JUnit 5 Workshop

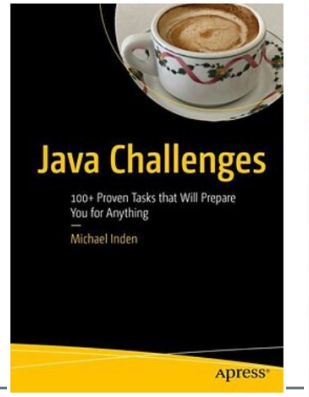
More fun and less stomach ache during development

Michael Inden

Speaker Intro

- Michael Inden, Year of Birth 1971
- Diploma Computer Science, C.v.O. Uni Oldenburg
- ~8 ¼ Years **SSE** at Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Years **TPL, SA** at IVU Traffic Technologies AG in Aachen
- ~4 ¼ Years **LSA / Trainer** at Zühlke Engineering AG in Zurich
- ~3 Years **TL / CTO** at Direct Mail Informatics / ASMIQ in Zurich
- Independent **Consultant, Conference Speaker and Trainer**
- **Since January 2022 Head of Development at Adcubum in Zurich**
- Author @ dpunkt.verlag and APress

E-Mail: michael_inden@hotmail.com
Blog: https://jaxenter.de/author/minden



https://github.com/Michaeli71/ADC_BOOTCAMP_SPRING

Agenda

Workshop Contents

- **PART 1: Why to test and good habits**
  - Why do we test?
  - Good habits
- **PART 2: JUnit 5 Intro**
  - Architecture
  - First Test
  - Tests with multiple asserts
  - Testing Exceptions
  - Tests with Timeouts

Workshop Contents

- **PART 3: JUnit 5 Advanced**
  - Parameterized Tests
  - Parameterized Tests Advanced
  - Repeated Tests
  - Simple Extensions
- **PART 4: Tips for migration JUnit 4 => JUnit 5**
  - Migration or / and parallel operation
  - AssertJ
- **PART 5: Test styles and dependencies**
  - State-based vs. behavior-based testing
  - Proxy objects

Workshop Contents

- **PART 6: Design For Testability**
  - Predetermined breaking points
  - Extract and Override
  - Mockito
- **PART 7: Test Smells**
- **PART 8: Test Coverage**
- **PART 9: Characterization Testing**

https://github.com/Michaeli71/AST-JUnit5

PART 1:
Why to test and good habits

What is testing?

- Testing is the process of comparing the actual behavior of a program or a part of it (actual) with the required behavior (target).
- Accordingly, testing does not correspond to the one-time start of a program with a few arbitrary operating actions.
- to test behavior of the software under different conditions.
- to act a little "maliciously" in order to uncover hidden problems or to provoke misbehavior.

Why do we test?

- Describe desired behavior
- Check functionality (also edge cases)
- Build a safety net
- Quality assurance
- Customer satisfaction
- less hassle, less nerves and more fun

External Quality

External Quality corresponds to user view

- **Works as expected**
- **Provides all required functionality**
- **Correctness**
- **Almost no (observable) bugs**
- **Well tested**
- **Usable**
- **Reliable**
- …

Internal Quality

Internal Quality ~ Developer view (code, build, testing):

- Readability
- Understandability
- No pitfalls and obstacles
- Extensible, maintainable
- Well/meaningfully documented
- Good test/code coverage
- …

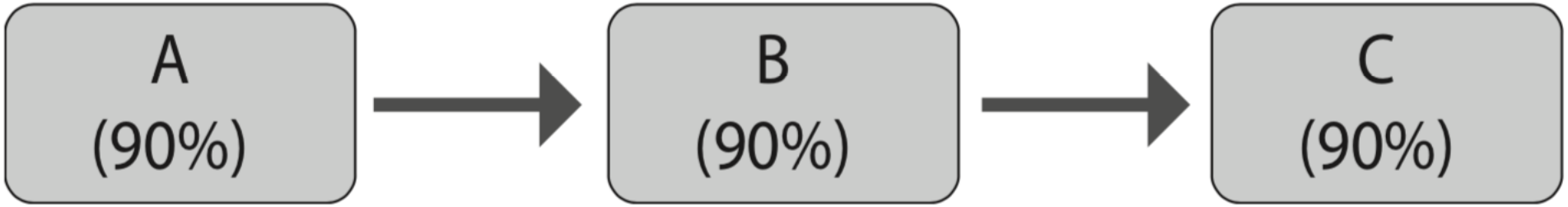
Why do we test? What is quality?

- By quality, everyone understands something slightly different:
  - good usability or
  - extensive functionality.
  - But we always want a high degree of **conformity to expectations** and **reliability**.
  - In real life, we almost always expect a quality of nearly **95 - 100 %.**

- Why do we often settle for less when it comes to software? What could be the causes of non-optimal quality?
  - Complexity in software is often quite high
  - Individual parts are partly not well tested
  - Computer science does not yet have the level of engineering like the automotive industry or mechanical engineering

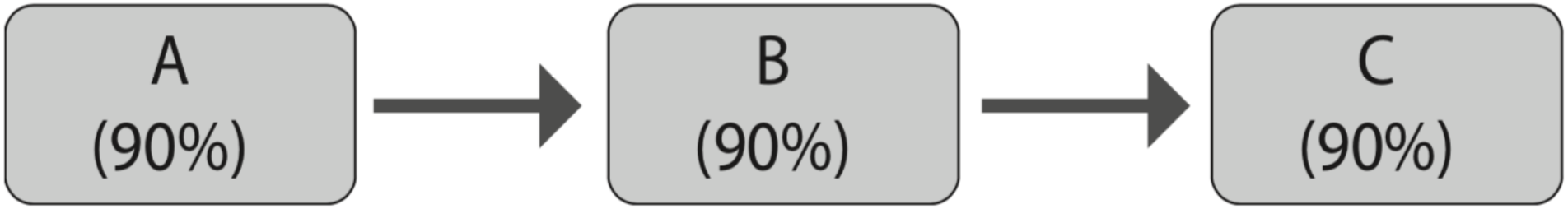
Influence of the quality of individual parts

- The industry has norms and standards. Quality as well as interoperability is given.
- The higher the quality of the individual parts, the higher the quality of the overall product. Let's look at a simple system with three building blocks:



- What is the overall quality?

Influence of the quality of individual parts



- According to system theory: product of individual qualities

  0.9 * 0.9 * 0.9 = 0.73 => 73 %

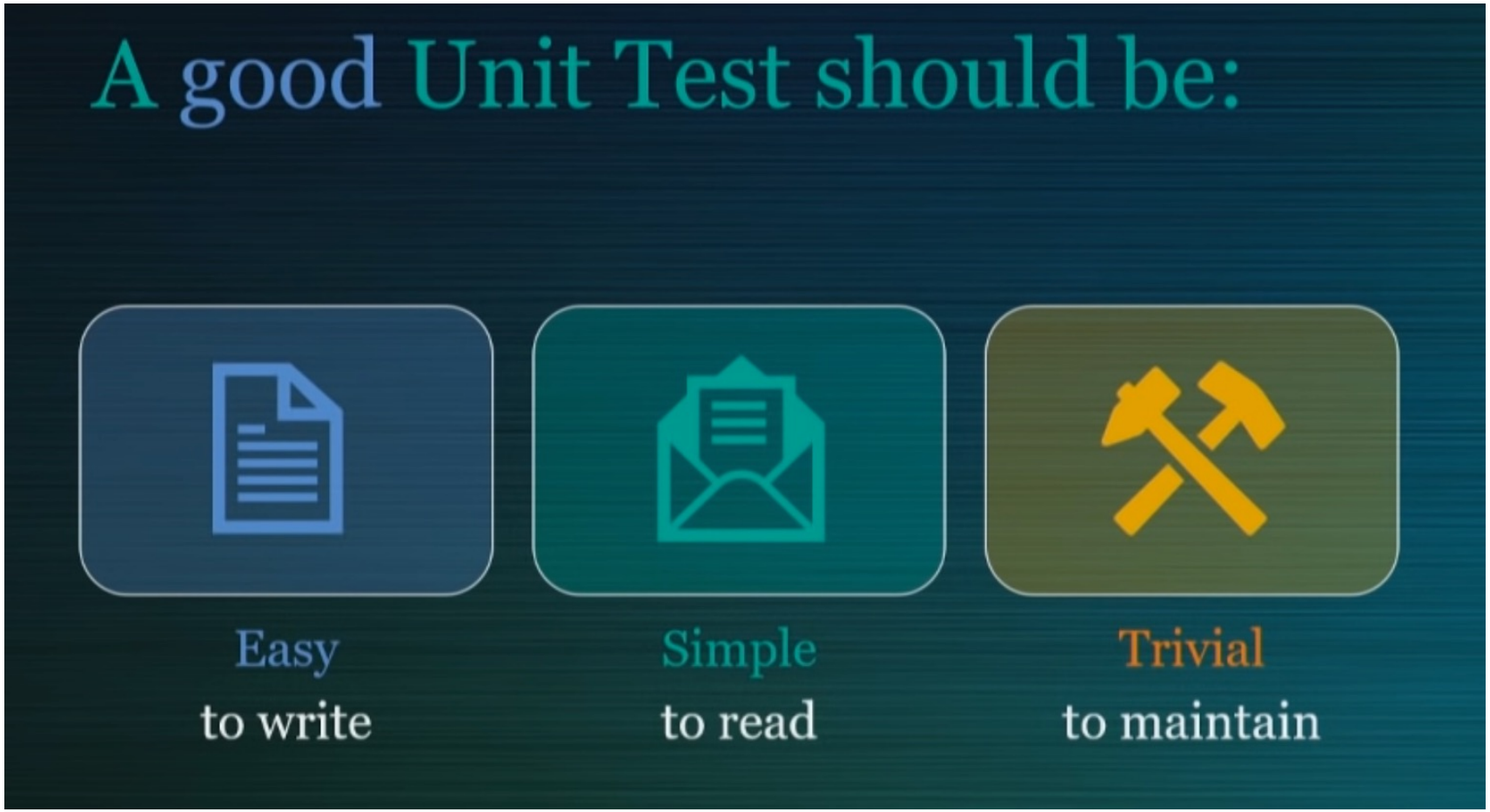
- Applied to software, we have the following problems:
  - Who has ever built systems with only 3 classes or components?
  - Normal applications do not only consist of a multiplicity of classes and objects, but these possess in particular also complicated, partly tricky dependencies.

Why do we create unit tests?

- We use it for **quality assurance at the small parts level**.
- Close integration into the development process => **fast feedback**
- Most of the time the errors can be easily corrected.

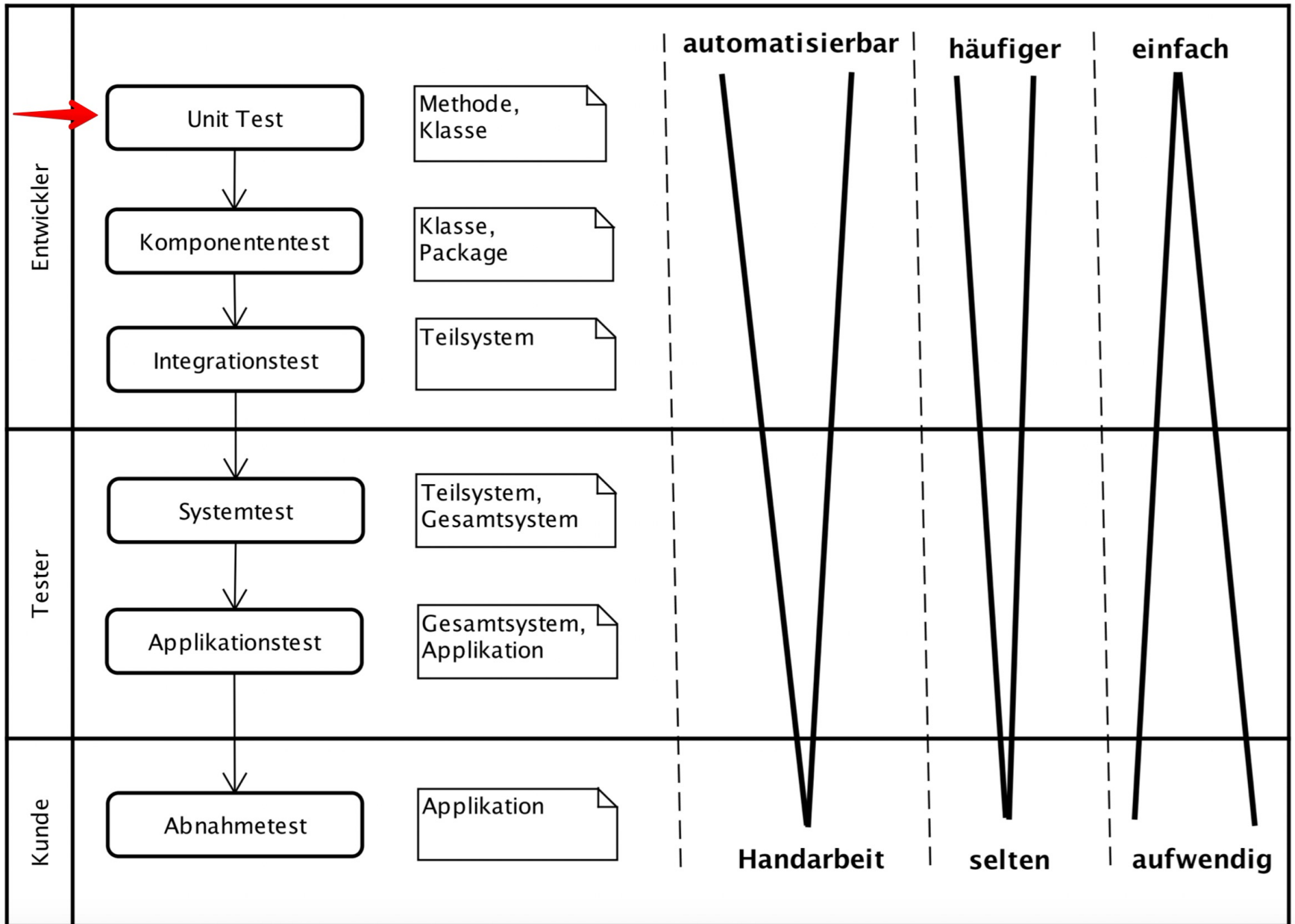
- Let's get back to standardization and quality: **Every single one of us can contribute a little bit to the improvement by following coding conventions, writing good unit tests and paying attention to a clean design.**

What makes a good unit test?

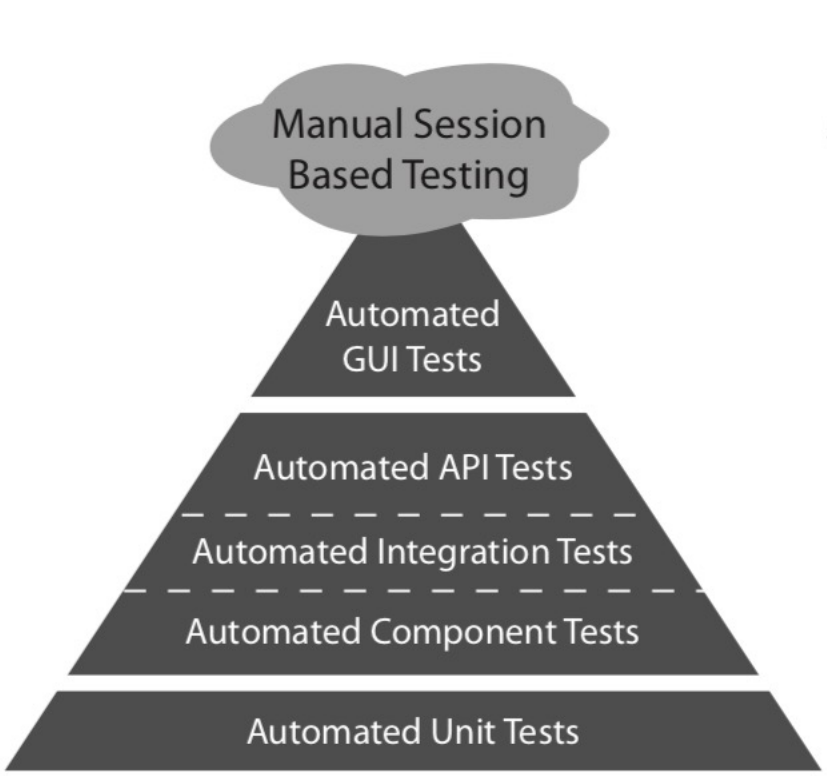


https://www.youtube.com/watch?v=V6MzzmJ9IPs -- The secret unit testing tools no one ever told you about - Dror Helper

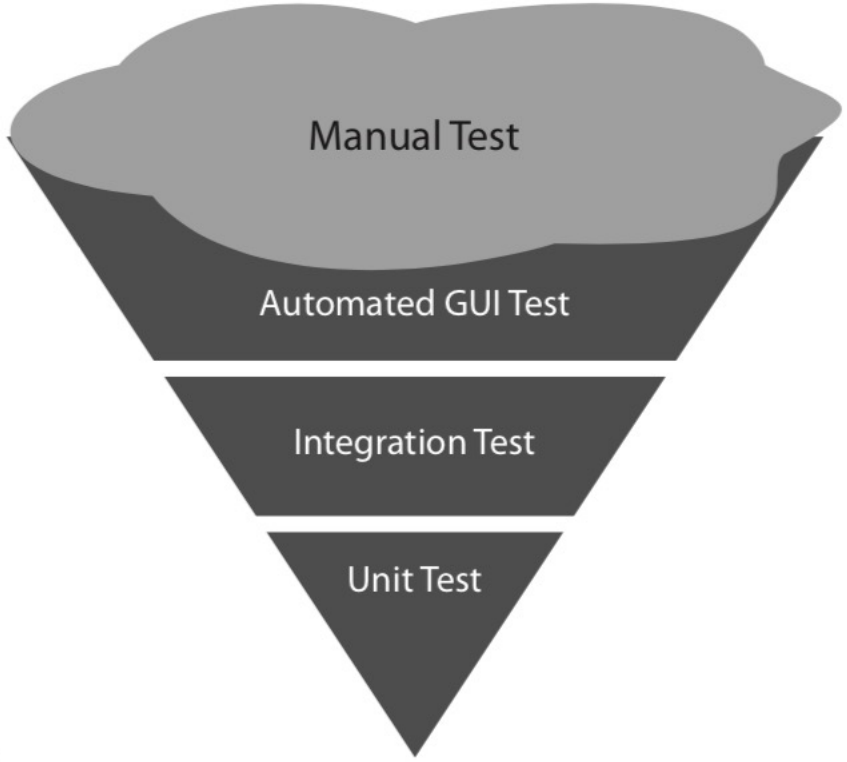
Types of tests



Test pyramid



The Ideal Testing Automation Pyramid



The Non-Ideal Testing Automation Inverted Pyramid

What makes it hard for us to write tests?

Difficulties in unit testing

- What makes it difficult for us to write good unit tests? - It's almost the same things that can make development difficult.
    - One should know BEFOREhand what is to be realized and how it should work.
    - If you develop the appropriate software to the requirements, then a successful naming and a problem-adapted design is cruical, especially clear responsibilities.
    - Often the reality looks different and classes offer too much functionality. As a result, there are usually too many dependencies on other classes.
    - This in turn leads to fragility, if you change something in one place, it breaks in another.

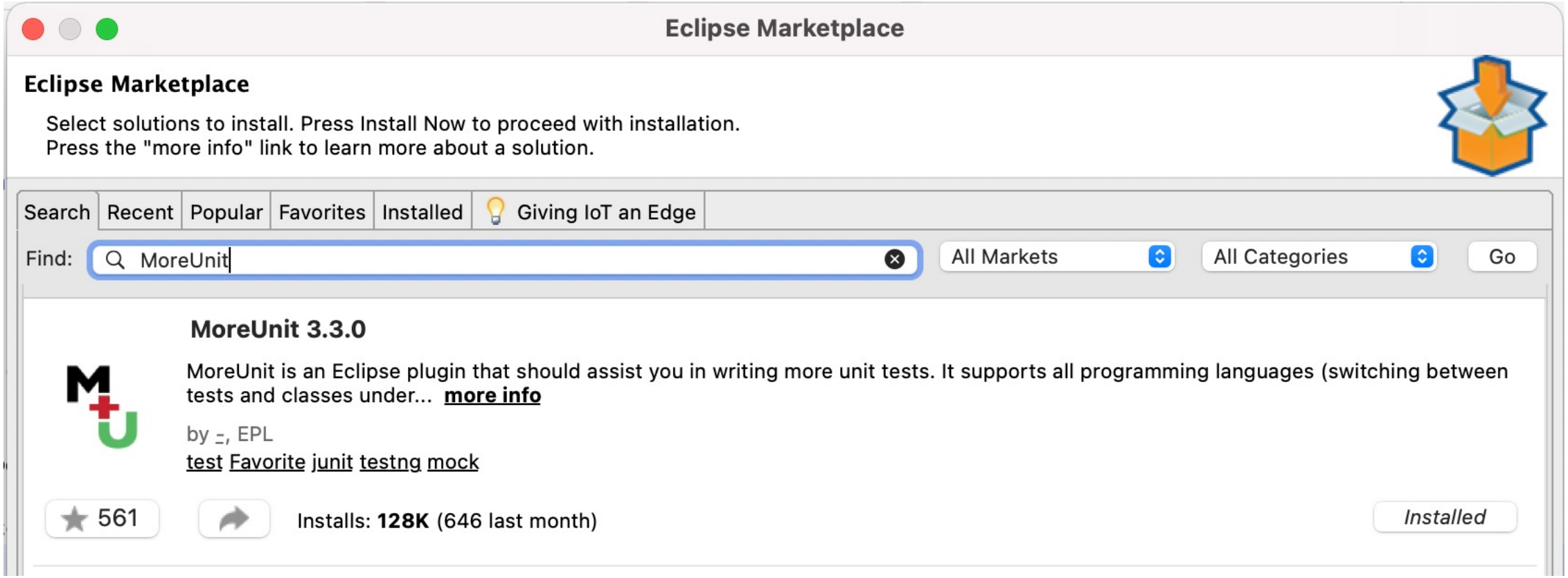
Difficulties in unit testing

- For unit tests, information about the desired functionality helps us to create adequate test cases and not just write pseudo tests for trivial `get()/set()` methods.
- well written, meaningful names of the test methods help to identify the essence of the test case
- To be able to formulate each test case as crisply as possible, not too many dependencies on other classes should exist.
- Remedy offers the test doubles presented later. The name is striking and resembles the stunt double from movies: A test double is a proxy for an application object to resolve dependencies and make a unit more testable.

Good habits

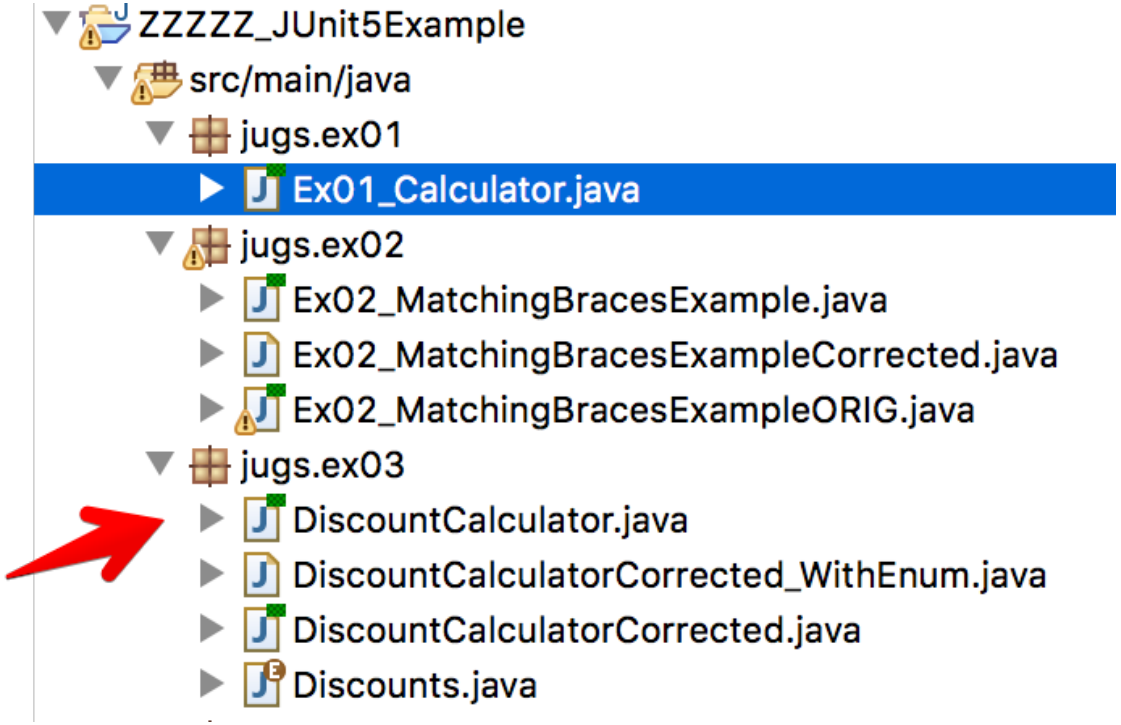
(Eclipse) Plugin MoreUnit

- http://moreunit.sourceforge.net/
- http://moreunit.sourceforge.net/update-site/



(Eclipse) Plugin MoreUnit

- Keyboard shortcuts to run (CTRL+R) and switch between class and test (CTRL+J).
- Icon decoration in the Package Explorer: green dot shows whether a test exists for a class.
- Refactorings: classes and corresponding test classes are moved or renamed synchronously to each other.



(Eclipse) Plugin MoreUnit

New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

JUnit 3 | JUnit 4 | JUnit 5 | Spock | TestNG

Source folder: ZZZZZ_JUnit5Refresher/src/test/java — Browse...

Package: newinrefresher.parameterized — Browse...

Name: Ex01_LeapYearTest

Superclass: — Browse...

Which method stubs would you like to create?

- setUpBeforeClass()
- tearDownAfterClass()
- setUp()
- tearDown()
- constructor

Do you want to add comments? (Configure templates and default value here)

- Generate comments

Class under test: newinrefresher.parameterized.Ex01_LeapYear — Browse...

< Back | Next > | Cancel | Finish

New JUnit Test Case

Test Methods

Select methods for which test method stubs should be created.

Available methods:

- Ex01_LeapYear
  - main(String[])
  - isLeap(int)
- Object
  - Object()
  - getClass()
  - hashCode()
  - equals(Object)
  - clone()
  - toString()
  - notify()
  - notifyAll()
  - wait()
  - wait(long)
  - wait(long, int)
  - finalize()

Select All | Deselect All

1 method selected.

- Create final method stubs
- Create tasks for generated test methods

< Back | Next > | Cancel | Finish

Maven Project Structure

```
.
├── build.gradle
├── libs
│   └── assertj-core-3.17.1.jar
├── pom.xml
└── src
    ├── main
    │   └── java
    │       ├── a_first_slides
    │       └── utils
    │           ├── FibonacciCalculator.java
    │           ├── PalindromeChecker.java
    │           └── PrimeNumberChecker.java
    └── test
        └── java
            ├── a_first_slides
            │   ├── FibonacciTest.java
            │   ├── PrimeNumberCheckerTestJUni5.java
            │   ├── PrimeNumberCheckerTestJUnit4.java
```

Define test cases

- Unit tests verify small components, mostly classes or methods
- As isolated as possible and without interaction with other components
- Tests are implemented in the form of methods
- Ideally: at least one test method for each relevant application method
- A test method tests exactly one functionality (or only a part of it) , ideally only 1 ASSERT !
- Keep test methods short, clear and understandable
- BUT: How to achieve this?

