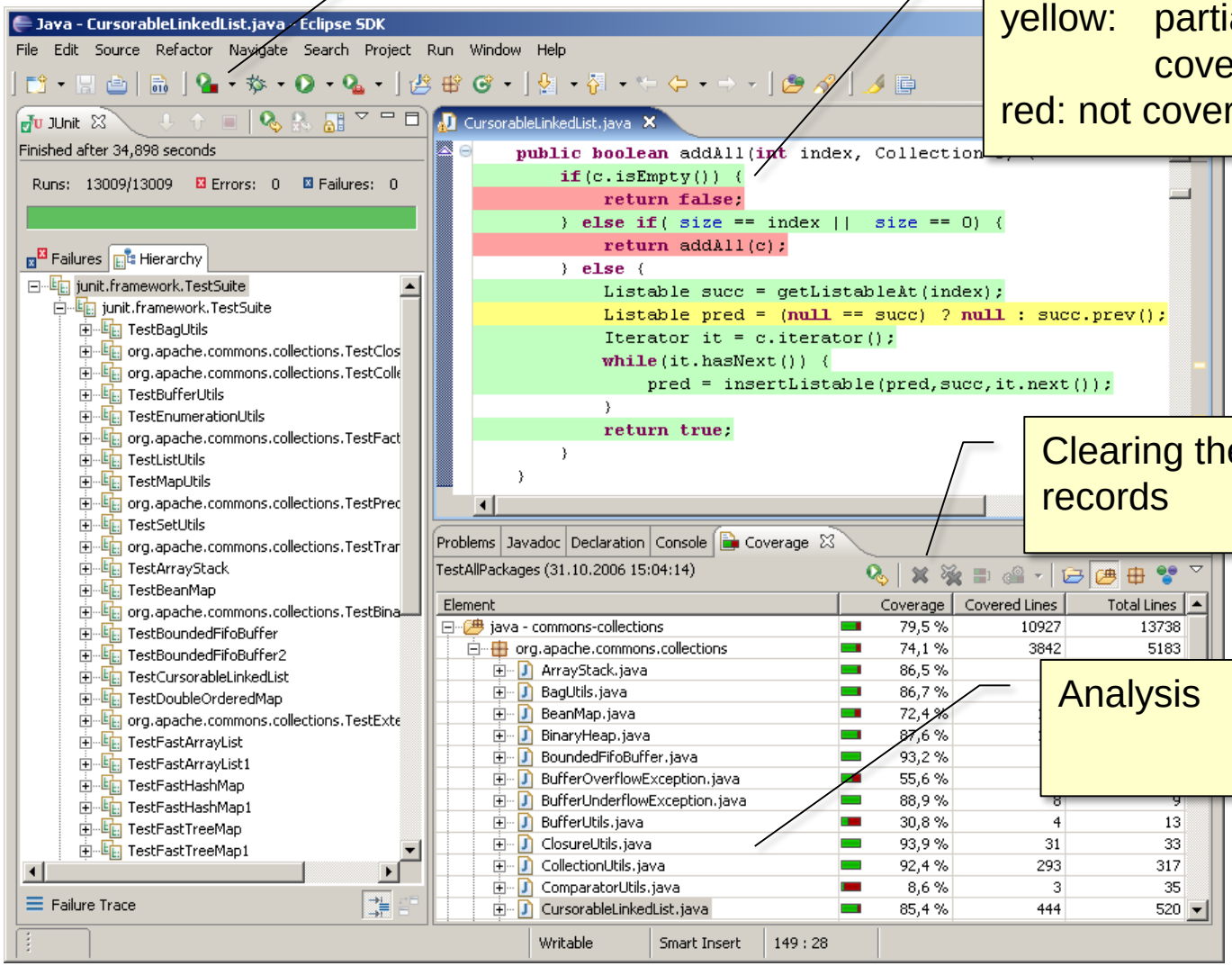
eclEmma – 1

- EMMA is a code coverage tool developed by Vlad Roubtsov, 2005 (named after his wife)
- EMMA is open source (http://emma.sourceforge.net/)
- eclEmma is the Eclipse plugin for EMMA, install from http://update.eclemma.org/

