JUnit 5 Workshop
Quick Refresher

Mehr Spass und weniger Bauchschmerzen beim Entwickeln durch clevere Tests

Michael Inden

https://github.com/Michaeli71/ADC_TESTING_XMAS_SPECIAL

Agenda

Workshop Contents

- **PART 1: Einführung Testing**
  - Warum testen?
  - Gute Angewohnheiten

- **PART 2: JUnit 5 Intro**
  - Architektur
  - First Test
  - Tests mit mehreren Asserts
  - Testing Exceptions
  - Tests mit Timeouts

Workshop Contents

- **PART 3: JUnit 5 Advanced**
  - Parameterized Tests
  - Simple Extensions
- **PART 4: Tipps zur Migration JUnit 4 => JUnit 5**
  - Migration oder / und Parallelbetrieb
  - AssertJ
- **PART 5: Testweisen und Abhängigkeiten**
  - Zustandsbasiertes vs. Verhaltensbasiertes Testen
  - Stellvertreterobjekte

https://github.com/Michaeli71/ADC_TESTING_XMAS_SPECIAL

Workshop Contents

- **PART 6: Design For Testability**
  - Sollbruchstellen
  - Extract and Override
  - Mockito
- **PART 7: Test Smells**
- **PART 8: Test Coverage**

PART 1:
Warum testen und gute Angewohnheiten

Warum testen wir?

- Gewünschtes Verhalten beschreiben
- Funktionalität prüfen (auch Randfälle)
- Sicherheitsnetz aufbauen
- Qualitätssicherung
- Kundenzufriedenheit
- weniger Ärger, Nerven und mehr Spass

External Quality

External Quality entspricht Benutzersicht

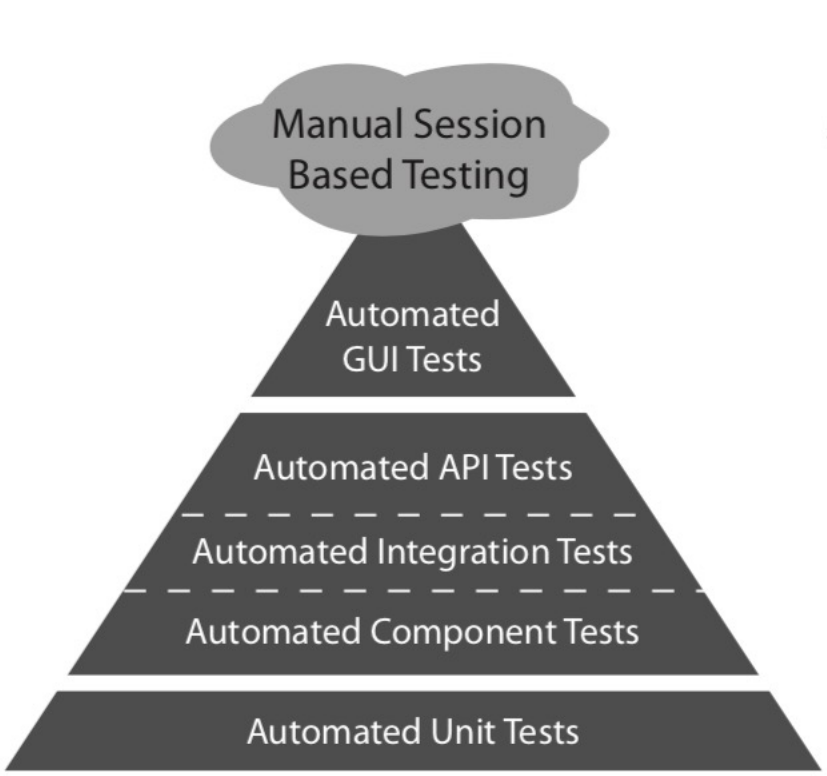
- Arbeitet wie erwartet
- Stellt alle benötigte Funktionalität bereit
- Korrektheit
  - Nahezu keine (beobachtbaren) Bugs
  - Gut getestet
- Benutzbar
- Verlässlich
- …

Internal Quality

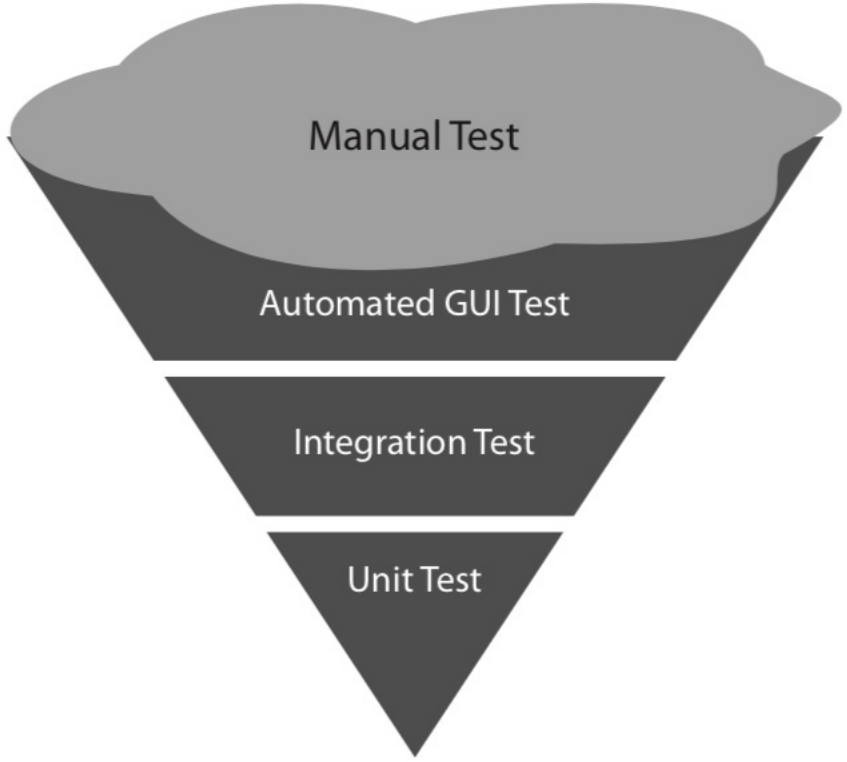
Internal Quality ~ Entwicklersicht (Code, Build, Testing):

- Lesbarkeit
- Verständlichkeit
- Keine Fallstricke und Hindernisse
- Erweiterbar, pflegbar
- Gut/sinnvoll dokumentiert
- Gute Test/Code Coverage
- …

Testpyramide

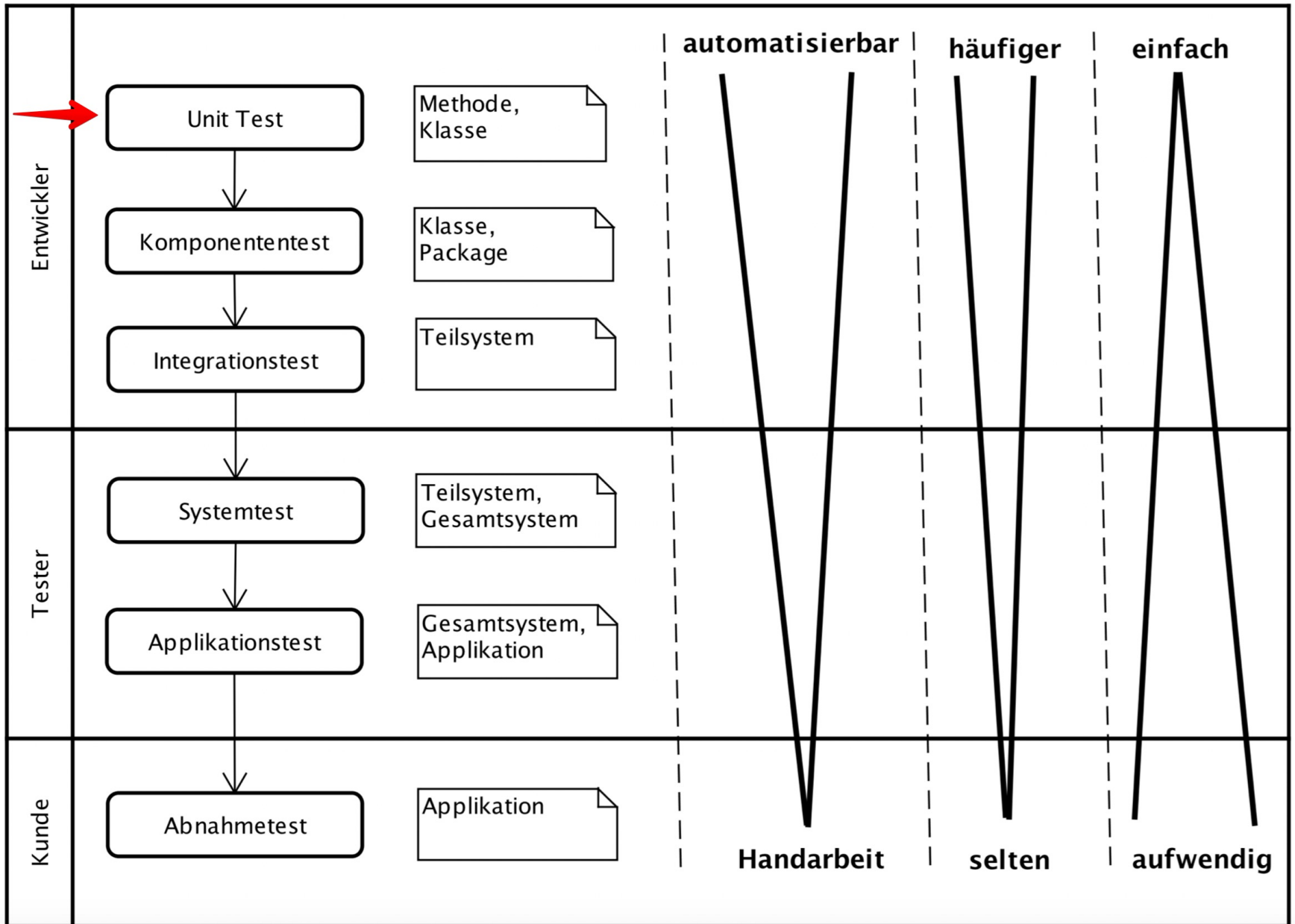


The Ideal Testing
Automation Pyramid



The Non-Ideal Testing
Automation Inverted Pyramid

Arten von Tests



Gute Angewohnheiten

Maven-Projektstruktur

```
.
├── build.gradle
├── 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
```

Namensgebung

- Klasse `Abc` => zugehörige Testklasse `AbcTest`

- Methoden:
  - Optional: Kürzel `test` als Start
  - Sinnvolle Beschreibung des Testfalls:
  - Methodenname, Bedingungen und Ergebnis im Namen kodieren => **CamelCase wird oft unleserlich**
  - Testing Guru Roy Osherove schlägt Folgendes vor

```
MethodName_StateUnderTest_ExpectedBehavior
MethodName_ExpectedBehavior_WhenTheseConditions
```

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

Testfälle definieren

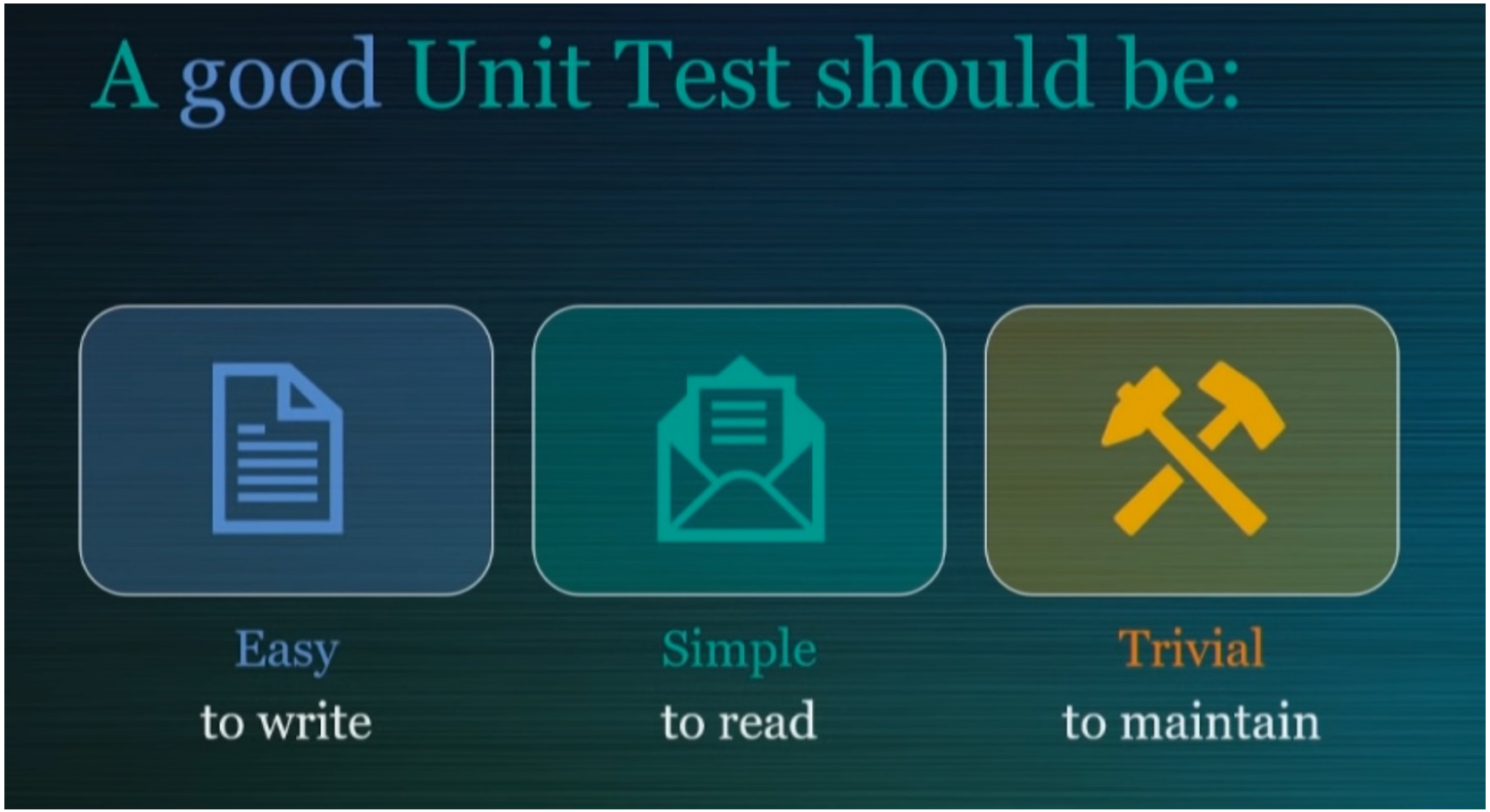
- Unit Tests **prüfen kleine Bausteine**, meistens Klassen oder Methoden
- **Möglichst isoliert** und ohne Interaktion mit anderen Komponenten
- Tests werden **in Form von Methoden** implementiert
- Im Idealfall: Für jede ***relevante*** **Applikationsmethode** mindestens **eine Testmethode**
- Eine Testmethode **prüft genau eine Funktionalität** (oder nur einen Teil davon) , **idealerweise nur 1 ASSERT !**
- Testmethoden **kurz, klar und verständlich** halten
- **ABER: Wie erreicht man das?**

AAA-Stil

- **ARRANGE** - **ACT** – **ASSERT** (Auch GWT genannt für GIVEN – WHEN – THEN)
- **ARRANGE**: Vorbedingungen und Initialisierungen (***Testfixture***)
- **ACT**: danach wird eine Aktion ausgeführt
- **ASSERT**: Prüfen, ob der erwartete Zustand eingetreten ist

```
@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");
}
```

Was macht einen guten Unit Test aus?