Naming

- Class `Abc` => associated test class `AbcTest`
- Methods
  - Optional: abbreviation test as start
  - Meaningful description of the test case:
  - Code method name, conditions and result in the name => CamelCase often becomes unreadable
  - Testing guru Roy Osherove suggests the following

```
MethodName_StateUnderTest_ExpectedBehavior
MethodName_ExpectedBehavior_WhenTheseConditions
```

```
calcSum_WithValidInputs_ShouldSumUpAllValues()
calcSum_ThrowsException_WhenNullInput()
```

JUnit – Definint test cases

- JUnit is a framework written in Java that assists in writing and automating test cases at the class level.
- For testing an application class usually a corresponding test class is written, in methods test assertions are set up and evaluated.
- Often one begins chekcing important functionality of own classes examing first some central methods by tests.
- This can be extended step by step afterwards

Example: A first unit test with JUnit 5

- Test cases get written in the form of special test methods marked with the annotation `@Test`

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

public class A_FirstTestWithJUnit5
{
    @Test
    void assertMethodsInAction()
    {
        String expected = "Tim";
        String actual = "Tim";

        assertEquals(expected, actual);
        assertEquals(expected, "XYZ", "Message if comaprison fails");
    }
}
```

AAA-Style

- **ARRANGE** - **ACT** – **ASSERT** (Also called GWT for GIVEN - WHEN - THEN)
- **ARRANGE:** preconditions and initializations **(test fixture)**
- **ACT:** then an action is executed
- **ASSERT:** check if the expected state has occurred

```
@Test
void listAdd_AAAStyle()
{
   // GIVEN: An empty list
   final List<String> names = new ArrayList<>();

   // WHEN: adding 2 elements
   names.add("Tim");
   names.add("Mike");

   // THEN: list should contain 2 elements
   assertEquals(2, names.size(), "list should contain 2 elements");
}
```

FAIR - Desired properties of unit tests

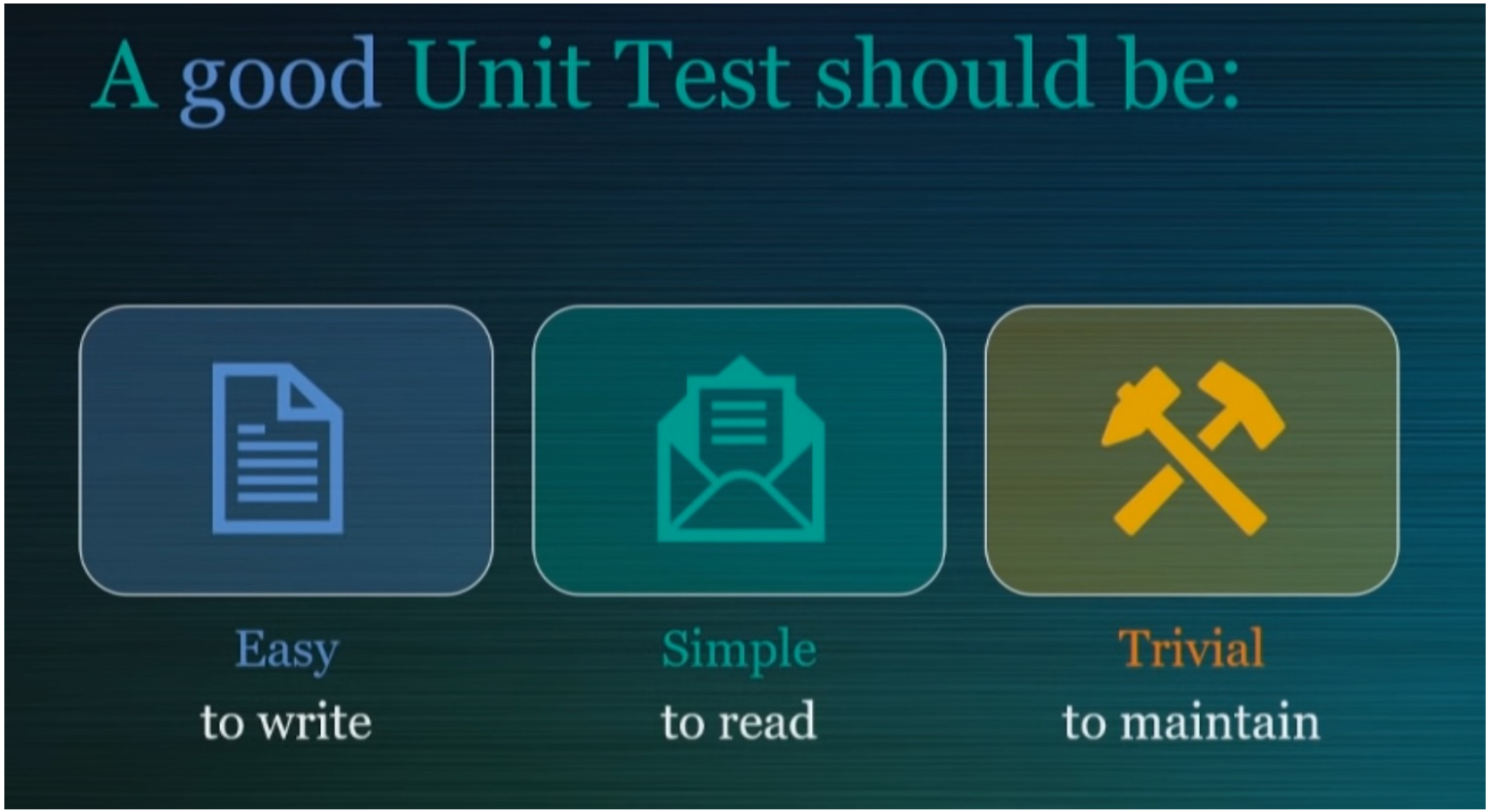
F – Fast, Focussed

A - Automated

I - Isolated

R – Reliable, Repeatable

What makes a good unit test?



https://www.youtube.com/watch?v=V6MzzmJ9IPs -- The secret unit testing tools no one ever told you about - Dror Helper

Unit Test Features

- Quality of individual components has a major influence on the quality of the overall system.
- Unit tests = quality assurance at the level of small component helping to increase the internal quality.

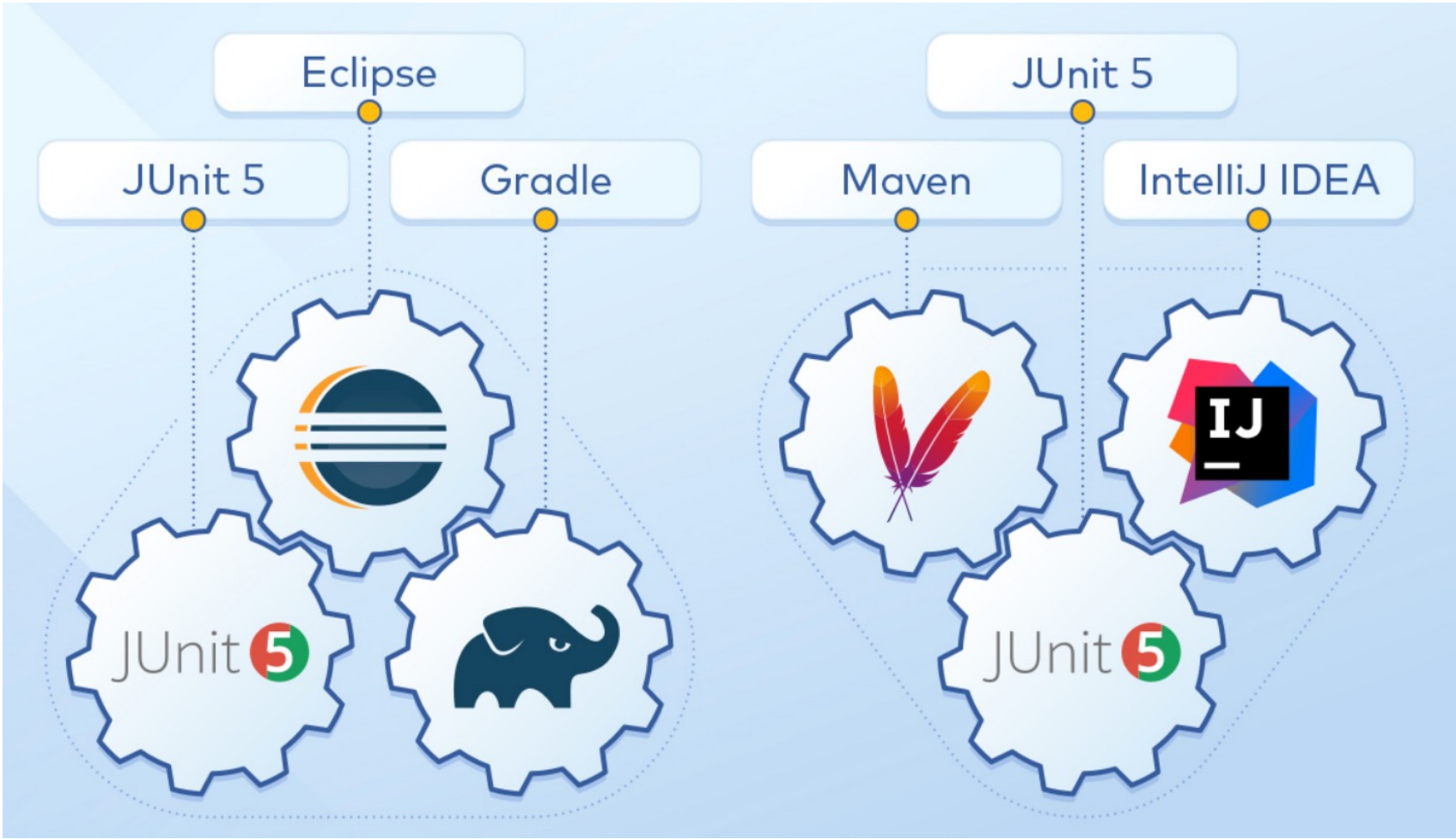
- The following properties of unit tests help with this:

  - **Clear result** - pass (green) or fail (red).
  - **Measurable** - The number of test cases and the program parts tested with them (test coverage) can be easily evaluated.
  - **Implementation-oriented and focused** - If unit tests are created and executed in parallel with the implementation, feedback is obtained quickly and errors can often be corrected easily.
  - **Gaining know-how** - You automatically become more intensively involved with the requirements and the existing implementations. This continuously builds up a good understanding.
  - **Repeatable** - After changes in the source code, tests can be executed again. With positive execution this increases the security8 to have inserted no defects.

- The above points make clear: You benefit the most from well formulated unit tests, if they are executed regularly.

Tests should be executed regularly

- mvn clean test
- gradle clean test
- Eclipse (MoreUnit): Ctrl + R
- IntelliJ: Ctrl + Shift + R



https://www.toptal.com/java/getting-started-with-junit

Unit Test Features: API Design & Documentation

- When writing unit tests, **design decisions** such as those regarding **cohesion, coupling and the design** of the API also play a role.
- By implementing test cases, you use the API of your own classes, which makes it easier to **judge whether the interfaces provided are useful and manageable**.
- Unit tests can thus **uncover possible weaknesses** in the APIs addressed by the tests before they are used in other components **and ensure more successful APIs**.
- **Unit tests as documentation of the expected program behavior**
- Documentation is **automatically always up to date**, otherwise the test cases would fail.

Exercises Part 1

https://github.com/Michaeli71/JUnit5_Workshop

PART 2:
JUnit 5 Intro

JUnit 5

JUnit 5

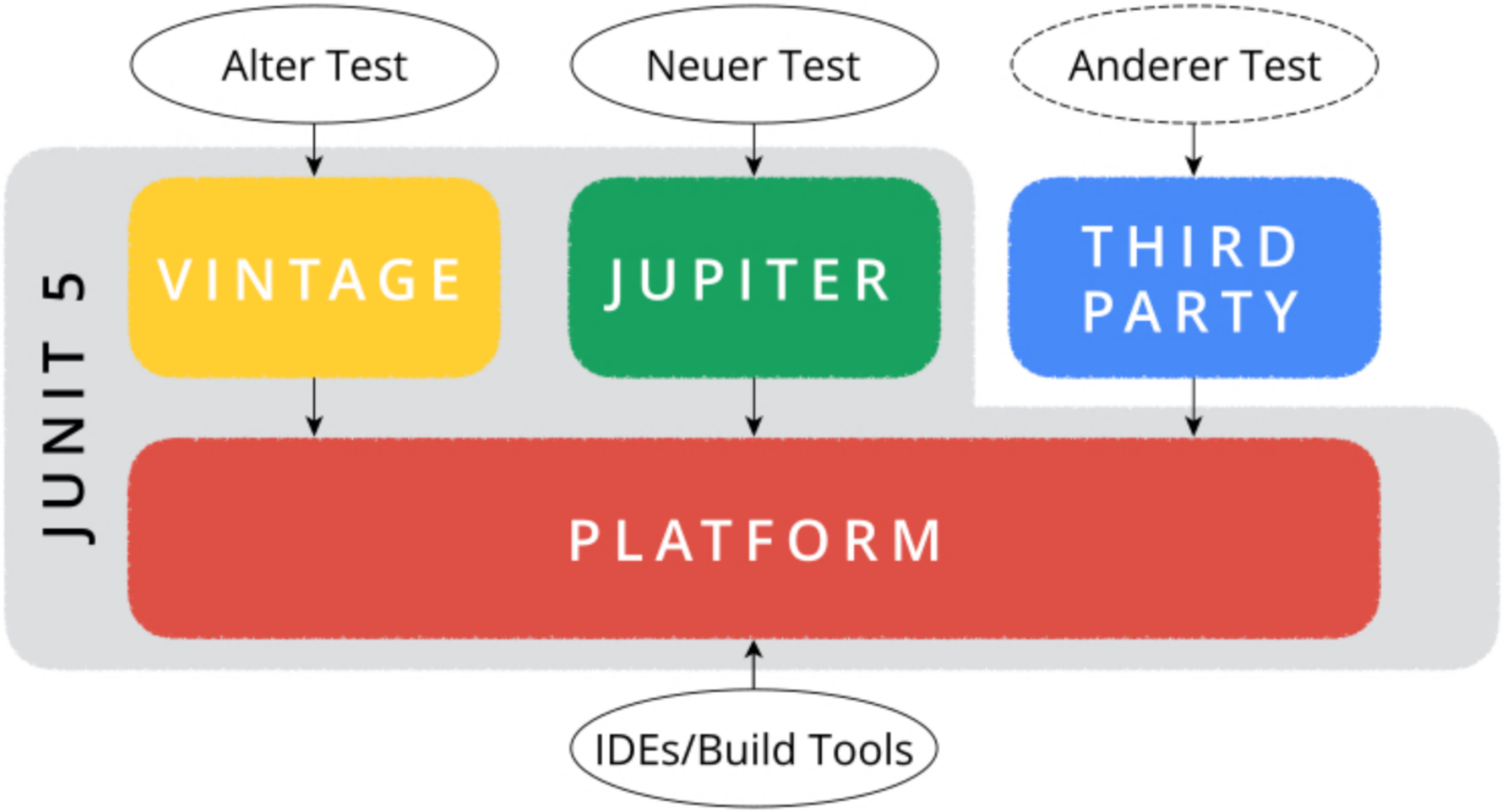
JUnit 4

The new major version of the programmer-friendly testing framework for Java

User Guide | Javadoc | Code & Issues | Q & A | Support JUnit

Architecture

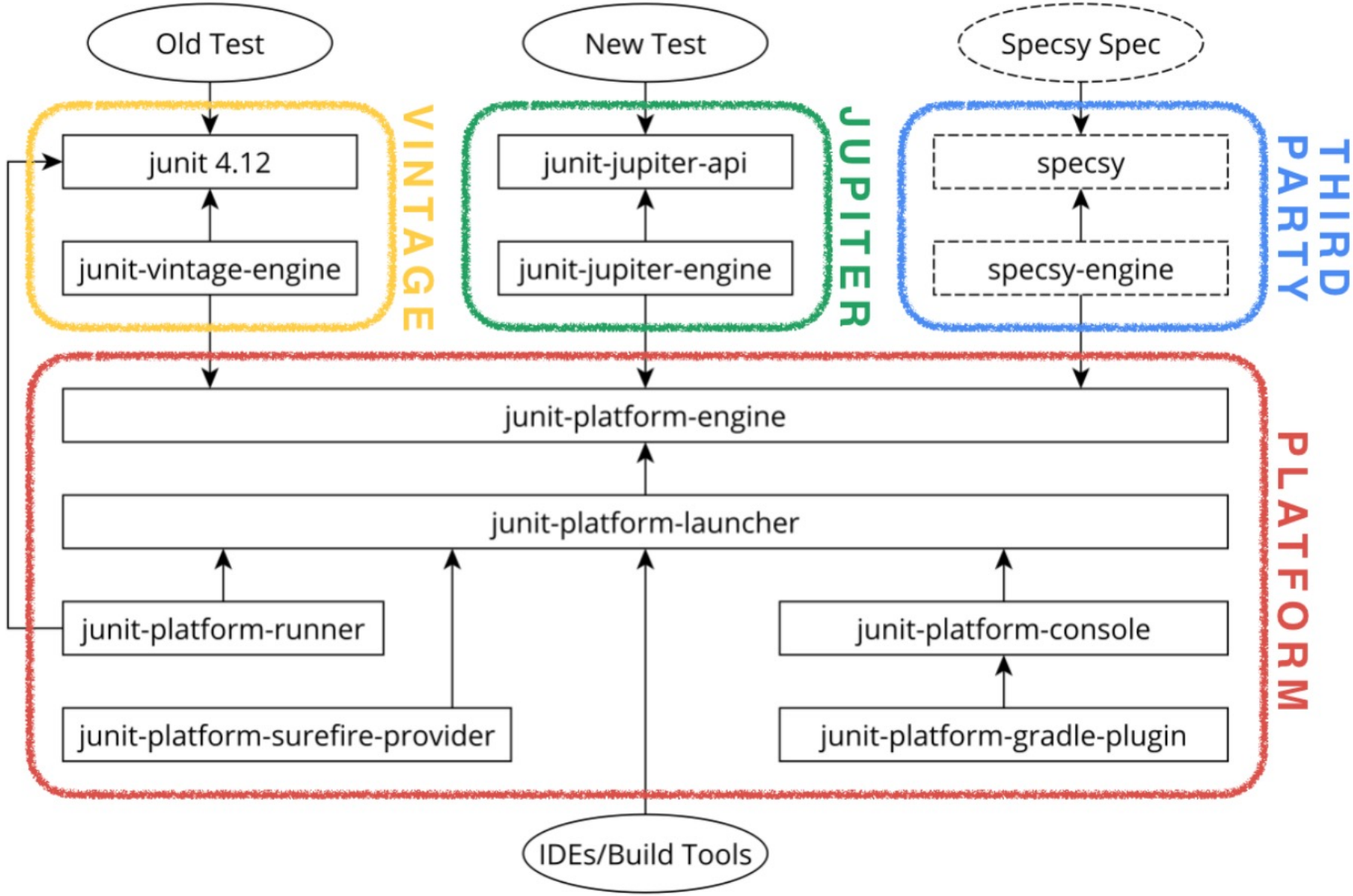
- JUnit 5 =
  JUnit Platform +
  JUnit Jupiter +
  JUnit Vintage



https://jaxenter.de/junit-5-beyond-testing-framework-81787

Architecture

- JUnit 5 =
  JUnit Platform +
  JUnit Jupiter +
  JUnit Vintage



https://jaxenter.de/highlights-junit-5-65986

Assertions - check conditions

- Evaluation of conditions - The `Assert` (JUnit4) / `Assertions` (JUnit 5) class provides a set of verification methods that can be used to express conditions and thereby verify assertions about the source code under test:
  - **`assertEquals()`** – check two objects for equality of content (call **`equals(Object)`**) or two variables of primitive type for equality*
  - **`assertTrue()`** and **`assertFalse()`** – check boolean conditions
  - **`assertNull()`** and **`assertNotNull()`** – check object references for == null and != null respectively
  - **`assertSame()`** and. **`assertNotSame()`** check object references for == or !=
  - **`fail()`** – deliberately make a test case fail

*) pay attention for floating point: `float` and `double`

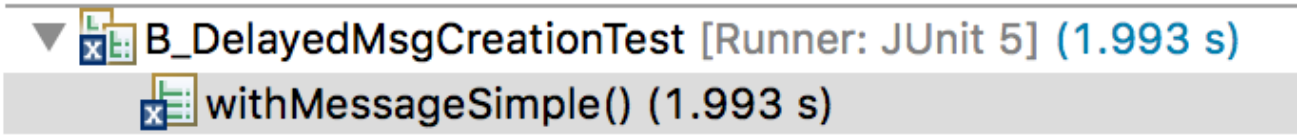
A second unit test with JUnit 5

- Test cases are written in the form of special test methods marked with the annotation @Test

```
@Test
void assertMethodsInAction()
{
    String expected = "Tim";
    String actual = "Tim";
    assertEquals(expected, actual);
    assertEquals(expected, "XYZ", "Hint if wrong");

    assertTrue(true);
    assertTrue(true, "Always true");

    assertFalse(false);
    assertNull(null);
    assertNotNull(new Object());
    assertSame(null, null);
    assertNotSame(null, new Object());
}
```

Complex message creation

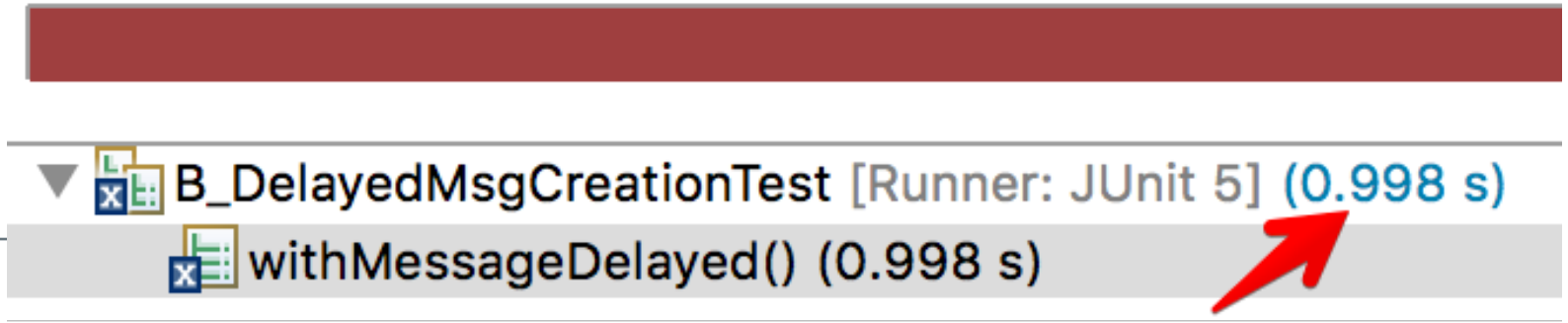
```
@Test
void withMessageSimple()
{
    String expected = "Tim";

    assertEquals(expected, "Tim", complicatedCalculation("Hint"));
    assertEquals(expected, "ALWAYS", complicatedCalculation("Hint"));
}

private String complicatedCalculation(String info)
{
    try
    {
        Thread.sleep(1_000);
    }
    catch (InterruptedException ignored) { }
    return info + info;
}
```



What is unattractive about it?

Complex message creation (just on demand)

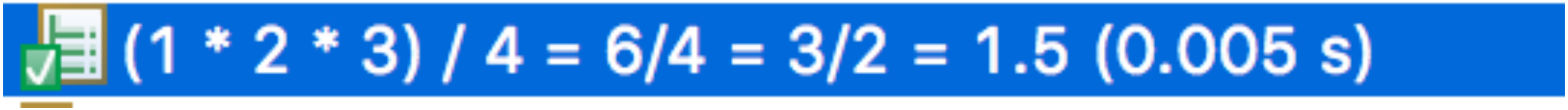
```
@Test
void withMessageDelayed()
{
    String expected = "Tim";

    // complicated msg is only calculated if comparison fails
    assertEquals(expected, "Tim", () -> complicatedCalculation("Hint"));
    assertEquals(expected, "ALWAYS", () -> complicatedCalculation("Hint"));
}
```



Special test names

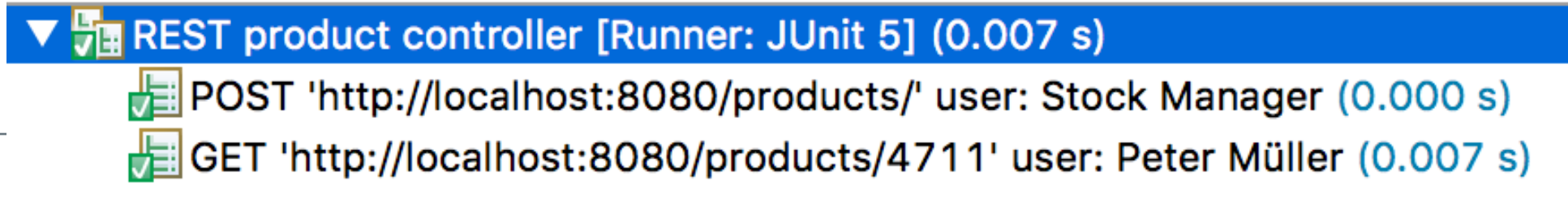
- With JUnit 4 one was limited to valid method names
- Special test names are now possible using the annotation **@DisplayName**

```
@Test
@DisplayName("(1 * 2 * 3) / 4 = 6/4 = 3/2 = 1.5")
void divideResultOfMultiplication()
{
    BigDecimal newValue = BigDecimal.ONE.multiply(BigDecimal.valueOf(2)).
                                         multiply(BigDecimal.valueOf(3)).
                                         divide(BigDecimal.valueOf(4));

    assertEquals(new BigDecimal("1.5"), newValue);
}
```



Special test names

```
@DisplayName("REST product controller")
public class C_DisplayNameDemo
{
    @Test
    @DisplayName("GET 'http://localhost:8080/products/4711' user: Peter Müller")
    public void getProductFor4711()
    {
        // ..
    }

    @Test
    @DisplayName("POST 'http://localhost:8080/products/' user: Stock Manager")
    public void addProductAsStockManager()
    {
        // ...
    }
}
```



Special classification

- Special classificationen with `@Tag`

```
@Test
@DisplayName("(1 * 2 * 3) / 4 = 6/4 = 3/2 = 1.5")
@Tag("multiplication")
@Tag("division")
void divideResultOfMultiplication()
{
    BigDecimal newValue = BigDecimal.ONE.multiply(BigDecimal.valueOf(2)).
                                         multiply(BigDecimal.valueOf(3)).
                                         divide(BigDecimal.valueOf(4));

    assertEquals(new BigDecimal("1.5"), newValue);
}
```

Info about context with `TestInfo`

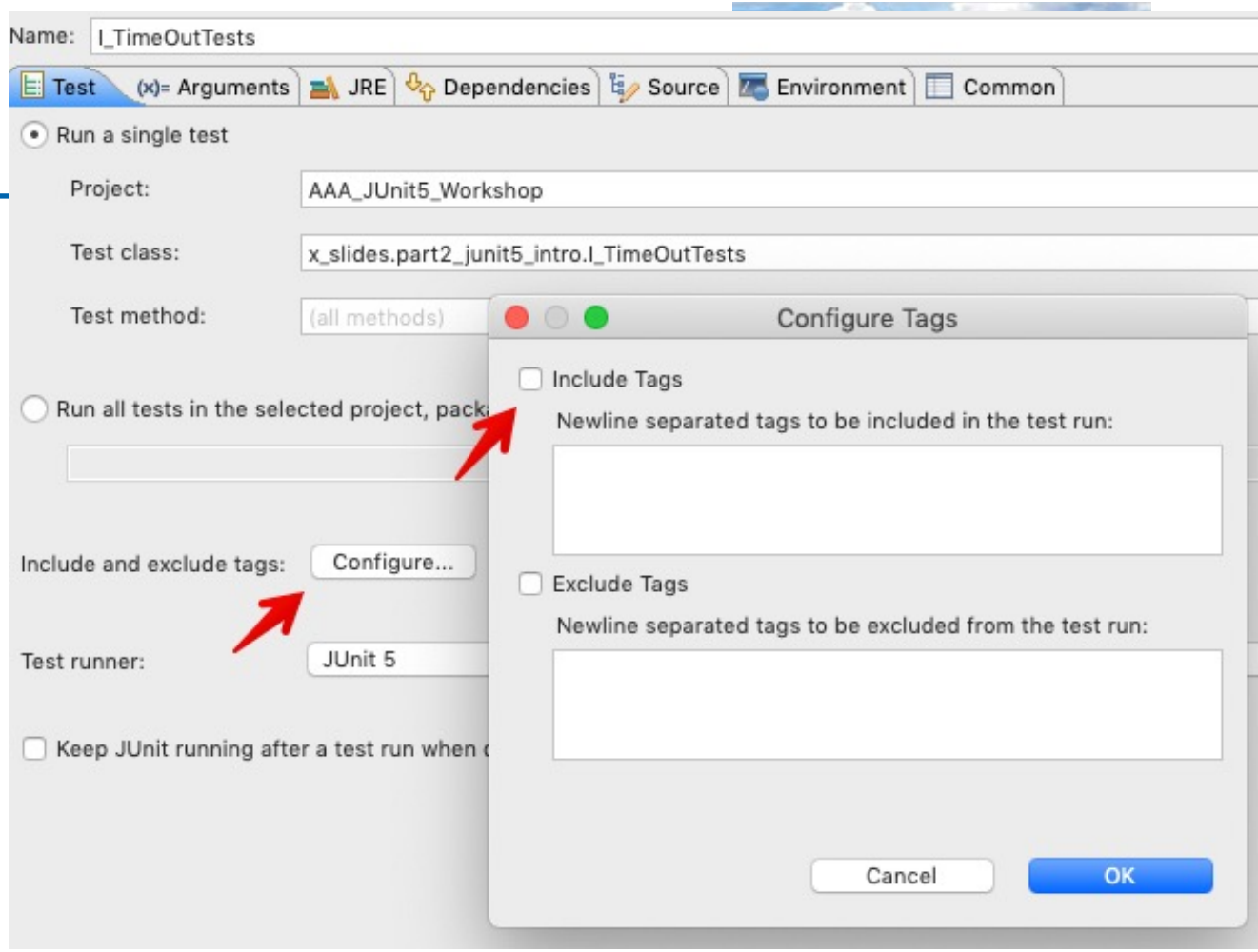
- TestInfo serves as a replacement for the JUnit 4 rule TestName.
- In addition: parameters in test methods are possible!