eclEmma – 2

Run with eclEmma

green: covered
yellow: partially covered
red: not covered



Clearing the records

Analysis

image: eclemma.org

eclEmma – 3

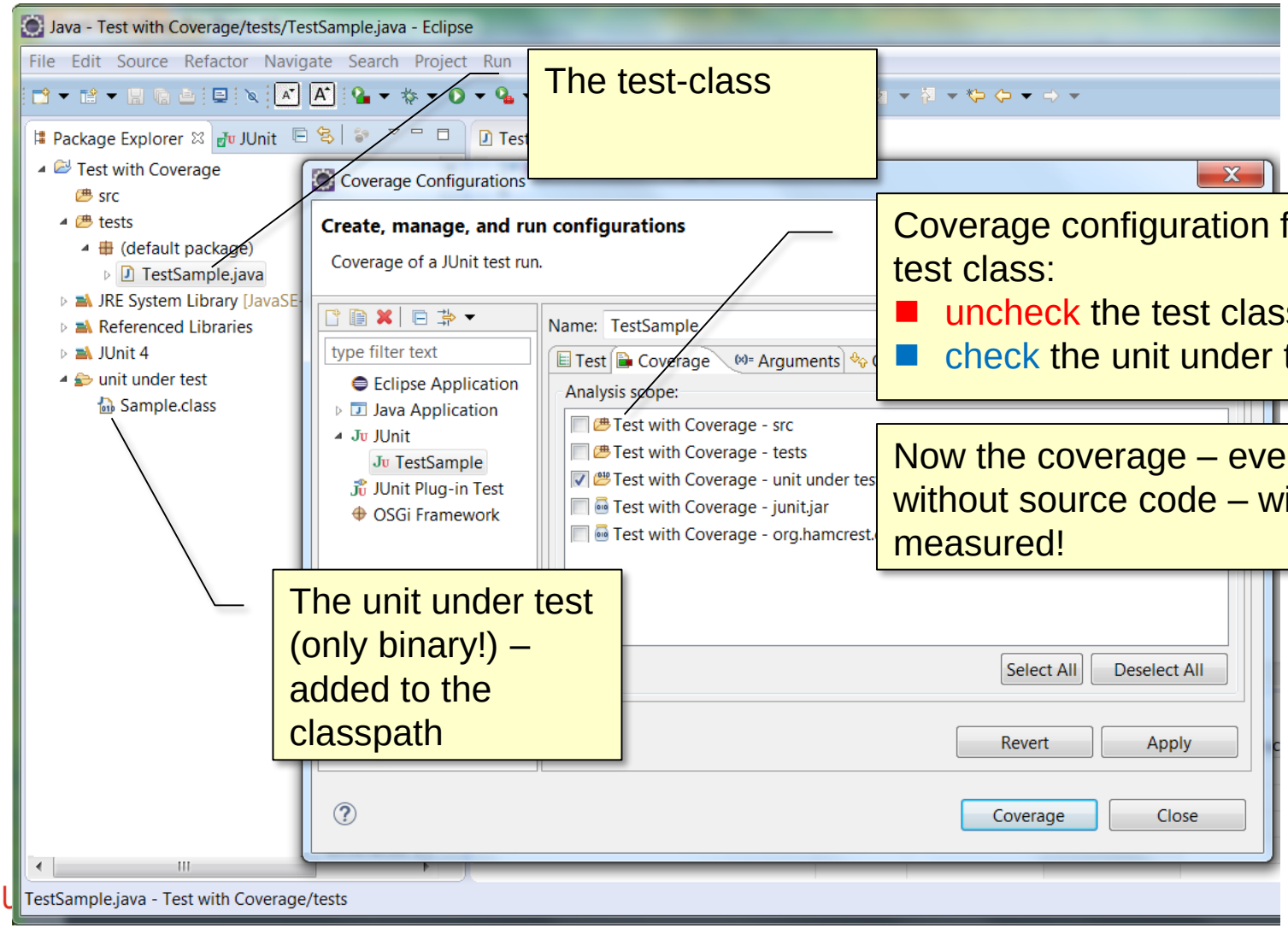


image: eclemma.org

Sometimes it is necessary, to access or change private (or even constant) fields for test purposes → Reflection can support this.

Native Java – Accessing Private Fields

```
Object target = … // The object containing the field
String fieldName… // The name of the field
Class<?> targetClass  = target.getClass();
```

Use reflection to access the field by name.

```
Field field = targetClass.getDeclaredField(fieldName);
field.setAccessible(true);
Object value = field.get(target);
```

Make a (possible) private field accessible.

Get the value – which has to be casted.

```
Object newValue = … // The new value to be set
field.set(target, newValue);
```

Set a new value – which has to be conform to the field type.

Native Java – Accessing private Methods

```
Object target = … // The object containing the method
String methodName … // The name of the method
Object[] argObjects = … // The invocation arguments

Class<?> targetClass  = target.getClass();
Class<?>[] argClasses = new Class[argObjects.length];
for(int i = 0; i < argObjects.length; i++)
    argClasses[i] = argObjects[i].getClass();

Method method = targetClass.getDeclaredMethod(methodName, argClasses);
method.setAccessible(true);
Object result = method.invoke(target, argObjects);
```

Use reflection to access the method by name and parameter classes (= signature).

Make a (possible) private method accessible.

Execute the method.