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

FAIR - Gewünschte Eigenschaften von Unit Tests

F – Fast, Focussed

A - Automated

I - Isolated

R – Reliable, Repeatable

PART 2:
JUnit 5 Intro

JUnit 5

JUnit 5

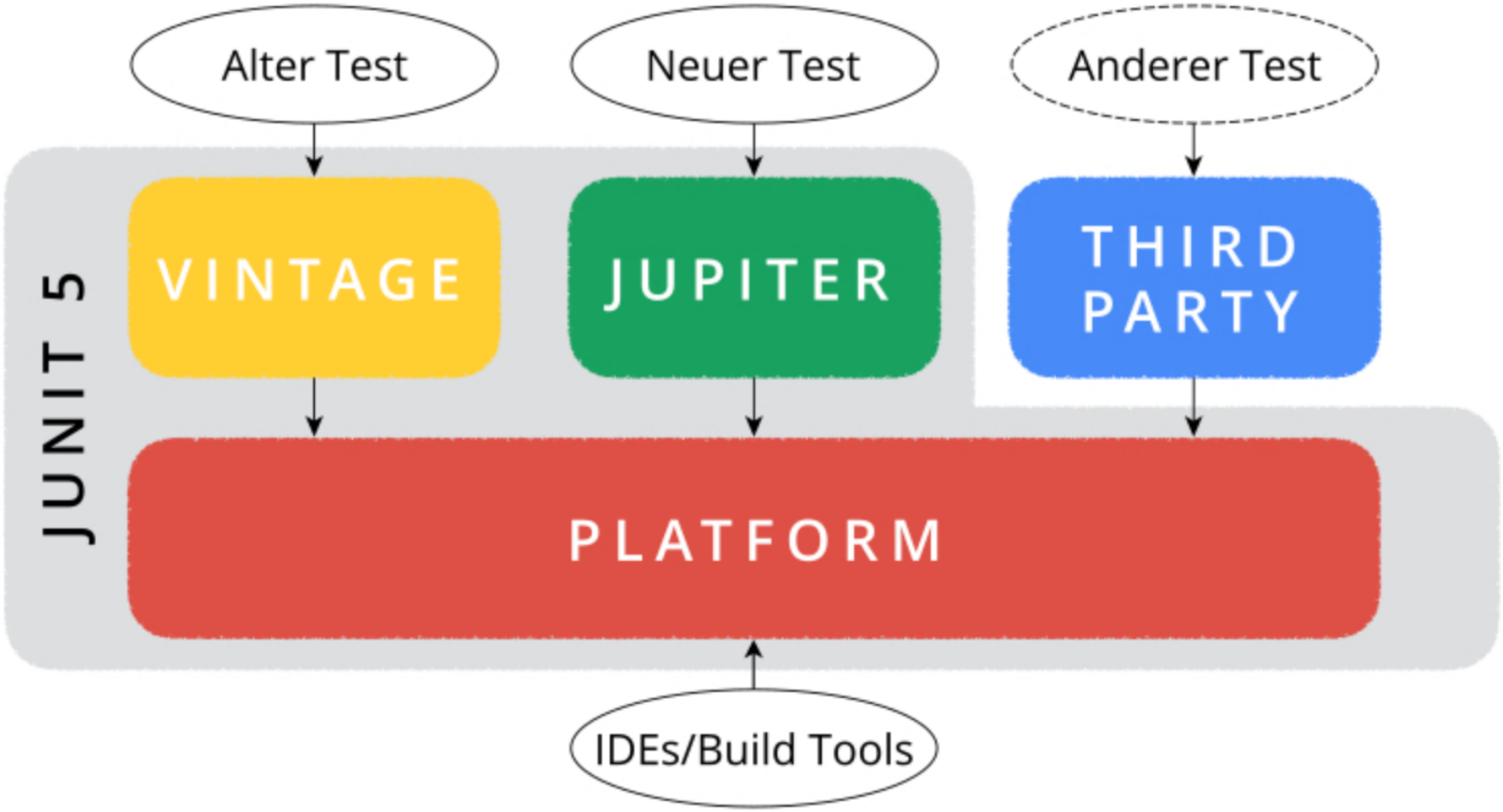
JUnit 4

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

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

Architektur

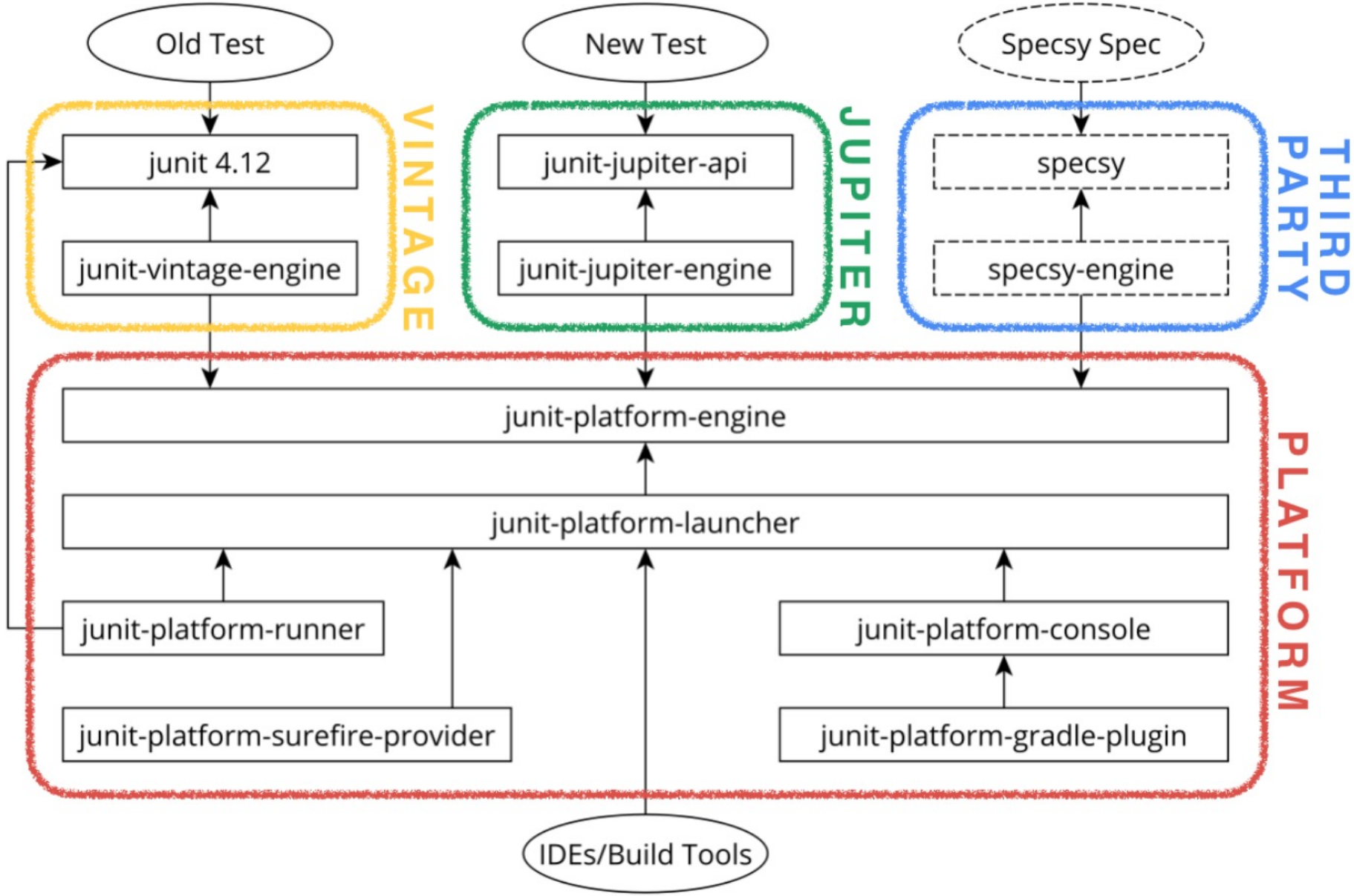
- JUnit 5 =
  JUnit Platform +
  JUnit Jupiter +
  JUnit Vintage



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

Architektur

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



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

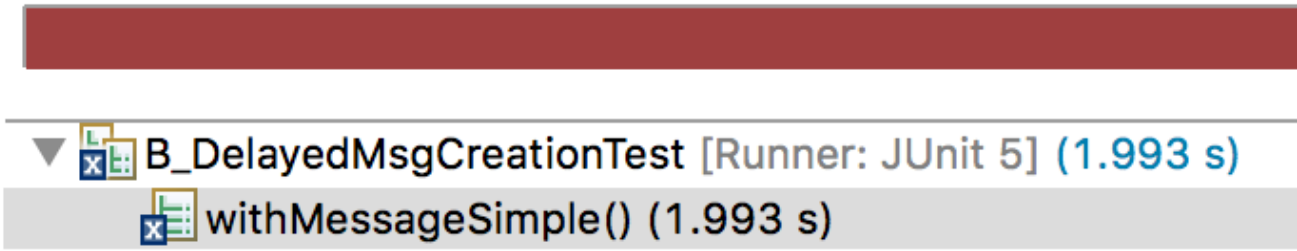
Ein erster Unit Test mit JUnit 5

- Testfälle in Form spezieller Testmethoden erstellt, die mit der Annotation `@Test` markiert

```
@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());
}
```

Komplexe Erzeugung von Messages

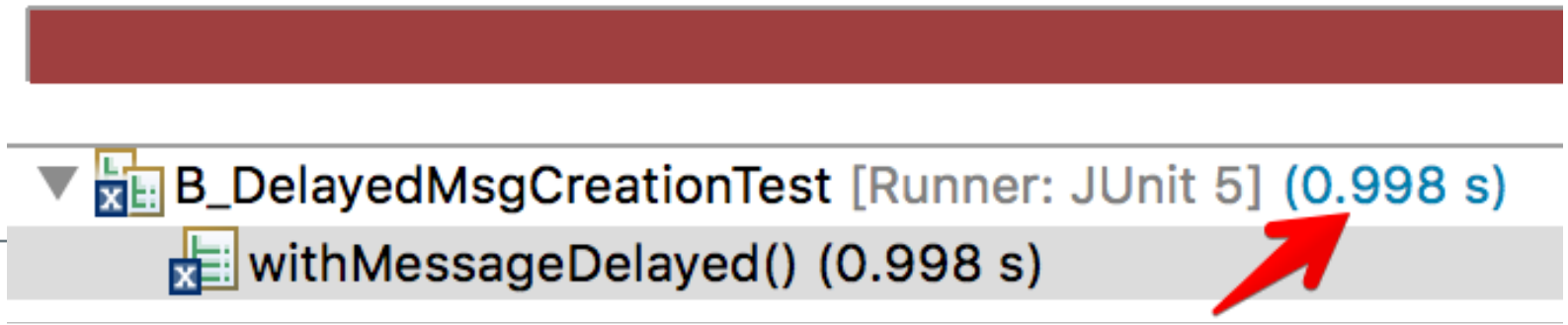
```
@Test
void withMessageSimple()
{
    String expected = "Tim";

    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;
}
```



Was ist daran unschön?

Komplexe Erzeugung von Messages (nur bei Bedarf)

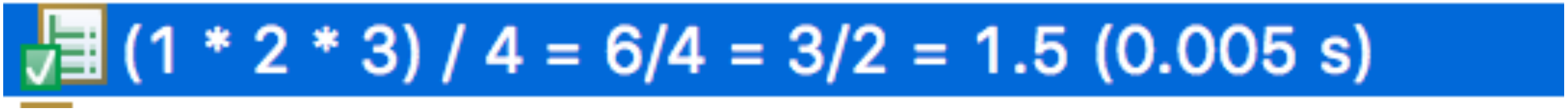
```
@Test
void withMessageDelayed()
{
    String expected = "Tim";

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



Spezielle Testnamen

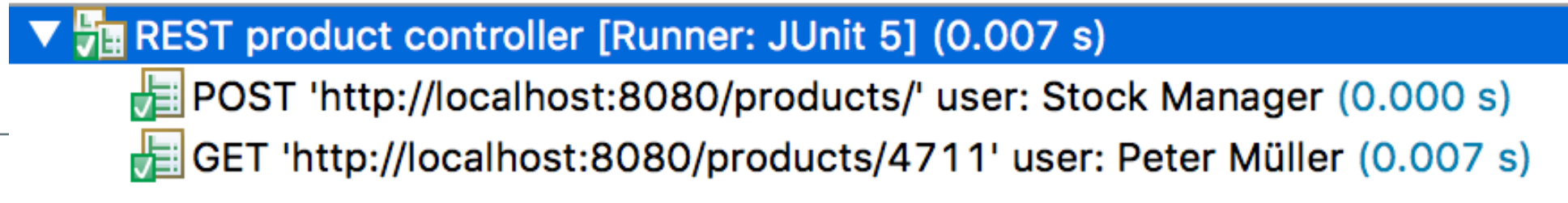
- Mit JUnit 4 war man auf valide Methodennamen eingeschränkt
- Spezielle Testnamen mit der Annotation **`@DisplayName`**

```
@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);
}
```



Spezielle Testnamen

```
@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()
    {
        // ...
    }
}
```



Spezielle 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));
}
```

Wo liegt der Unterschied?