```
@Test
void simpleTestInfo(TestInfo ti)
{
    assertEquals("simpleTestInfo", ti.getTestMethod().get().getName());
}

@Test
@DisplayName("DEMO-TAGS")
@Tag("FAST")
@Tag("COOL")
void moreTestInfo(TestInfo ti)
{
    assertEquals("DEMO-TAGS", ti.getDisplayName());
    assertEquals(Set.of("FAST", "COOL"), ti.getTags());
}
```

Tag based execution

- **Gradle**

```
useJUnitPlatform() {
    includeTags 'FAST', 'COOL', 'unit'
    excludeTags 'regression', 'bug-4711'
}

gradle clean test \
       -DincludeTags='FAST, COOL, unit' \
       -DexcludeTags='regression'
```



- **Maven**

```
<configuration>
    <argLine>@{argLine} --enable-preview</argLine>
    <!-- <argLine>enable-preview</argLine> -->
        <testFailureIgnore>true</testFailureIgnore>
       <!-- include tags -->
        <groups>FAST, unit</groups>
        <!-- exclude tags -->
        <excludedGroups>slow</excludedGroups>
</configuration>
```

Special Assertions

Multiple Asserts

```
@Test
void multipleAssertsforOneTopic()
{
    final Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

    // JUnit 4
    assertEquals("Mike", mike.name);
    assertEquals(LocalDate.of(1971, 2, 7), mike.dateOfBirth);
    assertEquals("Zürich", mike.homeTown);

    // JUnit 5
    assertAll(() -> assertEquals("Mike", mike.name),
              () -> assertEquals(LocalDate.of(1971, 2, 7), mike.dateOfBirth),
              () -> assertEquals("Zürich", mike.homeTown));
}
```

Where is the difference?

Multiple Asserts

```
@Test
void multipleAssertsforOneTopic_Diff1() {

    Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

    assertEquals("Tim", mike.name);
    assertEquals(LocalDate.of(1971, 3, 27), mike.dateOfBirth);
    assertEquals("Kiel", mike.homeTown);
}

@Test
void multipleAssertsforOneTopic_Diff2() {

    Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

    assertAll(() -> assertEquals("Tim", mike.name),
              () -> assertEquals(LocalDate.of(1971, 3, 27), mike.dateOfBirth),
              () -> assertEquals("Kiel", mike.homeTown));
}
```

Multiple Asserts

```java
@Test
void multipleAssertsforOneTopic_Diff1() {

    Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

    assertEquals("Tim", mike.name);
    assertEquals(LocalDate.of(1971, 3, 27), mike.dateOfBirth);
    assertEquals("Kiel", mike.homeTown);
}
```

```
org.opentest4j.AssertionFailedError: expected: <Tim> but was: <Mike>
    at a_first_slides.DisplayNameExample.multipleAssertsforOneTopic_Diff1(D
```

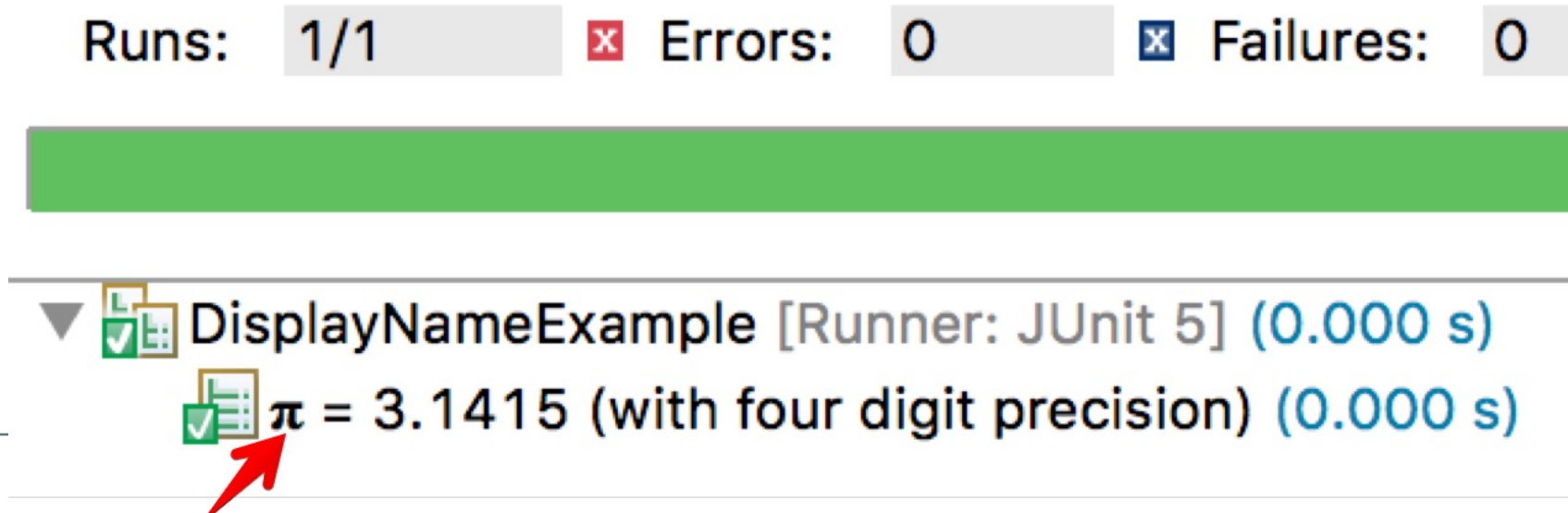
```java
@Test
void multipleAssertsforOneTopic_Diff2() {

    Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

    assertAll(() -> assertEquals("Tim", mike.name),
              () -> assertEquals(LocalDate.of(1971, 3, 27), mike.dateOfBirth),
              () -> assertEquals("Kiel", mike.homeTown));
}
```

```
org.opentest4j.MultipleFailuresError: Multiple Failures (3 failures)
expected: <Tim> but was: <Mike>
expected: <1971-03-27> but was: <1971-02-07>
expected: <Kiel> but was: <Zürich>
```

How do we test floating point numbers?

Special handling for floating point numbers

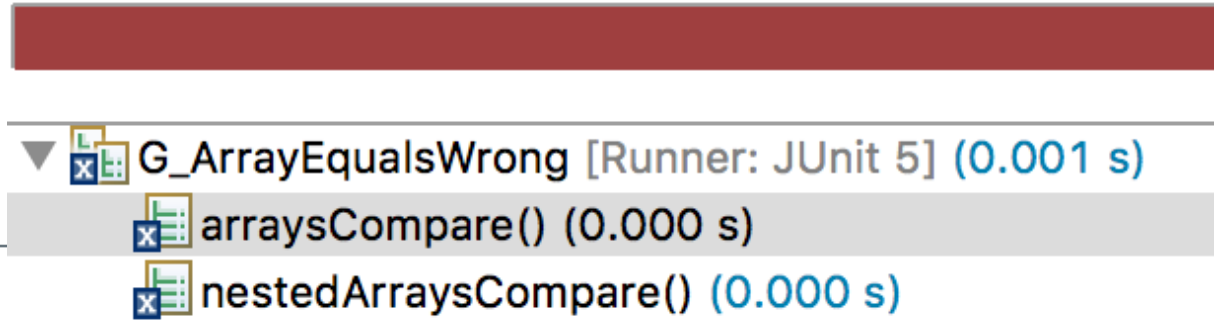
```
@Test
@DisplayName("\uD835\uDED1 = 3.1415 (with four digit precision)")
void floatingArithemticRoundingForPI()
{
    double value = calculatePI();
    double precision = 0.0001;

    assertEquals(3.1415, value, precision);
}

private double calculatePI()
{
    return Math.PI;
}
```



How do we test arrays?

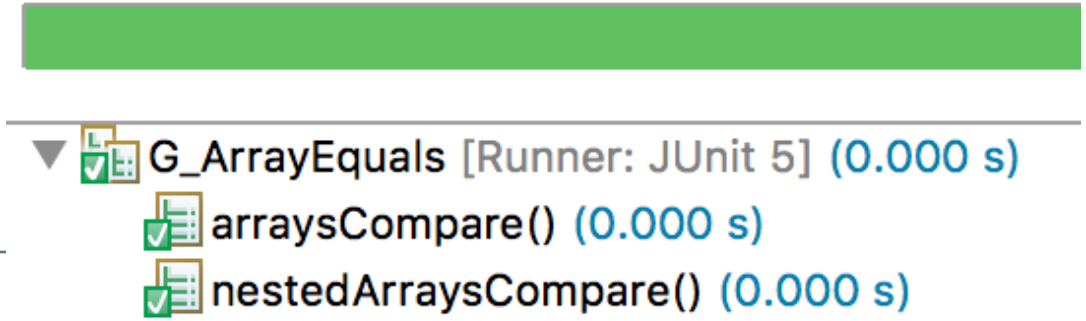
Compare arrays - Oops!

```
@Test
void arraysCompare()
{
    final String[] words = { "Word1", "Word2" };
    final String[] expected = { "Word1", "Word2" };

    assertEquals(expected, words);
}

@Test
void nestedArraysCompare()
{
    final String[][] nested = { { "Line1", "Word1" },
                                { "Line2", "Word2" } };
    final String[][] expected = { { "Line1", "Word1" },
                                  { "Line2", "Word2" } };

    assertEquals(expected, nested);
}
```



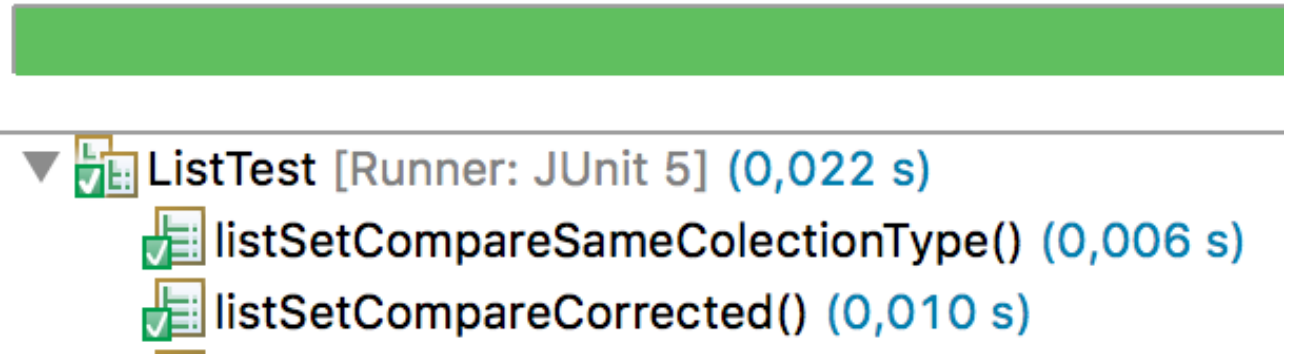
Compare arrays CORRECTLY

```
@Test
void arraysCompare()
{
    final String[] words = { "Word1", "Word2" };
    final String[] expected = { "Word1", "Word2" };

    assertArrayEquals(expected, words);
}

@Test
void nestedArraysCompare()
{
    final String[][] nested = { { "Line1", "Word1" },
                                { "Line2", "Word2" } };
    final String[][] expected = { { "Line1", "Word1" },
                                  { "Line2", "Word2" } };

    assertArrayEquals(expected, nested);
}
```



Collections vergleichen – problemlos?!

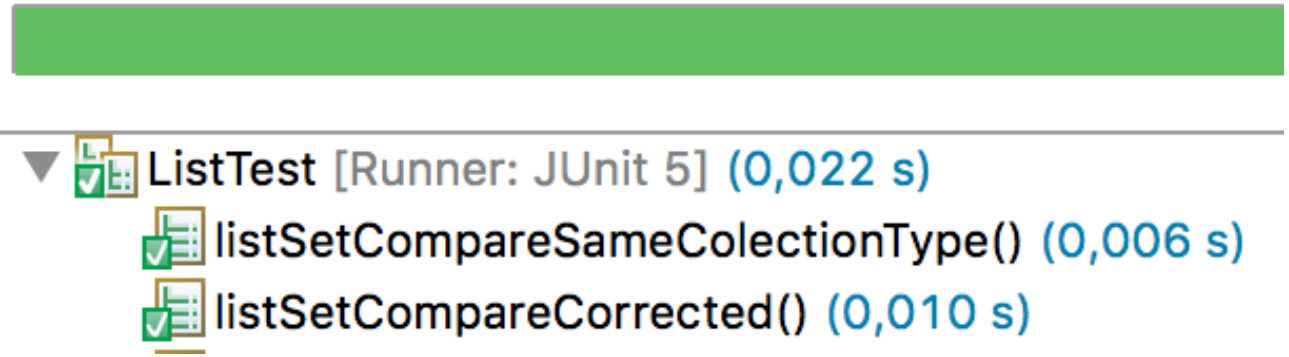
```java
@Test
void listSetCompareSameColectionType()
{
    final Collection<String> tags = new HashSet<>(Set.of("Fast", "Cool"));
    final Collection<String> names = List.of("Tim", "Mike", "Tom");

    assertEquals(Set.of("Fast", "Cool"), tags);
    assertEquals(List.of("Tim", "Mike", "Tom"), names);
}
```



How do we test Collections?

Compare Collections - no problem?!

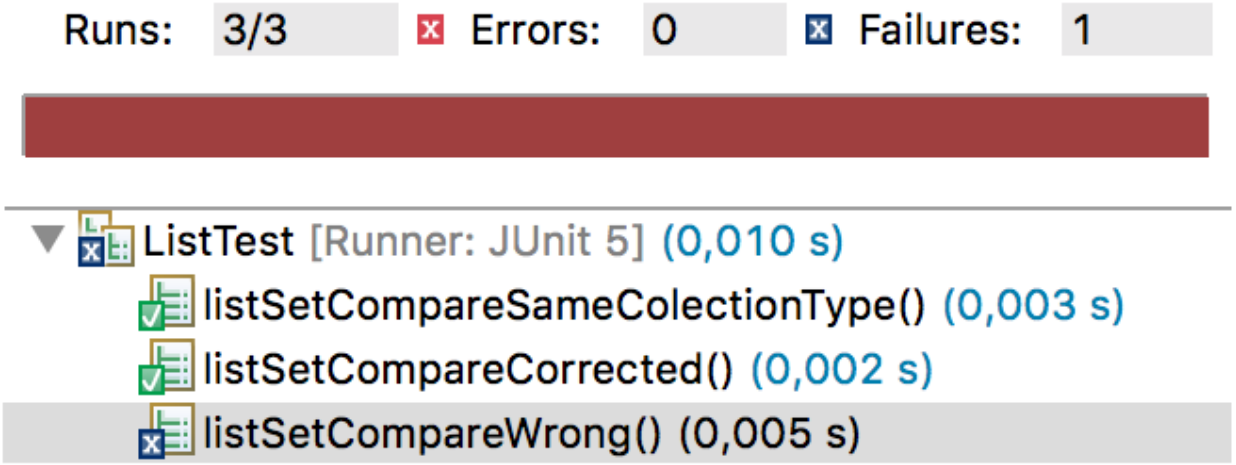
```java
@Test
void listSetCompareSameColectionType()
{
    final Collection<String> tags = new HashSet<>(Set.of("Fast", "Cool"));
    final Collection<String> names = List.of("Tim", "Mike", "Tom");

    assertEquals(Set.of("Fast", "Cool"), tags);
    assertEquals(List.of("Tim", "Mike", "Tom"), names);
}
```



Compare Collections - What to do in case of different type

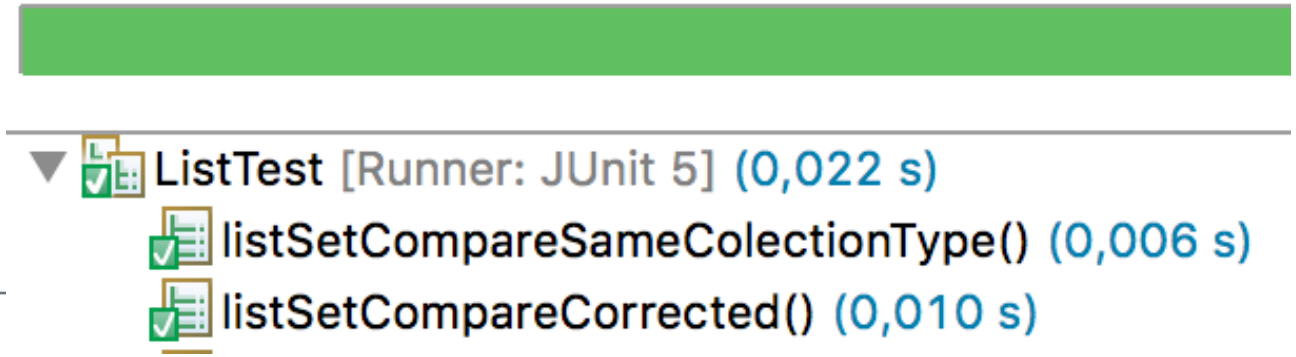
```java
@Test
public void listSetCompareWrong()
{
    final List<String> actual = List.of("a", "b", "c", "d");
    final Set<String> expected = new TreeSet<>(Set.of("c", "a", "d", "b"));

    // compare set and list
    assertEquals(expected, actual);
}
```

org.opentest4j.AssertionFailedError: expected: java.util.TreeSet@3b07a0d6<[a, b, c, d]> but was: java.util.ImmutableCollections$ListN@11a9e7c8<[a, b, c, d]>



Compare Collections SAFER

```java
@Test
public void listSetCompareCorrected()
{
    final List<String> actual = List.of("a", "b", "c", "d");
    final Set<String> expected = new TreeSet<>(Set.of("c", "a", "d", "b"));

    // compare set and list based on Iterable
    assertIterableEquals(expected, expected);
}
```



Testing Exceptions

Manual work

- Sometimes test cases should check the occurrence of exceptions

```
try
{
    actionsThrowingAnException();
    fail();                              // Sollte hier nicht hinkommen
}
catch (final ExpectedException e)
{
    assertTrue(true);                    // Erwarteter Fall
}
```

Checking Exceptions

- Since JUnit 4, an exception expected during test execution can be specified as a parameter named expected in the annotation @Test:

```
@Test(expected = java.lang.NumberFormatException.class)
public void testFailWithExceptionJUnit4()
{
    // Hier wird bewusst ein Fehler provoziert
    final int value = Integer.parseInt("Fehler simulieren!");
    fail("calculation should throw an exception!");
}
```

- Problematic: If you want to evaluate the message text.
- Also no AAA style

JUnit Rules: Checking Exceptions

- JUnit Rule **`ExpectedException`**

```
@Rule
public ExpectedException thrown = ExpectedException.none();


@Test
public void testSomething()
{
   thrown.expect(IllegalStateException.class);
   thrown.expectMessage("XYZ is not initialized");

   doSomethingCausingAnExcpetion();
 }
```

JUnit 5: assertThrows()

```
@Test
void cannotSetValueToNull()
{
    assertThrows(NullPointerException.class,
                 () -> new BigDecimal((String) null));
}
```

```
@Test
void assertThrowsException()
{
    assertThrows(IllegalArgumentException.class,
                 () -> { Integer.valueOf(null); });
}
```

JUnit 5: assertThrows() with return

```
@Test
void shouldThrowExceptionAndInspectMessage()
{
    UnsupportedOperationException exception =
                assertThrows(UnsupportedOperationException.class,
    () ->
    {
        throw new UnsupportedOperationException("Not supported");
    });

    assertEquals(exception.getMessage(), "Not supported");
}
```

JUnit 5: assertThrows() with return

```
@Test
@DisplayName("Exception test clearer")
void exceptionTestImproved()
{
    Executable executable = () -> {
        throw new UnsupportedOperationException("Not supported");
    };

    UnsupportedOperationException exception =
              assertThrows(UnsupportedOperationException.class, executable);

    assertEquals(exception.getMessage(), "Not supported");
}
```

Timeout Assertions

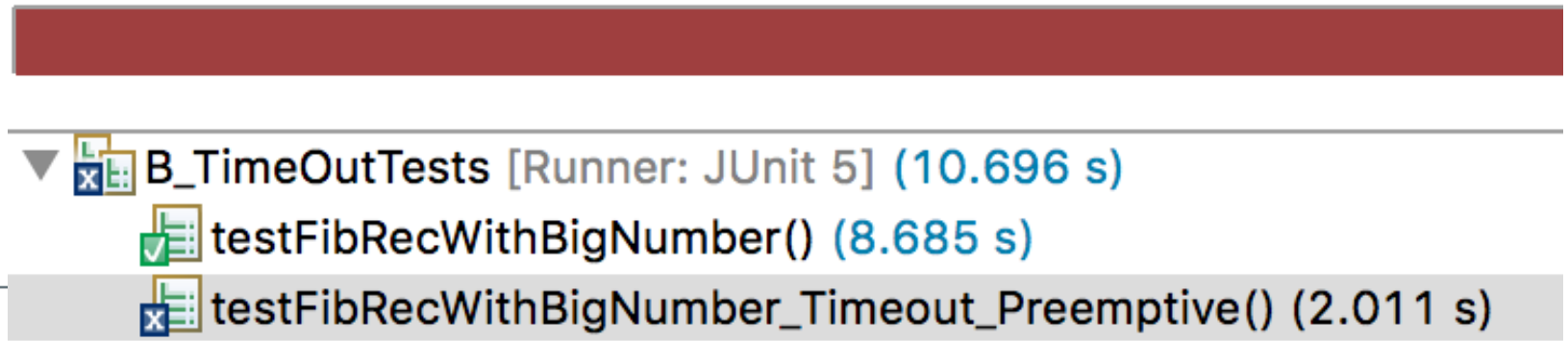
JUnit Rules: Checking for Time-outs (JUnit 4)

- Sometimes a test should last only for a maximum time => time-out

```
import org.junit.rules.Timeout;

public class TimeoutRuleTest
{
    @Rule
    public Timeout timeout = new Timeout(500, TimeUnit.MILLISECONDS);

    @Test
    public void longRunningAction() throws InterruptedException
    {
        for (int i=0; i < 20; i++)
        {
            TimeUnit.SECONDS.sleep(1);
        }
    }

    @Test
    public void loopForever() throws InterruptedException
    {
        for (;;)
        {
            TimeUnit.SECONDS.sleep(1);
        }
    }
}
```

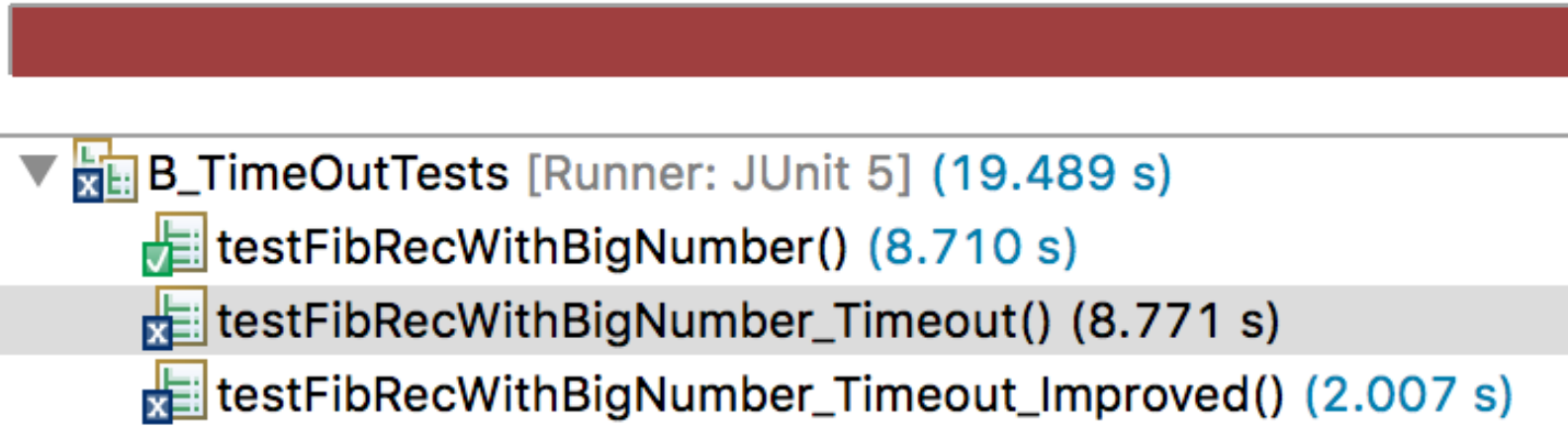
JUnit Rules: Checking for Time-outs (JUnit 5)

```
@Test
void testFibRecWithBigNumber()
{
    assertEquals(2971215073L, FibonacciCalculator.fibRec(47));
}

@Test
void testFibRecWithBigNumber_Timeout_Preemptive()
{
    assertTimeoutPreemptively(Duration.ofSeconds(2),
                              () -> FibonacciCalculator.fibRec(47));
}
```



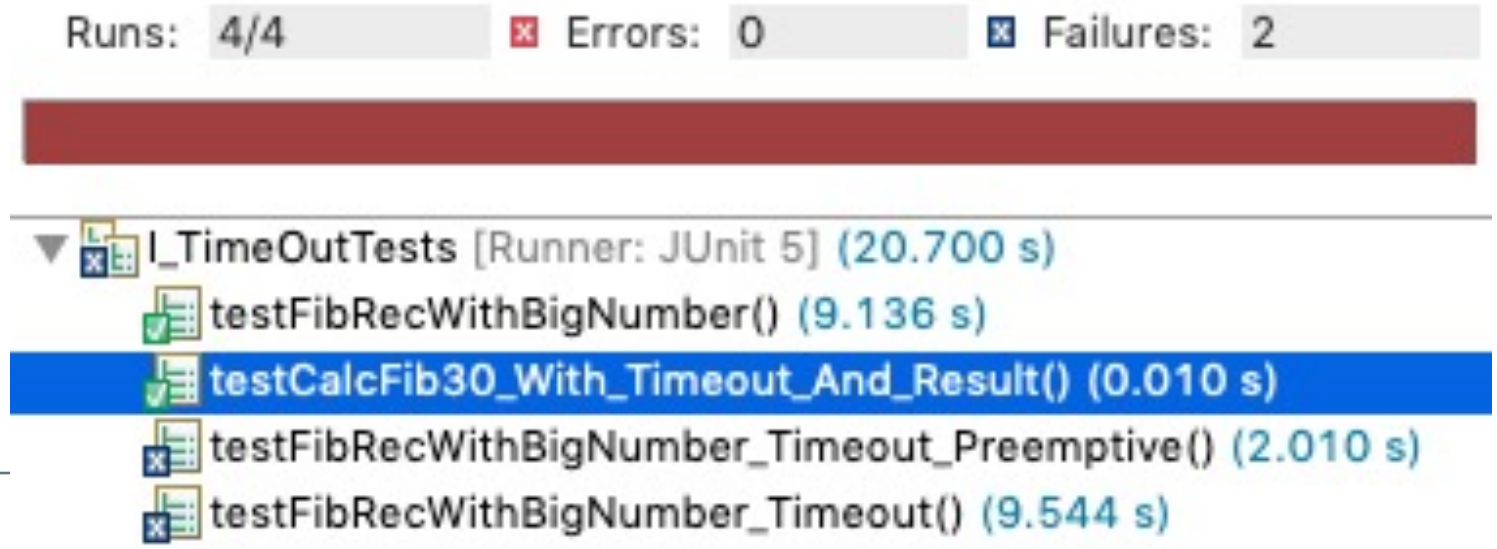
JUnit Rules: Checking for Time-outs (JUnit 5)

```
@Test
void testFibRecWithBigNumber_Timeout()
{
    assertTimeout(Duration.ofSeconds(2),
                  () -> FibonacciCalculator.fibRec(47));
}

@Test
void testFibRecWithBigNumber_Timeout_Improved()
{
    assertTimeoutPreemptively(Duration.ofSeconds(2),
                              () -> FibonacciCalculator.fibRec(47));
}
```



JUnit Rules: Checking for Time-outs (JUnit 5)

```
@Test
void testCalcFib30_With_Timeout_And_Result()
{
    // Provide result of calculation if with in time out
    ThrowingSupplier<Long> action = () -> FibonacciCalculator.fibRec(30);

    Long value = assertTimeoutPreemptively(Duration.ofSeconds(1),
                                           action);

    assertEquals(832040, value);
}
```



Disable a JUnit Test (temporarily)

```
@Test
@Disabled
void testFibRecWithBigNumber_Timeout()
{
    assertTimeout(Duration.ofSeconds(2),
                  () -> FibonacciCalculator.fibRec(47));
}
```

Life Cycle

JUnit 5 Life Cycle

```java
@BeforeAll
static void setupServer()
{
    System.out.println("setupServer");
}

@BeforeEach
void prepareDatabase()
{
    System.out.println("prepareDatabase");
}

@Test
void simpleTest()
{
    System.out.println("simpleTest");
}

@Test
void anotherTest()
{
    System.out.println("anotherTest");
}
```

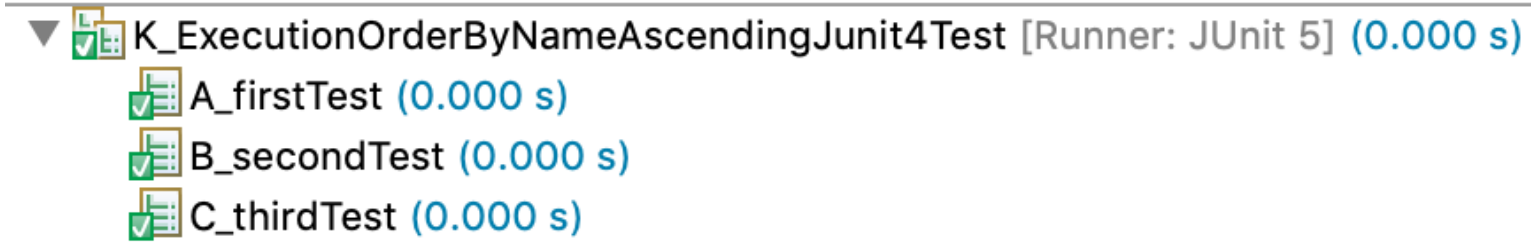
```java
@AfterEach
void cleanupDatabase()
{
    System.out.println("cleanupDatabase");
}