If the method returns a result you get it here (if not you would get an Exception here)

Example – Specification and Test cases

- `abs(int n):int` shall supply the absolute value of a given number
- if `Integer.MIN_VALUE` is entered, a runtime exception should be triggered (because there is no corresponding positive value)

Equivalence class	Input	expected outcome
Zero	`0`	`0`
Positive numbers	`1`	`1`
	`2`	`2`
	`MAX_VALUE`	`MAX_VALUE`
Negative numbers	`-1`	`1`
	`-2`	`2`
	`MIN_VALUE + 1`	`MAX_VALUE`
Overflow	`MIN_VALUE`	`RuntimeException`

Example – Extract from Test Class

```
package tests;

import static unit.AbsCalculator.*;

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

class AbsTest {
    …
    @Test
    void testAbsPos1() {
        int n = 1;
        int expected = 1;
        assertEquals(expected, abs(n));
    }
    …
    @Test
    void testAbsMinusMin() {
        int n = Integer.MIN_VALUE + 1;
        int expected = Integer.MAX_VALUE;
        assertEquals(expected, abs(n));
    }

    @Test
    void testAbsUnderflow() {
        int n = Integer.MIN_VALUE;
        assertThrows(RuntimeException.class, () -> abs(n));
    }
}
```

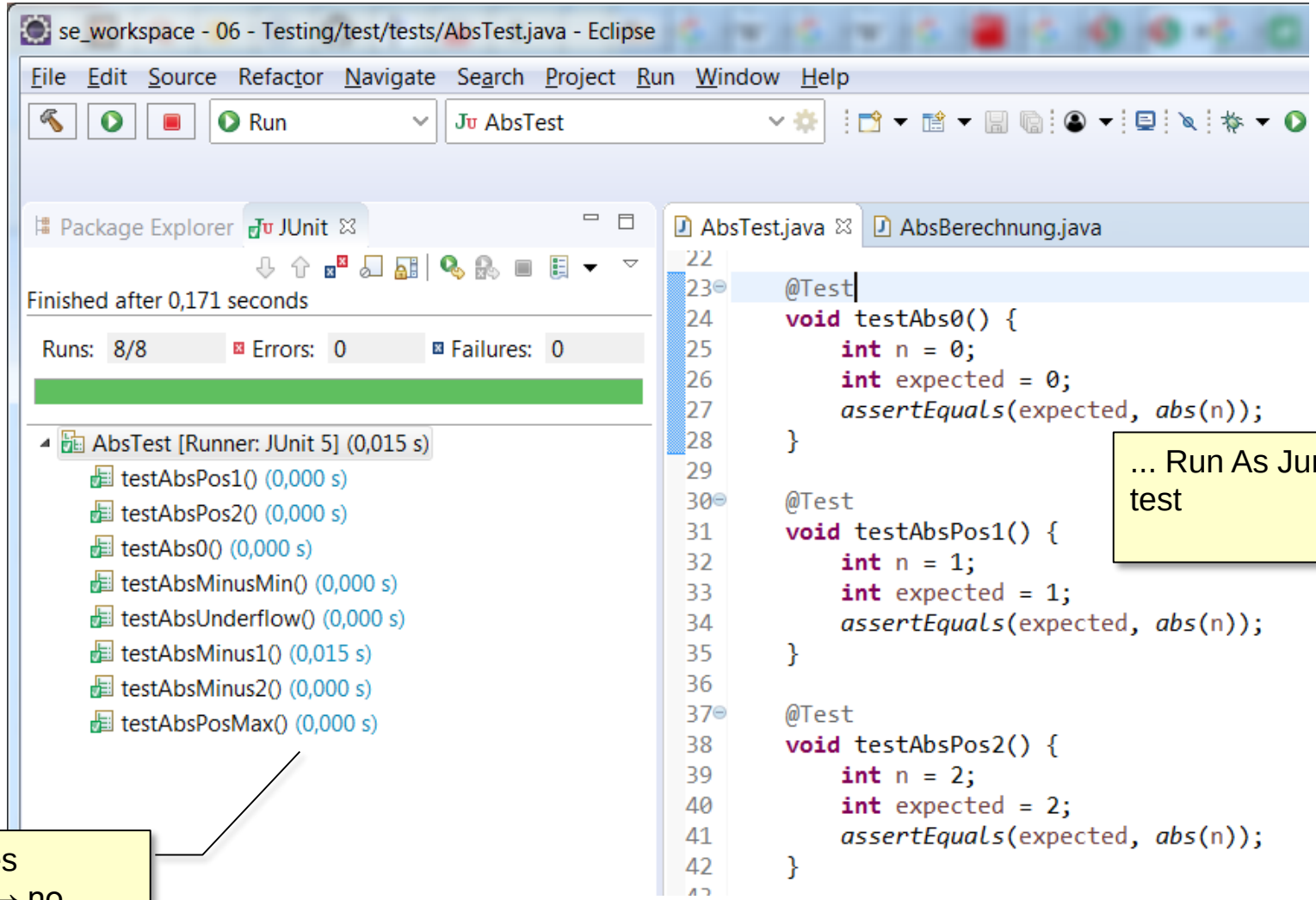
Example – Test Sample

```
package unit;

public class AbsCalculator {

    public static int abs(int n) {
        if(n == Integer.MIN_VALUE)
            throw new
              RuntimeException("Underflow");
        if(n >= 0)
            return n;
        else
            return -n;
    }
}
```

Example – Execution



... Run As Junit test

All test cases successful → no errors