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
```

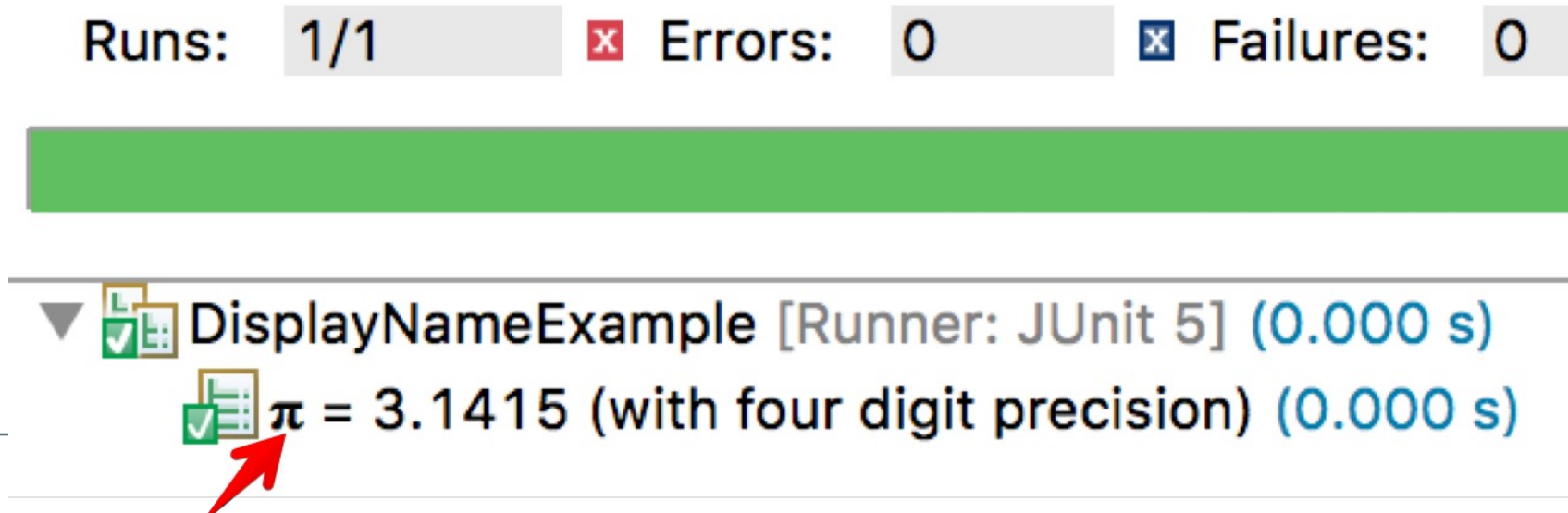
```java
@Test
void multipleAssertsforOneTopic_Diff2() {

    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>
```

Wie testen wir Gleitkommazahlen?

Spezielles Handling bei Gleitkommazahlen

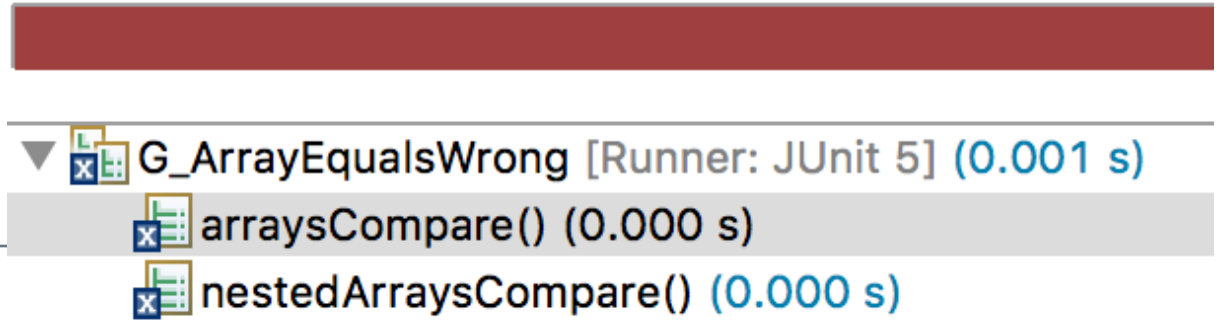
```
@Test
@DisplayName("\uD835\uDED1 = 3.1415 (with four digit precision)")
void floatingArithemticRoundingForPI()
{
    double value = calculatePI();
    double precision = 0.0001;

    assertEquals(3.1415, value, precision);
}

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



Wie testen wir Arrays?

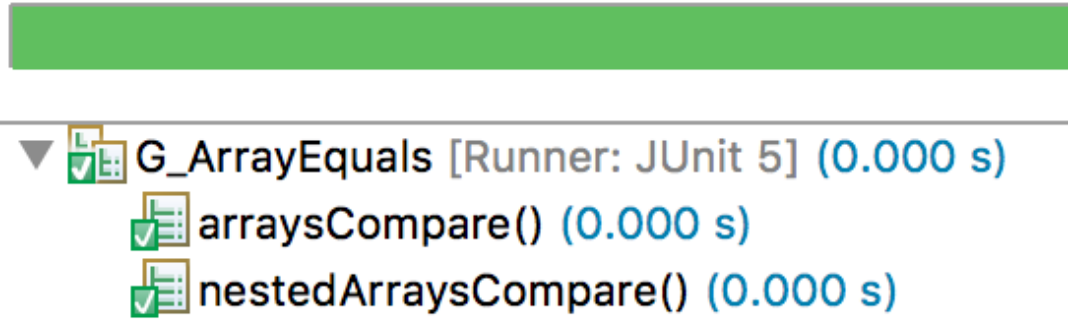
Arrays vergleichen – UPS!

```
@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);
}
```



Arrays RICHTIG vergleichen

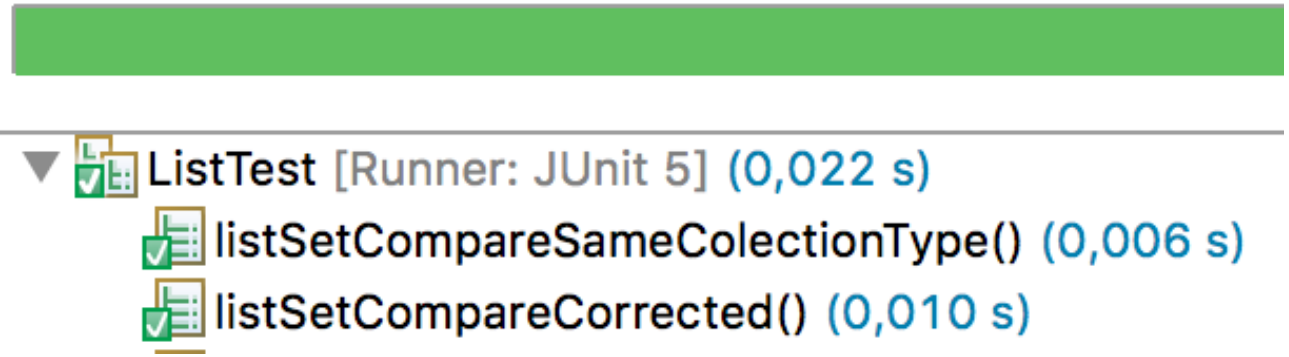
```
@Test
void arraysCompare()
{
    final String[] words = { "Word1", "Word2" };
    final String[] expected = { "Word1", "Word2" };

    assertArrayEquals(expected, words);
}

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

    assertArrayEquals(expected, nested);
}
```



Wie testen wir Collections?

Collections vergleichen – problemlos?!

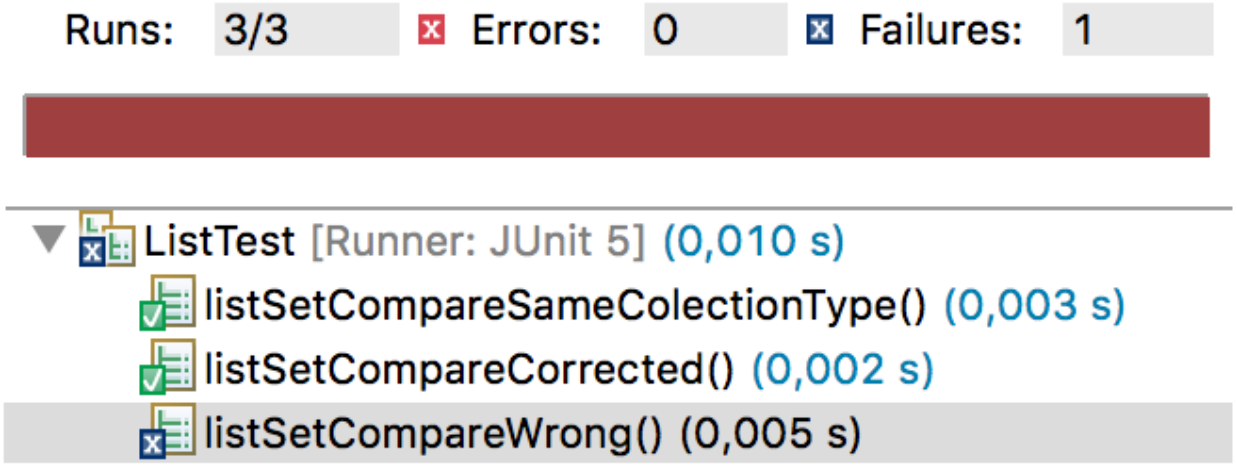
```java
@Test
void listSetCompareSameColectionType()
{
    final Collection<String> tags = new HashSet<>(Set.of("Fast", "Cool"));
    final Collection<String> names = List.of("Tim", "Mike", "Tom");

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



Collections vergleichen – Was tun bei abweichendem Typ

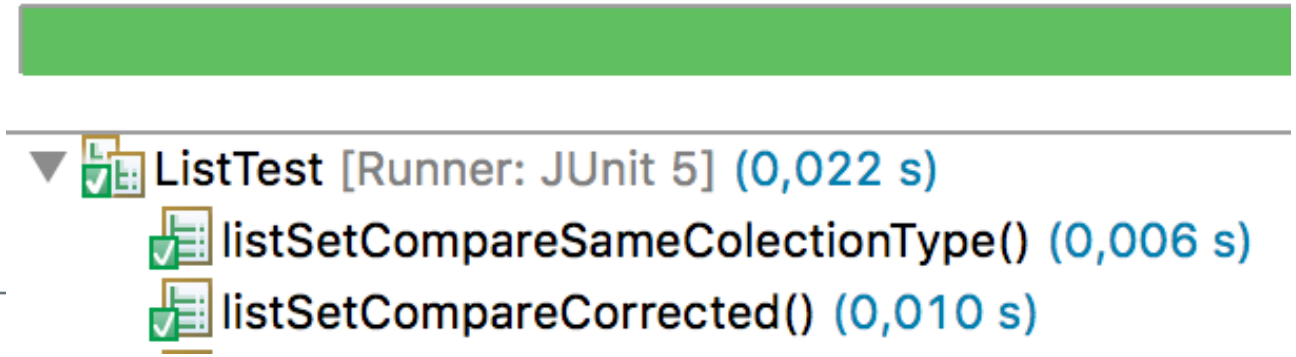
```java
@Test
public void listSetCompareWrong()
{
    final List<String> expected = List.of("a", "b", "c", "d");
    final Set<String> actual = new TreeSet<>(Set.of("c", "a", "d", "b"));

    // Set und List vergleichen
    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]>



Collections SICHERER vergleichen

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

    // Set und List basierend auf Iterable vergleichen
    assertIterableEquals(expected, actual);
}
```



Testing Exceptions

Handarbeit

- Manchmal sollen Testfälle das Auftreten von Exceptions prüfen

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

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() mit Rückgabe

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

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

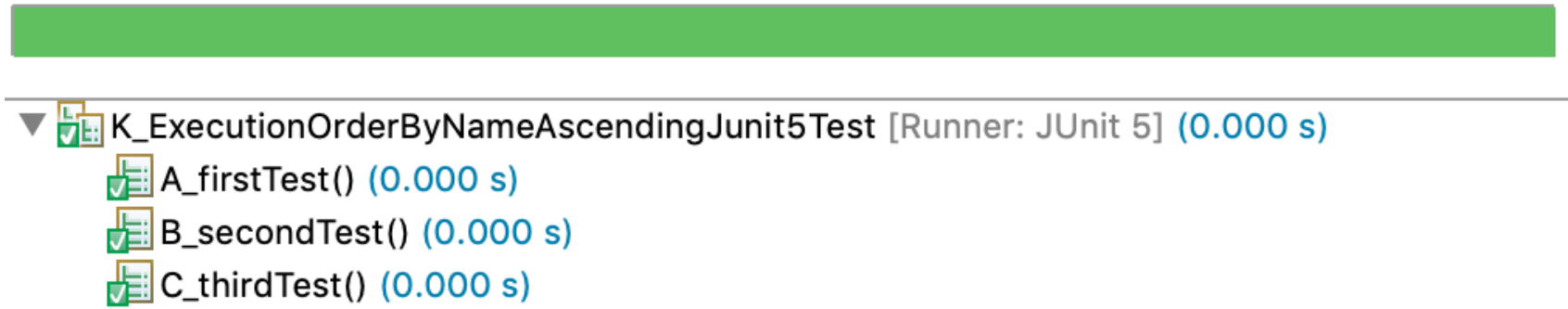
JUnit 5: assertThrows() mit Rückgabe