@AfterAll
static void tearDownServer()
{
    System.out.println("tearDownServer");
}
```

```
setupServer
prepareDatabase
simpleTest
cleanupDatabase
prepareDatabase
anotherTest
cleanupDatabase
tearDownServer
```

Execution order

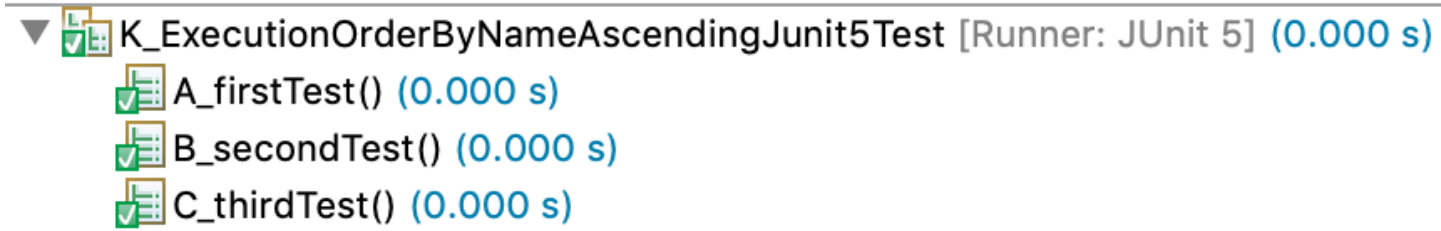
JUnit 4 Ordering of Test Methods (@FixMethodOrder)

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class K_ExecutionOrderByNameAscendingJUnit4Test
{
    private final static StringBuilder result = new StringBuilder("");

    @Test
    public void B_secondTest() {
        result.append("b");
    }

    @Test
    public void C_thirdTest() {
        result.append("c");
    }

    @Test
    public void A_firstTest() {
        result.append("a");
    }

    @AfterClass
    public static void assertOutput()
    {
        assertEquals("abc", result.toString());
    }
}
```



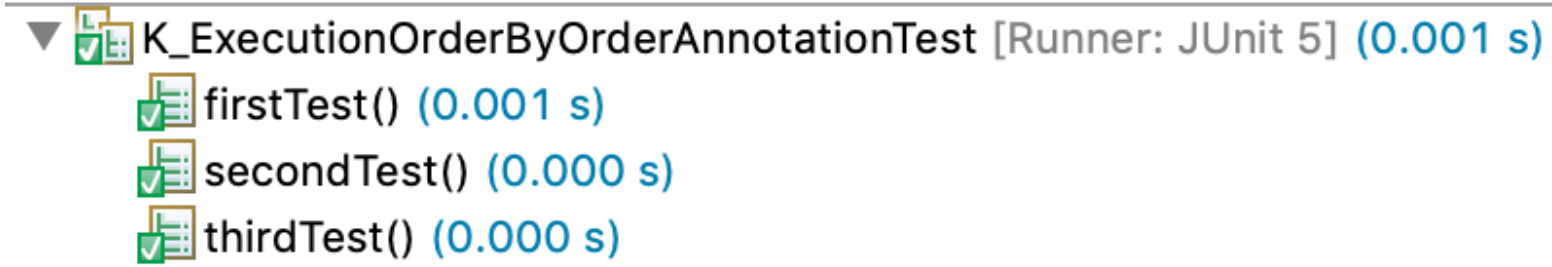
JUnit 5 Ordering of Test Methods (@TestMethodOrder)

```
@TestMethodOrder(MethodName.class)
public class K_ExecutionOrderByNameAscendingJUnit5Test
{
    private final static StringBuilder result = new StringBuilder("");

    @Test
    public void B_secondTest() {
        result.append("b");
    }

    @Test
    public void C_thirdTest() {
        result.append("c");
    }

    @Test
    public void A_firstTest() {
        result.append("a");
    }

    @AfterAll
    public static void assertOutput()
    {
        assertEquals("abc", result.toString());
    }
}
```



JUnit 5 Ordering of Test Methods (@TestMethodOrder)

```
@TestMethodOrder(OrderAnnotation.class)
public class K_ExecutionOrderByOrderAnnotationTest
{
    private static final StringBuilder output = new StringBuilder("");

    @Test
    @Order(2)
    public void secondTest() {
        output.append("b");
    }

    @Test
    @Order(3)
    public void thirdTest() {
        output.append("c");
    }

    @Test
    @Order(1)
    public void firstTest() {
        output.append("a");
    }

    @AfterAll
    public static void assertOutput()
    {
        assertEquals("abc", output.toString());
    }
```

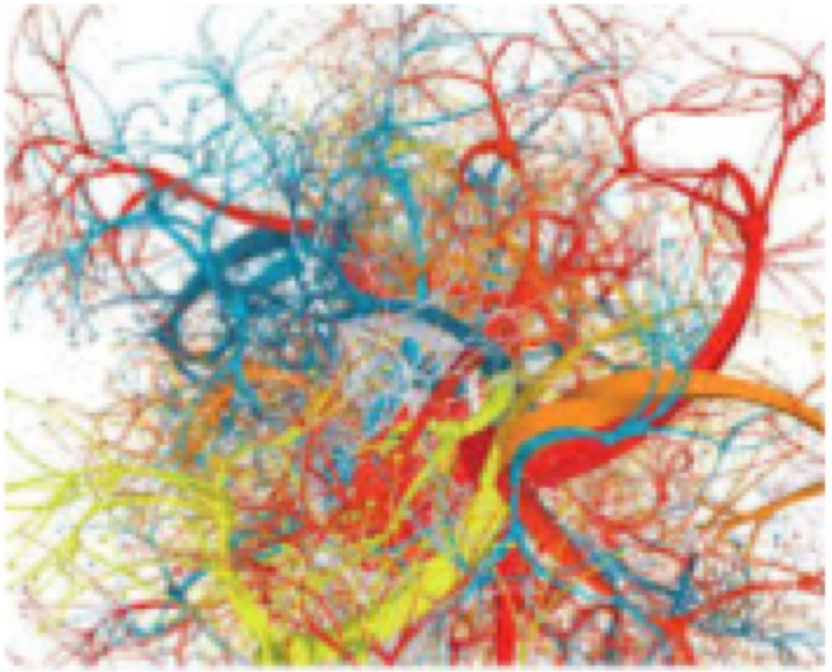


Exercises Part 2

https://github.com/Michaeli71/AST-JUnit5

PART 3
JUnit 5 Advanced

Combinatorics / Complexity



Complexity

3 key questions

1. Which value assignments should one test?

2. How do I avoid too much effort?

3. How do I find those test cases that allow a good and safe statement about the quality and functionality?

Complexity

- **Which value assignments should be tested?**
  - Even with two int int => $2^{32} * 2^{32} = 2^{64}$ combinations
- **How to avoid too much overhead?**
  - Test the important / complex things
  - Do not test getters / setters
  - Clever choice of inputs so that many variants are tested
- **How do I find those test cases that give a good and confident indication of quality and functionality?**
  - **Equivalence class test**
  - **Boundary test**

Equivalence classes

- Grouping of inputs: Different values => same result
- Typical example: discount calculation

Value range	discount
count < 100	0 %
100 <= count <= 1000	4 %
count > 1000	7 %

- How many and which equivalence classes result?

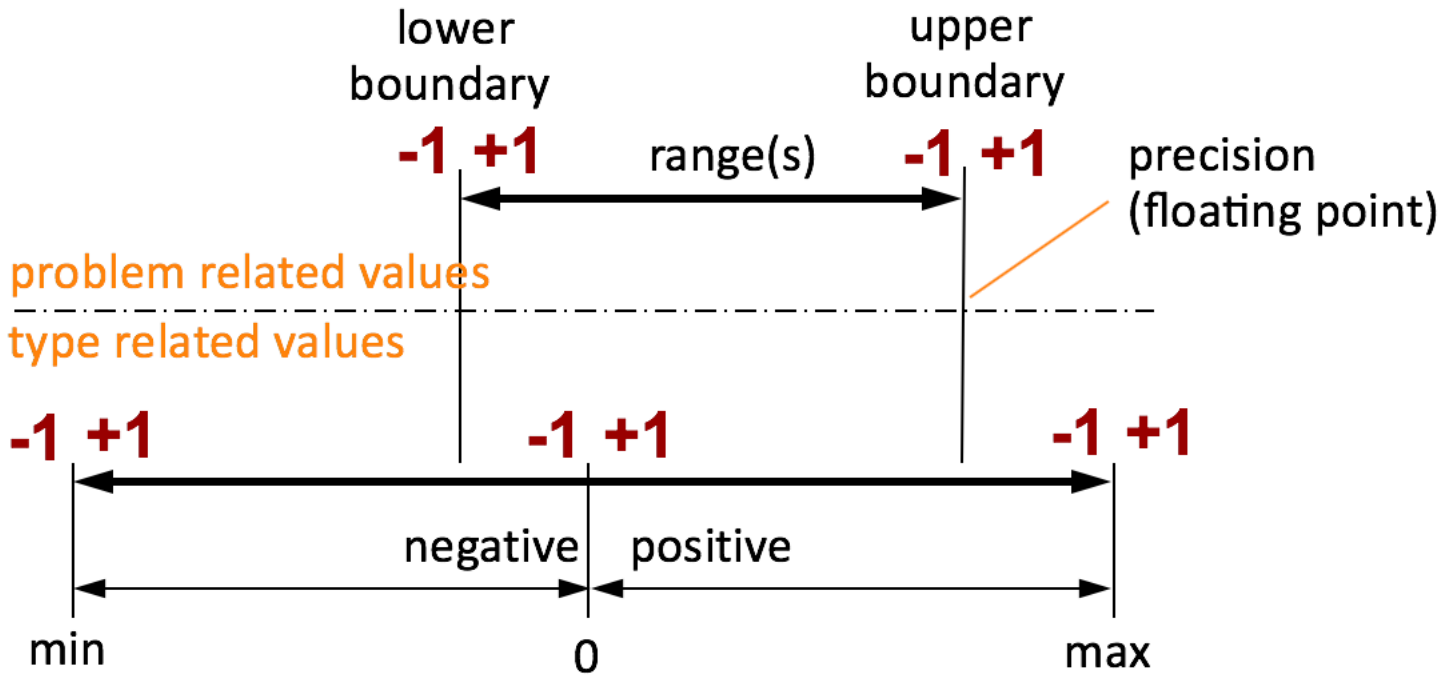
Equivalence class test

- **So let's write 3 test methods. But: Are these tests enough?**

```
@Test
public void testCalcDiscount_SmallOrder_NoDiscount()
{
    final int smallAmount = 20;
    assertEquals(0, calculator.calcDiscount(smallAmount), "no discount");
}

@Test
public void testCalcDiscount_MediumOrder_MediumDiscount()
{
    final int mediumAmount = 200;
    assertEquals(4, calculator.calcDiscount(mediumAmount), "4 % discount");
}

@Test
public void testCalcDiscount_BigOrder_BigDiscount()
{
    final int bigAmount = 2000;
    assertEquals(7, calculator.calcDiscount(bigAmount), "7 % discount");
}
```

Boundary tests

- **NO!** The experience from the practice shows, besides equivalence class tests one needs still others, why?
- Often we still find **problems at the edges**, i.e. in the transition of the value ranges:



Boundary tests

- For the discount calculation we still find problems at the edges, i.e. in the transition of the value ranges, here thus
  - 99, 100, 101
  - 999, 1000, 1001
- Why? Often errors in comparisons with < <= == != >= >
- Other potential candidates are:
  - Values < 0 or
  - Values > than an intended maximum

Are we supposed to write individual methods for all these values?

☹ ☹ ☹

Parameterized Tests

Parameterized Test

- **Remedy by so-called parameterized test**
- **Execute test case with different data again and again with new value assignment**
- **Thus covering all desired combinations to be tested**

- **Realization variants**
  - Manual work: for loop: delivers only successive results
  - JUnit 4 spasmodic, syntactically unattractive
  - JUnit 4 with Expected Exception better, but again some manual work
  - JUnit 5 **finally good**

Parameterized Test: Short look back: for loop

```
@Test
public void testCheckMatchingBracesAllOkay() throws Exception
{
	List<String> inputs = List.of("()", "()[]{}", "[((()[]{}))]");

	for (String current : inputs)
	{
		assertTrue("Checking " + current,
				MatchingBracesChecker.checkMatchingBraces(current));
	}
}

@Test
public void testCheckMatchingBracesAllWrong() throws Exception
{
	for (String current : List.of("(()", "(()}", "(()}", ")()("))
	{
		assertFalse("Checking " + current,
				MatchingBracesChecker.checkMatchingBraces(current));
	}
}
```

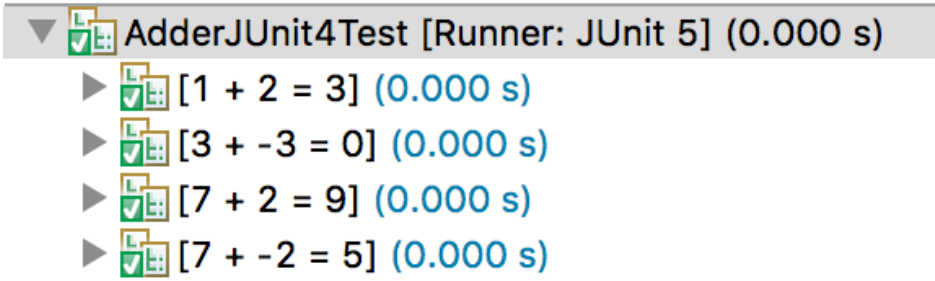
Parameterized Test: Short look back: JUnit 4 Parameterized

- **How do we test the following class Adder with different value combinations?**

```
public class Adder
{
    public int addNumbers(int a, int b)
    {
        return a + b;
    }
}
```

- **So let's write a test class with**
  - @RunWith(Parameterized.class)
  - A constructor and all inputs and Expected
  - A static method to generate the test data.
  - A test method

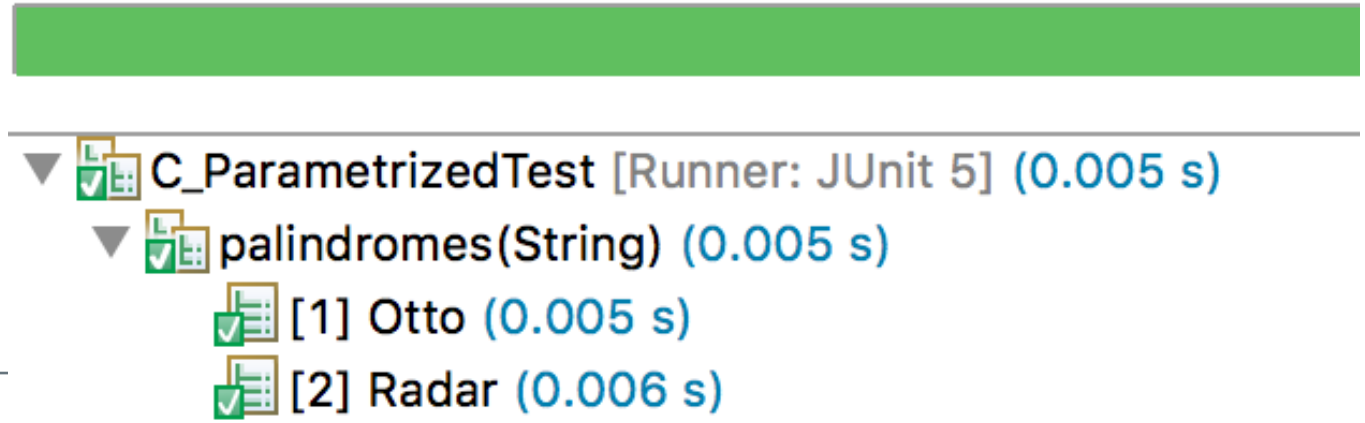
```java
@RunWith(Parameterized.class)
public class AdderJUnit4Test
{
    private int first;
    private int second;
    private int expected;

    public AdderJUnit4Test(int firstNumber, int secondNumber, int expectedResult)
    {
        this.first = firstNumber;
        this.second = secondNumber;
        this.expected = expectedResult;
    }

    @Parameters(name="{0} + {1} = {2}")
    public static Collection<Integer[]> inputAndExpectedNumbers()
    {
        return Arrays.asList(new Integer[][] { { 1, 2, 3 }, { 3, -3, 0 },
                                               { 7, 2, 9 }, { 7, -2, 5 } });
    }

    @Test
    public void sum()
    {
        assertEquals(expected, Adder.add(first, second));
    }
}
```



That can't really be it now?!

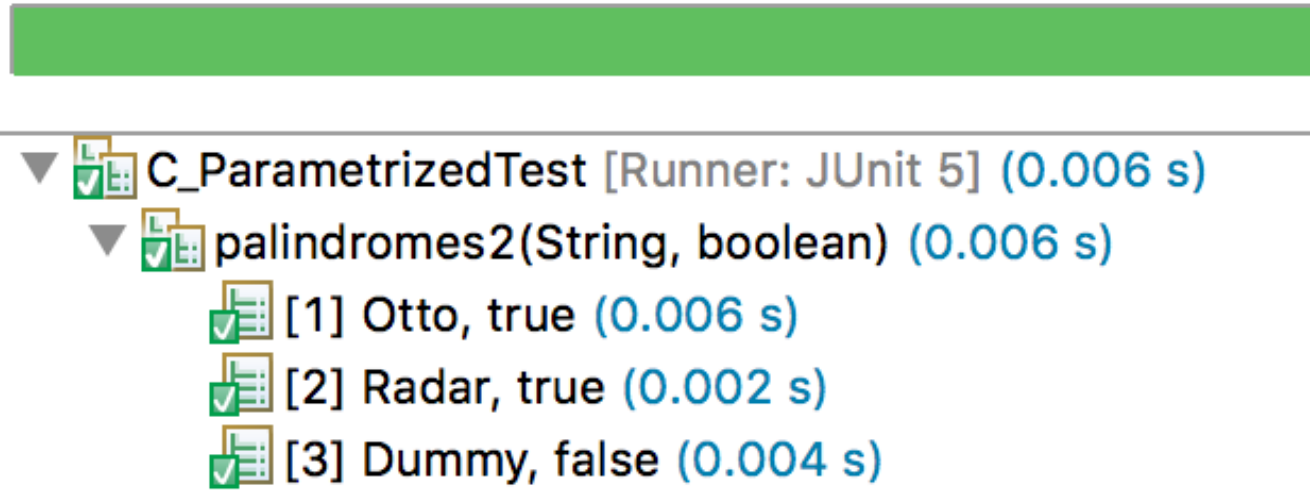
Parameterized Test – JUnit 5 @ParameterizedTest / @ValueSource

```
@ParameterizedTest
@ValueSource(strings = { "Otto", "Radar" })
void palindromes(String candidate)
{
    boolean isPalindrome = PalindromeChecker.isPalindrome(candidate));

    assertTrue(isPalindrome);
}
```



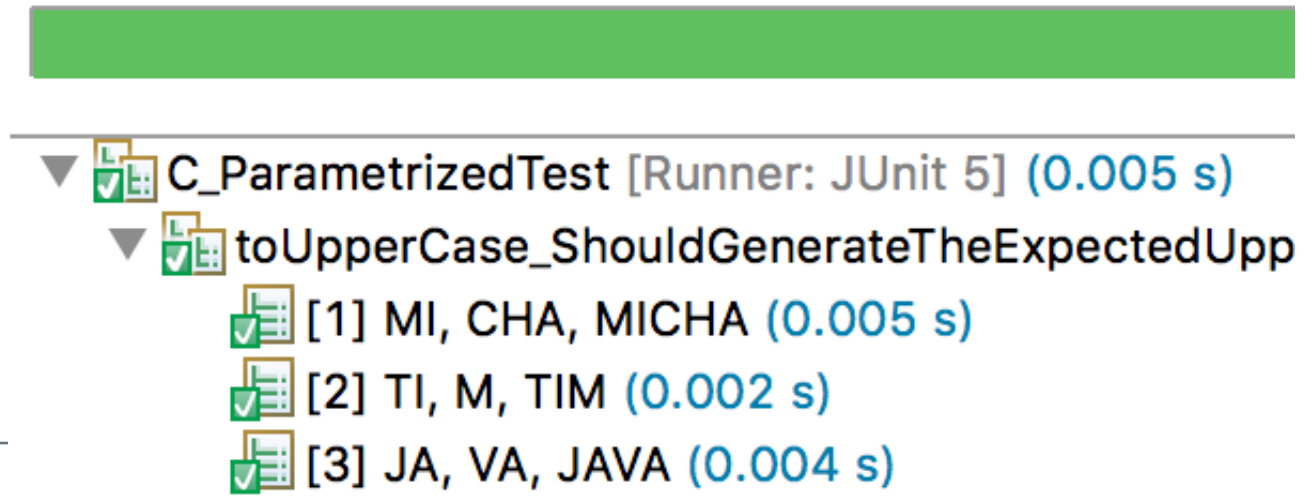
Parameterized Test – @CsvSource

```
@ParameterizedTest
@CsvSource({"Otto,true", "Radar,true", "Dummy,false" })
void palindromes2(String candidate, boolean expected)
{
    boolean isPalindrome = PalindromeChecker.isPalindrome(candidate));

    assertEquals(expected, isPalindrome);
}
```



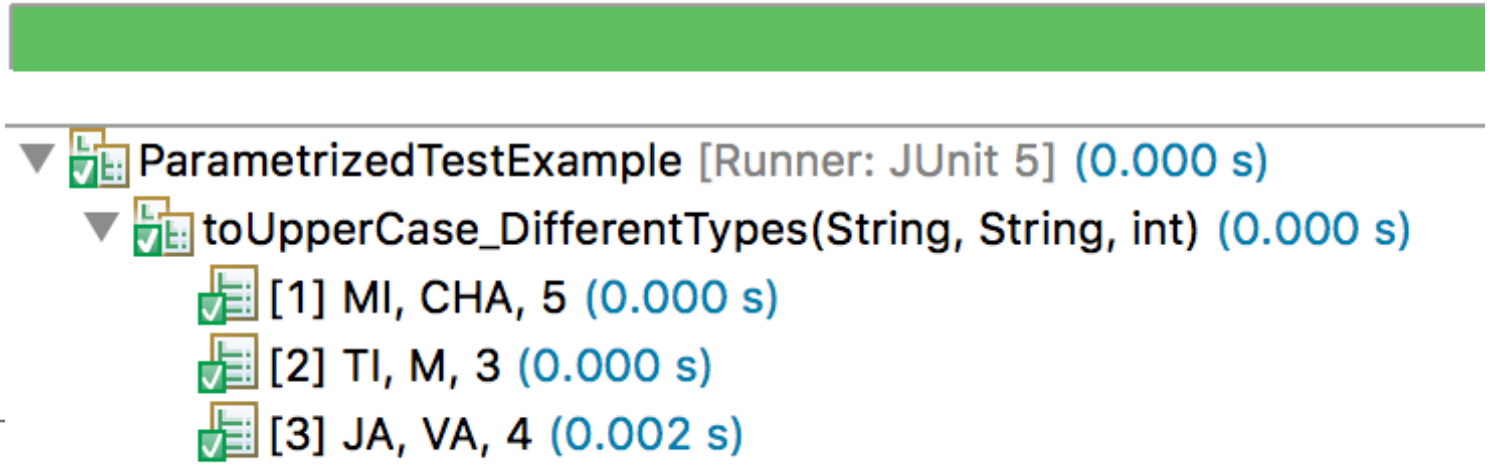
Parameterized Test – multiple inputs

```
@ParameterizedTest
@CsvSource({"MI,CHA,MICHA", "TI,M,TIM", "JA,VA,JAVA"})
void toUpperCase_ShouldGenerateTheExpectedUppercaseValue(String input1,
                                                         String input2,
                                                         String expected)
{
    String actualValue = input1.concat(input2);

    assertEquals(expected, actualValue);
}
```



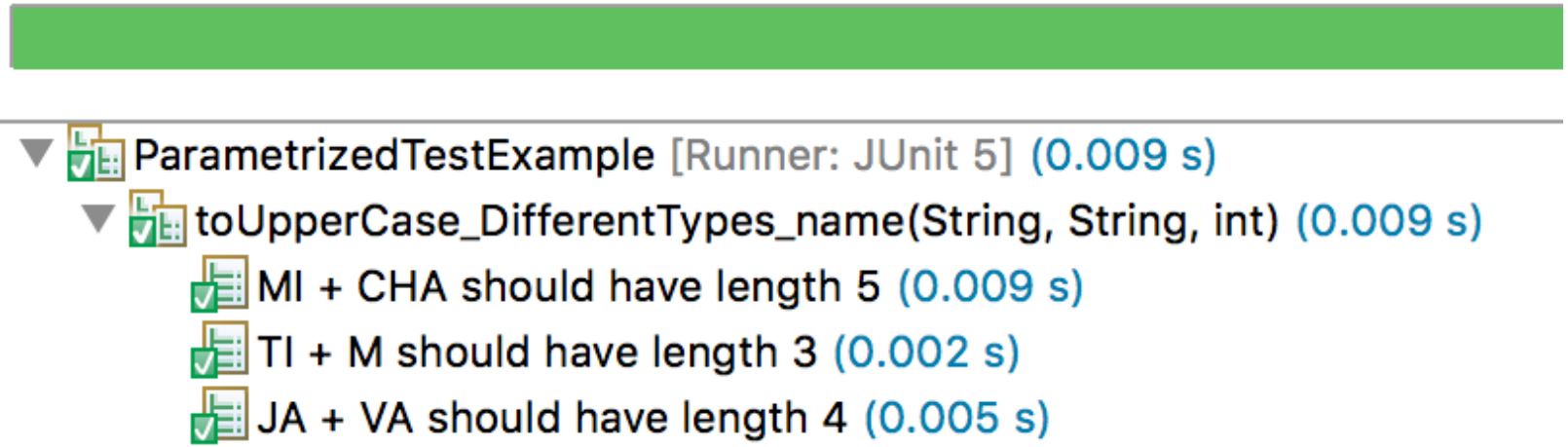
Parameterized Test – several inputs different types

```
@ParameterizedTest
@CsvSource({"MI,CHA,5", "TI,M,3", "JA,VA,4"})
void toUpperCase_DifferentTypes(String input1,
                                String input2,
                                int expectedLength)
{
    int actualValue = input1.concat(input2).length();

    assertEquals(expectedLength, actualValue);
}
```



Parameterized Test – better naming

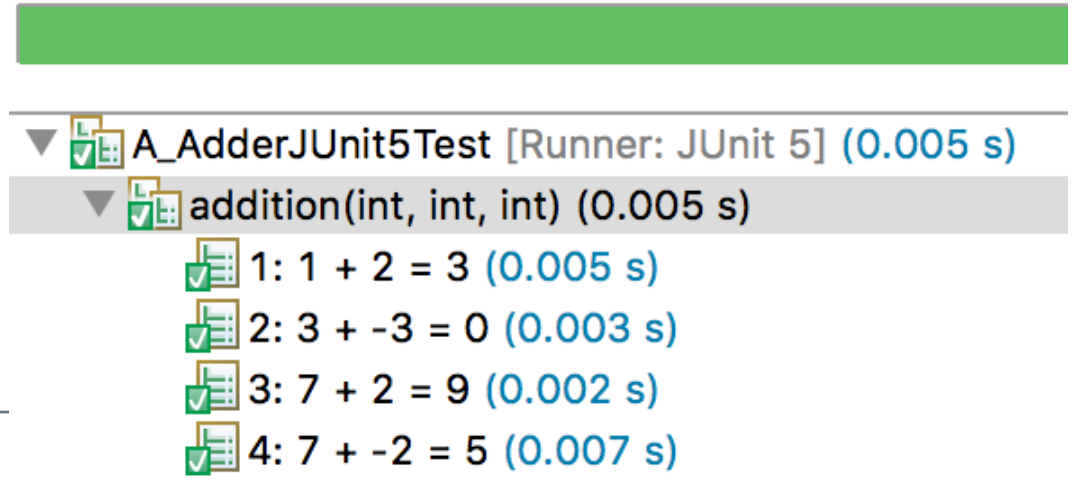
```
@ParameterizedTest(name = "{0} + {1} should have length {2}")
@CsvSource({"MI,CHA,5", "TI,M,3", "JA,VA,4"})
void toUpperCase_DifferentTypes_name(String input1,
                                     String input2,
                                     int expectedLength)
{
    int actualValue = input1.concat(input2).length();

    assertEquals(expectedLength, actualValue);
}
```



So what does the test for the Adder look like with JUnit 5?

