



JUnit 5 Workshop – Quick Start

**Mehr Spass und weniger Bauchschmerzen beim Entwickeln
durch clevere Tests**

Michael Inden

Freiberuflicher Consultant und Trainer

Speaker Intro



- Michael Inden, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg
- ~8 ¼ Jahre SSE bei Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Jahre TPL, SA bei IVU Traffic Technologies AG in Aachen
- ~4 ¼ Jahre LSA / Trainer bei Zühlke Engineering AG in Zürich
- ~3 Jahre TL / CTO bei Direct Mail Informatics / ASMIQ in Zürich
- Freiberuflicher Consultant, Trainer und Konferenz-Speaker
- Autor und Gutachter beim dpunkt.verlag

E-Mail: michael_inden@hotmail.com

Blog: <https://jaxenter.de/author/minden>

Kurse: Bitte sprecht mich an!





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

- **PART 3: JUnit 5 Advanced**

- Parameterized Tests
-

Workshop Contents



- **PART 4: Testweisen und Abhangigkeiten**
 - Zustandsbasiertes vs. Verhaltensbasiertes Testen
 - Stellvertreterobjekte
- **PART 5: Design For Testability**
 - Sollbruchstellen
 - Extract and Override
 - Mockito



PART 1: Warum testen und gute Angewohnheiten



Was ist Testen?



- Unter *Testen* versteht man den Vorgang, das tatsächliche Verhalten eines Programms oder eines Teils davon (*ist*) mit dem geforderten Verhalten (*Soll*) zu vergleichen.
- Demnach entspricht Testen **nicht** dem **einmaligen** Start eines Programms mit ein paar **willkürlichen** Bedienhandlungen.
- Verhalten der Software unter verschiedenen Bedingungen zu prüfen.
- ein wenig »bösaig« agieren, um versteckte Probleme aufdecken oder Fehlverhalten provozieren zu können.
- Etwa bewusste Fehlbedienung, etwa die Eingabe ungültiger Werte sowie von Extrem- oder Randwerten.

Warum testen wir?



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

Was macht einen guten Unit Test aus?



A good Unit Test should be:



Easy
to write

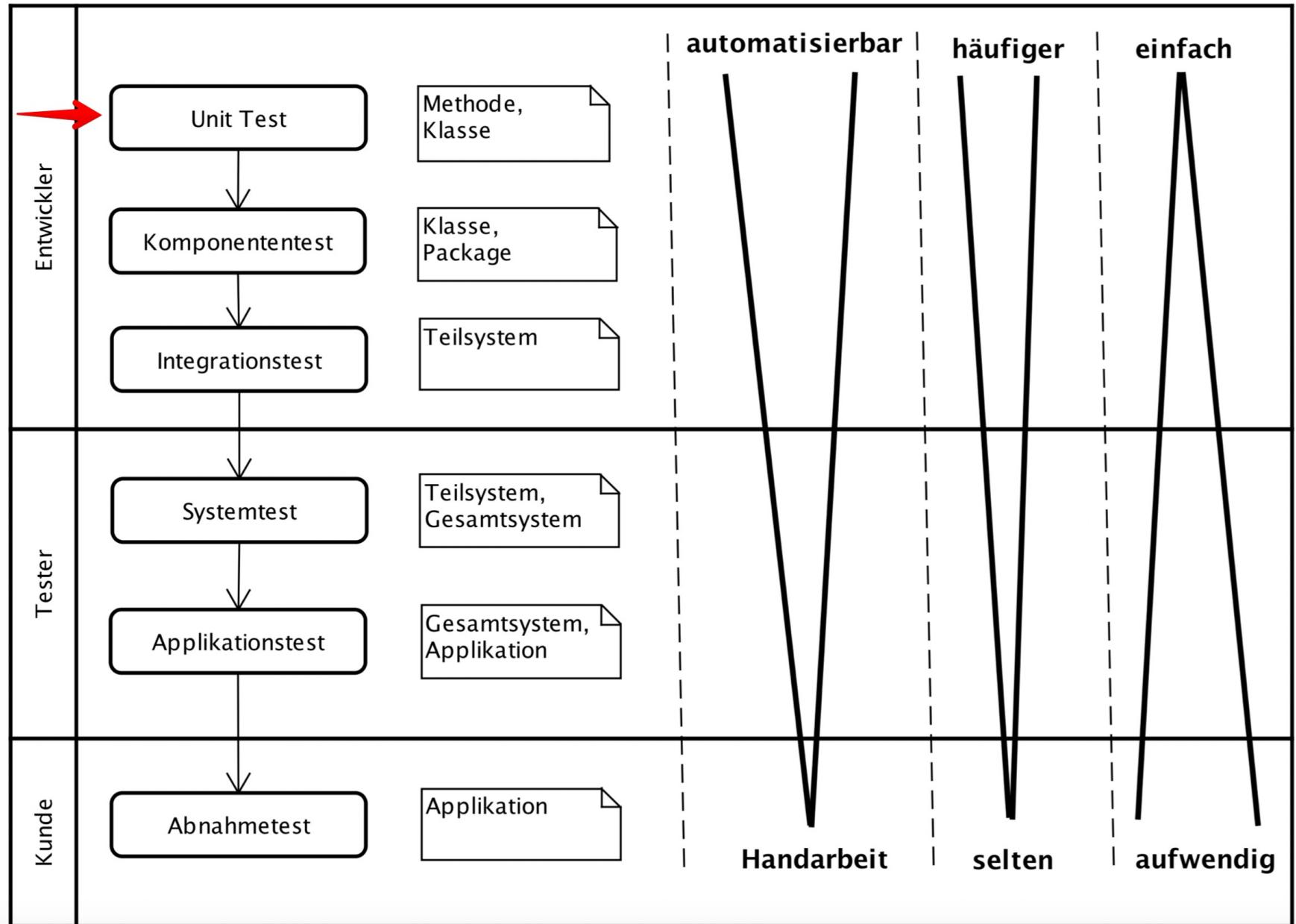


Simple
to read

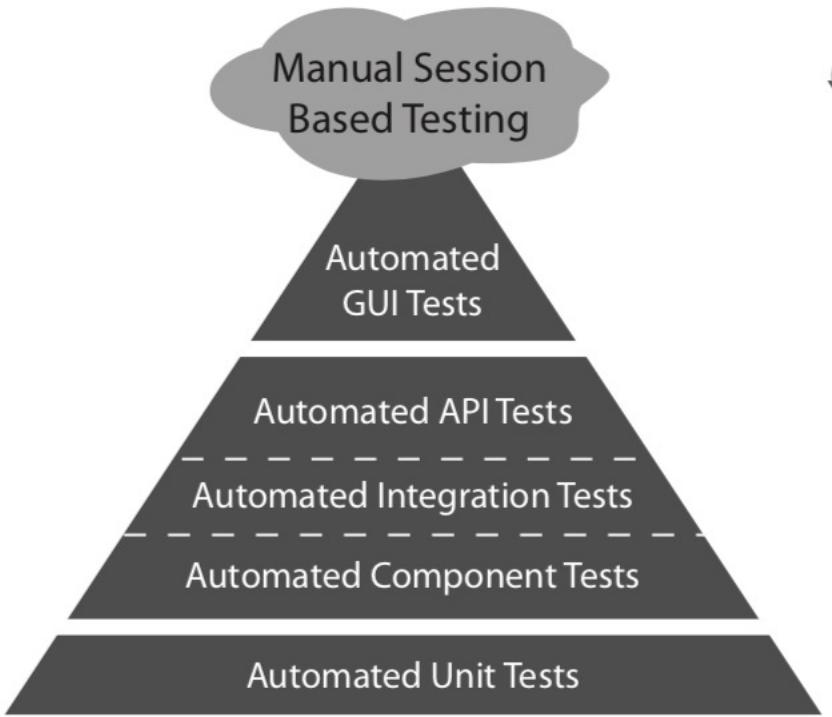


Trivial
to maintain

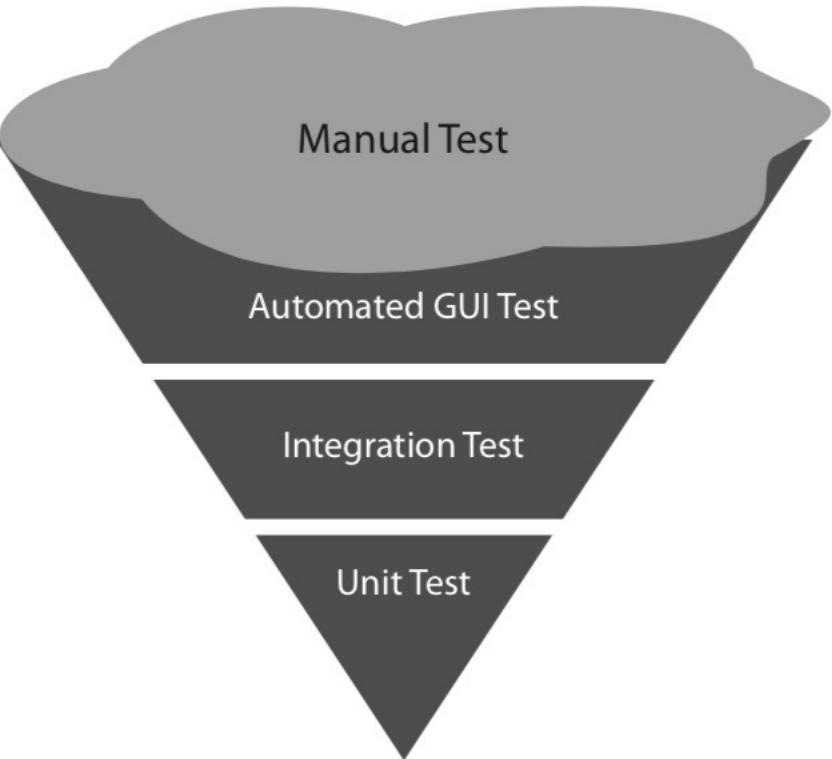
Arten von Tests



Testpyramide



The Ideal Testing
Automation Pyramid



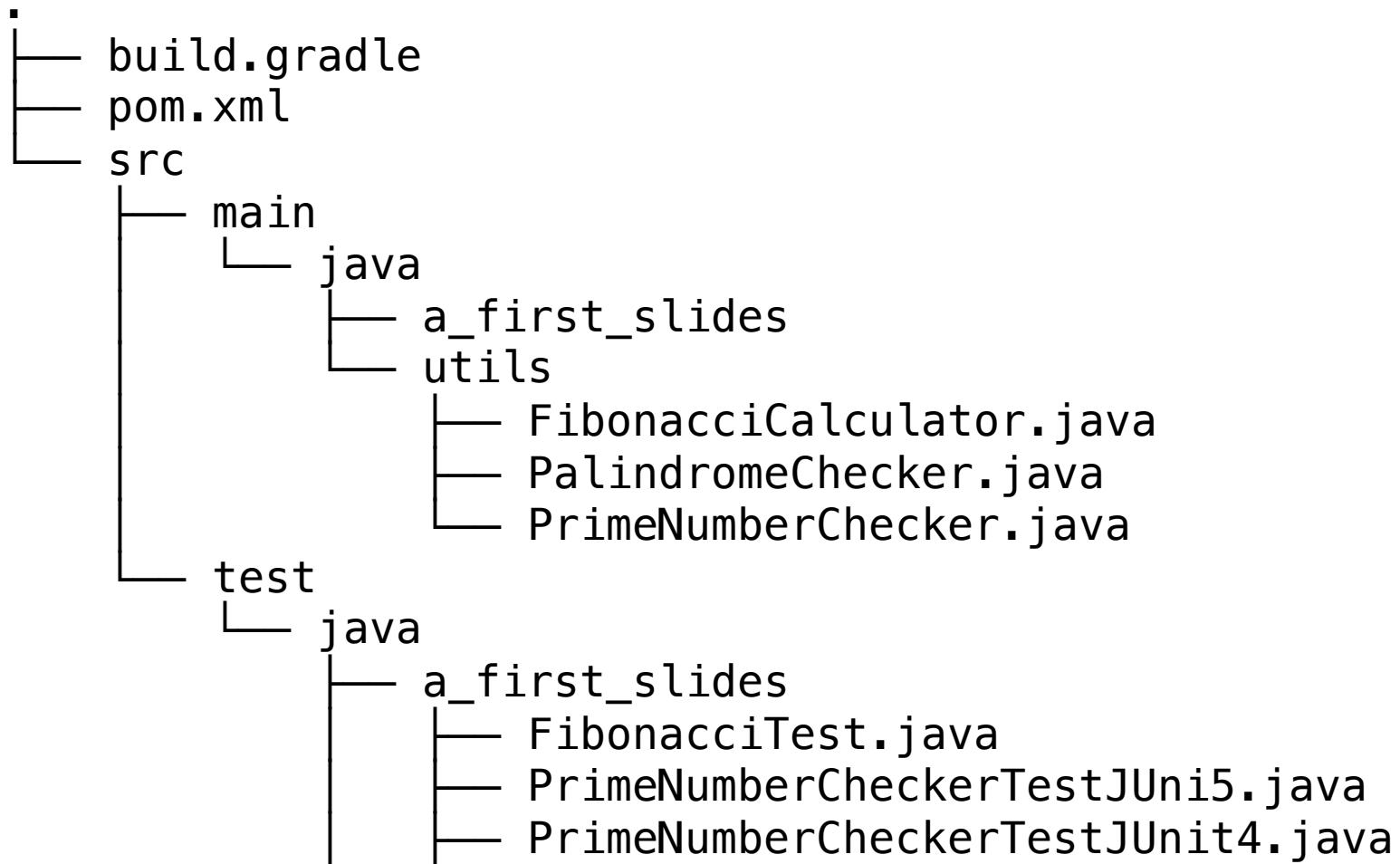
The Non-Ideal Testing
Automation Inverted Pyramid