```
@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");
}
```

Ausführungsreihenfolge

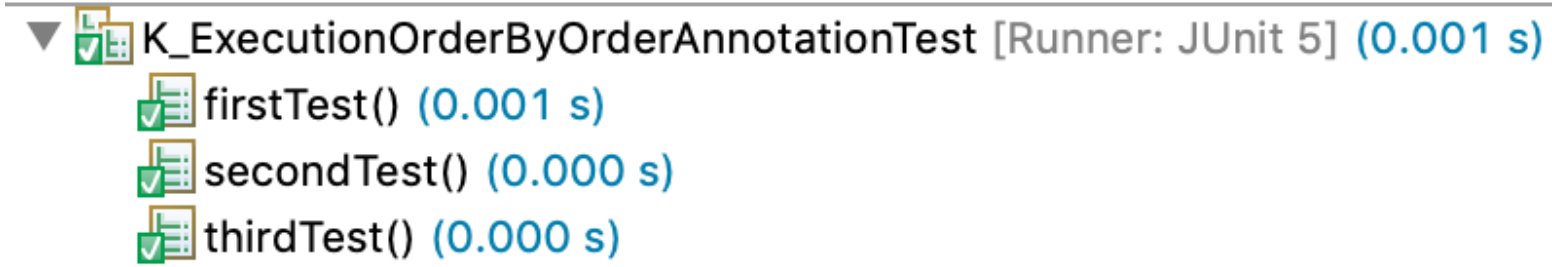
JUnit 5 Test Methoden ordnen (@TestMethodOrder)

```
@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 Test Methoden ordnen (@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());
    }
```

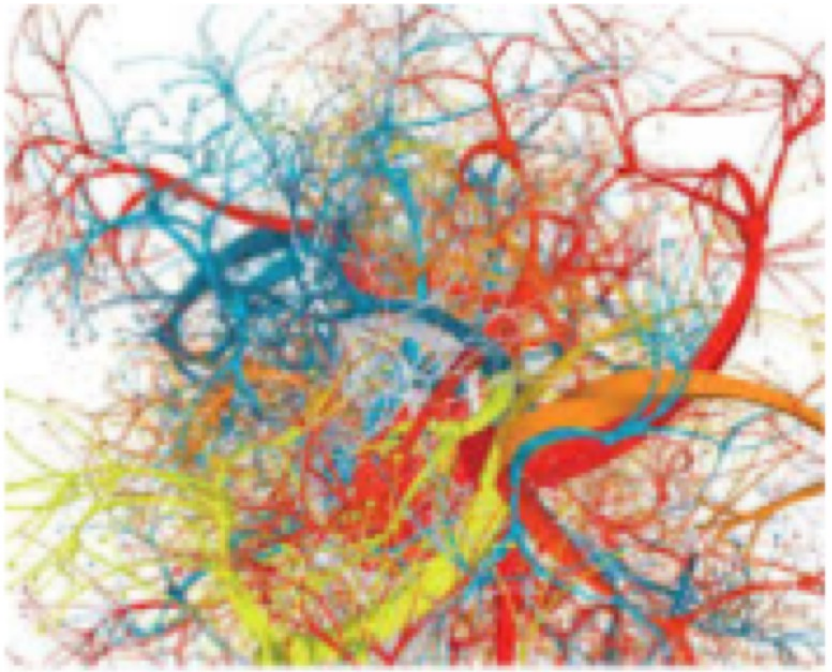


Exercises Part 2

https://github.com/Michaeli71/ADC_TESTING_XMAS_SPECIAL

PART 3
JUnit 5 Advanced

Kombinatorik / Komplexität



Komplexität

3 zentrale Fragen

1. Welche Wertebelegungen soll man testen?

2. Wie vermeidet man zu viel Aufwand?

3. Wie finde ich diejenigen Testfälle, die eine gute und sichere Aussage über die Qualität und die Funktionalität ermöglichen?

Komplexität

- **Welche Wertebelegungen soll man testen?**
  - Selbst bei zwei `int` => $2^{32} * 2^{32} = 2^{64}$ Kombinationen
- **Wie vermeidet man zu viel Aufwand?**
  - Die wichtigen / komplexen Dinge testen
  - Keine Getter / Setter testen
  - Geschickte Wahl von Eingaben, so dass viele Varianten abgeprüft werden
- **Wie finde ich diejenigen Testfälle, die eine gute und sichere Aussage über die Qualität und die Funktionalität ermöglichen?**
  - **Äquivalenzklassentest**
  - **Grenzwerttest**

Äquivalenzklassen

- Gruppierung von Eingaben: Verschiedene Werte => gleiches Ergebnis
- Typisches Beispiel: Rabattberechnung

Wertebereich	Rabatt
count < 100	0 %
100 <= count <= 1000	4 %
count > 1000	7 %

- **Wie viele und welche Äquivalenzklassen ergeben sich?**

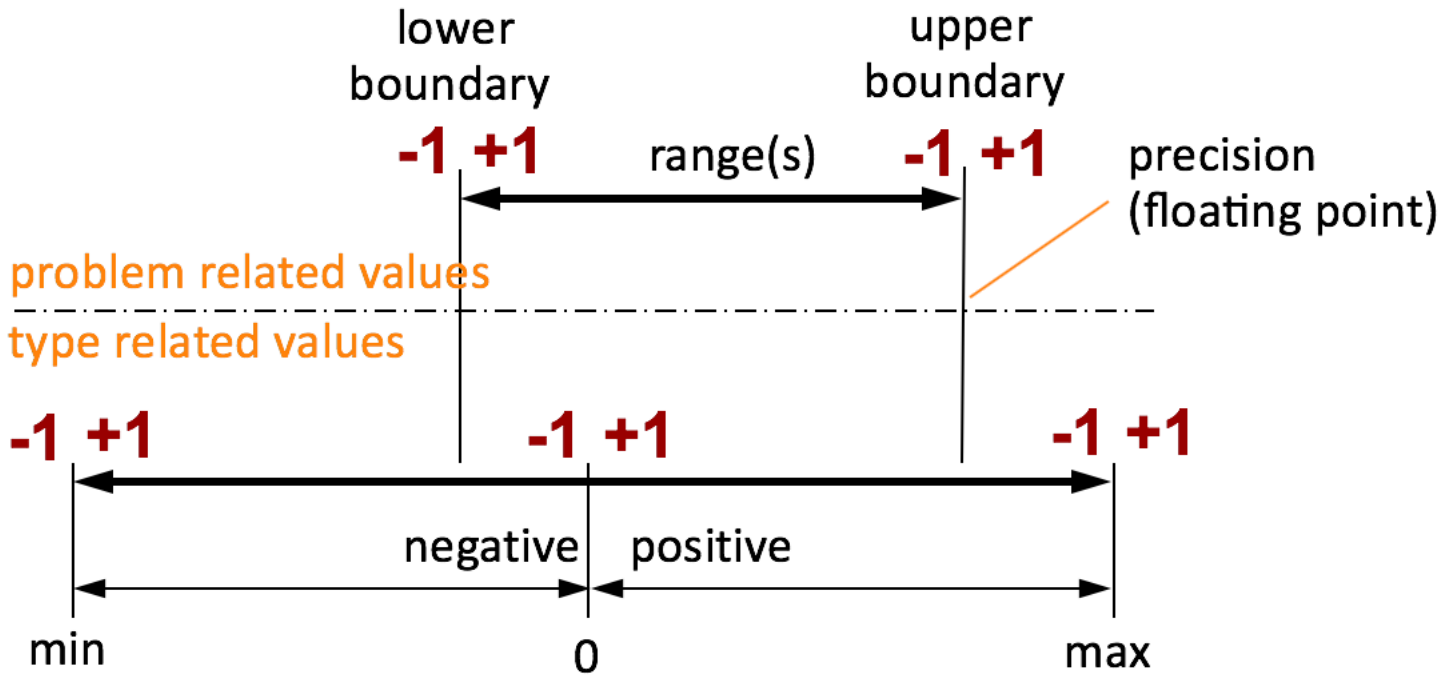
Äquivalenzklassentest

- **Schreiben wir also 3 Testmethoden. Aber: Reichen diese Tests aus?**

```
@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");
}
```

Grenzwerttests

- **NEIN!** Die Erfahrung aus der Praxis zeigt, man benötigt neben Äquivalenzklassentests noch weitere, warum?
- Oftmals finden wir **an den Rändern noch Probleme**, also im Übergang der Wertebereiche:



Grenzwerttests

- **Für die Rabattberechnung finden wir an den Rändern noch Probleme**, also im Übergang der Wertebereiche, hier also
  - 99, 100, 101
  - 999, 1000, 1001
- Wieso? Oftmals Fehler bei Vergleichen mit `<  <=  ==  !=  >=  >`
- Weitere potenzielle Kandidaten sind:
  - Werte < 0 oder
  - Werte > als ein vorgesehenes Maximum

Sollen wir etwa für alle diese Werte einzelne Methoden schreiben?

☹ ☹ ☹

Parameterized Tests

Parameterized Test

- **Abhilfe durch sogenannte Parameterized Test**
- **Testfall mit verschiedenen Daten immer wieder mit neuer Wertebelegung auszuführen**
- **Dadurch alle gewünschten, zu prüfenden Kombinationen abdecken**

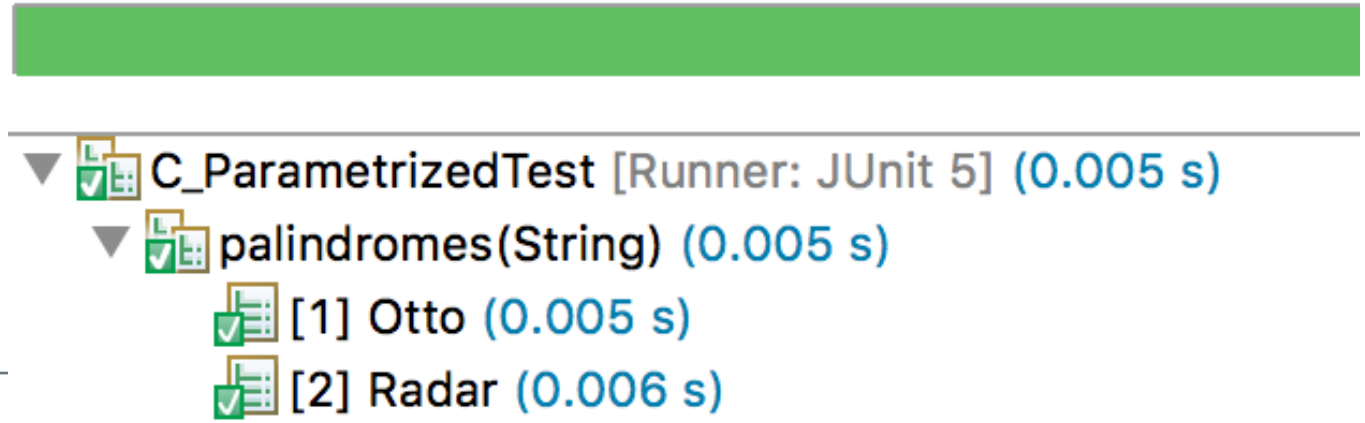
- **Realisierungsvarianten**
  - Handarbeit: for-Schleife: liefert nur sukzessive Ergebnisse
  - JUnit 4 krampfig, syntaktisch unschön
  - JUnit 4 mit Expected Exception besser, aber wieder einiges an Eigenarbeit
  - JUnit 5 **endlich gut**

Parameterized Test: Kurzer Blick zurück: for-Schleife

```
@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));
	}
}
```

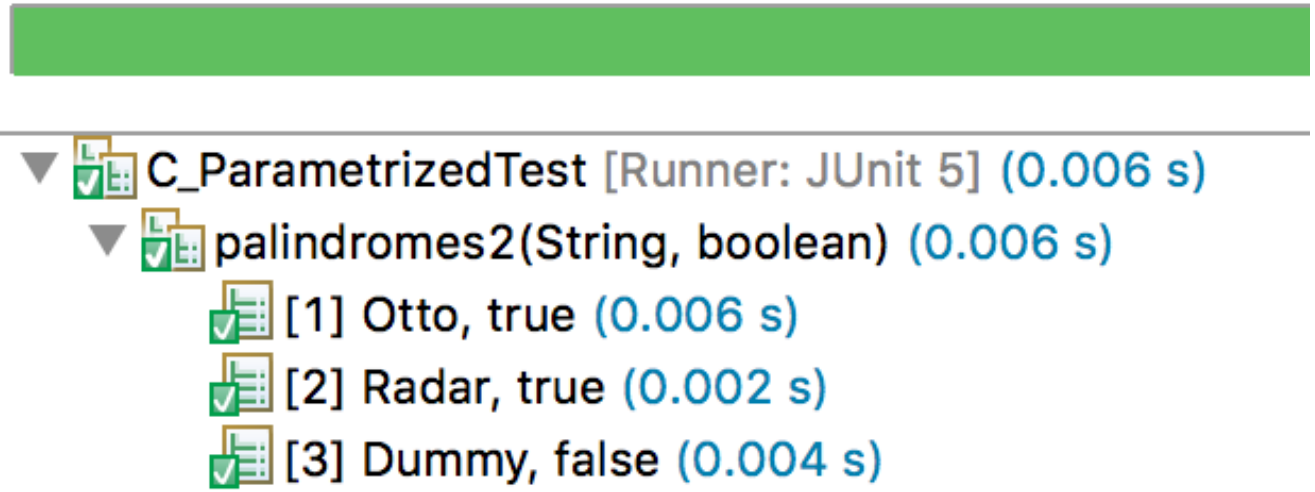
Parameterized Test – JUnit 5 @ParameterizedTest / @ValueSource

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

    assertTrue(isPalindrome);
}
```