Parameterized Test – Adder Test & Better Naming

```
public class A_AdderJUnit5Test
{
    @ParameterizedTest(name = "{index}: {0} + {1} = {2}")
    @CsvSource({ "1,2,3", "3, -3, 0", "7, 2, 9", "7,-2,5" })
    void addition(int a, int b, int result)
    {
        int sum = Adder.add(a, b);

        assertEquals(result, sum);
    }
}
```

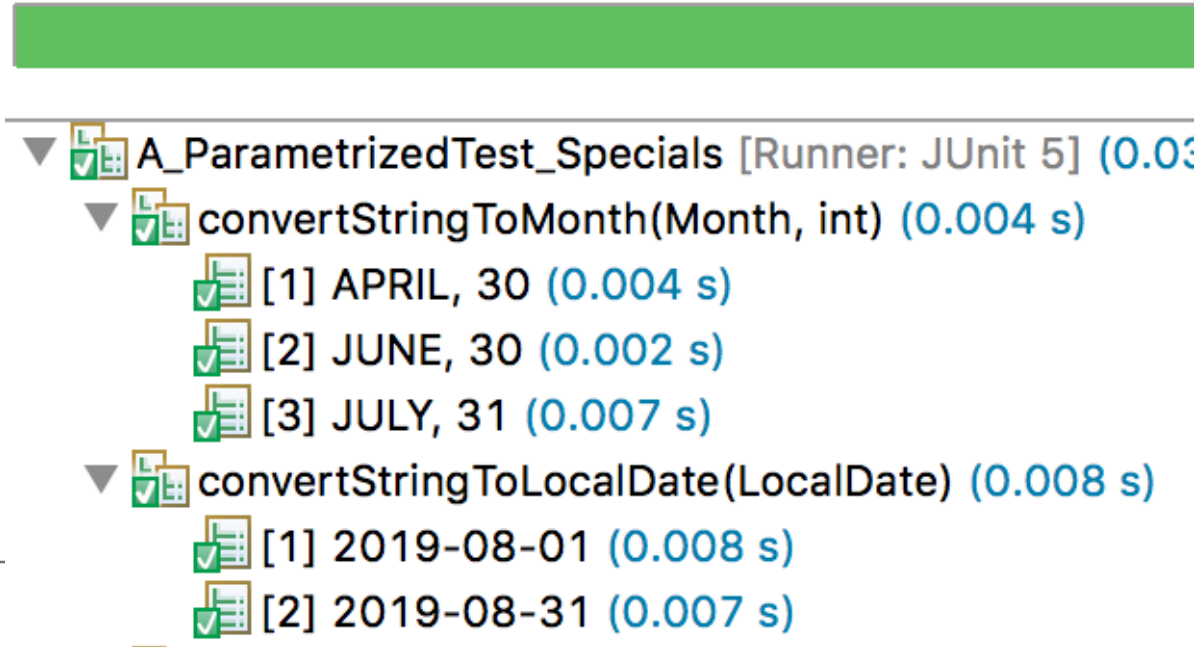


Parameterized Test – Various conversions and help

- *LocalDate, LocalTime, LocalDateTime, Year, Month, etc.*

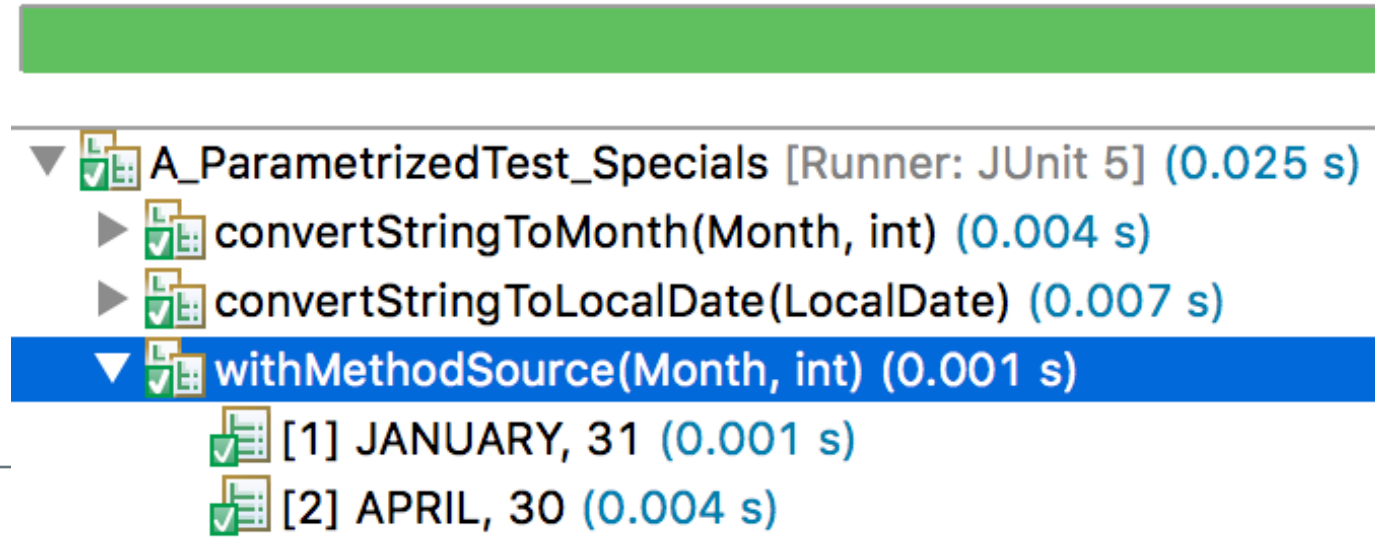
```
@ParameterizedTest
@ValueSource(strings = { "2019-08-01", "2019-08-31" })
void convertStringToLocalDate(LocalDate localDate)
{
    assertEquals(Month.AUGUST, localDate.getMonth());
}
```

```
@ParameterizedTest
@CsvSource(value= {"APRIL:30", "JUNE:30",
          "JULY:31"}, delimiter = ':')
void convertStringToMonth(Month month,
                          int length)
{
    assertEquals(length,
                 month.length(false));
}
```



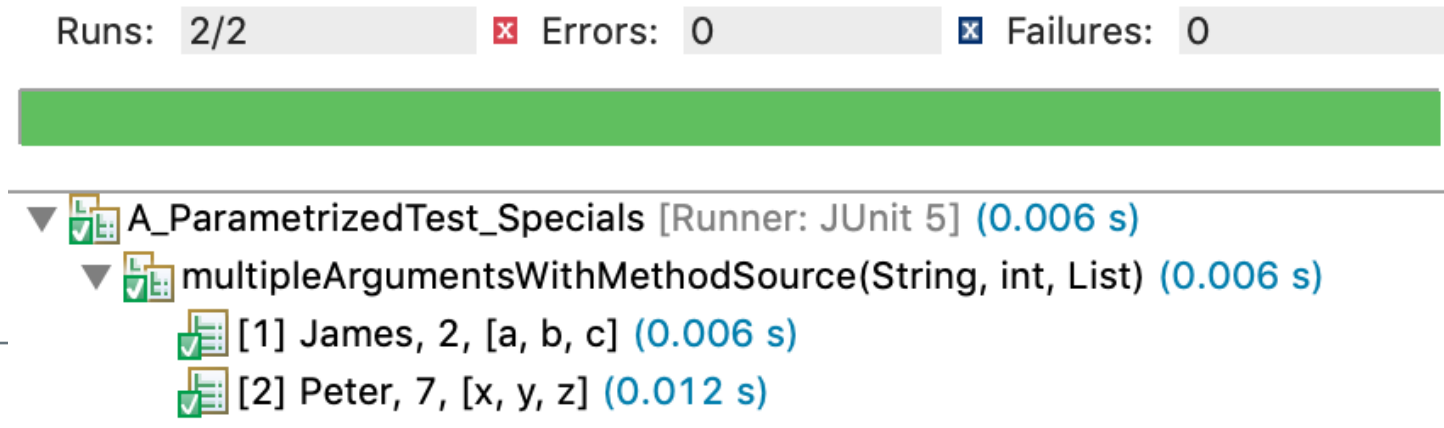
Parameterized Test – @MethodSource, e. g. for Enums

```
@ParameterizedTest
@MethodSource("createMonthsWithLength")
void withMethodSource(Month month, int expectedLength)
{
    assertEquals(expectedLength, month.length(false));
}

private static Stream<Arguments> createMonthsWithLength()
{
    return Stream.of(Arguments.of(Month.JANUARY, 31),
                     Arguments.of(Month.APRIL, 30));
}
```



Parameterized Test – @MethodSource

```java
@ParameterizedTest
@MethodSource("stringIntAndListProvider")
void multipleArgumentsWithMethodSource(String str, int num, List<String> list)
{
    assertEquals(5, str.length());
    assertTrue(num < 10);
    assertEquals(3, list.size());
}

static Stream<Arguments> stringIntAndListProvider()
{
    return Stream.of(Arguments.arguments("James", 2, List.of("a", "b", "c")),
                     Arguments.arguments("Peter", 7, List.of("x", "y", "z")));
}
```



Parameterized Test – @MethodSource, e.g. for lists as parameters

```
@ParameterizedTest(name = "removeDuplicates({0}) = {1}")
@MethodSource("listInputsAndExpected")
void removeDuplicates(List<Integer> inputs, List<Integer> expected)
{
    List<Integer> result = Ex02_ListRemove.removeDuplicates(inputs);

    assertEquals(expected, result);
}


static Stream<Arguments> listInputsAndExpected()
{
    return Stream.of(Arguments.of(List.of(1, 1, 2, 3, 4, 1, 2, 3),
                                  List.of(1, 2, 3, 4)),
                     Arguments.of(List.of(1, 3, 5, 7),
                                  List.of(1, 3, 5, 7)),
                     Arguments.of(List.of(1, 1, 1, 1),
                                  List.of(1)));
}
```

What else can be done with a Parameterized Test?

Parameterized Test – @CsvSource, e.g. for large amounts of data

```
@ParameterizedTest(name = "fromRomanNumber(''{1}'') => {0}")
@CsvSource({ "1, I", "2, II", "3, III", "4, IV", "5, V", "7, VII", "9, IX",
             "17, XVII", "40, XL", "90, XC", "400, CD", "444, CDXLIV", "500, D",
             "900, CM", "1000, M", "1666, MDCLXVI", "1971, MCMLXXI",
             "2018, MMXVIII", "2019, MMXIX", "2020, MMXX", "3000, MMM"})
@DisplayName("Convert roman number to arabic number")
void fromRomanNumber(final int arabicNumber, final String romanNumber)
{
    int result = RomanNumbers.fromRomanNumber(romanNumber);

    assertEquals(arabicNumber, result);
}
```

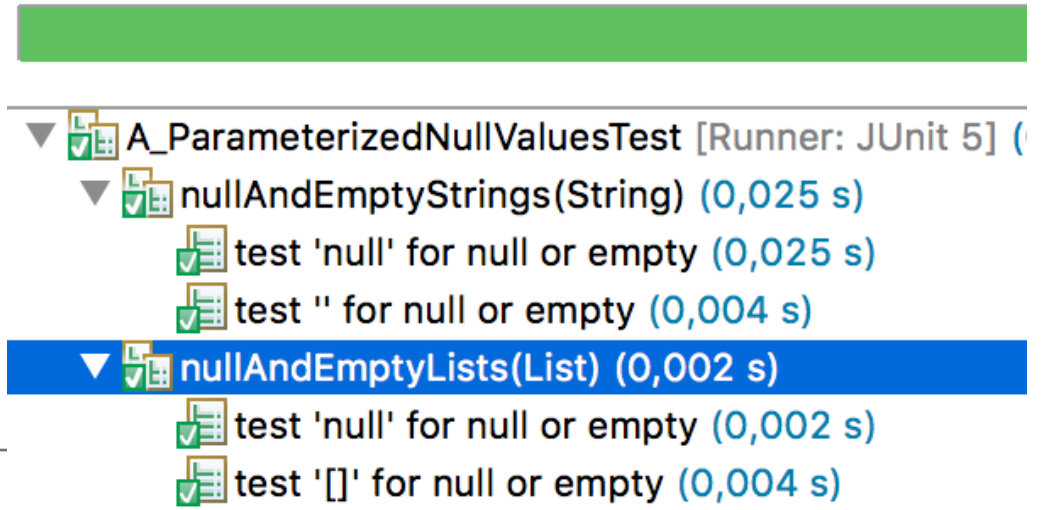
Parameterized Test – @CsvSource, e.g. for large amounts of data

```
@ParameterizedTest(name = "fromRomanNumber(''{1}'') => {0}")
@CsvFileSource(resources = "arabicroman.csv", numLinesToSkip = 1)
@DisplayName("Convert roman number to arabic number ")
void fromRomanNumber(final int arabicNumber, final String romanNumber)
{
    int result = RomanNumbers.fromRomanNumber(romanNumber);

    assertEquals(arabicNumber, result);
}
```

```
arabic,roman
1, I
2, II
3, III
4, IV
5, V
7, VII
9, IX
17, XVII
40, XL
90, XC
```

Parameterized Test – Check edge cases

```
@ParameterizedTest(name = "test ''{0}'' for null or empty")
@NullSource
@EmptySource
void nullAndEmptyStrings(String str)
{
    assertTrue(str == null || str.isEmpty());
}

@ParameterizedTest(name = "test ''{0}'' for null or empty")
@NullSource
@EmptySource
void nullAndEmptyLists(List<?> list)
{
    assertTrue(list == null || list.isEmpty());
}
```



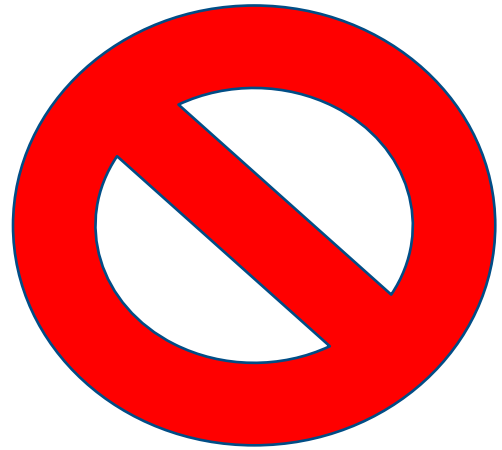
Parameterized Test – Check edge cases II

```
class DateRange
{
    // one of both can be null
    LocalDate from;
    LocalDate to;

    public DateRange(LocalDate from, LocalDate to)
    {
        if (from == null && to == null) {
            throw new IllegalArgumentException("invalid range, two nulls");
    }

    this.from = from;
    this.to = to;
}
```

- **Prüfe `(null, dateTo)`, `(dateFrom, null)` und `(null, null)`**

Parameterized Test – Check edge cases, @MethodSource

```
@ParameterizedTest(name = "validateDateRange({0} => {1}")
@MethodSource("allCombinations")
void validateDateRangeConstruction(LocalDate from, LocalDate to,
                                   Class<? extends Throwable> expectedException)
{
    // Mehr Infos später ...
    if (expectedException != null)
        assertThrows(expectedException, () -> new DateRange(from, to));
    else
        assertNotNull(new DateRange(from, to));
}

static Stream<Arguments> allCombinations()
{
    return Stream.of(Arguments.of(null, LocalDate.now(), null),
                     Arguments.of(LocalDate.now(), null, null),
                     Arguments.of(null, null, IllegalArgumentException.class));
}
```



Is this really a good idea?

Parameterized Test – Check edge cases, @MethodSource

- **NO, too ambitious because confusing due to case distinctions:**
  - because two cases represented a permitted input and
  - one represents an error case.
- **It is more natural to have this test performed by two test methods.**
- **Interestingly, one can then get rid of the parameterized test and simplify the whole thing enormously -- this is also a desirable goal in testing:**

```
@Test
void testInvalidDateRangeConstruction()
{
    assertThrows(IllegalArgumentException.class, () -> new DateRange(null, null));
}

@Test
void testValidDateRangeConstruction()
{
    assertAll(()-> assertNotNull(new DateRange(null, LocalDate.now())),
              ()-> assertNotNull(new DateRange(LocalDate.now(), null)));
}
```

Case study
@Test => @Parameterized Test

@Test => @Parameterized Test – e. g. list as expected

```
@Test
void sundaysBetween()
{
    final LocalDate start = LocalDate.of(2020, 1, 1);
    final LocalDate end = LocalDate.of(2020, 3, 3);

    final List<LocalDate> sundays =
          SundayCalculator.allSundaysBetween(start, end);

    final List<LocalDate> expectedSundays =
          Arrays.asList(LocalDate.parse("2020-01-05"),
          LocalDate.parse("2020-01-12"), LocalDate.parse("2020-01-19"),
          LocalDate.parse("2020-01-26"), LocalDate.parse("2020-02-02"),
          LocalDate.parse("2020-02-09"), LocalDate.parse("2020-02-16"),
          LocalDate.parse("2020-02-23"));

    assertEquals(expectedSundays, sundays);
}
```

How do we solve the difficulties with lists when "Expected" is more elaborate to construct?

@Test => @Parameterized Test

```
// Schritt 1: Simplify logic to prepare for further extraction

@Test
void sundaysBetweenImproved()
{
    final LocalDate start = LocalDate.of(2020, 1, 1);
    final LocalDate end = LocalDate.of(2020, 3, 1);

    final List<LocalDate> sundays =
          SundayCalculator.allSundaysBetween(start, end);

    final List<LocalDate> expectedSundays = Stream.of("2020-01-05", "2020-01-12",
          "2020-01-19", "2020-01-26", "2020-02-02", "2020-02-09", "2020-02-16",
          "2020-02-23").
          map(day -> LocalDate.parse(day)).
          collect(Collectors.toList());

    assertEquals(expectedSundays, sundays);
}
```

@Test => @Parameterized Test

```
// Schritt 2: Move conversion from String to LocalDate to method

@Test
void sundaysBetweenImproved_V2()
{
    final LocalDate start = LocalDate.of(2020, 1, 1);
    final LocalDate end = LocalDate.of(2020, 3, 3);

    final List<LocalDate> sundays = SundayCalculator.allSundaysBetween(start, end);
    final List<String> expectedSundaysAsStrings = List.of("2020-01-05", "2020-01-12",
           "2020-01-19", "2020-01-26", "2020-02-02", "2020-02-09", "2020-02-16",
           "2020-02-23", "2020-03-01");
    final List<LocalDate> expectedSundays = convertToLocalDates(expectedSundaysAsStrings);

    assertEquals(expectedSundays, sundays);
}

private List<LocalDate> convertToLocalDates(final List<String> datesAsStings)
{
    return datesAsStings.stream().map(day -> LocalDate.parse(day)).collect(Collectors.toList());
}
```

@Test => @Parameterized Test

```
//Schritt 3: Deploy Expected in Method

@Test
void sundaysBetweenImproved_V3()
{
    final LocalDate start = LocalDate.of(2020, 1, 1);
    final LocalDate end = LocalDate.of(2020, 3, 3);

    final List<LocalDate> sundays = SundayCalculator.allSundaysBetween(start, end);

    final List<String> expectedSundaysAsStrings = allExpectedDates();
    final List<LocalDate> expectedSundays = convertToLocalDates(expectedSundaysAsStrings);

    assertEquals(expectedSundays, sundays);
}

private List<String> allExpectedDates()
{
    return List.of("2020-01-05", "2020-01-12", "2020-01-19", "2020-01-26", "2020-02-02",
                   "2020-02-09", "2020-02-16", "2020-02-23", "2020-03-01");
}
```

@Test => @Parameterized Test

```
// Schritt 4: Include date range as parameter

@Test
void sundaysBetweenImproved_V4()
{
    final LocalDate start = LocalDate.of(2020, 1, 1);
    final LocalDate end = LocalDate.of(2020, 3, 3);

    final List<LocalDate> result = SundayCalculator.allSundaysBetween(start, end);

    final List<String> expectedSundaysAsStrings = allExpectedDatesForDateRange(start, end);
    final List<LocalDate> expected = convertToLocalDates(expectedSundaysAsStrings);

    assertEquals(expected, result);
}

private List<String> allExpectedDatesForDateRange(final LocalDate start, final LocalDate end)
{
    return List.of("2020-01-05", "2020-01-12", "2020-01-19", "2020-01-26", "2020-02-02",
                   "2020-02-09", "2020-02-16", "2020-02-23", "2020-03-01");
}
```

@Test => @Parameterized Test

```
// Schritt 5: Conversion to Parameterized Test

@ParameterizedTest(name = "sundays between {0} and {1} => {2}")
@MethodSource("startAndEndDateAndResults_V5")
void sundaysBetweenImproved_V5(LocalDate start, LocalDate end, List<String> expectedDates)
{
    final List<LocalDate> result = SundayCalculator.allSundaysBetween(start, end);

    final List<LocalDate> expected = convertToLocalDates(expectedDates);

    assertEquals(expected, result);
}

private static Stream<Arguments> startAndEndDateAndResults_V5()
{
    return Stream.of(Arguments.of(LocalDate.of(2020, 1, 1), LocalDate.of(2020, 3, 3),
                     List.of("2020-01-05", "2020-01-12", "2020-01-19", "2020-01-26",
                             "2020-02-02", "2020-02-09", "2020-02-16", "2020-02-23",
                             "2020-03-01")));
}
```

@Test => @Parameterized Test

```
// Schritt 6: Add more test parameterizations

@ParameterizedTest(name = "sundays between {0} and {1} => {2}")
@MethodSource("startAndEndDateAndResults")
void sundaysBetweenImproved(LocalDate start, LocalDate end, List<String> expectedAsStream)
{
    final List<LocalDate> result = SundayCalculator.allSundaysBetween(start, end);

    final List<LocalDate> expected = convertToLocalDates(expectedAsStream);

    assertEquals(expected, result);
}

private static Stream<Arguments> startAndEndDateAndResults()
{
    return Stream.of(Arguments.of(LocalDate.of(2020, 1, 1), LocalDate.of(2020, 3, 3),
                     List.of("2020-01-05", "2020-01-12", "2020-01-19", "2020-01-26", "2020-02-02",
                             "2020-02-09", "2020-02-16", "2020-02-23", "2020-03-01")),
                     Arguments.of(LocalDate.of(1971, 2, 1), LocalDate.of(1971, 3, 1),
                     List.of("1971-02-07", "1971-02-14", "1971-02-21", "1971-02-28")),
                     Arguments.of(LocalDate.of(2020, 4, 1), LocalDate.of(2020, 5, 1),
                     List.of("2020-04-05", "2020-04-12", "2020-04-19", "2020-04-26")));
}
```

Can it be any more elegant?
Yes, ArgumentConverter!

@Parameterized Test – Trick: Argument Converter for «Expected»

```
@ParameterizedTest(name = "sundays between {0} and {1} => {2}")
@MethodSource("startAndEndDateAndResults")
void sundaysBetween(LocalDate start, LocalDate end,
     @ConvertWith(ToLocalDateListConverter.class) List<LocalDate> expected)
{
    final List<LocalDate> result = SundayCalculator.allSundaysBetween(start, end);

    assertEquals(expected, result);
}

static class ToLocalDateListConverter implements ArgumentConverter
{
    @Override
    public Object convert(Object source, ParameterContext context)
                                         throws ArgumentConversionException
    {
        // leider nicht prüfbar, ob List<String>
        if (source instanceof List)
            return convertToLocalDates((List<String>) source);

        throw new ArgumentConversionException(source + " is no list");
    }
}
```

Wouldn't it be cool to be able to process JSON?

@Parameterized Test – Trick: Argument Converter for «Input»

```
@ParameterizedTest
@CsvSource(value = {
"{ name:'Peter', dateOfBirth: '2012-12-06', homeTown : 'Köln'} | false",
"{ name:'Mike', dateOfBirth: '1971-02-07', homeTown : 'Zürich'} | true" },
delimiter = '|')
void jsonPersonAdultTest(@ConvertWith(JsonToPerson.class)
                         Person person, boolean expected)
{
    final long age = ChronoUnit.YEARS.between(person.dateOfBirth, LocalDate.now());

    assertEquals(expected, age >= 18);
}
```

@Parameterized Test – Trick: JSON Argument Converter

```
static class JsonToPerson extends SimpleArgumentConverter
{
    private static final Gson gson =
                                new GsonBuilder()./* TODO IN EXERCISES */.create();


    @Override
    public Person convert(Object source, Class<?> targetType)
    {
        return gson.fromJson((String) source, Person.class);
    }
}
```

```
// https://mvnrepository.com/artifact/com.google.code.gson/gson
testCompile group: 'com.google.code.gson', name: 'gson', version: '2.10.1'
```

How to create Dynamic Tests?

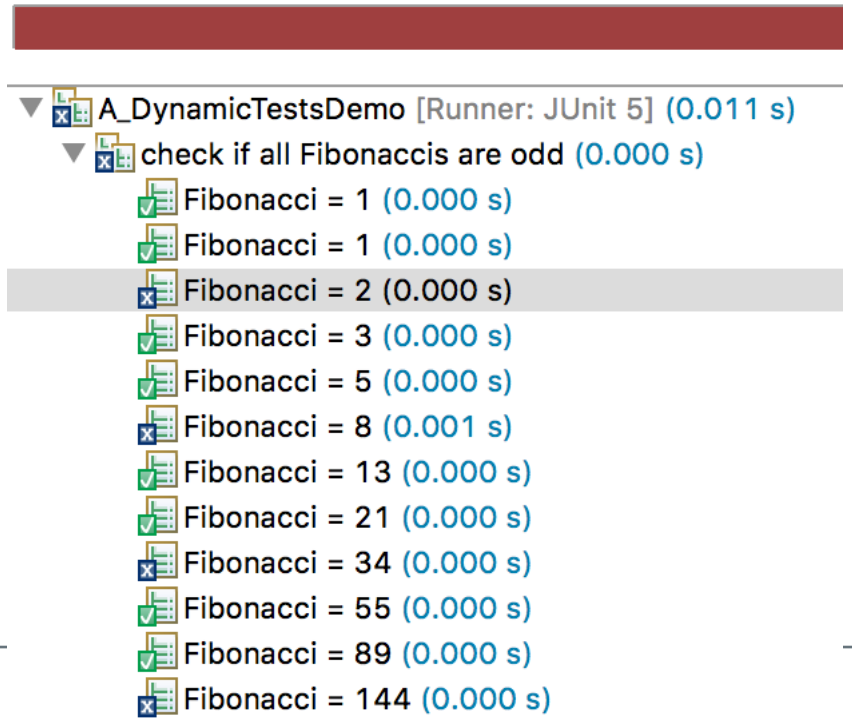
Dynamic Tests

```
@TestFactory
Stream<DynamicTest> dynamicTestsFromIntStream()
{
    return IntStream.iterate(0, n -> n + 2).limit(10)
                    .mapToObj(n -> dynamicTest("test" + n,
                                               () -> assertTrue(n % 2 == 0)));
}


@TestFactory
Collection<DynamicTest> dynamicTestsWithCollection()
{
    return Arrays.asList(dynamicTest("Add test",
                                     () -> assertEquals(2, Math.addExact(1,
                                     dynamicTest("Multiply Test",
                                     () -> assertEquals(4, Math.multiplyExact(2, 2))));
}
```

- dynamicTestsFromIntStream() (0.000 s)
  - test0 (0.000 s)
  - test2 (0.000 s)
  - test4 (0.000 s)
  - test6 (0.000 s)
  - test8 (0.000 s)
  - test10 (0.000 s)
  - test12 (0.000 s)
  - test14 (0.000 s)
  - test16 (0.000 s)
  - test18 (0.000 s)
- dynamicTestsWithCollection() (0.000 s)
  - Add test (0.000 s)
  - Multiply Test (0.000 s)

Dynamic Tests

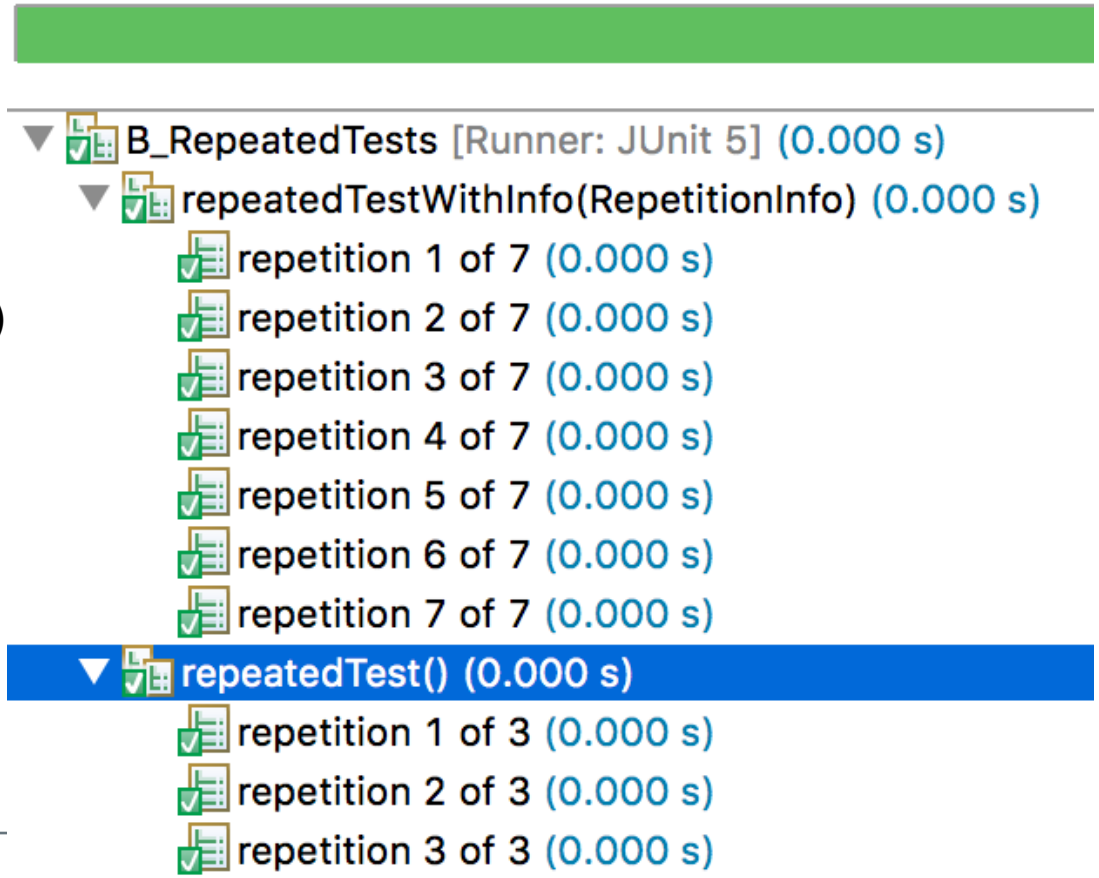
```
@TestFactory
@DisplayName("check if all Fibonaccis are odd")
Stream<DynamicTest> allFibonaccisAreOdd()
{
    return IntStream.range(1, 13).map(MathUtils::fib)
                    .mapToObj(number -> dynamicTest("Fibonacci = " + number,
                                                    () -> assertTrue(MathUtils.isOdd(number))));
}
```



How can you run tests multiple times and why would you want to?