Gute Angewohnheiten



Maven-Projektstruktur





Namensgebung

- Klasse **Abc** => zugehörige Testklasse **AbcTest**
- Methoden:
 - Optional: Kürzel **test** als Start
 - Sinnvolle Beschreibung des Testfalls:
 - Methodename, 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()



- **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");
}
```

FAIR - Gewünschte Eigenschaften von Unit Tests



F – Fast, Focussed

A - Automated

I - Isolated

R – Reliable, Repeatable



Eigenschaften von Unit Tests: API-Design & Dokumentation



- Beim Schreiben von **Unit Tests** spielen auch **Entwurfsentscheidungen**, etwa solche zu **Kohäsion**, **Kopplung** und zum **Design des APIs** eine Rolle.
- Durch das Implementieren von Testfällen nutzt man das API der eigenen Klassen, wodurch die Beurteilung leichter fällt, ob die **angebotenen Schnittstellen sinnvoll und handhabbar** sind.
- **Unit Tests** können also **mögliche Schwächen** in den von den Tests angesprochenen APIs vor einer Nutzung in anderen Komponenten **aufdecken** und **für gelungenere APIs sorgen**.
- Unit Tests als **Dokumentation des erwarteten Programmverhaltens**
- **Dokumentation** ist automatisch **immer aktuell**, da die Testfälle ansonsten fehlschlagen würden.



PART 2: JUnit 5 Intro



JUnit 5

5 JUnit 5

JUnit 4

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

User Guide

Javadoc

Code & Issues

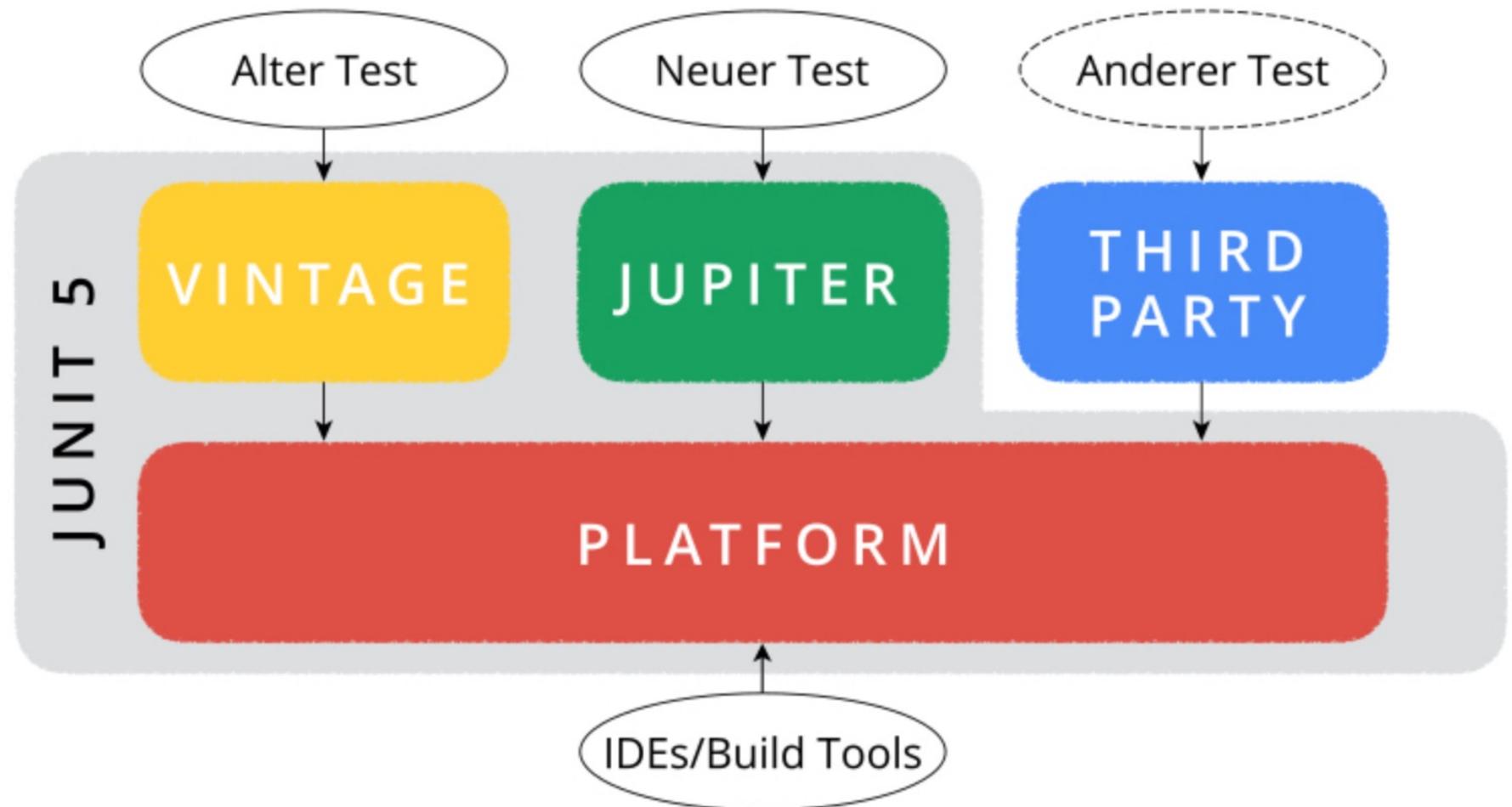
Q & A

Support JUnit



Architektur

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





Assertions – Bedingungen prüfen

- Auswertung von Bedingungen – Die Klasse **Assert** (JUnit4) / **Assertions** (JUnit 5) stellt eine Menge von Prüfmethoden bereit, mit denen Bedingungen formuliert und dadurch **Zusicherungen** über den zu testenden Sourcecode geprüft werden können:
 - **assertEquals()** – zwei Objekte auf inhaltliche Gleichheit (Aufruf von **equals(Object)**) bzw. zwei Variablen primitiven Typs auf Gleichheit prüfen*
 - **assertTrue()** und **assertFalse()** – boolesche Bedingungen prüfen
 - **assertNull()** bzw. **assertNotNull()** – Objektreferenzen auf == null bzw. != null prüfen
 - **assertSame()** bzw. **assertNotSame()** – Objektreferenzen auf == bzw. != prüfen
 - **fail()** – einen Testfall bewusst fehlschlagen lassen

*) Achtung für Floating Point: float und double



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");

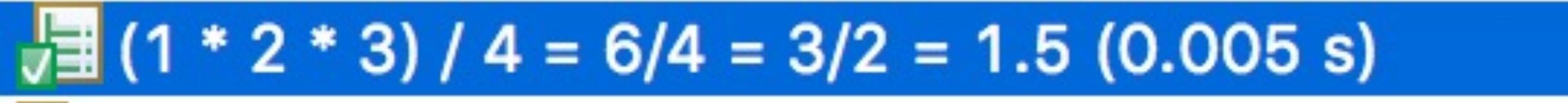
    assertFalse(false);
    assertNull(null);
    assertNotNull(new Object());
    assertSame(null, null);
    assertNotSame(null, new Object());
}
```