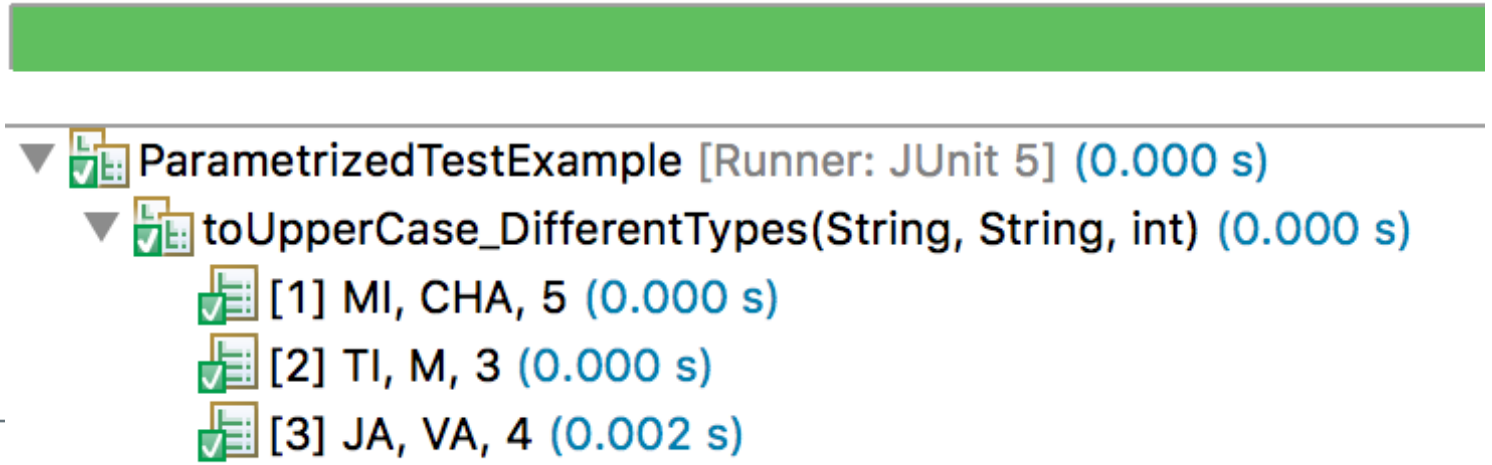
Parameterized Test – @CsvSource

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

    assertEquals(expected, isPalindrome);
}
```



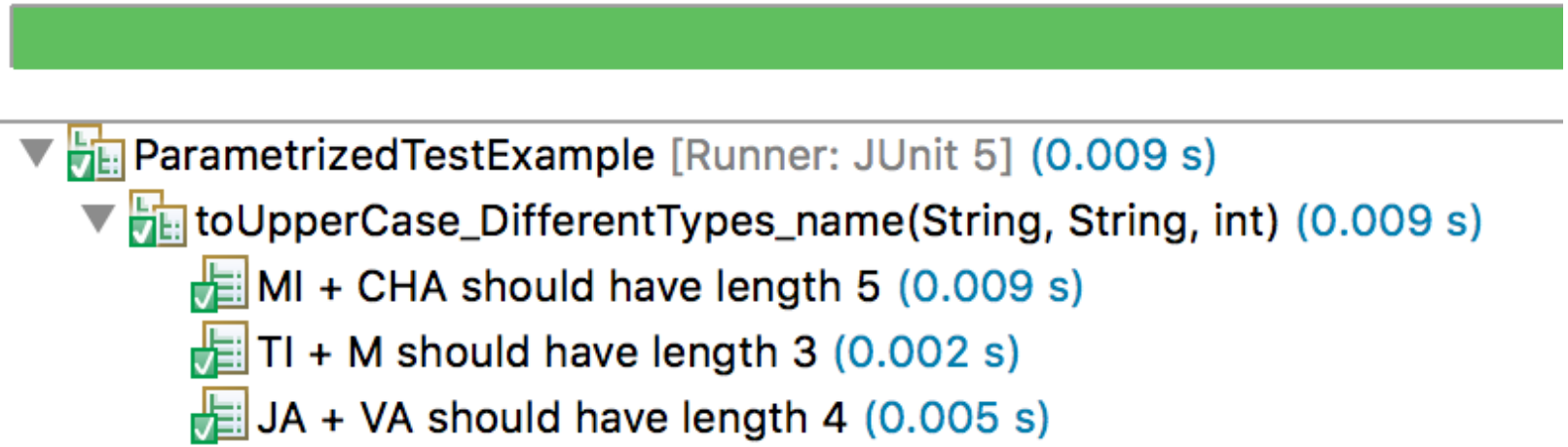
Parameterized Test – mehrere Eingaben verschiedene Typen

```
@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);
}
```



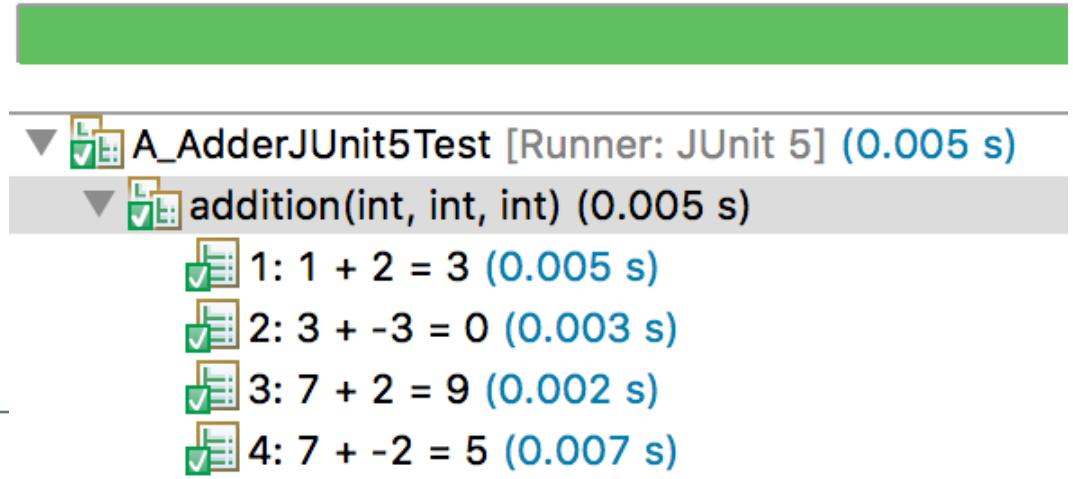
Parameterized Test – bessere Benennung

```
@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);
}
```



Parameterized Test – Adder Test & bessere Benennung

```
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);
    }
}
```

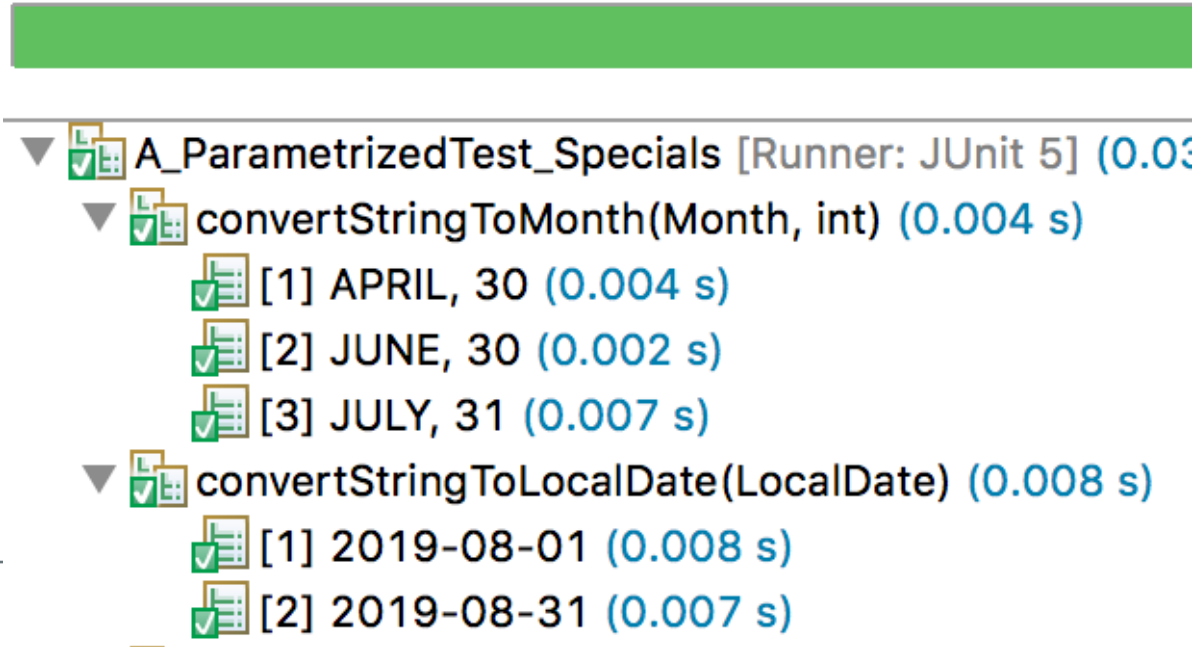


Parameterized Test – Diverse Konvertierungen und Hilfen

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

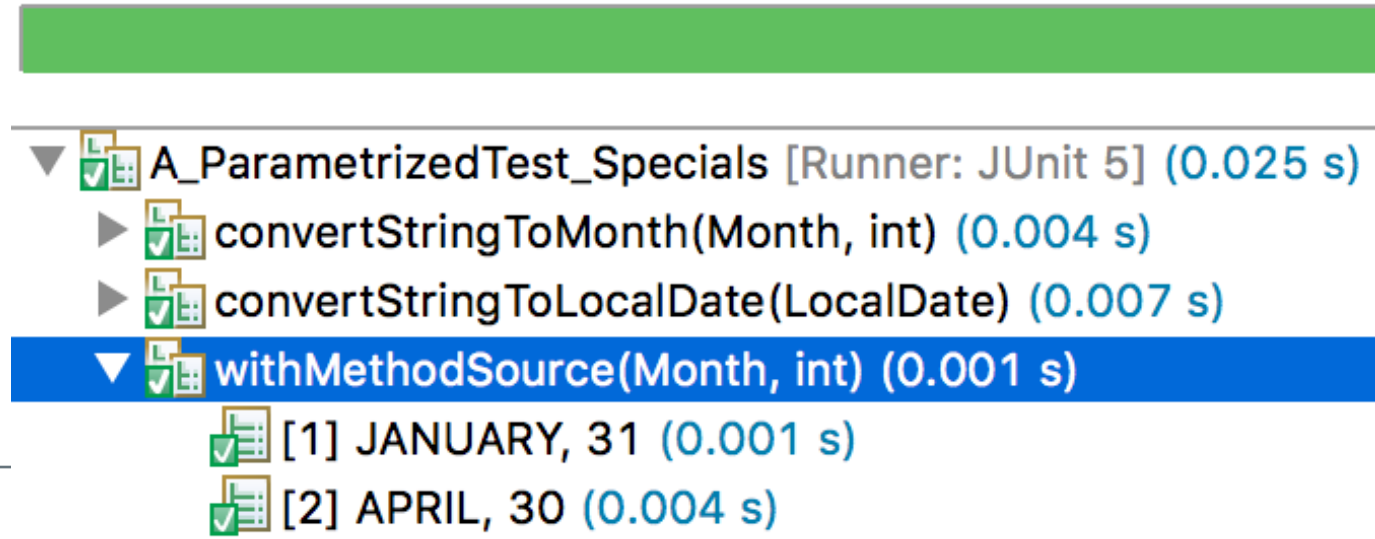
```
@ParameterizedTest
@ValueSource(strings = { "2019-08-01", "2019-08-31" })
void convertStringToLocalDate(LocalDate localDate)
{
    assertEquals(Month.AUGUST, localDate.getMonth());
}
```

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



Parameterized Test – @MethodSource

```
@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));
}
```



Was lässt sich mit Parameterized Test denn noch so machen?

Parameterized Test – @CsvSource, z. B. für grosse Datenmengen

```
@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("Konvertiere römische in arabische Zahl")
void fromRomanNumber(final int arabicNumber, final String romanNumber)
{
    int result = RomanNumbers.fromRomanNumber(romanNumber);

    assertEquals(arabicNumber, result);
}

// wie oben
void toRomanNumber(final int arabicNumber, final String romanNumber)
{
    int result = RomanNumbers.toRomanNumber(arabicNumber);

    assertEquals(romanNumber, result);
}
```

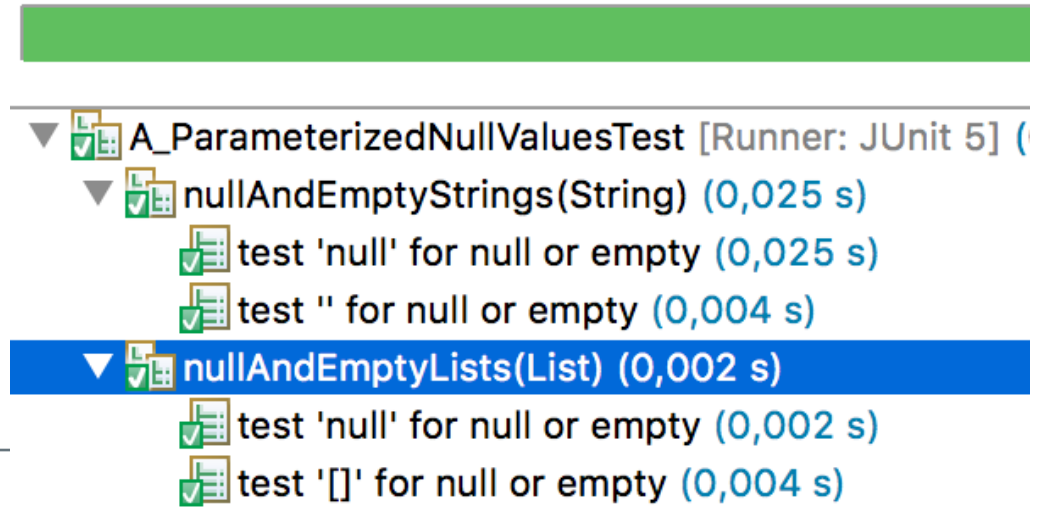
Parameterized Test – @CsvSource, z. B. für große Datenmengen

```java
@ParameterizedTest(name = "fromRomanNumber(''{1}'') => {0}")
@CsvFileSource(resources = "arabicroman.csv", numLinesToSkip = 1)
@DisplayName("Konvertiere römische in arabische Zahl")
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 – Randfälle prüfen

```
@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 – @MethodSource, z. B. für Listen als Parameter

```
@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)));
}
```

Wäre es nicht cool JSON verarbeiten zu können?

@Parameterized Test – Trick: Argument Converter für «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: '2.8.7'
```

PART 4:
Tipps zur Migration

Wissenswertes zum direkten Starten

- Vermutlich habt ihr schon eine größere Basis an Tests => **Migrationsplan** nötig
- **Migration** oder / und **Parallelbetrieb** möglich: **Parallelbetrieb** bietet sich an
- Praktisch: Zwar ähnliche Annotations aber in anderen Packages
- JUnit 4 parallel zu JUnit 5:

```
// JUnit 4 Support
testCompile "junit:junit:4.13.2"
testRuntime "org.junit.vintage:junit-vintage-engine:5.8.0"
```

- Einige JUnit 4 Rules nutzen:

```
// Migration Support to Enable Rules in JUnit 5
testCompile "org.junit.jupiter:junit-jupiter-migrationsupport:5.8.0"
```

Wissenswertes zum direkten Starten

- Beide: Testfälle in Form spezieller Testmethoden, die mit `@Test` markiert sein müssen.