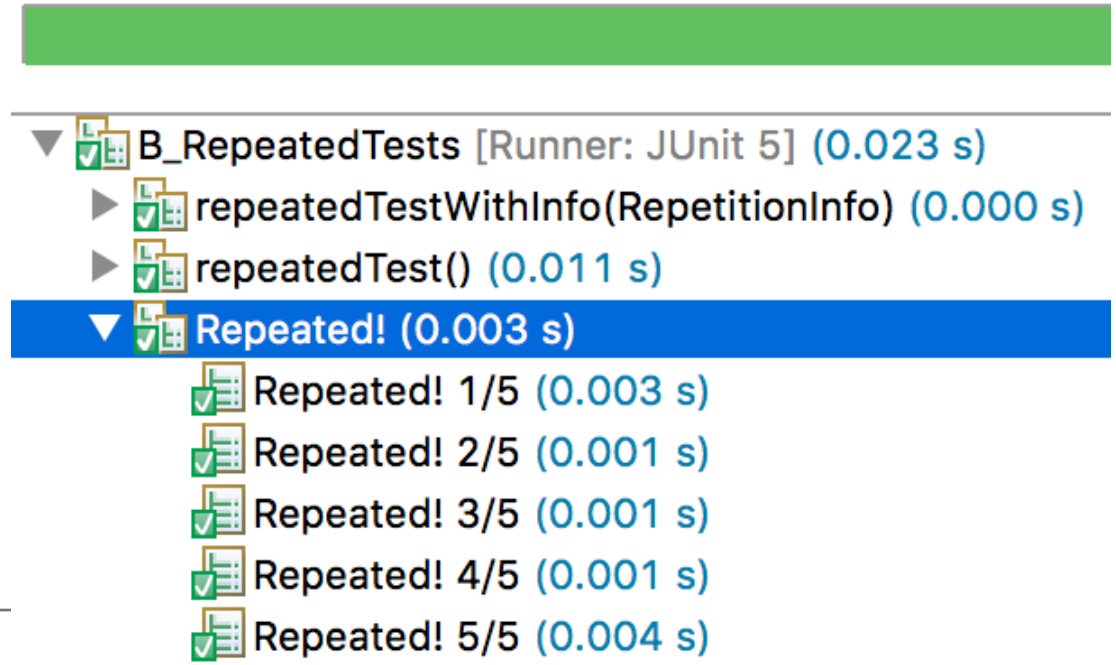
Repeated Tests – Basics

```
@RepeatedTest(3)
void repeatedTest()
{
    assertTrue(true);
}

@RepeatedTest(7)
void repeatedTestWithInfo(RepetitionInfo info)
{
    assertEquals(7, info.getTotalRepetitions())
}
```



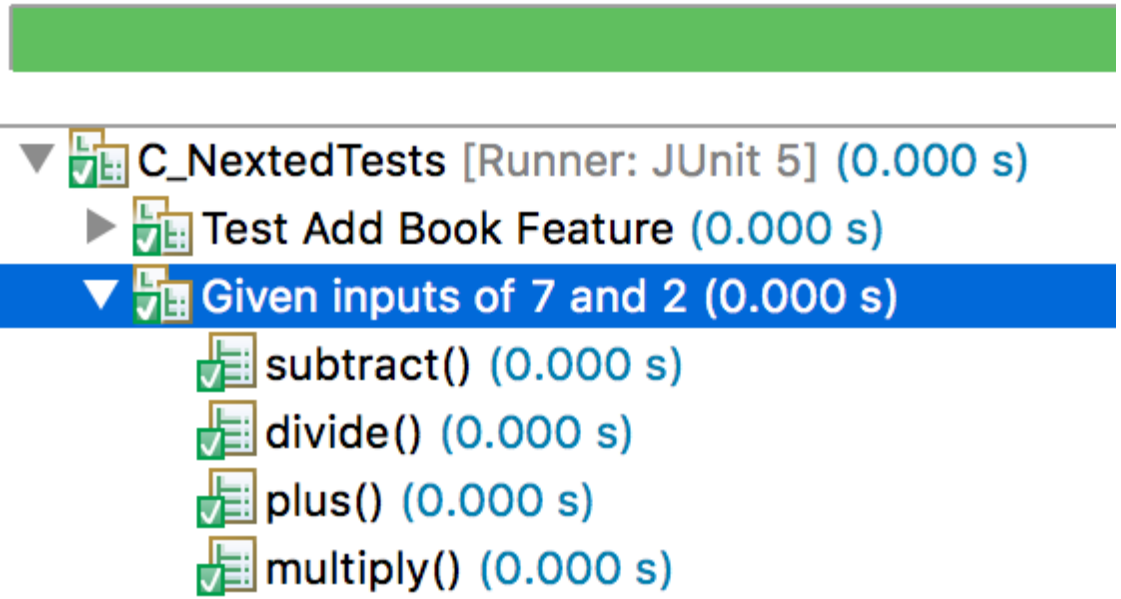
Repeated Tests – Naming and info

```
@RepeatedTest(value = 5,
              name = "{displayName} {currentRepetition}/{totalRepetitions}")
@DisplayName("Repeated! ")
void customDisplayName(TestInfo testInfo, RepetitionInfo repetitionInfo)
{
    int current = repetitionInfo.getCurrentRepetition();
    int total = repetitionInfo.getTotalRepetitions();

    assertEquals(testInfo.getDisplayName(),
                 "Repeated! " + current +
                 "/" + total);
}
```



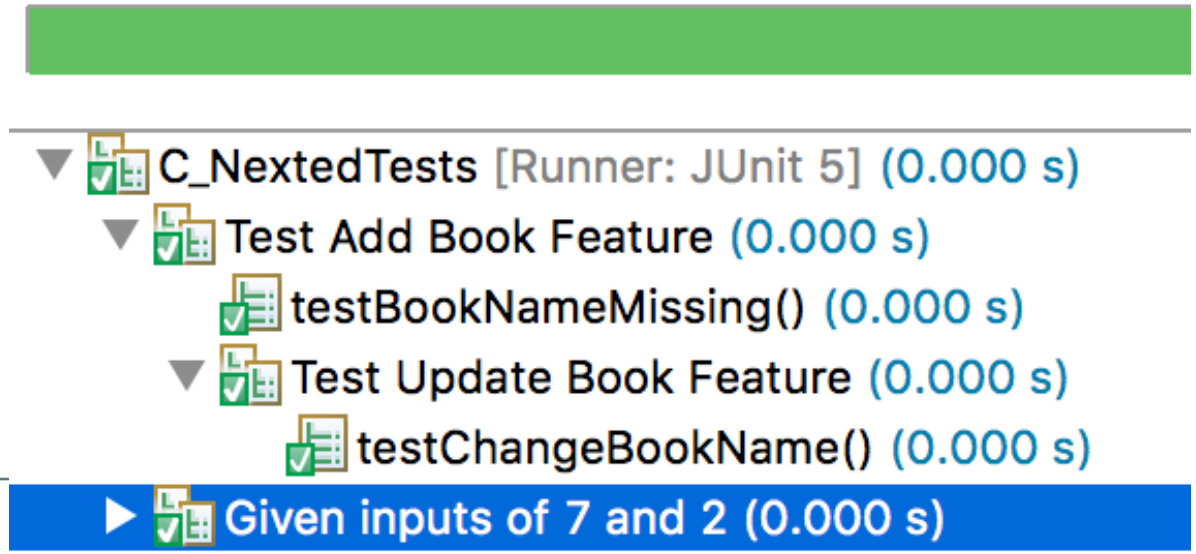
Nested Tests

```
@Nested
@DisplayName("Given inputs of 7 and 2")
class PredefinedValues
{
    final int input1 = 7;
    final int input2 = 2;

    @Test
    void plus()
    {
        assertEquals(9, input1 + input2);
    }

    @Test
    void subtract()
    {
        assertEquals(5, input1 - input2);
    }

    // …
}
```

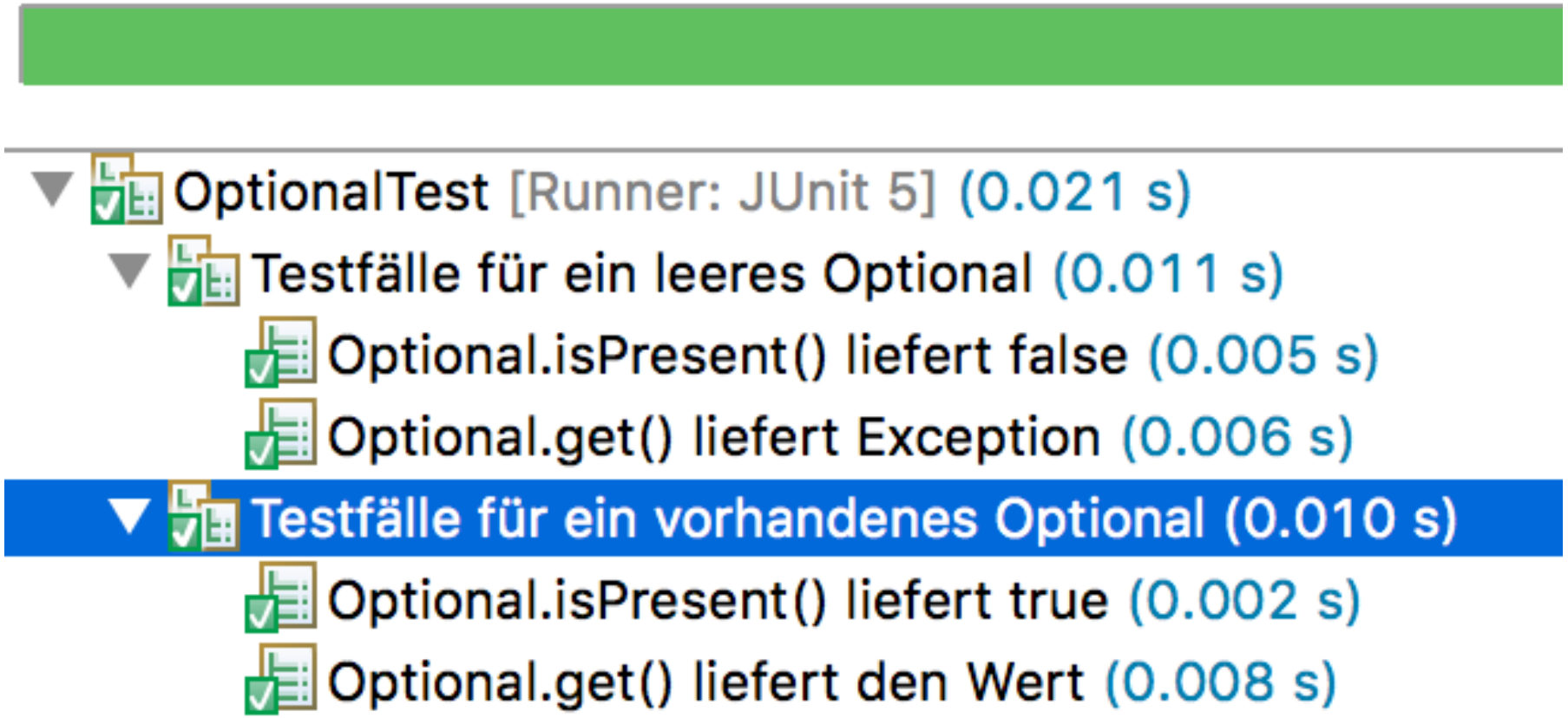


Nested Tests

```
@Nested
@DisplayName("Test Add Book Feature")
class AddFeature
{
    @Test
    void testBookNameMissing()
    {
    }

    @Nested
    @DisplayName("Test Update Book Feature")
    class UpdateFeature
    {
        @Test
        void testChangeBookName()
        {
        }
    }
}
```



Nested Tests



https://www.viadee.de/wp-content/uploads/JUnit5_javaspektrum.pdf

Simple Extensions

Simple Extensions: Benchmarking

```
public class BenchmarkExtension implements BeforeTestExecutionCallback,
                                           AfterTestExecutionCallback
{
    private long start;

    @Override
    public void beforeTestExecution(ExtensionContext ctx) throws Exception
    {
        start = System.currentTimeMillis();
    }

    @Override
    public void afterTestExecution(ExtensionContext ctx) throws Exception
    {
        System.err.println("Test " + ctx.getDisplayName() + " took " +
                           (System.currentTimeMillis() - start) + " ms");
    }
}
```

Simple Extensions: Benchmarking

```
@ExtendWith(BenchmarkExtension.class)
public class BenchmarkedFibonacciTest
{
    @Test
    void testFibRecWithBigNumber()
    {
        long value = FibonacciCalculator.fibRec(47);

        assertEquals(2971215073L, value);
    }

    @Test
    void testFibRecWithBigNumber_Timeout()
    {
        assertTimeoutPreemptively(Duration.ofSeconds(2),
                                  () -> FibonacciCalculator.fibRec(47));
    }
}
```

```
Test testFibRecWithBigNumber() took 8675 ms
Test testFibRecWithBigNumber_Timeout() took 2012 ms
```

PART 4:
Migration tips

Things to know to get started right away

- Presumably you already have a larger base of tests => migration plan necessary
- Migration or / and parallel operation possible: Parallel operation is possible
- Practical: Similar annotations but in other packages
- JUnit 4 parallel to JUnit 5:

```
// JUnit 4 Support
testCompile "junit:junit:4.13.2"
testRuntime "org.junit.vintage:junit-vintage-engine:5.9.3"
```

- Use some JUnit 4 Rules:

```
// Migration Support to Enable Rules in JUnit 5
testCompile "org.junit.jupiter:junit-jupiter-migrationsupport:5.9.3"
```

Things to know to get started right away

- Both: Test cases in the form of special test methods, which must be marked with @Test.

- JUnit 4:
  - Test methods `public`
  - No parameters allowed
  - Package: `org.junit`
  - Parameter order: : `assertTrue("Always true", true);`
  - `assertThat()` and some Hamcrest matchers included

- JUnit 5:
  - Test methods no longer mandatory `public`
  - Can have parameters
  - Package: `org.junit.jupiter.api`
  - Parameter order: : `assertTrue(true, "Always true");`
  - No `assertThat()` and no Hamcrest matchers

Annotations JUnit 4 vs JUnit 5

JUnit 4	JUnit 5		Beschreibung
`org.junit`	`org.junit.jupiter.api`		Package
`@Test`	`@Test`		Defines a test case
`@Ignore`	`@Disabled`		Deactivate test case (temporarily)
`@BeforeClass`	`@BeforeAll`		Action ***once*** before all test methods
`@Before`	`@BeforeEach`		Action in ***each case*** before all test methods
`@After`	`@AfterEach`		Action in ***each case*** after all test methods
`@AfterClass`	`@AfterAll`		Action ***once*** after all test methods
`@Rule`	`-/-`		Extensions, in JUnit 5 with special methods

Conversion to JUnit 5

- In the long run a migration / switch to JUnit 5 is advisable

- Step by step class by class, package by package

    - Delete imports and adapt to package: org.junit.jupiter.api
    - Adjust annotations slightly if necessary @BeforeXXX, @AfterXXX
    - Note parameter order
    - @Rule ExpectedException replace by JUnit 5 features, e.g. assertThrows()
    - @Rule TimeOut replace by JUnit 5 features, e.g. assertTimeout()

Conversion to JUnit 5

- Step by step class by class, package by package
  - Include @EnableRuleMigrationSupport from "org.junit.jupiter:junit-jupiter-migration-support:5.9.3".
    - TemporaryFolder
    - ErrorCollector
    - ExpectedException
  - replace assertThat() with AssertJ or Hamcrest

What benefits AssertJ offers me?

AssertJ – https://assertj.github.io/doc/

- Include JAR in IDE / customize build file

```
testCompile group: 'org.assertj', name: 'assertj-core', version: '3.20.2'
```

- **import static** org.assertj.core.api.Assertions.*;

```
@Test
void assertJAssertionsBasics()
{
    String peter = "Peter";
    assertThat(peter).isEqualTo("Peter");

    assertThat(peter.isEmpty()).isFalse();
    assertThat("".isEmpty()).isTrue();

    assertThat(7.271).isEqualTo(7.2, withPrecision(0.1d));
}
```

AssertJ – Advantages with strings compared to JUnit assertXyz()

- JUnit 5 offers only the check for (non-)match.
- especially for strings in practice desirable to be able to check legibly whether a string is not empty or contains desired characters.

```
@Test
void someStringAsserts()
{
    // JUnit style
    assertFalse("ABC".isEmpty());
    assertTrue("ONE TWO THREE".contains("TWO"));

    // AssertJ
    assertThat("ABC").isNotEmpty();
    assertThat("ONE TWO THREE").contains("TWO");
}
```

AssertJ – Advantages with numbers compared to JUnit assertXyz()

```
@Test
void someNumberAsserts()
{
    // JUnit style
    assertEquals(42, 42);
    assertEquals(3.415, 3, 0.15d);

    IntPredicate greaterThan27 = n -> n >= 27;
    assertTrue(greaterThan27.test(42));
    assertTrue(isBetweenOWN(9, 0, 10));

    // AssertJ
    assertThat(42).isEqualTo(42);
    assertThat(3.1415).isEqualTo(3, withPrecision(0.15d));
    assertThat(3.1415).isCloseTo(3, withPrecision(0.15d));
    assertThat(42).isGreaterThanOrEqualTo(27);
    assertThat(7).isBetween(0, 10);
}

boolean isBetweenOWN(int n, int lowerBound, int upperBound)
{
    return n >= lowerBound && n < upperBound;
}
```

AssertJ – Lists I

```
@Test
void assertJCollectionsBasics()
{
    List<String> names = List.of("Tim", "Tom", "Mike");

    assertThat(names).isNotEmpty();
    assertThat(names).contains("Mike");
    assertThat(names).startsWith("Tim");

    assertAll(()-> assertThat(names).isNotEmpty(),
              ()-> assertThat(names).contains("Mike"),
              ()-> assertThat(names).startsWith("Tim"));

    // AssertJ Variante Chaining
    assertThat(names).isNotEmpty().
                      contains("Mike").
                      startsWith("Tim");
}
```

AssertJ – Lists II

- JUnit 5 bietet nicht die gleiche Aussagekraft wie AssertJ
- Bei Strings noch durch Tricks erreichbar, aber für Listen nur sehr umständlich

```
@Test
void someListAsserts()
{
    List<String> list = List.of("2", "3", "5", "7", "11", "13");

    // JUnit style
    assertNotNull(list);
    assertFalse(list.isEmpty());

    // AssertJ
    assertThat(list).isNotNull();
    assertThat(list).isNotEmpty();
    assertThat(list).startsWith("2").contains("7").endsWith("11", "13");
    assertThat(list).doesNotContain("42").containsSequence("3", "5", "7");
}
```

AssertJ – Maps

```
@Test
void assertJMapsBasics()
{
    Map<String, Integer> personsAgeMap = Map.of("Tim", 48, "Tom", 7, "Mike", 48);


    assertThat(personsAgeMap).isNotEmpty()
                             .containsKey("Mike")
                             .doesNotContainKeys("Peter")
                             .contains(Map.entry("Tim", 48));
}
```

Tipp:

```
assertThat(personList).contains(mike).isSortedAccordingTo(byAge);
```

AssertJ – Handling Exceptions

```
@Test
void testException()
{
    ThrowingCallable action = () -> {
        throw new Exception("PENG!");
    };

    assertThatThrownBy(action).isInstanceOf(Exception.class).
                               hasMessageContaining("PENG");
}

@Test
void testException_2()
{
    ThrowingCallable action = () -> {
        throw new IOException("PENG!");
    };

    assertThatExceptionOfType(IOException.class).isThrownBy(action).
                                                 withMessage("%s!", "PENG").
                                                 withMessageContaining("NG").
                                                 withNoCause();
}
```

Initial situation: Entity class and plain AssertJ

```
@Entity
public class ToDo {

    enum CompletionStage { OPEN, CLOSED, ONGOING };
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String description;
    private LocalDate deadline;
    private CompletionStage completionStatus;
    // change tracking
    private LocalDateTime createdAt;
    private LocalDateTime modifiedAt;


    public ToDo(String description)
    {
        this.description = description;
        this.completionStatus = CompletionStage.OPEN;
        this.createdAt = LocalDateTime.now();
        this.modifiedAt = LocalDateTime.now();
    }
```

Trip: Plain AssertJ vs. Custom Asserts

```
@Test
void savedToDoHasCreationDateV1() {
    ToDo toDo = new ToDo("Solve Exercises");

    ToDo savedToDo = todoRepository.save(toDo);

    assertThat(savedToDo.getCreatedAt()).isNotNull();
    assertThat(savedToDo.getCompletionStatus()).isEqualTo(ToDo.CompletionStage.OPEN);
}

@Test
void savedToDoHasCreationDateV2() {
   ToDo toDo = new ToDo("Solve Exercises");

   ToDo savedToDo = todoRepository.save(toDo);

   ToDoAssert.assertThat(savedToDo).hasCreationDate();
   ToDoAssert.assertThat(savedToDo).isInOpenState();
   // ToDoAssert.assertThat(savedToDo).hasCreationDate().isInOpenState();
}
```

Trip: Own asserts

```
class ToDoAssert extends AbstractAssert<ToDoAssert, ToDo> {

    ToDoAssert(ToDo toDo) {
        super(toDo, ToDoAssert.class);
    }

    static ToDoAssert assertThat(ToDo actual) {
        return new ToDoAssert(actual);
    }

    public ToDoAssert hasCreationDate() {
        isNotNull();
        if (actual.getCreatedAt() == null) {
            failWithMessage("Expected ToDo to have a creation date, but it was
null");
        }
        return this;
    }

    …
}
```

Exercises Part 3 + 4

https://github.com/Michaeli71/AST-JUnit5

PART 5:
Test modes and dependencies

State-based vs. behavior-based testing

State-based testing

- **Changes in the object state are examined: For this purpose, various properties or attributes are read and checked against expected values.**

- **According to the AAA style**
  - **first the correct context is provided,**
  - **then the desired functionality to be tested is executed**
  - **and finally the result is checked.**

```
// GIVEN: An empty list
final List<String> names = new ArrayList<>();

// WHEN: adding 2 elements
names.add("Tim");
names.add("Mike");

// THEN: list should contain 2 elements
assertEquals(2, names.size(), "list should contain 2 elements");
```

State-based testing

```
// GIVEN: An empty list
final List<String> names = new ArrayList<>();

// WHEN: adding 2 elements
names.add("Tim");
names.add("Mike");

// THEN: list should contain 2 elements
assertEquals(2, names.size(), "list should contain 2 elements");
```

- **BUT: The checked state may also have been created by other calls!**
- **SO: How do you check interactions and their sequences? For example, that the method add() was called twice?**

Behavior-based testing

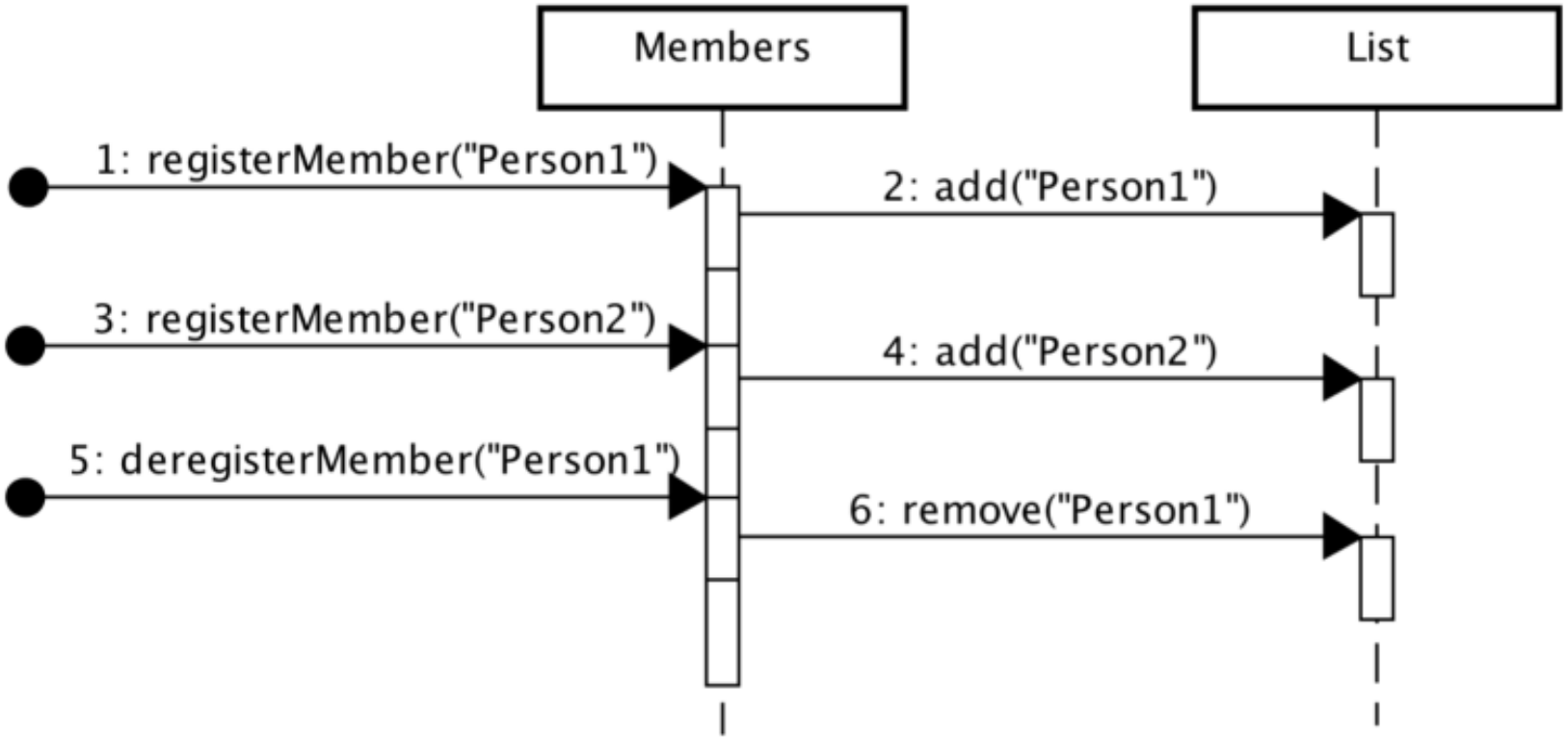
- The point here is to examine the interactions rather than the specific changes in state that have been triggered.

```
public final class Members
{
    private final List<String> members;

    Members(final List<String> persons)
    {
        this.members = persons;
    }

    public boolean registerMember(final String member)
    {
        return members.add(member);
    }

    public boolean deregisterMember(final String member)
    {
        return members.remove(member);
    }
...
```

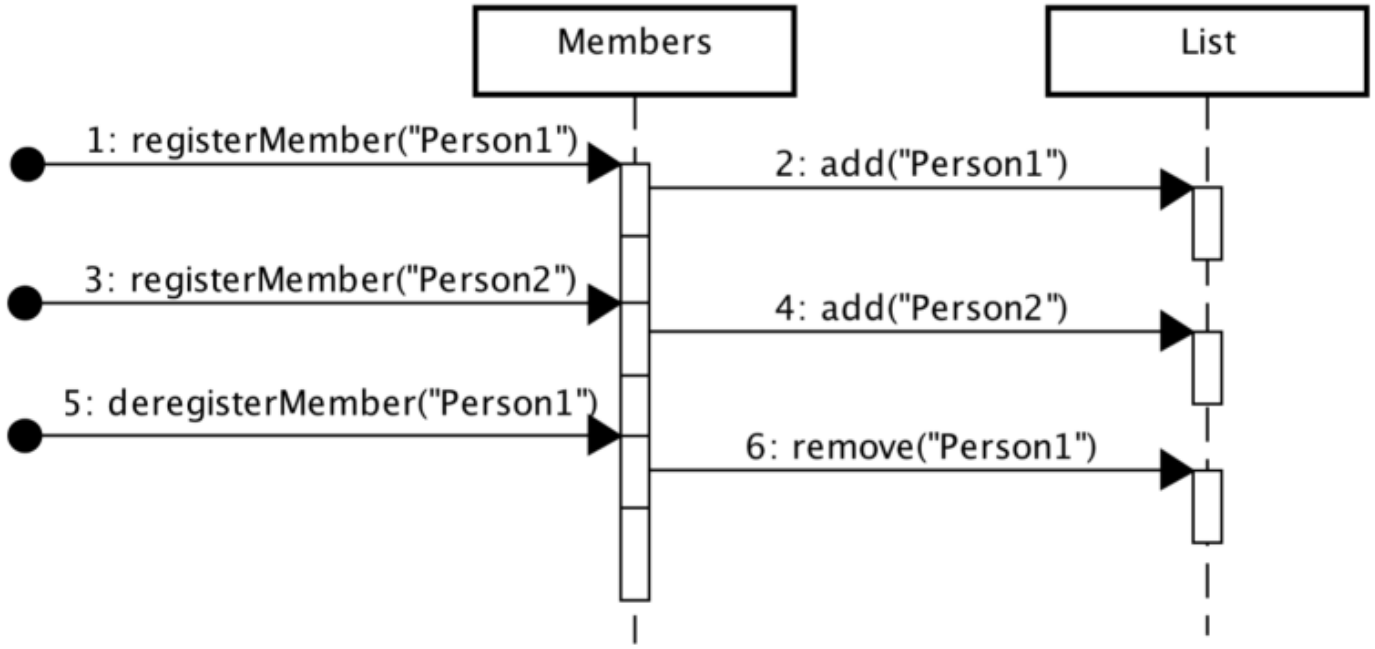
Behavior-based testing

- Behavioral testing looks at the **interactions**, i.e. the **methods called**, rather than changes in the object state.
- Accordingly, one is not interested, for example, in whether the number of stored elements has increased after a call to the `registerMember(String)` method.
- In state-based testing, one would simply use a corresponding call to an `assertXYZ()` method to check.
- **But how can one then check whether there is correct behavior?**

Behavior-based testing



Behavior-based testing



- Expectation based on method calls:
  - two calls to `add(String`
  - followed by one call to `remove(String)`

- In behavior-based testing, we check for exactly this.
- We do this by creating a proxy object that logs the interactions and allows us to match them against expectations later.