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

▼ REST product controller [Runner: JUnit 5] (0.007 s)

- POST 'http://localhost:8080/products/' user: Stock Manager (0.000 s)
- GET 'http://localhost:8080/products/4711' user: Peter Müller (0.007 s)



Spezielle Assertions



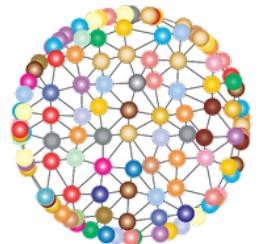


Multiple Asserts

```
@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

```
@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);
}
```

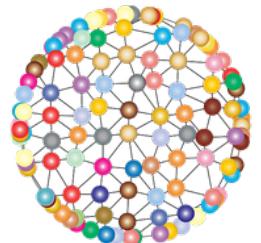
J! org.opentest4j.AssertionFailedError: expected: <Tim> but was: <Mike>
≡ at a_first_slides.DisplayNameExample.multipleAssertsforOneTopic_Diff1(D)

```
@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));
}
```

J! 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("\u03c0 = 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;  
}
```

Runs: 1/1 ✘ Errors: 0 ✘ Failures: 0

▼ DisplayNameExample [Runner: JUnit 5] (0.000 s)
 $\pi = 3.1415 \text{ (with four digit precision)}$ (0.000 s)



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");
}
```



JUnit Test (temporär) ausschalten

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



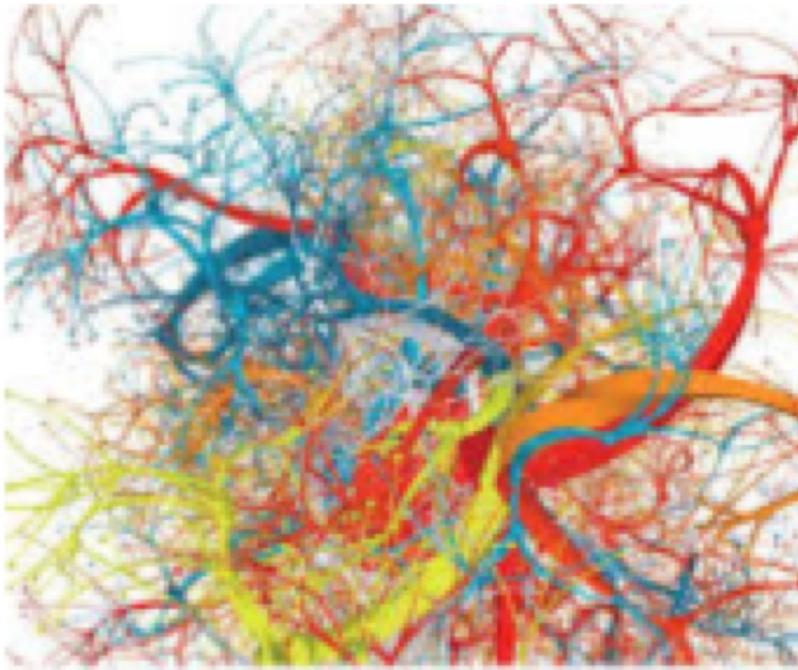
PART 3

JUnit 5 Advanced





Kombinatorik / Komplexität





3 zentrale Fragen

1. Welche Wertebereiche 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?
-



- **Welche Wertbelegungen 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");  
}
```