- JUnit 4:
  - Testmethoden `public`
  - *Keine* Parameter erlaubt
  - Package: `org.junit`
  - Parameterreihenfolge: `assertTrue("Always true", true);`
  - `assertThat()` und einige Hamcrest-Matcher inkludiert

- JUnit 5:
  - Testmethoden nicht mehr zwingend `public`
  - Kann Parameter haben
  - Package: `org.junit.jupiter.api`
  - Parameterreihenfolge: `assertTrue(true, "Always true");`
  - *Kein* `assertThat()` und *keine* Hamcrest-Matcher

Annotations JUnit 4 vs JUnit 5

JUnit 4	JUnit 5		Beschreibung
`org.junit`	`org.junit.jupiter.api`		Package
`@Test`	`@Test`		Definiert einen Testfall
`@Ignore`	`@Disabled`		Testfall (temporär) deaktivieren
`@BeforeClass`	`@BeforeAll`		Aktion ***einmalig*** vor allen Testmethoden
`@Before`	`@BeforeEach`		Aktion ***jeweils*** vor allen Testmethoden
`@After`	`@AfterEach`		Aktion ***jeweils*** nach allen Testmethoden
`@AfterClass`	`@AfterAll`		Aktion ***einmalig*** nach allen Testmethoden
`@Rule`	`-/-`		Erweiterungen, in JUnit 5 mit speziellen Methoden

Umstellung auf JUnit 5

- **Auf lange Sicht ist eine Migration / Umstellung auf JUnit 5 ratsam**
- **Schrittweise** Klasse für Klasse, Package für Package
  - Imports löschen und auf Package: `org.junit.jupiter.api` anpassen
  - Annotations ggf. leicht anpassen @BeforeXXX, @AfterXXX
  - Parameterreihenfolge beachten
  - `@Rule ExpectedException` durch JUnit-5-Features ersetzen, z. B. `assertThrows()`
  - `@Rule TimeOut` durch JUnit-5-Features ersetzen, z. B. `assertTimeout()`

Umstellung auf JUnit 5

- Schrittweise Klasse für Klasse, Package für Package
  - @EnableRuleMigrationSupport aus "org.junit.jupiter:junit-jupiter-migrationsupport:5.8.0" einbinden
    - TemporaryFolder
    - ErrorCollector
    - ExpectedException
  - `assertThat()` durch `AssertJ` oder `Hamcrest` ersetzen

Was bringt mir AssertJ?

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

- JAR in die IDE aufnehmen / Build-Datei anpassen

```
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 – Vorteile bei Strings gegenüber JUnit assertXyz()

- JUnit 5 bietet lediglich die Prüfung auf (Nicht-)Übereinstimmung bietet
- gerade bei Strings in der Praxis wünschenswert, lesbar prüfen zu können, ob ein String nicht leer ist oder gewünschte Zeichen enthält.

```
@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 – Vorteile bei Zahlen gegenüber 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 – Listen 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 – Listen 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 – Exception-Abfragen

```
@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();
}
```

Ausgangslage: Entity-Klasse und Plain Assert J

```
@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();
    }
```

Ausflug: Plain Assert J vs. Eigene 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();
}
```

Ausflug: Eigene 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/ADC_TESTING_XMAS_SPECIAL

PART 5:
Testweisen und Abhängigkeiten

Zustandsbasiertes vs. verhaltensbasiertes Testen

Zustandsbasiertes Testen

- Veränderungen im Objektzustand werden untersucht: Dazu werden verschiedene Eigenschaften bzw. **Attribute ausgelesen** und gegen **erwartete Werte geprüft**.

- Gemäß dem AAA-Stil wird
  - zunächst für den **richtigen Kontext** gesorgt,
  - dann die gewünschte, **zu testende Funktionalität ausgeführt** und
  - schließlich das **Ergebnis geprüft**.

```
// 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");
```

Zustandsbasiertes Testen

```java
// 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");
```

- **ABER: Der geprüfte Zustand kann auch durch andere Aufrufe entstanden sein!**
- **ALSO: Wie prüft man Interaktionen und deren Abfolgen? Also etwa, dass zweimal die Methode `add()` aufgerufen wurde?**

Verhaltensbasiertes Testen

- Hierbei geht es darum, die **Interaktionen** zu prüfen und **nicht** die konkret ausgelösten **Zustandsänderungen**.

```
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);
    }
...
```

Verhaltensbasiertes Testen

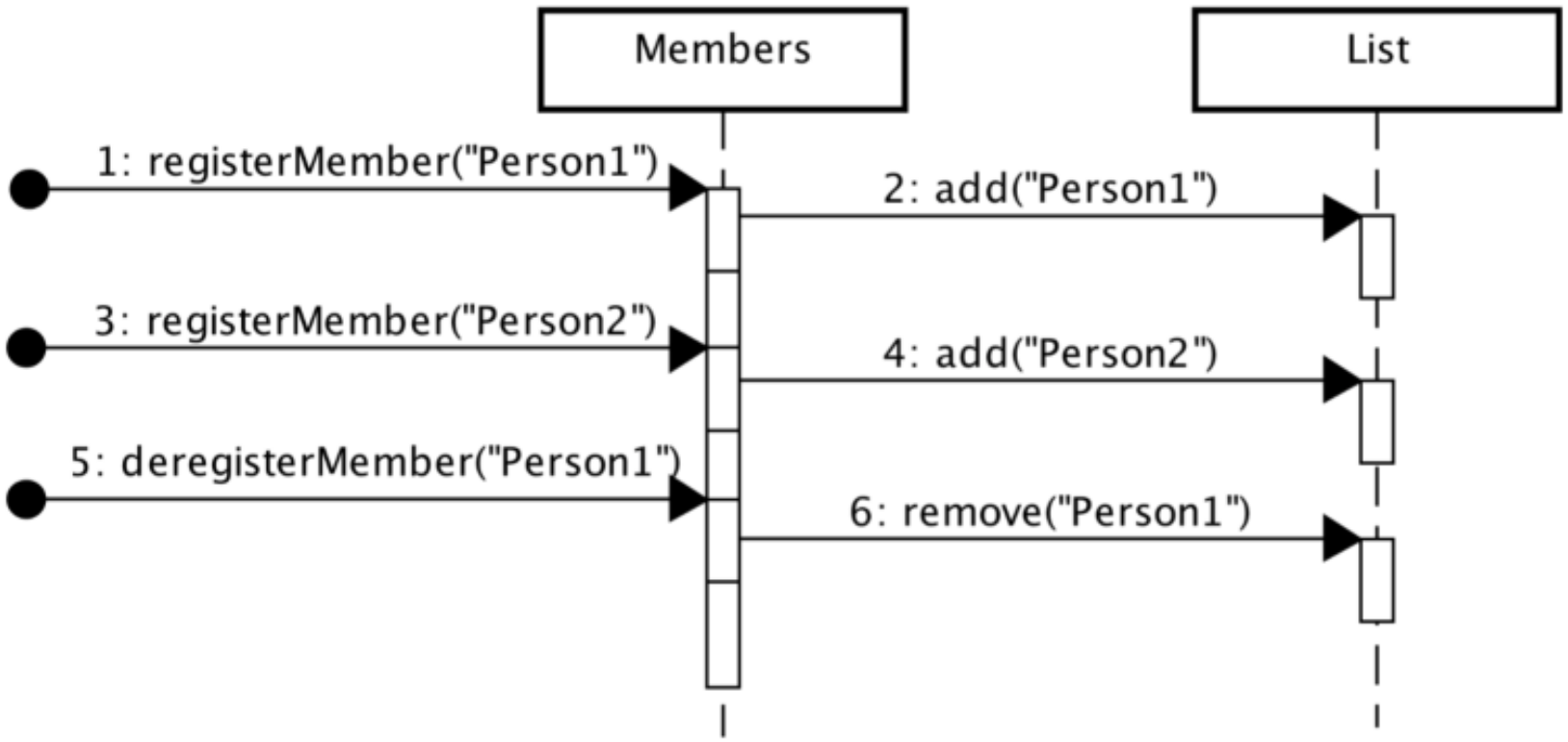
- Beim verhaltensbasierten Testen betrachtet man die **Interaktionen**, also die **aufgerufenen Methoden**, statt Veränderungen im Objektzustand.

- Demnach ist man beispielsweise nicht daran interessiert, ob sich nach einem Aufruf der Methode `registerMember(String)` die Anzahl der gespeicherten Elemente erhöht hat.

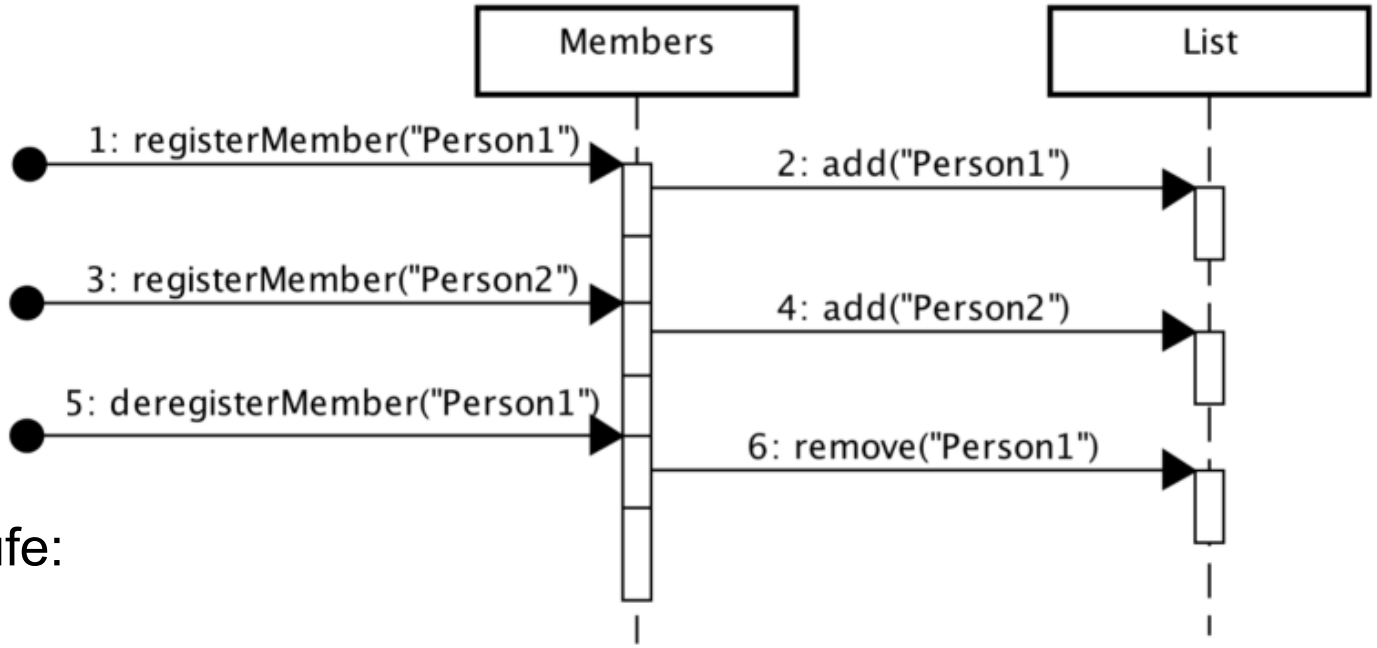
- Beim zustandsbasierten Testen würde man zur Prüfung einfach einen entsprechenden Aufruf einer `assertXYZ()`-Methode nutzen.

- **Wie kann man dann aber prüfen, ob ein korrektes Verhalten vorliegt?**

Verhaltensbasiertes Testen



Verhaltensbasiertes Testen



- Erwartung aufgrund der Methodenaufrufe:
  - zwei Aufrufen von `add(String)`,
  - gefolgt von einem Aufruf von `remove(String)`

- Beim verhaltensbasierten Testen prüfen wir genau dies ab.
- Wir erstellen dazu ein **Stellvertreterobjekt**, das die Interaktionen protokolliert und es später ermöglicht, diese mit den Erwartungen abzugleichen.

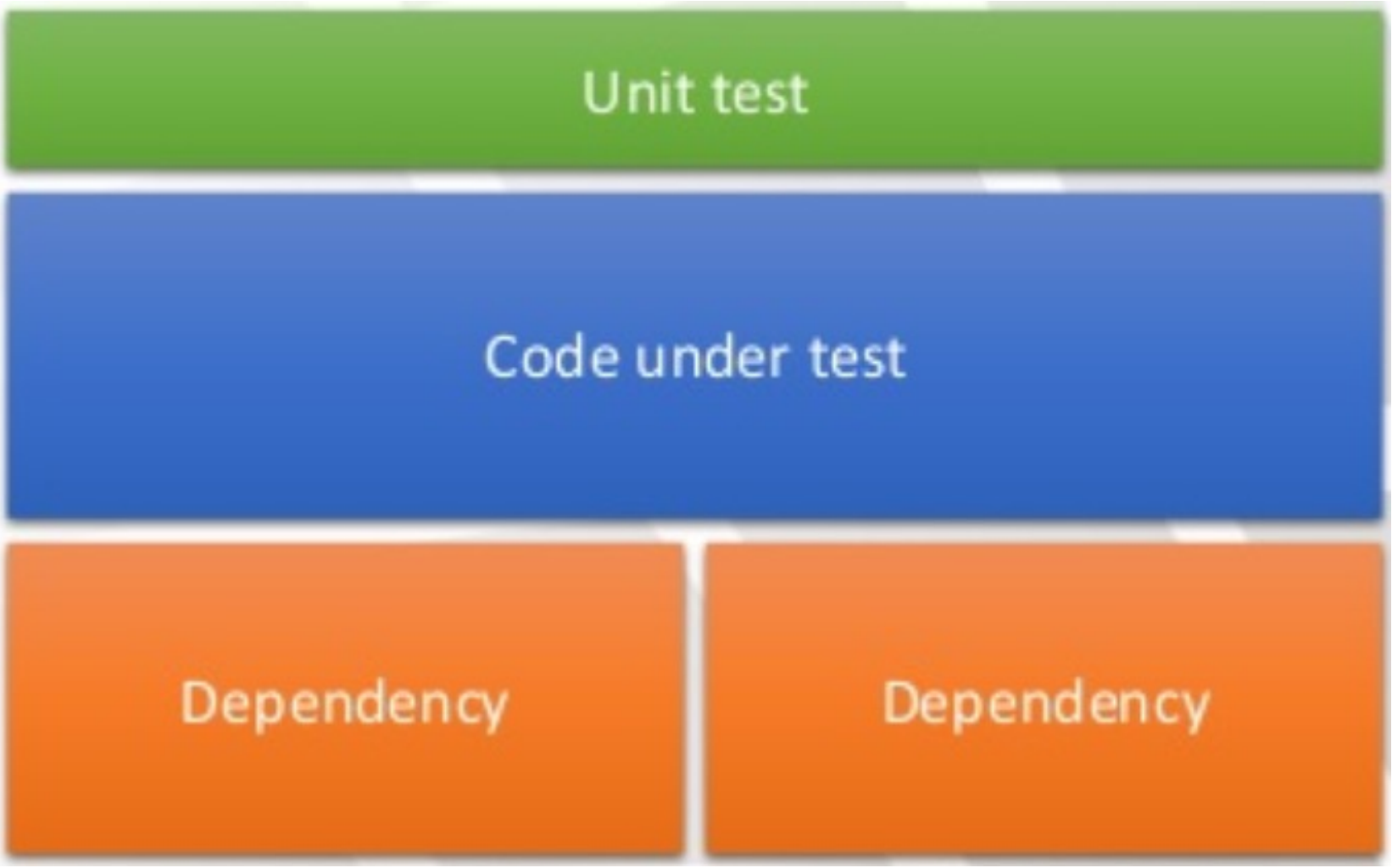
Verhaltensbasiertes Testen

- Nehmen wir an, wir würden einen **speziellen Teststellvertreter** durch Aufruf von `mock()` erhalten und Erwartungen durch Aufruf von `verify()` prüfen:

```
// 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");
```

Stellvertreter

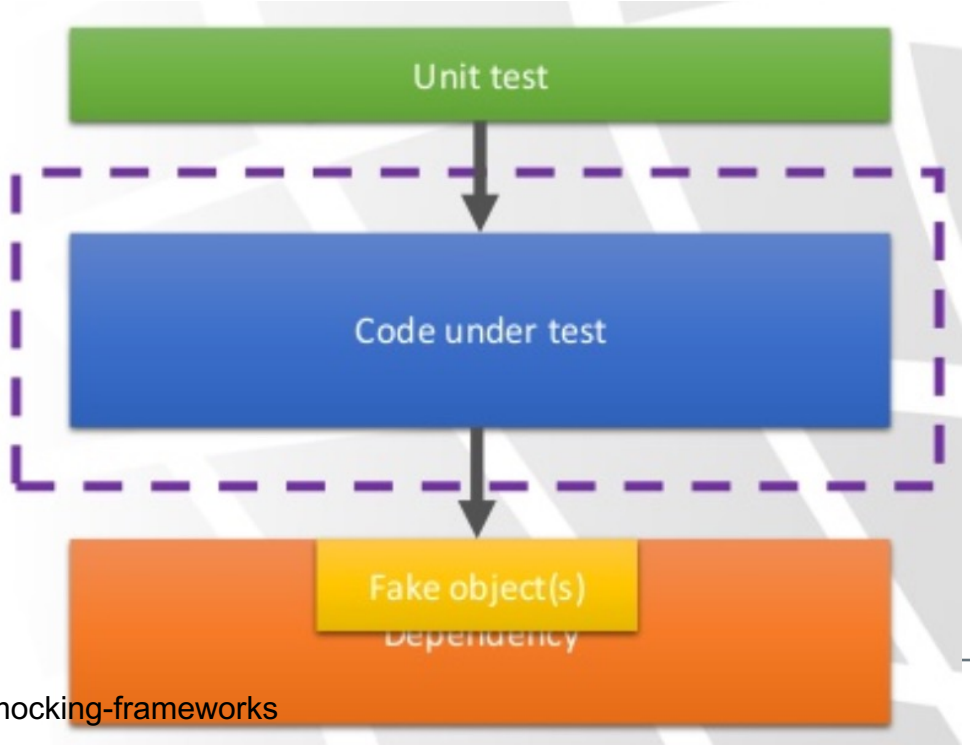
Stellvertreterobjekte – Warum sind sie nötig?



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

Stellvertreterobjekte

- *Auch Test-Doubles* genannt, sind Objekte, die als Stellvertreter für Applikationsobjekte oder Fremdsysteme zur Erleichterung von Tests
- Durch den Einsatz von Test-Doubles wird es möglich, **andere Komponenten für Testfälle zu ersetzen bzw. deren Verhalten zu simulieren**.
- Auf diese Weise kann man **Zustand oder Verhalten prüfen, ohne immer ein ganzes System oder eine spezielle Konfiguration bereitstellen zu müssen**.



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

Test Doubles

- Vielen sind die Begriffe Stubs und Mocks sicher geläufiger.

- Beide leiten sich aus dem Englischen ab:
  - »stub« = Stumpf, Stummel und
  - »to mock« = nachahmen, vortäuschen.

- Man findet vor allem folgende Varianten von Test Doubles:
  - Dummy
  - Stub und Fake
  - Mock

Dummy

- Unter einem **Dummy** versteht man einen **Platzhalter**, der **keine Funktionalität bereitstellt**. Manchmal muss man eine eine Methode gewisse Parameter übergeben, um Abhängigkeiten aufzulösen.

```
final Object optionalDataDummy = null;

doSomething(mandatoryValue, optionalDataDummy);
```

- Deutlich komplexer als Dummies sind ***Stub*** und **Fake**: Für mich stellen beide jeweils eine rudimentäre, manchmal auch nur leicht abgespeckte, aber funktionierende Implementierung einer anderen Klasse dar.

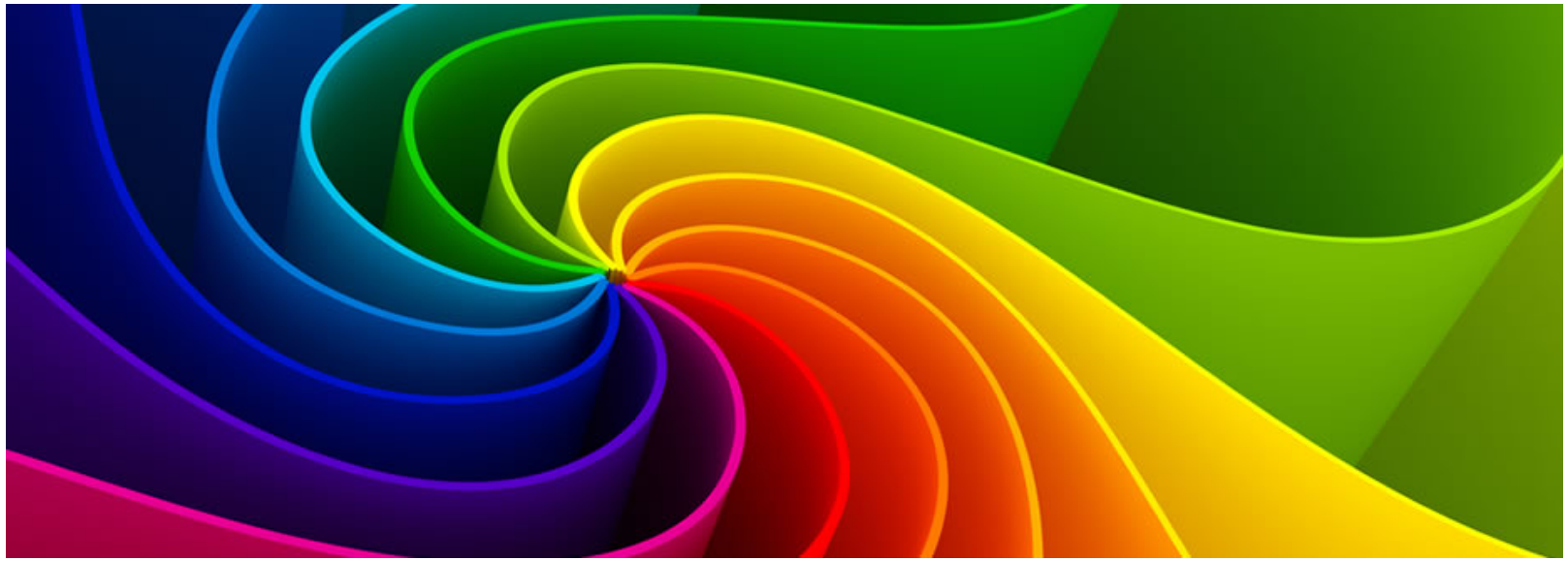
```
public final class SimulationDisplayStubBasic implements IDisplay
{
    @Override
    public void displayMsg(final MessageDto msg)
    {
        System.out.println("SimulationDisplay - got msg '" + msg + "'");
    }
}
```

Mock

- Mocks dienen zum **Überprüfen von Verhalten in Form von erwarteten Methodenaufrufen**. Werden durch die Anwendungsfunktionalität nicht die zuvor spezifizierten Methoden aufgerufen, wird dies als Fehler gewertet.

```
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



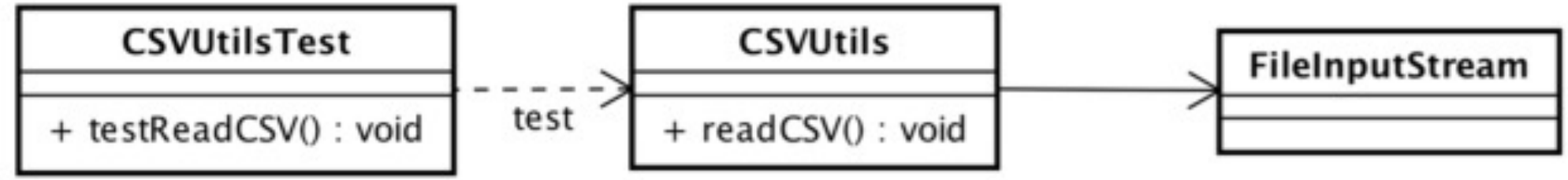
Sollbruchstellen

Sollbruchstellen – Injection Points

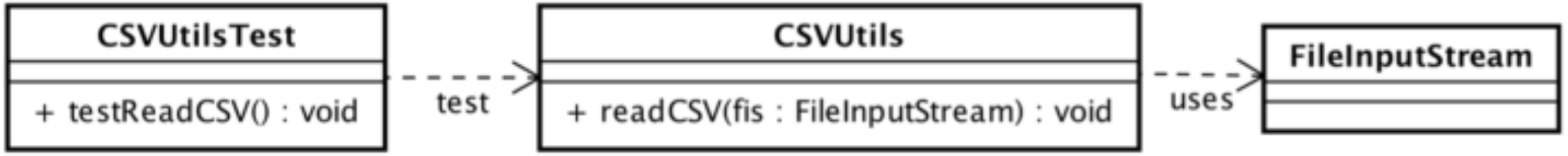
- Oftmals erschweren direkte Abhängigkeiten das Testen
- Besser gegen Abstraktion arbeiten, also
  - Abstrakte Klasse
  - Interface
- Zum Teil gibt es diese und sie sind nur geeignet in das Design einzufügen
- Manchmal muss erst eine Abstraktion erzeugt und dann genutzt werden (**Dependency Injection**)

Sollbruchstellen – Injection Points

- Oftmals erschweren direkte Abhängigkeiten das Testen

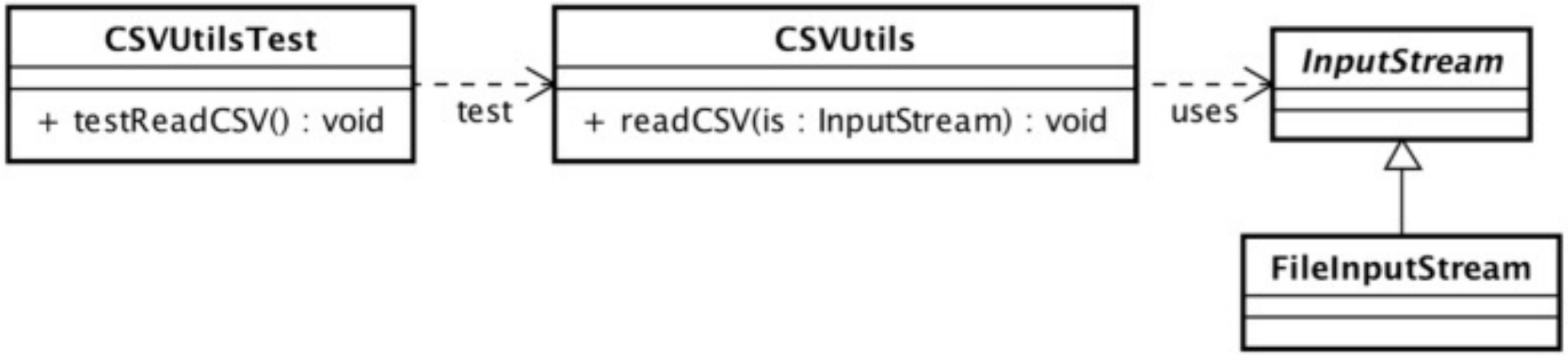


- Indirektion nutzen: Method Injection

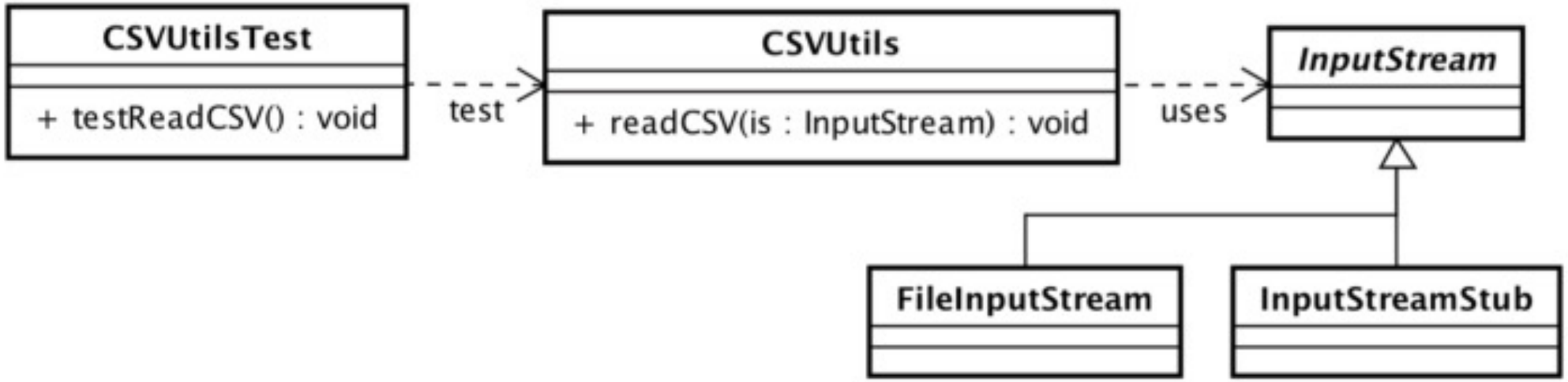


Sollbruchstellen – Injection Points

- Abstraktion nutzen



- Stub zur Testbarkeit nutzen




EXTRACT
Override

Extract And Override

- Eine **Variante**, wie man eine **Sollbruchstelle** in den **Applikationscode** einfügt
- Nutzt **Ableiten** und **Überschreiben**
- Umgeht Fallstricke, die eine Ausführung von Unit Tests erschweren

Extract And Override

- **Beginnen wir mit einem Taschenrechner ...**

```
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");
        }
    }
}
```

- **Wo ist das Problem?**

Extract And Override -- Schreiben wir ein paar 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**: Die **Testausführung** wird **unterbrochen**, bis jemand den Dialog schließt!