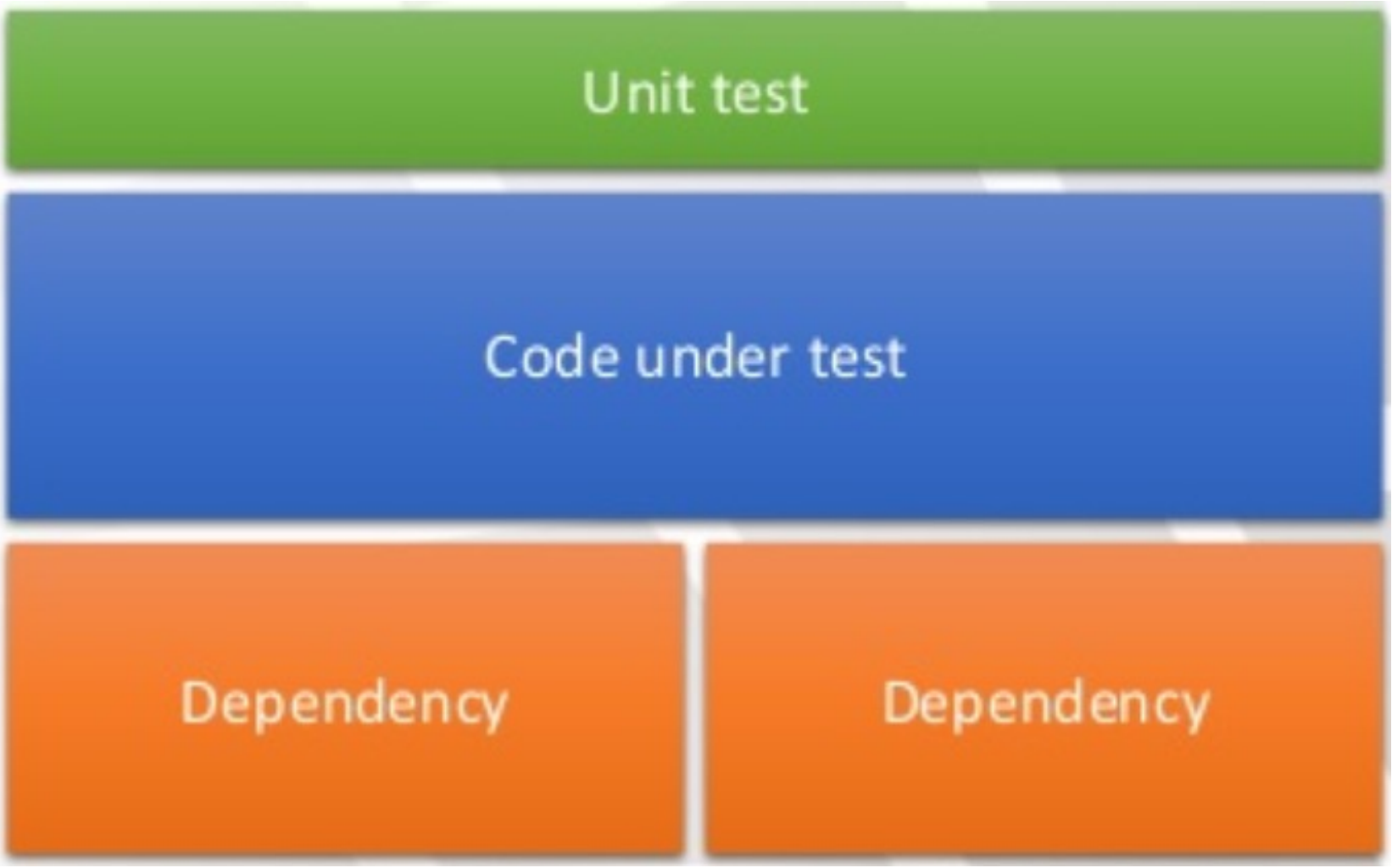
Behavior-based testing

- Suppose we were to obtain a special test proxy by calling mock() and check expectations by calling verify():

```
// Arrange
final List<String> mockedList = mock(List.class);
final Members members = new Members(mockedList);

// Act
members.registerMember("Person1");
members.registerMember("Person2");
members.deregisterMember("Person1");
// Assert
verify(mockedList).add("Person1");
verify(mockedList).add("Person2");
verify(mockedList).remove("Person1");
```

Substitute ... Proxy … Double

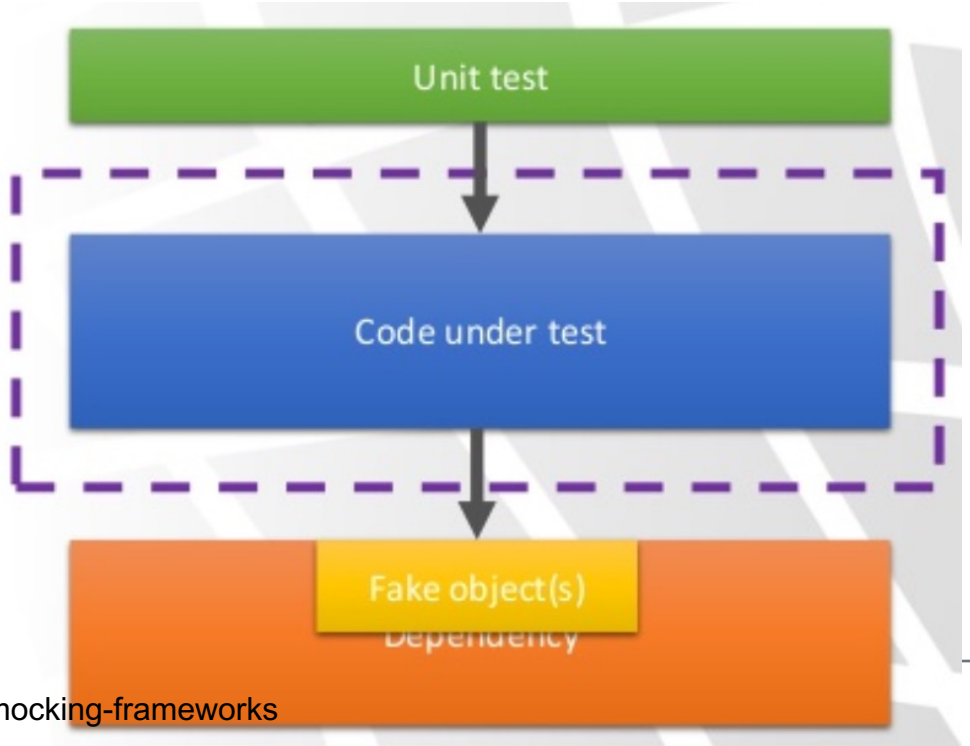
Substitute objects - Why are they necessary?



https://www.slideshare.net/dhelper/battle-of-the-mocking-frameworks

Substitute objects - Why are they necessary?

- Also called test doubles, these are objects that are used as substitutes for application objects or third-party systems to facilitate tests.
- By using test doubles it is possible to substitute other components for test cases or to simulate their behavior.
- In this way, it is possible to test state or behavior without always having to provide an entire system or a special configuration.



https://www.slideshare.net/dhelper/battle-of-the-mocking-frameworks

Test Doubles

- Many people are probably more familiar with the terms stubs and mocks.

- Both are derived from English:
  - "stub" = stump, stub and
  - "to mock" = to imitate, to pretend.

- You can find the following variants of Test Doubles:
  - Dummy
  - Stub and Fake
  - Mock

Dummy, Stub and Fake

- A dummy is a placeholder that does not provide any functionality. Sometimes you have to pass certain parameters to a method to resolve dependencies.

```
final Object optionalDataDummy = null;

doSomething(mandatoryValue, optionalDataDummy);
```

- Much more complex than dummies are stubs and fakes: To me, each represents a rudimentary, sometimes only slightly stripped-down, but working implementation of another class.

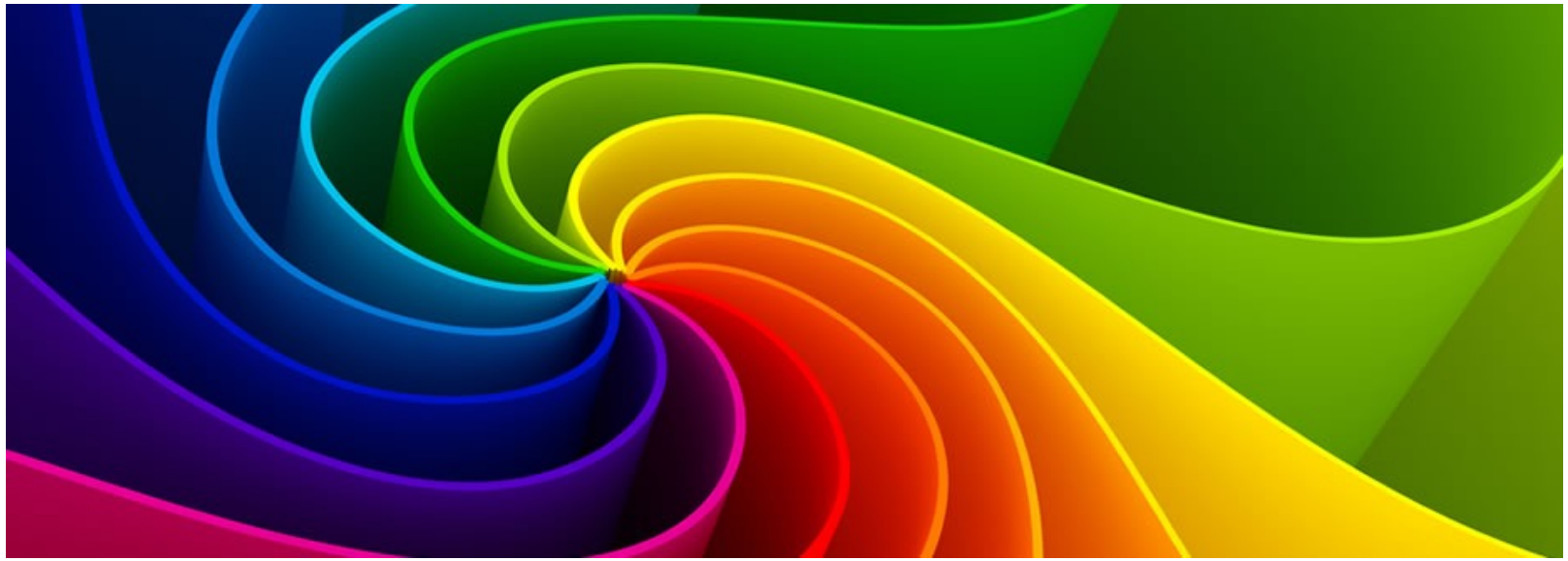
```
public final class SimulationDisplayStubBasic implements IDisplay
{
    @Override
    public void displayMsg(final MessageDto msg)
    {
        System.out.println("SimulationDisplay - got msg '" + msg + "'");
    }
}
```

Mock

- Mocks are used to check behavior in the form of expected method calls. If the application functionality does not call the previously specified methods, this is considered an error.

```
public final class SimulationDisplayMock implements IDisplay
{
    private boolean displayMsgCalled = false;

    @Override
    public void displayMsg(final MessageDto msg)
    {
        displayMsgCalled = true;
    }

    // MOCK
    public void verifyDisplayMsgWasCalled() throws AssertionError
    {
        PreConditions.checkState(displayMsgCalled,
                                 "method 'displayMsg' has not been called");
    }
}
```

PART 6:
Design For Testability



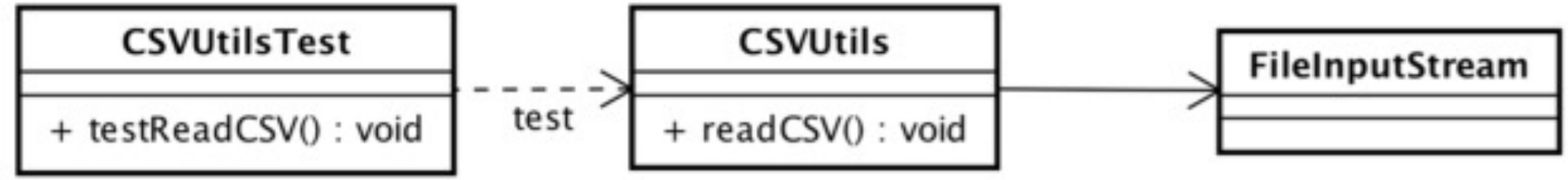
Target breaking points

Target breaking points – Injection Points

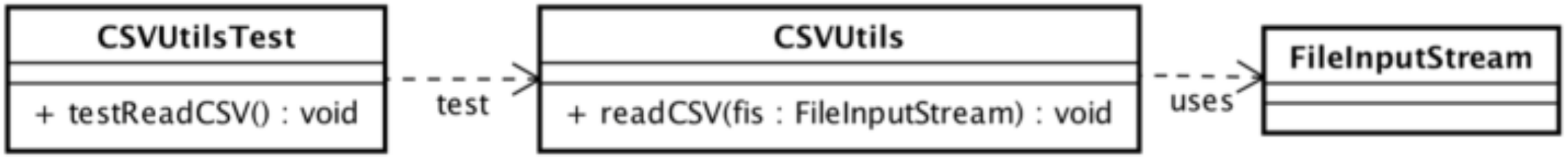
- Often direct dependencies hinder testing
- It is preferable to work against abstraction, i.e.
  - Abstract class
  - Interface
- Sometimes these exist and are only suitable to be inserted into the design
- Sometimes an abstraction must first be created and then used (**Dependency Injection**)

Target breaking points – Injection Points

- Often direct dependencies complicate testing

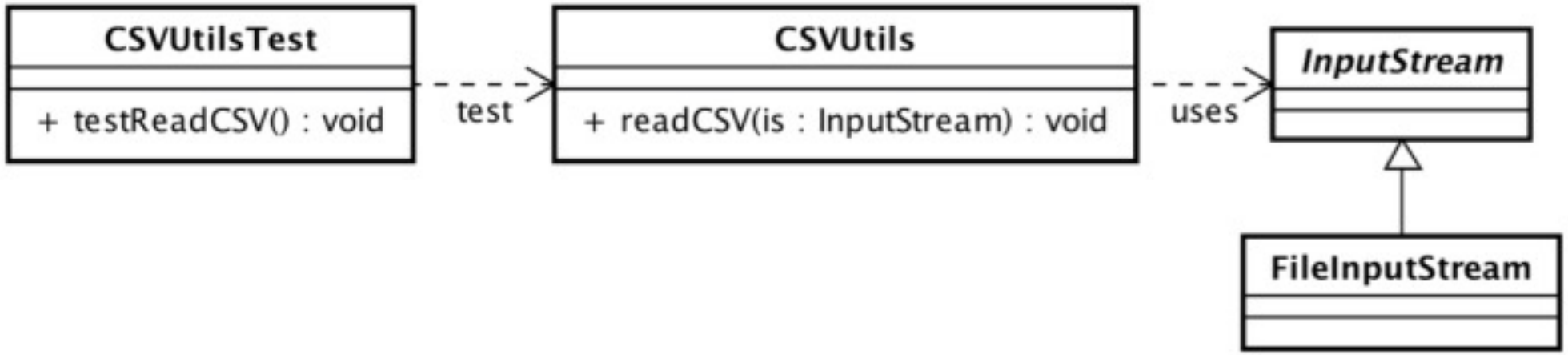


- Use Indirection: Method Injection

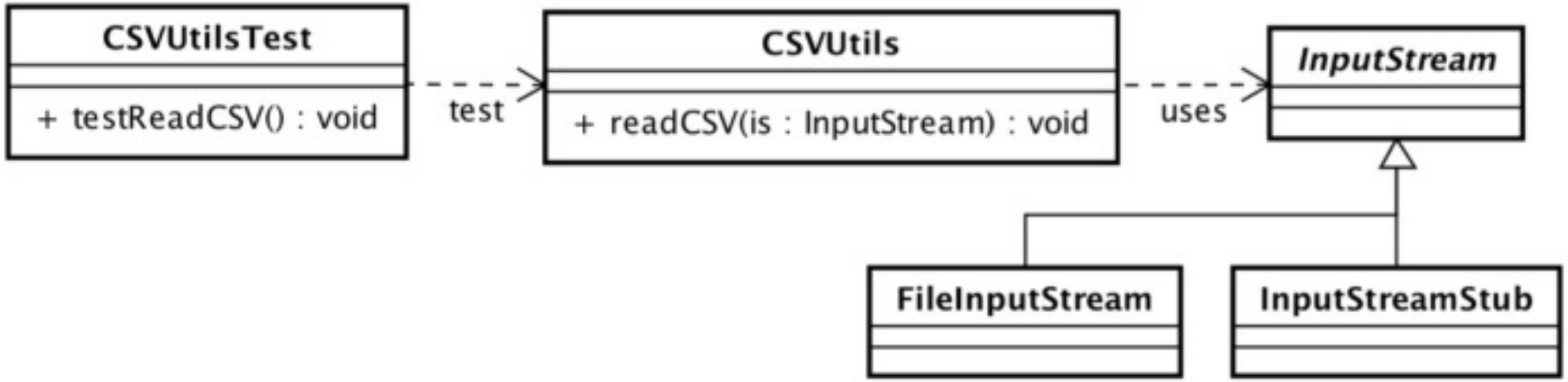


Target breaking points – Injection Points

- Use Abstraction



- Use stub for testability




EXTRACT
Override

Extract And Override

- A variant of inserting a breaking point into the application code
- Uses derivation and overwriting
- Circumvents pitfalls that make unit tests difficult to execute

Extract And Override

- **Let's start with a calculator ...**

```
public class Calculator
{
    public int calc(final String strNum1, final String strNum2)
    {
        try
        {
            final int num1 = Integer.parseInt(strNum1);
            final int num2 = Integer.parseInt(strNum2);

            return num1 + num2;
        }
        catch (final NumberFormatException ex)
        {
            JOptionPane.showConfirmDialog(null, "Keine gültige Ganzzahl");
            throw new IllegalArgumentException("Keine gültige Ganzzahl");
        }
    }
}
```

- **Do you spot the problem?**

Extract And Override -- Let's write some tests ...

```
public class CalculatorTest
{
    @Test
    void testCalc_TwoNumbers_ShouldReturnSum()
    {
        final Calculator calculator = new Calculator();
        assertEquals(5, calculator.calc("2", "3"));
    }

    @Test
    void testCalc_WithEqualNumbersButDifferentSigns_ShouldReturn0()
    {
        final Calculator calculator = new Calculator();
        assertEquals(0, calculator.calc("7", "-7"));
    }

    @Test
    void testCalc_IllegalInputs_ShouldRaiseException()
    {
        final Calculator calculator = new Calculator();
        assertThrows(IllegalArgumentException.class, () -> calculator.calc("a2", "b3"));
    }
}
```

Extract And Override

- AUA: Test execution is interrupted until someone closes the dialog!

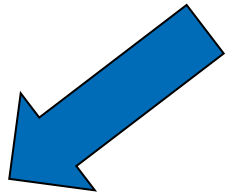
- User interaction is already annoying when running a unit test manually (e.g. in the IDE)
- Showstopper in an automated build on Continuous Integration server.
- So what now? Let's briefly consider what we can do.

Extract And Override – Breaking Point

```
public class Calculator
{
    public int calc(final String strNum1, final String strNum2)
    {
        try
        {
            final int num1 = Integer.parseInt(strNum1);
            final int num2 = Integer.parseInt(strNum2);
            return num1 + num2;
        }
        catch (final NumberFormatException ex)
        {
            showWarning("Keine gültige Ganzzahl");
            throw new IllegalArgumentException("Keine gültige Ganzzahl");
        }
    }

    protected void showWarning(final String message)
    {
        JOptionPane.showConfirmDialog(null, message);
    }
}
```

Extract And Override – Breaking Point

```
@Test
void testCalc_IllegalInputs_ShouldRaiseException()
{
    final Calculator calculator = new Calculator()
    {
        @Override
        protected void showWarning(final String message)
        {
            // JOptionPane.showConfirmDialog(null, message);
        }
    };

    assertThrows(IllegalArgumentException.class, () -> calculator.calc("a2", "b3"));
}
```



- Pretty good workaround for many minor difficulties
- BUT: What do we do if we don't have access to the source code?


JUNIT
&
MOCKITO

Mocking Motivation

- Classes are often executed in the context of other classes. In general, we therefore often have the challenge in unit testing that a class depends on one or more others.

- It would now be extremely time-consuming, possibly even almost impossible, and also undesirable to parameterize or initialize these appropriately. Always then the use of a mocking framework is appropriate.

- A classic example is the data access layer or an e-mail service. In both cases, one would like to be able to safely execute the unit tests without an external dependency.

- The corresponding mocks would be created as follows:

```
BookDAO mockedBookDAO = mock(BookDAO.class);
EMailService mockedEMailService = mock(EMailService.class);
```

Mocking in Maven/Gradle-Build

```xml
<dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>5.4.0</version>
        <scope>test</scope>
</dependency>

<dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-junit-jupiter</artifactId>
        <version>5.4.0</version>
        <scope>test</scope>
</dependency>
```

```
dependencies {
   testCompile 'junit:junit:4.13‘
   // Mockito
   testCompile "org.mockito:mockito-core:5.4.0"
   testCompile "org.mockito:mockito-junit-jupiter:5.4.0 "
}
```

Mockito Basics

- Our goal is to create a test double
- Starting point a simple class

```
public class Greeting
{
    public String greet()
    {
        return "Hello world!";
    }
}
```

- When greet() is called, a return value different from the original implementation should be delivered, for example "Changed by Mockito".

Mockito Basics

- `mock()` - create mock / stub
- `when()` and `thenReturn()` - describe behavior
- With Mockito you can obey the ARRANGE-ACT-ASSERT style

```
@Test
void testGreetingReturnValue()
{
    // Arrange
    final Greeting greeting = mock(Greeting.class);
    when(greeting.greet()).thenReturn("Changed by Mockito");

    // Act
    final String result = greeting.greet();

    // Assert
    assertEquals("Changed by Mockito", result);
}
```

Mockito Basics

- `mock()` - create mock / stub
- `when()` and `thenReturn()` - describe behavior
- With Mockito you can obey the ARRANGE-ACT-ASSERT style

```
@Test
void testGreetingReturnValue()
{
    // Arrange
    final Greeting greeting = mock(Greeting.class);
    when(greeting.greet()).thenReturn("Changed by Mockito");

    // Act
    final String result = greeting.greet();

    // Assert
    assertEquals("Changed by Mockito", result);
}
```

the test double created by `mock()` is not a mock but a stub

state-based testing

Mockito Basics

- Mocking is used for collaborators, so let's create a class that uses Greeting:

```
public class Application
{
    private final Greeting greeting;

    public Application(final Greeting greeting)
    {
        this.greeting = greeting;
    }

    public String generateMsg(final String name)
    {
        return greeting.greet(name);
    }
}
```

Specific returns and exceptions

- Specify sequential operations:
  - `when()`,`anyString()` and `thenReturn()` – specify behavior for all input values.
  - `when()`,parameter value and `thenReturn()` - specify behavior for specific input values
  - `when()` und `thenThrow()` – raise exception

```
@Test
public void testGreetingReturnValue()
{
    // Arrange
    final Greeting greeting = Mockito.mock(Greeting.class);

    // Attention: Sequence important
    when(greeting.greet(anyString())).thenReturn("Welcome to Mockito");
    when(greeting.greet("Mike")).thenReturn("Mister Mike");
    when(greeting.greet("ERROR")).thenThrow(new IllegalArgumentException());

    ...
```

Mockito Basics

```
@Test
public void testGreetingReturnValue()
{
    // Arrange
    final Greeting greeting = Mockito.mock(Greeting.class);

    // Attention: Sequence important
    when(greeting.greet(anyString())).thenReturn("Welcome to Mockito");
    when(greeting.greet("Mike")).thenReturn("Mister Mike");
    when(greeting.greet("ERROR")).thenThrow(new IllegalArgumentException());

    // Act
    final Application app = new Application(greeting);
    final String result1 = app.generateMsg("Mike");
    final String result2 = app.generateMsg("ABC");

    // Assert
    assertEquals("Mister Mike", result1);
    assertEquals("Welcome to Mockito", result2);
    assertThrows(IllegalArgumentException.class,
                 () -> greeting.greet("ERROR")); // => Exception
}
```

Subsequent return values

- Specify more than one return with `thenReturn()` simply comma-separated:

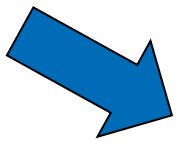
```
@Test
public void testGreetingMultipleReturns()
{
    // Arrange
    final Greeting greeting = mock(Greeting.class);
    when(greeting.greet(anyString())).
                  thenReturn("Hello Mockito1", "Hello Mockito2");

    // Act
    final Application app = new Application(greeting);
    final String result1 = app.generateMsg("One");
    final String result2 = app.generateMsg("Two");
    final String result3 = app.generateMsg("Three");

    // Assert
    assertEquals("Hello Mockito1", result1);
    assertEquals("Hello Mockito2", result2);
    assertEquals("Hello Mockito2", result3);
}
```

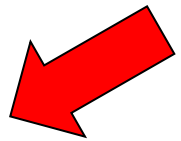
- **So far still all state-based! How do we check behavior in the form of calls?**

Checking calls

- **The `verify()` method for behavior-based testing, checks whether calls have been made:**

```
@Test
public void testVerifyCallsAndParams()
{
    // Arrange
    final Greeting greeting = mock(Greeting.class);
    when(greeting.greet(anyString())).thenReturn("Hello Mockito1", "Hello Mockito2");

    // Act
    final Application app = new Application(greeting);
    final String result1 = app.generateMsg("One");
    final String result2 = app.generateMsg("Two");
    final String result3 = app.generateMsg("Three");

    // Assert
    verify(greeting).greet("One");
    verify(greeting).greet("Two");
    verify(greeting).greet("Three");
    // verify(greeting).greet(anyString());
}
```





MockitoVerifyCallsAndParamsExample.java

Frequency of calls

- The methods `atLeast()`, `atMost()` and `times()` define how often a call is expected:
  - Minimum,
  - Maximum and
  - Exactly the specified number.

```
// Act
final Application app = new Application(greeting);
final String result1 = app.generateMsg("Tim");
final String result2 = app.generateMsg("Mike");
final String result3 = app.generateMsg("Tim");

// Assert
verify(greeting, atLeast(1)).greet("Tim");
verify(greeting, atMost(2)).greet("Tim");
verify(greeting, times(3)).greet(anyString());
```

InOrder() and the order of calls

```
@Test
public void testMethodCallsAreInOrder()
{
    ServiceClassA firstMock = Mockito.mock(ServiceClassA.class);
    ServiceClassB secondMock = Mockito.mock(ServiceClassB.class);

    Mockito.doNothing().when(firstMock).methodOne();
    Mockito.doNothing().when(secondMock).methodTwo();
    Mockito.doNothing().when(firstMock).methodThree();

    // InOrder records the order of the methods of the given mocks
    InOrder inOrder = Mockito.inOrder(firstMock, secondMock);

    // ACT
    firstMock.methodOne();
    secondMock.methodTwo();
    firstMock.methodThree();

    // Check that the calls occur in the correct order
    inOrder.verify(firstMock).methodOne();
    inOrder.verify(secondMock).methodTwo();
    inOrder.verify(firstMock).methodThree();
}
```

Mockito: It's the dose that counts ☺

http://clipart-library.com/fun-word-cliparts.html

https://www.severe-headache-expert.com/sharp-head-pain/

How to avoid (too) many mocks?