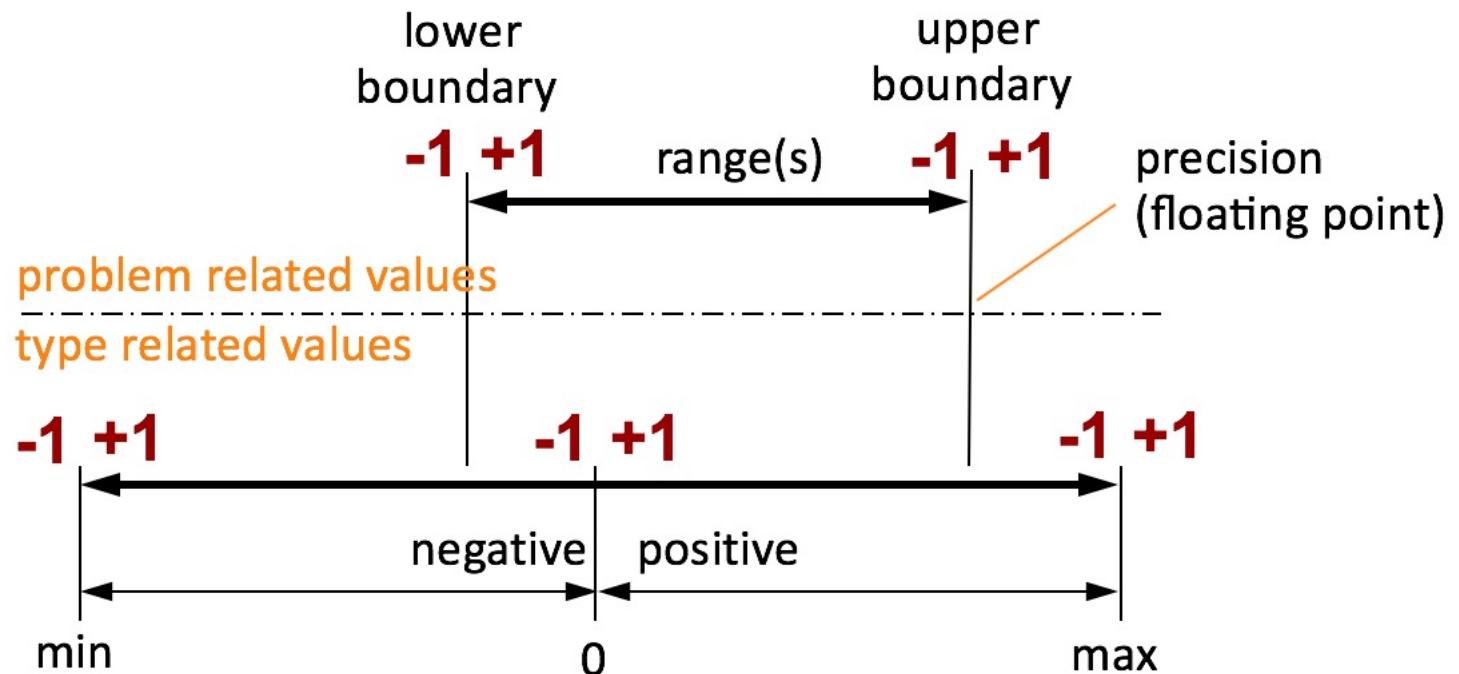
```
@Test  
public void testCalcDiscount_MediumOrder_MediumDiscount()  
{  
    final int mediumAmount = 200;  
    assertEquals(4, calculator.calcDiscount(mediumAmount), "4 % discount");  
}
```

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



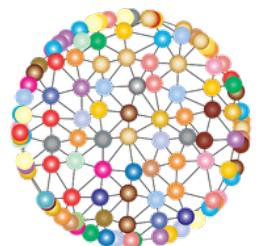
Grenzwerttests

- **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:





- 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





- **Abhilfe durch sogenannte Parameterized Test**
- **Testfall mit verschiedenen Daten immer wieder mit neuer Werteverteilung 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