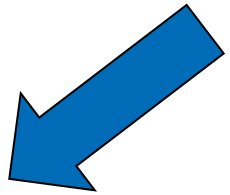
- **Benutzerinteraktion** ist beim manuellen Ausführen eines Unit Tests (z. B. in der IDE) schon **störend**
- **Showstopper** in einem automatischen Build auf Continuous-Integration-Server.
- **Also was nun? Überlegen wir kurz, was wir machen können.**

Extract And Override – Sollbruchstelle

```
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 – Sollbruchstelle

```
@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"));
}
```



- Ziemlich gute Abhilfe für viele kleinere Schwierigkeiten
- **ABER: Was machen wir, wenn wir den Sourcecode nicht im Zugriff haben?**


JUNIT
&
MOCKITO

Mocking Motivation

- Klassen werden oft im Kontext anderer Klassen ausgeführt. Generell haben wir deshalb beim Unit Testen vielfach die **Herausforderung, dass eine Klasse von einer oder mehreren anderen abhängt**.

- Es wäre nun **extrem aufwendig**, möglicherweise sogar fast unmöglich, und auch nicht wünschenswert, diese **geeignet zu parametrieren** oder zu initialisieren. Immer dann bietet sich der **Einsatz eines Mocking-Frameworks** an.

- Ein klassisches Beispiel ist der **Data Access Layer** oder ein **E-Mail-Service**. In beiden Fällen möchte man sicher ohne die externe Abhängigkeit die Unit Tests ausführen können.

- Die korrespondierenden Mocks würde man wie folgt erstellen:

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

Mocking im Maven/Gradle-Build

```xml
<dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>4.10.0</version>
        <scope>test</scope>
</dependency>

<dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-junit-jupiter</artifactId>
        <version>4.10.0</version>
        <scope>test</scope>
</dependency>
```

```
dependencies {
   testCompile 'junit:junit:4.13‘
   // Mockito
   testCompile "org.mockito:mockito-core:4.10.0"
   testCompile "org.mockito:mockito-junit-jupiter:4.10.0"
}
```

Mockito Basics

- Unser Ziel ist es, ein Test-Double zu erstellen
- Ausgangsbasis eine simple Klasse

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

- Beim Aufruf von `greet()` soll ein von der Originalimplementierung abweichender Rückgabewert geliefert werden, etwa `"Changed by Mockito"`

Mockito Basics

- `mock()` – Mock / Stub erstellen
- `when()` und `thenReturn()` – Verhalten beschreiben
- Mit Mockito kann man dem ARRANGE-ACT-ASSERT-Stil folgen

```
@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()` – Mock / Stub erstellen
- `when()` und `thenReturn()` – Verhalten beschreiben
- Mit Mockito kann man dem ARRANGE-ACT-ASSERT-Stil folgen

```
@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);
}
```

das durch `mock()` erstellte Test-Double kein Mock, sondern ein Stub

zustandsbasiertes Testen

Mockito Basics

- Mocking nutzt man für **Kollaborateure,** erstellen wir also eine Klasse, die `Greeting` nutzt:

```
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);
    }
}
```

Spezifische Rückgaben und Exceptions

- Abläufe spezifizieren:
  - `when()`,`anyString()` und `thenReturn()` – Verhalten für alle Eingabewerte beschreiben
  - `when()`, Parameterwert und `thenReturn()` – Verhalten für spezifische Eingabewerte festlegen
  - `when()` und `thenThrow()` – Exception auslösen

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

    // Achtung: Reihenfolge wichtig
    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);

    // Achtung: Reihenfolge wichtig
    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
}
```

Mehrere Rückgabewerte

- Mehrere Rückgaben mit `thenReturn()` einfach komma-separiert vorgeben:

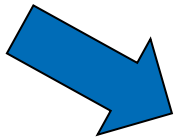
```
@Test
public void testGreetingMultipleReturns()
{
    // Arrange
    final Greeting greeting = mock(Greeting.class);
    when(greeting.greet(anyString())).
                  thenReturn("Hello Mockito1", "Hello Mockito2");

    // 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);
}
```

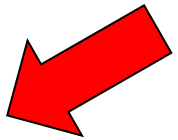
- **Bislang noch alles zustandsbasiert! Wie prüfen wir Verhalten in Form von Aufrufen?**

Prüfen von Aufrufen

- Die Methode `verify()` zum verhaltensbasierten Testen, prüft ob Aufrufe erfolgt sind:

```
@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

Häufigkeit von Aufrufen

- Die Methoden `atLeast()`, `atMost()` und `times()` festlegen, wie häufig ein Aufruf erwartet wird:
  - Mindestens,
  - Höchstens und
  - Exakt die angegebene Anzahl.

```
// 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() und die Reihenfolge von Aufrufen

```
@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 zeichnet Reihenfolge der Methoden der übergebnen Mocks auf
    InOrder inOrder = Mockito.inOrder(firstMock, secondMock);

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

    // Prüfe, dass die Aufrufe in der richtigen Reihenfolge erfolgen
    inOrder.verify(firstMock).methodOne();
    inOrder.verify(secondMock).methodTwo();
    inOrder.verify(firstMock).methodThree();
}
```

Mockito: Auf die Dosis kommt es an ☺

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

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

Wie kann man (zu) viele Mocks vermeiden?

Beispiel: Mocks vermeiden

```
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

Beispiel: Mocks vermeiden, durch 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/ADC_TESTING_XMAS_SPECIAL

PART 7:
Test Smells

Test Smells High Level

- **Ratschlag: Folge deiner Nase ☺**

- Prinzipiell sind Tests auch nur Sourcecode wie Businesscode
- Test Smells sind ähnlich zu Bad Smells / Code Smells (Details im “Java-Profi“)
- Bad Smell Examples: https://www.youtube.com/watch?v=9E6_zpx3q2c

- Man findet darüber hinaus folgende Test Smells auf High Level
  - Tests sind schwierig zu schreiben
  - Merkwürdige, komplizierte Dinge nötig, um Tests zu schreiben / zum Laufen zu bringen
  - Es benötigt sehr viel Mocking
  - Testen erfordert manchmal sogar Spezialbehandlungen im Businesscode
  - Die Testausführung ist (zu) langsam
  - Die Testausführung liefert schwankende Resultate (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 **=> Gilt NICHT für einfache Initialisierungen.**
Test Logic in Production		The code that is put into production contains logic that should be exercised only during tests. **=> Gilt NICHT für getter!* Anekdote 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: Falsche Nutzung von `assertTrue()/False()`

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

Test Smell: Falsche Nutzung von 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);
```

Einmal hin alles drin ☹ 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");
}
```

Einmal hin alles drin ☹ 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: Zu viele 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: Einsatz von `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)); // Wird nicht ausgeführt
}


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

gut gemeint != gut gemacht

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)); // Wird nicht ausgeführt
}
```

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

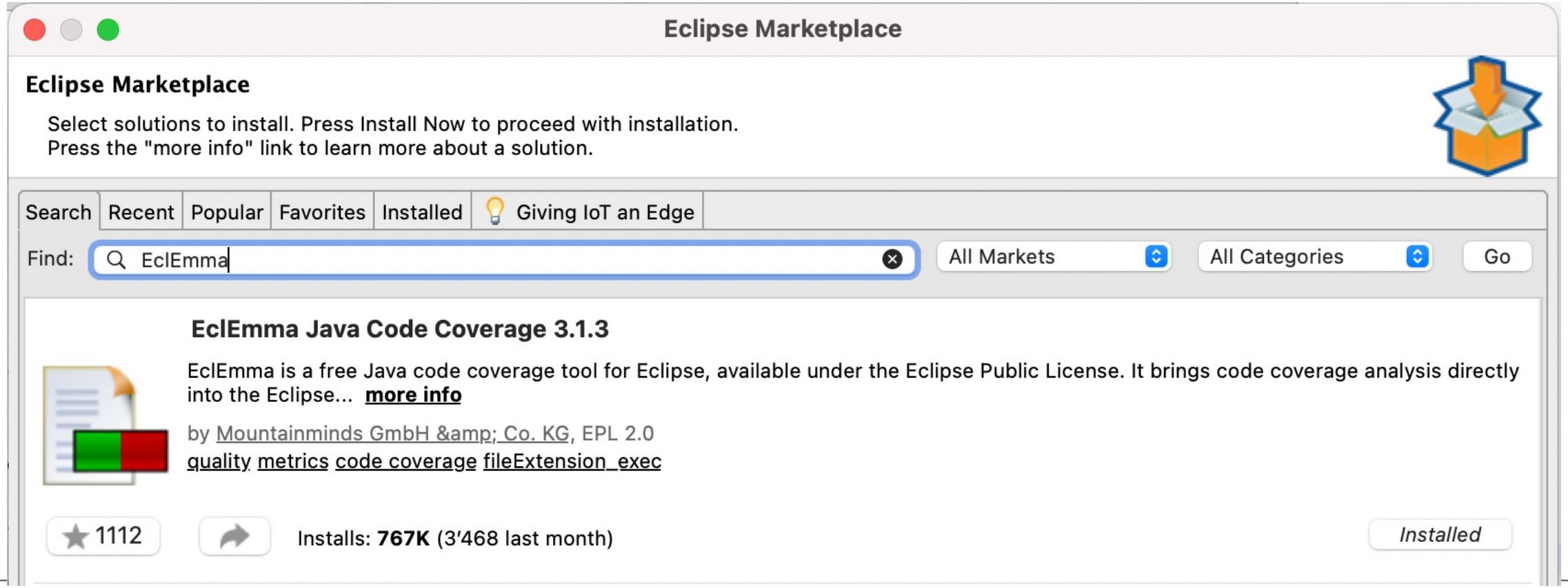
```
@Test
void reasonableAssertProvidingGoodFeedback()
{
    List<String> result = calcResultList();

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

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

PART 8:
Test Coverage

Test Coverage

- EclEmma Plugin ermittelt Testabdeckung
- Testabdeckung = der von Tests durchlaufene Sourcecode
- Im Marketplace frei verfügbar



Test Coverage

- Neuer Ausführungsmodus mit dem Namen „Coverage“, neben Debug und Run (oder als Context Menü)

- Dadurch werden während der Programm- bzw. Testausführung entsprechende Informationen über ausgeführte Sourcecode-Teile gesammelt
- Das Ergebnis wird nach Ausführungsende automatisch in einer speziellen Coverage-View angezeigt
- Praktischerweise von Projektebene bis hinunter zu einzelnen Java-Methoden werden sich verschiedene Metriken ermitteln und präsentiert je nach Wunsch: Instruktionen, Verzweigungen, zyklomatische Komplexität usw.

EclEmma – Eclipse Plugin

- Konfigurierbar

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

- Einfach, schnell, grafisch, übersichtlich

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

- Praktischer Dekorator
- Ungetester Code direkt sichtbar
- Durch Prozente => «Dringlichkeit» erkennbar

Test Coverage

- Gelungene Editor-Integration, dort wird die Testabdeckung farblich gekennzeichnet:
  - Grün: vollständig ausgeführt / abgedeckt
  - Gelb: teilweise ausgeführt / abgedeckt
  - Rot: nicht ausgeführt / abgedeckt

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

- Generell: Je kürzer und weniger Parameter und Verzweigungen eine Methode hat, desto besser lässt sich für diese eine hohe Testabdeckung erzielen
- Das erreicht man fast automatisch, wenn man Grundregeln guten OO-Designs befolgt:
  - Cyclomatic Complexity gering halten
  - SRP einhalten
- Wenn man die Smells vermeidet
  - Long Parameter List
  - Long Method

Besonderheit I: Wieso 80% und nicht 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

Besonderheit I: Wieso nicht 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
```

```
// ACHTUNG: wird automatisch generiert
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

Besonderheit I: Private Konstruktor => 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-Klasse sollte keinen
// Default-Ctor haben!
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

Besonderheit II: 100 % Testabdeckung keine Funktionalität getestet

```
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

Besonderheit II: 100 % Testabdeckung keine Funktionalität getestet

```
public class SpecialCounterTest
{
    // VERY BAD "TEST" ... 100% Coverage, aber KEINE semantische Prüfung
    @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();
    }
}
```

Besonderheiten III: Trotz 100 % Testabdeckung eine NPE

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

100 % Testabdeckung aber 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 – Testabdeckung mit Gradle und Maven

- In Kombination mit Tests ausführbar, produziert HTML-Report
- Gradle

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

- Maven

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

- **Interessanterweise unterscheiden sich die Reports leicht im Ergebnis?!**

JaCoCo – Testabdeckung mit Gradle und 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/ADC_TESTING_XMAS_SPECIAL

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