Example: Avoid mocks

```
public long placeOrder(long userId, Cart cart) {
    User user = userRepo.findById(userId);
    // heavy logic
    Order order = new Order();
    order.setDeliveryCountry(user.getAddress().getCountry());
    // more heavy logic
    orderRepo.save(order);
    return order.getId();
}
```

https://www.youtube.com/watch?v=1Z_h55jMe-M -- Victor Rentea - Unit Testing like a Pro: The Circle of Purity

Example: Avoiding mocks by method extraction

```
public long placeOrder(long userId, Cart cart) {
    User user = userRepo.findById(userId);
    Order order = createOrder(user, cart);
    orderRepo.save(order);
    return order.getId();
}

Order createOrder(User user, Cart cart) {
    // heavy logic
    Order order = new Order();
    order.setDeliveryCountry(
            user.getAddress().getCountry());
    // more heavy logic
    return order;
}
```

Mock-Free Unit Tests

= Pure Function

https://www.youtube.com/watch?v=1Z_h55jMe-M -- Victor Rentea - Unit Testing like a Pro: The Circle of Purity

Exercises Part 5 + 6

https://github.com/Michaeli71/AST-JUnit5

PART 7:
Test Smells

Test Smells High Level

- **Advice: Follow your nose ☺**

- Basically tests are also just source code like business code
- Test Smells are similar to Bad Smells / Code Smells (details in "Java-Profi")
- Bad Smell Examples: https://www.youtube.com/watch?v=9E6_zpx3q2c

- You can also find the following test smells on high level
    - Tests are difficult to write
    - Strange, complicated things needed to write / run tests
    - It needs a lot of mocking* (or even worse PowerMock for private / static mocking)
    - Testing sometimes even requires special handling in business code
    - Test execution is (too) slow
    - Test execution gives fluctuating results (random failures)

Test Smell		Symptom
Hard-to-Test Code		Code is difficult to test
Fragile Test		A test fails to compile or run when the SUT is changed in ways that do not affect the part the test is exercising.
Erratic Test		One or more tests are behaving erratically; sometimes they pass and sometimes they fail.
Obscure Test		It is difficult to understand the test at a glance
Assertion Roulette		It is hard to tell which of several assertions within the same test method caused a test failure.
Slow Tests		The tests take too long to run.
Test Code Duplication		The same test code is repeated many times **=> Does NOT apply to simple initializations.**
Test Logic in Production		The code that is put into production contains logic that should be exercised only during tests. **=> Does NOT apply to getter!* Anecdote Reflection !!!!**

http://xunitpatterns.com/Test%20Smells.html

Assertion Roulette / Obscure Test – Not SRP – Multiple Test Cases

```
@Test
public void testFlightMileage_asKm2() throws Exception
{
    // setup fixture
    String validFlightNumber = "LX 857";
    // exercise contructor
    Flight newFlight = new Flight(validFlightNumber);
    // verify constructed object
    assertEquals(validFlightNumber, newFlight.number);
    assertEquals("LX", newFlight.airlineCode);
    // setup mileage
    newFlight.setMileage(1111);
    int actualKilometres = newFlight.getMileageAsKm();
    // verify results
    int expectedKilometres = 1777;
    assertEquals(expectedKilometres, actualKilometres);
    // now try it with a canceled flight:
    newFlight.cancel();
    Throwable th = assertThrows(InvalidRequestException.class,
                                () -> newFlight.getMileageAsKm());
    assertEquals("Cannot get cancelled flight mileage", th.getMessage());
}
```

(Basierend auf http://xunitpatterns.com/Assertion%20Roulette.html)

Test Smell: Wrong use of `assertTrue()/False()`

```
assertTrue(db.writeCount == 10);
assertTrue(tasks.totalProcessed == 10);
assertTrue(tasks.errorCount == 0);
```

Test Smell: Wrong use of assertTrue()/False()

```
assertTrue(db.writeCount == 10);
assertTrue(tasks.totalProcessed == 10);
assertTrue(tasks.errorCount == 0);
```

```
assertEquals(10, db.writeCount);
assertEquals(10, tasks.totalProcessed);
assertEquals(0, tasks.errorCount);
```

Test Smell: ☹ assertTrue() forever? ☹

```
@Test
void assertTrueForever()
{
    final Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");
    final Person sameMike = mike;
    final Person otherMike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

    assertTrue(mike != null, "mike not null");
    assertTrue(mike == sameMike, "same obj");
    assertTrue(mike.equals(otherMike), "same content");
}
```

Test Smell: ☹ assertTrue() forever? ☹

```java
@Test
void assertTrueForever()
{
    final Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");
    final Person sameMike = mike;
    final Person otherMike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

    assertTrue(mike != null, "mike not null");
    assertTrue(mike == sameMike, "same obj");
    assertTrue(mike.equals(otherMike), "same content");
}
```

```java
@Test
void rightAssertsForTheJob()
{
    final Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");
    final Person sameMike = mike;
    final Person otherMike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

    assertNotNull(mike, "mike not null");
    assertSame(mike, sameMike, "same obj");
    assertEquals(mike, otherMike, "same content");
}
```

Test Smell: Too many Asserts

```
assertEquals(10, db.readCount);
assertEquals(10, db.writeCount);
assertEquals(10, db.commitCount);

assertEquals(10, tasks.totalProcessed);
assertEquals(0, tasks.errorCount);
assertEquals(SUCCESS, tasks.status);
```

Test Smell: Use of `toString()` in `assertEquals()`

```
assertEquals("mongodb.writeConcern.timeout=10000," +
             "mongodb.writeConcern.writes=1, " +
             "mongodb.port=27017, " +
             "mongodb.password=ksdjsa2455aAYdsj," +
             "mongodb.user=ABCD",
             dbConnection.toString())
```

Test Smell: Conditional Logic

```
@Test
void badConditionalLogic()
{
    final Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");
    final Person otherMike = new Person("Mike", LocalDate.of(1971, 2, 7), "Kiel");

    assertNotNull(mike, "mike not null");
    if (mike.getHomeTown().equals("Zürich"))
    {
        assertEquals(LocalDate.of(1971, 2, 7), mike.getDateOfBirth());
    }
    else
    {
        assertTrue(mike.equals(otherMike), "same content");
    }
}
```

Test Smell: Over Asserting

```java
@Test
void overAssertingAndLoosingHelpfulInfo()
{
    List<String> result = calcResultList();

    assertEquals(1, result.size());
    assertEquals("Tom", result.get(0)); // Will not be executed
}


private List<String> calcResultList()
{
    return List.of("Tom", "Jerry");
}
```

well meant != well done

org.opentest4j.AssertionFailedError: expected: <1> but was: <2>

Test Smell: Over Asserting

```
@Test
void overAssertingAndLoosingHelpfulInfo()
{
    List<String> result = calcResultList();

    assertEquals(1, result.size());
    assertEquals("Tom", result.get(0)); // Will not be executed
}
```

org.opentest4j.AssertionFailedError: expected: <1> but was: <2>

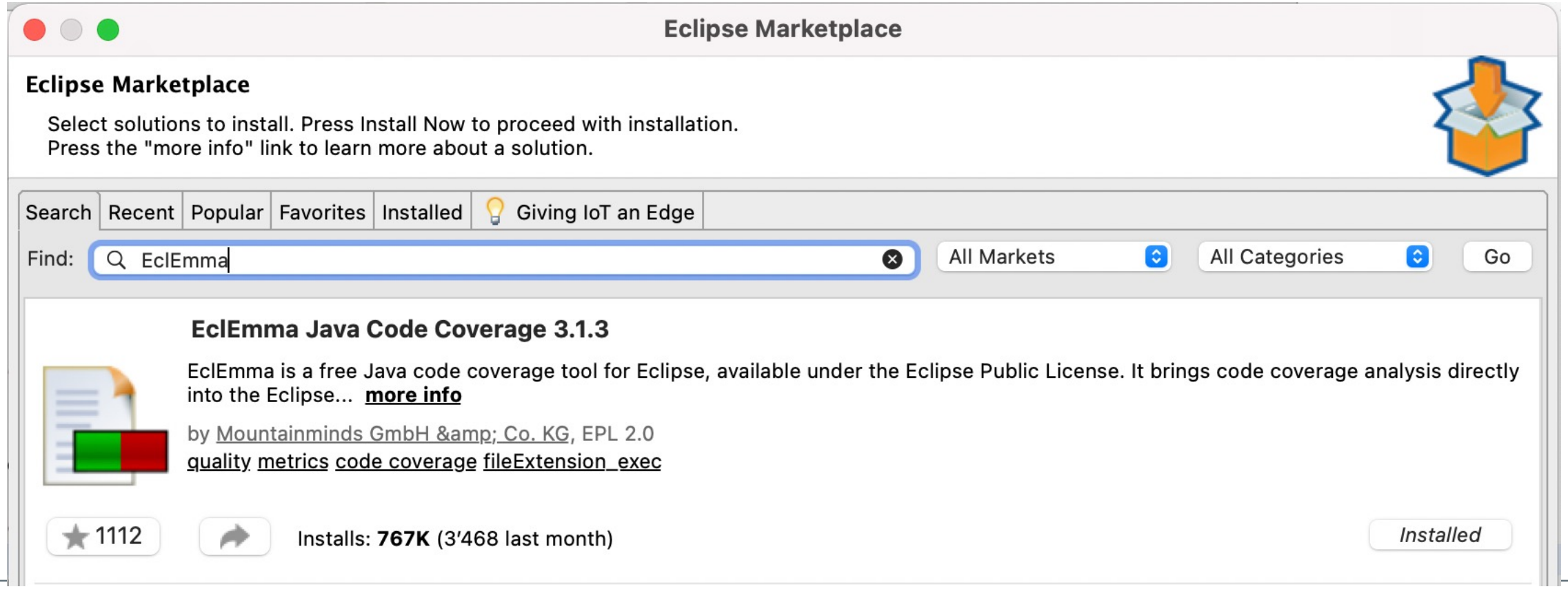
```
@Test
void reasonableAssertProvidingGoodFeedback()
{
    List<String> result = calcResultList();

    assertEquals(List.of("Tom"), result);
}
```

org.opentest4j.AssertionFailedError: expected: <[Tom]> but was: <[Tom, Jerry]>

PART 8:
Test Coverage

Test Coverage

- EclEmma plugin determines test coverage
- Test coverage = the source code passed by tests
- Freely available in the Marketplace



Test Coverage

- New execution mode named "Coverage", next to Debug and Run (or as Context menu)

- Thereby, during the execution of the program or test, the corresponding information about the executed source code parts is collected.
- The result is automatically displayed in a special coverage view at the end of the execution.
- Conveniently, from the project level down to individual Java methods, various metrics are determined and presented as desired: instructions, branches, cyclomatic complexity, etc.

EclEmma – Eclipse Plugin

- Configurable

Problems | Javadoc | Declaration | Console | Coverage

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
src/test/java	85.3 %	2'480	429	2'909
jugs.solutions.part2	81.1 %	557	130	687
slides.part2_junit5_intro	80.7 %	459	110	569
jugs.exercises.part2	69.7 %	221	96	317
jugs.solutions.part1	82.1 %	225	49	274
jugs.exercises.part1	58.8 %	50	35	85
slides.part3_junit5_advanced	99.2 %	890	7	897
slides.part5_testsmells	97.5 %	78	2	80
src/main/java	66.6 %	500	251	751
utils	54.7 %	82	68	150

274M of 591M

- ✓ Show Projects
- Show Package Roots
- Show Packages
- Show Types
- Instruction Counters
- Branch Counters
- Line Counters
- Method Counters
- Type Counters
- Complexity
- Hide Unused Elements

EclEmma – Eclipse Plugin

- Simple, fast, graphical, clear

Problems ■ Javadoc ■ Declaration ■ Search Console Progress Coverage

AAA_JUnit_5_Workshop (08.09.2020 18:22:27)

Element	Coverage	Covered Complexity	Missed Complexity	Total Complexity
AAA_JUnit_5_Workshop	76.6 %	820	250	1'070
src/main/java	64.2 %	194	108	302
domain	46.2 %	6	7	13
exercises.part3	52.6 %	40	36	76
utils	59.0 %	23	16	39
exercises.part7_8	60.4 %	29	19	48
exercises.part5_6	63.2 %	12	7	19
solutions.part3	67.9 %	36	17	53
solutions.part5_6	80.0 %	8	2	10
exercises.part2	85.0 %	17	3	20
solutions.part7_8	95.2 %	20	1	21
solutions.part2	100.0 %	3	0	3
src/test/java	81.5 %	626	142	768
x_slides.part5_6_dependencies_testability.testdoubles	26.7 %	4	11	15
x_slides.part5_6_dependencies_testability	58.3 %	14	10	24
x_slides.part7_testsmells	60.0 %	3	2	5
exercises.part7_8	61.5 %	16	10	26
exercises.part5_6.cheatsheet	65.7 %	23	12	35
exercises.part3	70.7 %	41	17	58
exercises.part5_6	75.0 %	12	4	16
a_questions	77.8 %	7	2	9
solutions.part7_8	79.3 %	111	29	140
x_slides.part5_6_dependencies_testability.step1	80.0 %	8	2	10
solutions.part3	84.7 %	105	19	124
x_slides.part5_6_dependencies_testability.step0	85.7 %	6	1	7
x_slides.part2_junit5_intro	86.6 %	71	11	82
solutions.part5_6	90.9 %	30	3	33
x_slides.part3_junit5_advanced	94.1 %	95	6	101
solutions.part2	95.1 %	58	3	61
exercises.part2	100.0 %	22	0	22

EclEmma – Eclipse Plugin

- Practical decorator
- Untested code directly visible
- Through percentages => "urgency" recognizabl

Test Coverage

- Successful editor integration, there the test coverage is color-coded:
  - Green: fully executed / covered
  - Yellow: partially executed / covered
  - Red: not executed / covered

DiscountCalcula | Ex01_Calculator | Ex01_Calculator | DynamicTests.ja | Ex02_MatchingBr | Ex02_MatchingBr | Ex02_Matchin

```
1  package jugs.ex03;
2
3  public class DiscountCalculatorCorrected
4  {
5      public int calcDiscount(final int count)
6      {
7          if(count < 0)
8              throw new IllegalArgumentException("Count must be positive");
9
10         if (count < 50)
11             return 0;
12         if (count >= 50 && count <= 1000)
13             return 4;
14         if (count > 1000)
15             return 7;
16
17         throw new IllegalStateException("programming problem: should never " +
18                 "reach this line. value " + count + " is not handled!");
19     }
20 }
```

Problems | Javadoc | Declaration | Search | Console | Coverage | SonarLint Rule Description

Test Coverage

- **In general: The shorter and fewer parameters and branches a method has, the better a high test coverage can be achieved for it.**
- **This is achieved almost automatically by following the basic rules of good OO design:**
  - Keep Cyclomatic Complexity low
  - Adhere to SRP
- **If one avoids the Smells**
  - Long Parameter List
  - Long Method

Special feature I: Why 80% and not 100%?

Ex01_MatchingBr | ScoreCalculator | ScoreCalculator

```
1  package jugs.solutions.part6;
2
3  public class ScoreCalculator
4  {
5      public static String calcScore(int score)
6      {
7          if (score < 45)
8              return "fail";
9          else
10         {
11             if (score > 95)
12                 return "pass with distinction";
13
14             return "pass";
15         }
16     }
17 }
18
```

jugs.solutions.part6	80.0 %	12	3	15
ScoreCalculator.java	80.0 %	12	3	15
ScoreCalculator	80.0 %	12	3	15
calcScore(int)	100.0 %	12	0	12

Feature I: Why not 100%?

Ex01_MatchingBr | ScoreCalculator | ScoreCalculator

```
1  package jugs.solutions.part6;
2
3  public class ScoreCalculator
4  {
5      public static String calcScore(int score)
6      {
7          if (score < 45)
8              return "fail";
9          else
10         {
11             if (score > 95)
12                 return "pass with distinction";
13
14             return "pass";
15         }
16     }
17 }
18
```

```
// ATTENTION: is generated automatically
public ScoreCalculator()
{
    super();
}
```

jugs.solutions.part6	80.0 %	12		3	15
ScoreCalculator.java	80.0 %	12		3	15
ScoreCalculator	80.0 %	12		3	15
calcScore(int)	100.0 %	12		0	12

Feature I: Private constructor => 100%

Ex01_MatchingBr | ScoreCalculator | ScoreCalculator

```
1  package jugs.solutions.part6;
2
3  public class ScoreCalculator
4  {
5      public static String calcScore(int score)
6      {
7          if (score < 45)
8              return "fail";
9          else
10         {
11             if (score > 95)
12                 return "pass with distinction";
13
14             return "pass";
15         }
16     }
17 }
18
```

```
// Util class should not have a
// default ctor!
private ScoreCalculator()
{
}
```

jugs.solutions.part6	100.0 %	12	0	12
ScoreCalculator.java	100.0 %	12	0	12
ScoreCalculator	100.0 %	12	0	12
calcScore(int)	100.0 %	12	0	12

Code Coverage != Software Quality

Special feature II: 100 % test coverage no functionality tested

```
class SpecialCounter
{
    private int count;

    public void countIfHundredOrAbove(final int value)
    {
        if (value >= 100)
        {
            count++;
        }
    }

    public void reset()
    {
        count = 0;
    }

    public int currentCount()
    {
        return count;
    }
}
```

Idee basiert auf https://www.oracle.com/corporate/features/mutation-testing.html

Special feature II: 100 % test coverage no functionality tested

```
public class SpecialCounterTest
{
    // VERY BAD "TEST" ... 100% coverage, but NO semantic test
    @Test
    @DisplayName("Boss says he wants 100% coverage. Here you go!")
    public void veryBadTrickyAssertNothing()
    {
        SpecialCounter counter = new SpecialCounter();

        counter.reset();
        counter.countIfHundredOrAbove(111);
        counter.countIfHundredOrAbove(99);

        counter.currentCount();
    }
}
```

Special features III: Despite 100% test coverage, one NPE

```
public class Coverage
{
    public String coverage100ButNPE(final boolean condition)
    {
        String value = null;
        if (condition)
        {
            value = String.valueOf(condition);
        }
        return value.trim();
    }
    // ...
```

100% test coverage but NPE

```
public class CoverageTest
{
    final Coverage coverage = new Coverage(4711, "4711");

    @Test
    public void testCoverage100ButNPE1()
    {
        coverage.coverage100ButNPE(true);
    }

    @Test
    public void testCoverage100ButNPE2()
    {
        // Löst eine NullPointerException aus
        coverage.coverage100ButNPE(false);
    }
}
```

JaCoCo – Test coverage with Gradle and Maven

- Executable in combination with tests, produces HTML report
- Gradle

```
gradle test
open build/reports/jacoco/test/html/index.html
```

- Maven

```
mvn test
open target/site/jacoco/index.html
```

- **Interestingly, the reports differ slightly in the results?!**

JaCoCo – Test coverage with Gradle and Maven

Mail - Michael | Speedtest DSL | Michaeli71/JAX | JDT Core/Plan/J | Amazon.de Best | jacoco maven - | Intro to JaCoCo

file:///Users/michaeli/Desktop/Vorträge/ZZZ_JUnit5/ZZZ_CH_OPEN_2020_JUnit5_Workshop/ZZZZZ_CH_O

Meistbesucht | Erste Schritte | Apple | iCloud | Yahoo | Bing | Google | Wikipedia | Facebook | Twitter | LinkedIn | Wetter Online | Yelp

ZZZZZ_CH_OPEN_JUnit5Example

ZZZZZ_CH_OPEN_JUnit5Example

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
exercises.part7_8		58%		63%	19	48	31	96	9	26	1	7
exercises.part3		78%		58%	36	76	36	154	8	26	1	9
utils		64%		69%	16	39	25	56	10	21	0	7
solutions.part3		76%		76%	17	53	17	66	4	14	1	5
domain		70%		50%	7	13	4	19	1	7	0	1
exercises.part5_6		84%		n/a	7	19	7	35	7	19	3	9
exercises.part2		90%		100%	3	20	4	34	3	18	0	5
solutions.part5_6		85%		n/a	2	10	2	18	2	10	0	3
solutions.part7_8		100%		96%	1	21	0	36	0	6	0	4
solutions.part2		100%		n/a	0	3	0	8	0	3	0	1
Total	544 of 2'239	75%	94 of 304	69%	108	302	126	522	44	150	6	51

Exercises Part 7 + 8

https://github.com/Michaeli71/AST-JUnit5

Part 9:
Characterization Testing
aka Pinning Test / Pin-Down Tests

Characterization Testing

- **Legacy code … big chunk of code but no tests!**
- **How to pin down / nail down the behaviour of this method?**

```
public static String formatText(String text)
{
  StringBuffer result = new StringBuffer();
  for (int n = 0; n < text.length(); ++n) {
    int c = text.charAt(n);
    if (c == '<') {
      while(n < text.length() && text.charAt(n) != '/' && text.charAt(n) != '>')
        n++;
      if (n < text.length() && text.charAt(n) == '/')
        n+=4;
      else
        n++;
    }
    if (n < text.length())
      result.append(text.charAt(n));
  }
  return new String(result);
}
```

https://michaelfeathers.silvrback.com/characterization-testing

Characterization Testing

- What about even larger chunks of code?

```
public class TravelsAdapter {
    public List<Travel> adapt(JsonNode jsonNode) throws InvalidTravelException {
        List<Travel> travels = new ArrayList<>();
        JsonNode payloadNode = jsonNode.with("data");
        if (payloadNode.findValue("orderId") == null ||
                StringUtils.isBlank(payloadNode.findValue("orderId").textValue())) {
            throw new InvalidTravelException("Invalid order id");
        }
        long orderId = payloadNode.findValue("orderId").asLong();
        JsonNode flights = payloadNode.withArray("flights");
        if (flights.size() == 0) {
            throw new InvalidTravelException("Invalid json (no flights)");
        }
        flights.iterator().forEachRemaining(flight -> {
            ObjectNode nodeFlight = (ObjectNode) flight;
            if (nodeFlight.get("flightId") == null || StringUtils.isBlank(nodeFlight.get("flightId").textVal
                try {
                    throw new InvalidTravelException("Invalid flightNumber value");
                } catch (InvalidTravelException e) {
                    e.printStackTrace();
                }
            }
            String flightNumber = nodeFlight.get("flightId").textValue();
            String arrivalAirport = nodeFlight.get("to").textValue();
            String departureAirport = nodeFlight.get("from").textValue();
            String airline = nodeFlight.get("airline").textValue();
            travels.add(new Travel(
                            orderId,
                            flight.toString(),
                            flightNumber,
                            airline,
                            departureAirport,
                            arrivalAirport));
        });
        return travels;
    }
}
```

How to describe the behaviour if you just have the source code?

Just execute with different inputs and observe the results!

Characterization Testing

- **The purpose of characterization testing is to document your system's actual behavior, not check for the behavior you wish your system had.**
- **More concrete:**
  - Create test methods
  - Step 1: "silly result"

```
@Test
public void formatsPlainText1() {
    assertEquals("XXXX", UnknownFormatterFunctionality.formatText("plain text"));
}
```

```
public static String formatText(String text)
{
  StringBuffer result = new StringBuffer();
  for (int n = 0; n < text.length(); ++n) {
    int c = text.charAt(n);
    if (c == '<') {
      while(n < text.length() && text.charAt(n) != '/' && text.charAt(n) != '>')
        n++;
      if (n < text.length() && text.charAt(n) == '/')
        n+=4;
      else
        n++;
    }
    if (n < text.length())
      result.append(text.charAt(n));
  }
  return new String(result);
}
```

https://michaelfeathers.silvrback.com/characterization-testing

Characterization Testing

- **Just execute with different inputs and observe the result**
- **More concrete:**
  - Create test methods
  - Step 1: "silly result"
  - Step 2: adjust "silly result" to what is delivered by method

```
@Test
public void formatsPlainText2() {
    assertEquals("plain text", UnknownFormatterFunctionality.formatText("plain text"));
}
```

```
public static String formatText(String text)
{
  StringBuffer result = new StringBuffer();
  for (int n = 0; n < text.length(); ++n) {
    int c = text.charAt(n);
    if (c == '<') {
      while(n < text.length() && text.charAt(n) != '/' && text.charAt(n) != '>')
        n++;
      if (n < text.length() && text.charAt(n) == '/')
        n+=4;
      else
        n++;
    }
    if (n < text.length())
      result.append(text.charAt(n));
  }
  return new String(result);
}
```

https://michaelfeathers.silvrback.com/characterization-esting

Characterization Testing

- **Just execute with different inputs and observe the result**
- **More concrete:**
  - Create test methods
  - Step 1: "silly result»
  - Step 2: adjust "silly result" to what is delivered by method
  - Step 3: adjust test name to gain better understanding

```
@Test
public void doesNotChangeUntaggedText() {
    assertEquals("plain text", UnknownFormatterFunctionality.formatText("plain text"));
}
```

```
public static String formatText(String text)
{
  StringBuffer result = new StringBuffer();
  for (int n = 0; n < text.length(); ++n) {
    int c = text.charAt(n);
    if (c == '<') {
      while(n < text.length() && text.charAt(n) != '/' && text.charAt(n) != '>')
        n++;
      if (n < text.length() && text.charAt(n) == '/')
        n+=4;
      else
        n++;
    }
    if (n < text.length())
      result.append(text.charAt(n));
  }
  return new String(result);
}
```

https://michaelfeathers.silvrback.com/characterization-testing

Characterization Testing … other trys

```
@Test
public void emptyInput() {
    assertEquals("XXXX", UnknownFormatterFunctionality.formatText("<>"));
}
```

org.opentest4j.AssertionFailedError: expected: <XXXX> but was: <>

```
@Test
public void removesTagTextBetweenAngleBracketPairs() {
    assertEquals("XXXX", UnknownFormatterFunctionality.formatText("<TAGGED>"));
}
```

org.opentest4j.AssertionFailedError: expected: <XXXX> but was: <>

```
@Test
public void justKeepsContent() {
    assertEquals("XXXX", UnknownFormatterFunctionality.formatText("<TAG>CONTENT<TAG>"));
}
```

org.opentest4j.AssertionFailedError: expected: <XXXX> but was: <CONTENT>

https://michaelfeathers.silvrback.com/characterization-testing

Characterization Testing … adjusted

```
@Test
public void emptyInput2() {
    assertEquals("", UnknownFormatterFunctionality.formatText("<>"));
}

@Test
public void removesTagTextBetweenAngleBracketPairs2() {
    assertEquals("", UnknownFormatterFunctionality.formatText("<TAGGED>"));
}

@Test
public void justKeepsContent2() {
    assertEquals("CONTENT",
                 UnknownFormatterFunctionality.formatText("<TAG>CONTENT<TAG>"));
}
```

https://michaelfeathers.silvrback.com/characterization-testing

Demo

Characterization Testing

- https://michaelfeathers.silvrback.com/characterization-testing
- https://www.artima.com/weblogs/viewpost.jsp?thread=198296
- https://daedtech.com/characterization-tests/
- https://www.fabrizioduroni.it/2018/03/20/golden-master-test-characterization-test-legacy-code/
- https://freol35241.medium.com/pinning-tests-in-python-using-pytest-c678a2a3cb3b

Questions?

Hilfe


HELP
IS ON THE
WAY


5.
Auflage
Michael Inden
Der Weg zum
Java-Profi
Konzepte und Techniken für die
professionelle Java-Entwicklung
dpunkt.verlag


Java
Michael Inden
Challenge
Fit für das Job-Interview und die Praxis –
mit mehr als 100 Aufgaben
und Musterlösungen
dpunkt.verlag

Weitere Infos / Quellen

- **JUnit 5**
  - https://junit.org/junit5/
  - https://jaxenter.de/highlights-junit-5-65986
  - https://jaxenter.de/junit-5-beyond-testing-framework-81787
  - https://gul.gu.se/public/pp/public_courses/course82759/published/1524658283418/resourceId/40520654/content/junit-tdd-mocking.pdf
  - https://www.viadee.de/wp-content/uploads/JUnit5_javaspektrum.pdf

- **AssertJ**
  - https://assertj.github.io/doc/
  - https://joel-costigliola.github.io/assertj/assertj-core-quick-start.html
  - https://www.vogella.com/tutorials/AssertJ/article.html
  - https://dzone.com/articles/assertj-and-collections-introduction
  - https://de.slideshare.net/tsveronese/assert-j-techtalk (Hamcrest vs. AssertJ)

Thank You