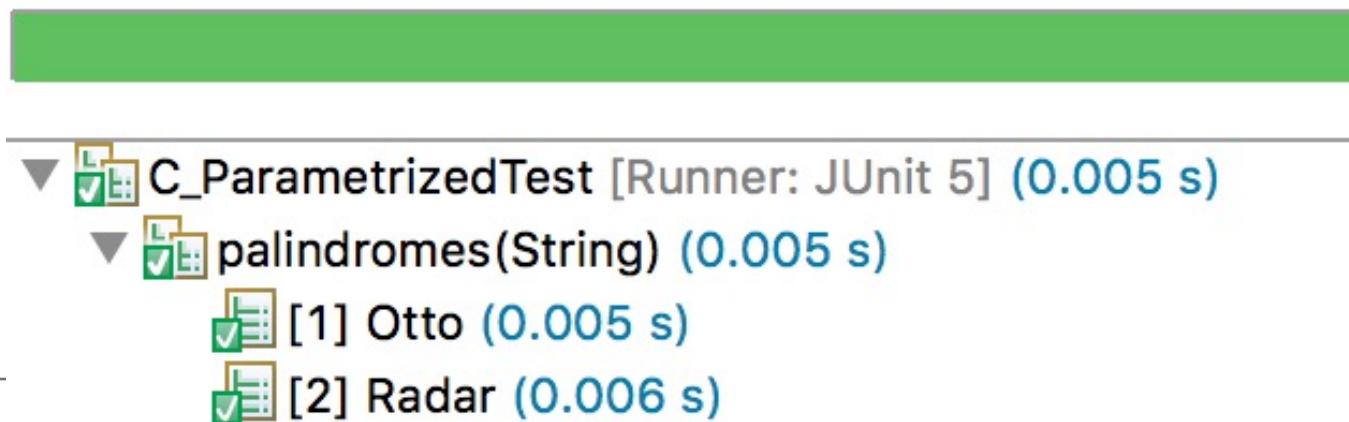
```
@Test
public void testCheckMatchingBracesAllOkay() throws Exception
{
    List<String> inputs = List.of("()", "()[]{}", "[((())[]{}))]");
    for (String current : inputs)
    {
        assertTrue("Checking " + current,
                   MatchingBracesChecker.checkMatchingBraces(current));
    }
}

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

Parameterized Test – JUnit 5 @ParameterizedTest / @ValueSource



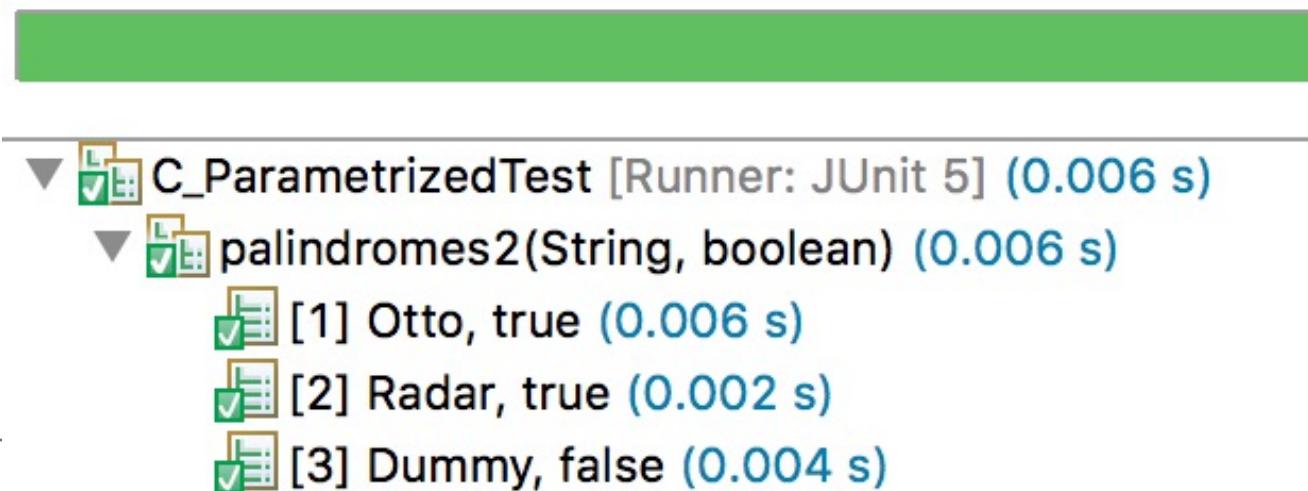
```
@ParameterizedTest  
@ValueSource(strings = { "Otto", "Radar" })  
void palindromes(String candidate)  
{  
    boolean isPalindrome = PalindromeChecker.isPalindrome(candidate));  
  
    assertTrue(isPalindrome);  
}
```



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);  
}
```

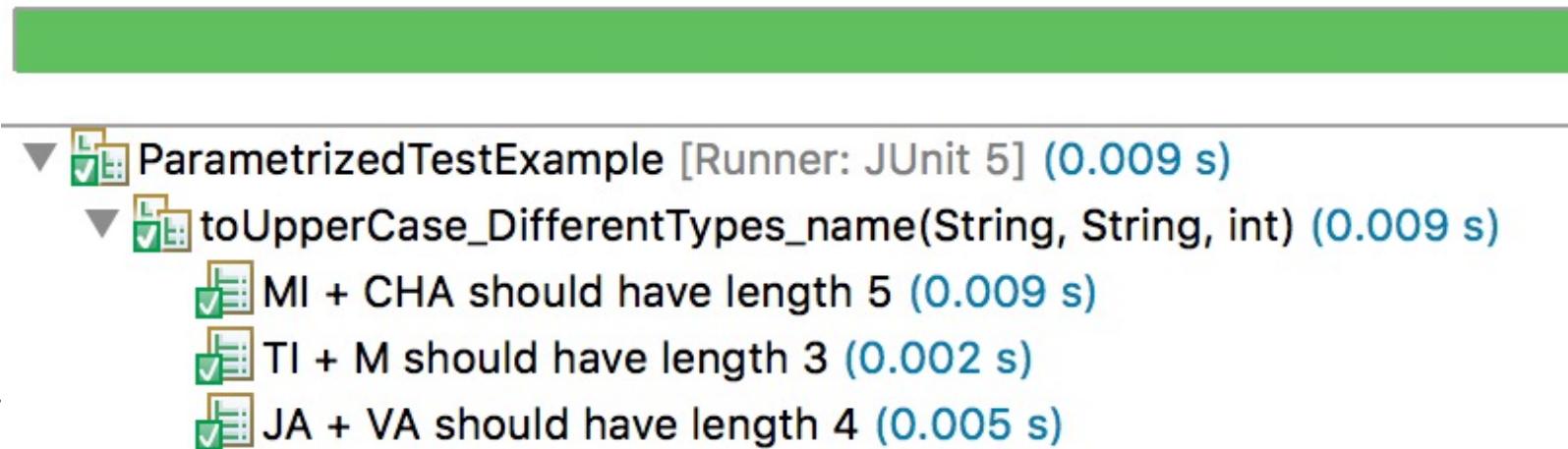


Parameterized Test – bessere Benennung



```
@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);
}
```



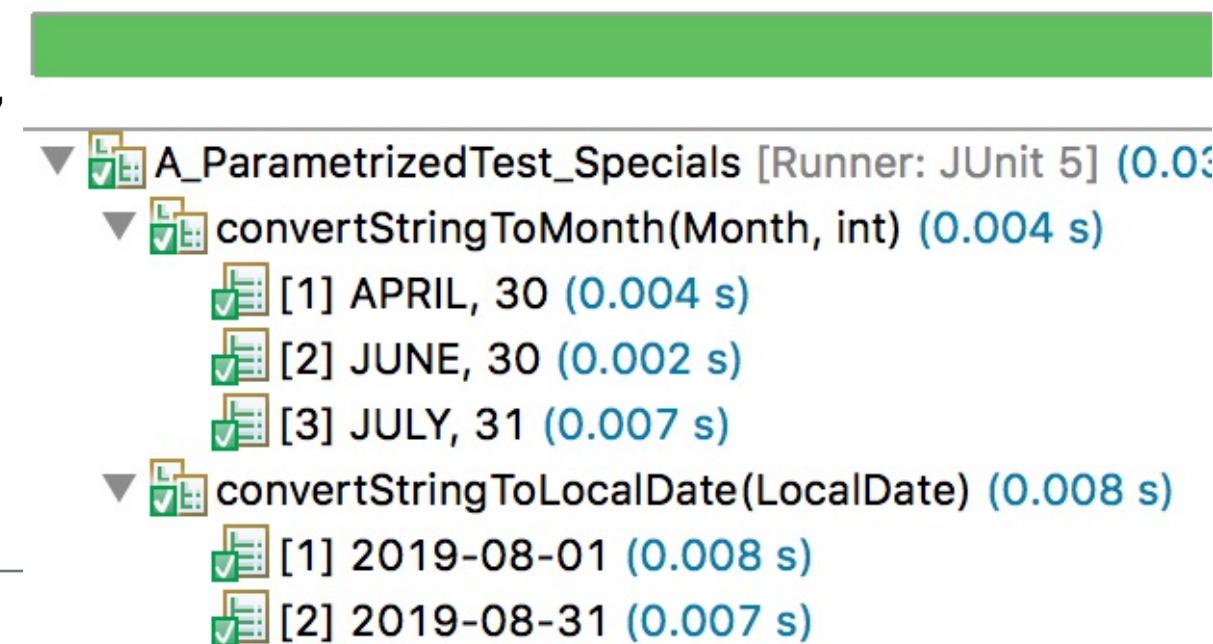
Parameterized Test – Diverse Konvertierungen und Hilfen



- *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));
}
```

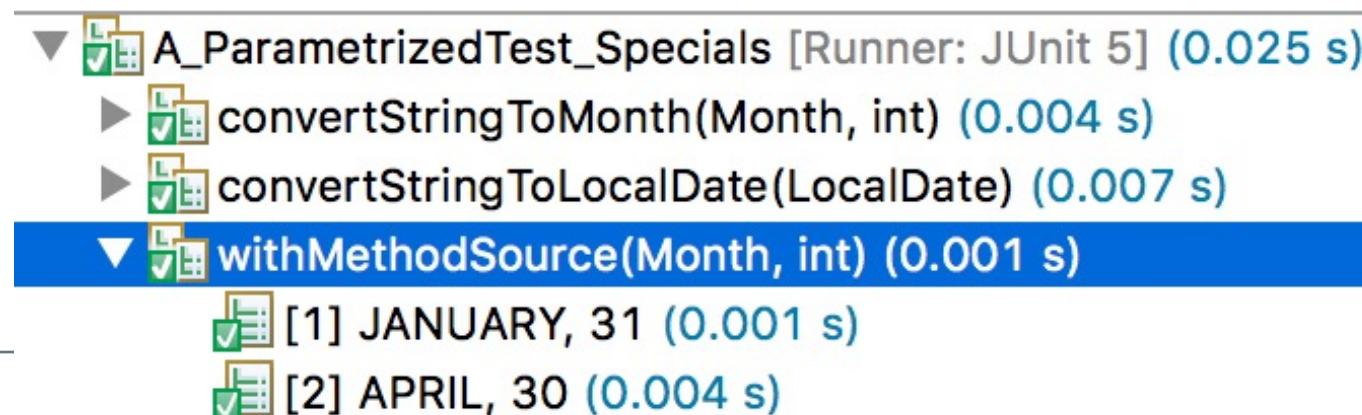


Parameterized Test – @MethodSource



```
@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));
}
```

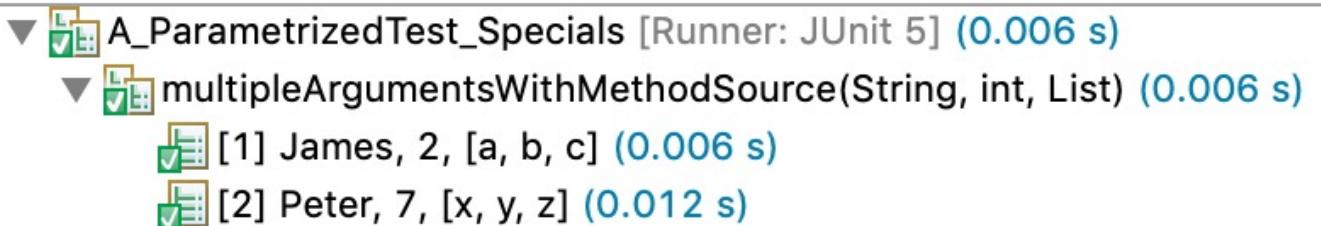


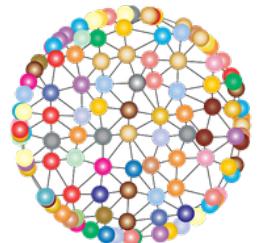
Parameterized Test – @MethodSource



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

Runs: 2/2 ✘ Errors: 0 ✘ Failures: 0





**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);
}
```

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



```
@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);
}
```

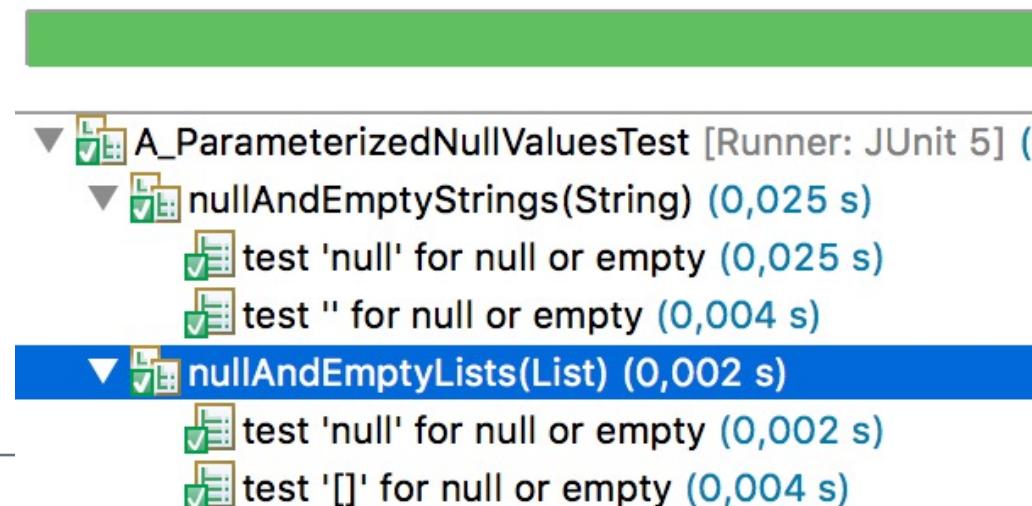
```
arabic,roman
1, I
2, II
3, III
4, IV
5, V
7, VII
9, IX
17, XVII
40, XL
90, XC
```

Parameterized Test – Randfälle prüfen



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

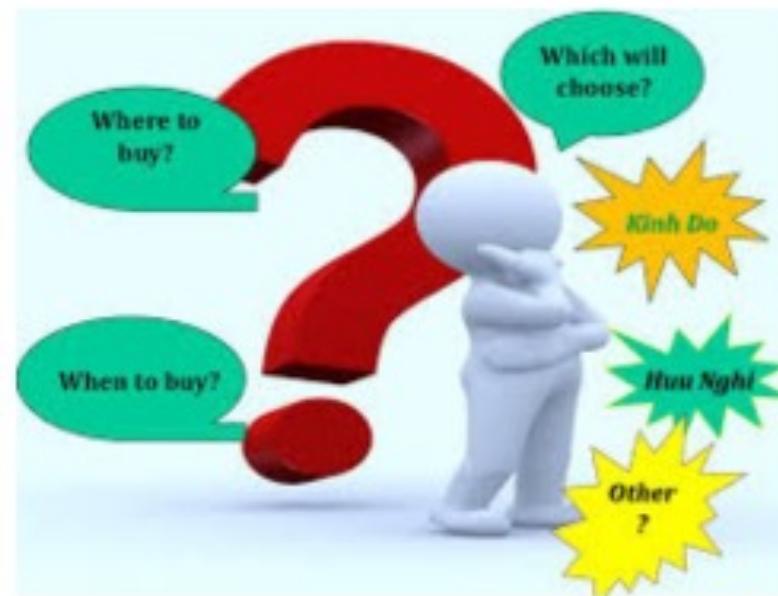
```
@ParameterizedTest(name = "test '{0}' for null or empty")
@NullSource
@EmptySource
void nullAndEmptyLists(List<?> list)
{
    assertTrue(list == null || list.isEmpty());
}
```





PART 4:

Testweisen und Abhangigkeiten





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



```
// 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);
    }

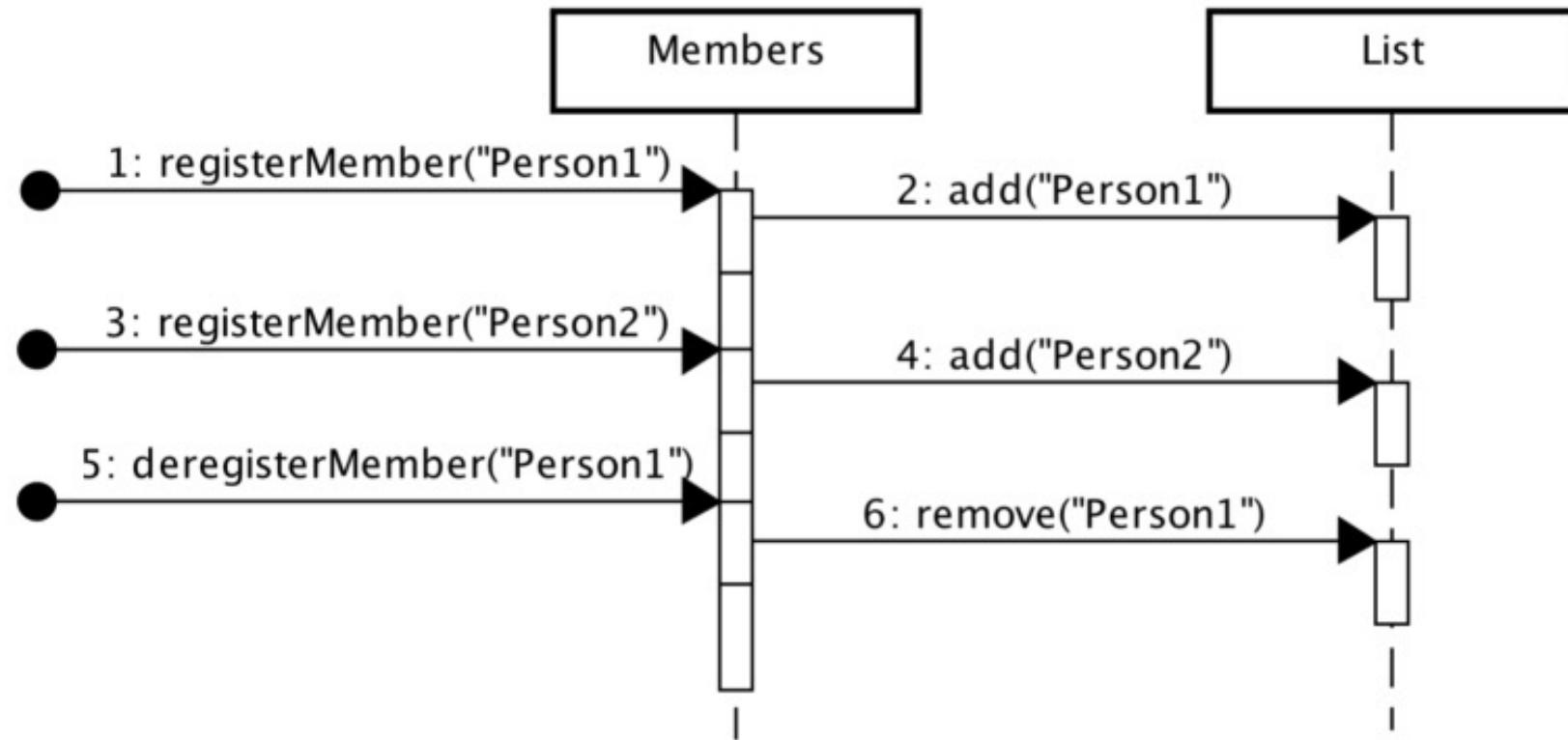
    ...
}
```

Verhaltensbasiertes Testen

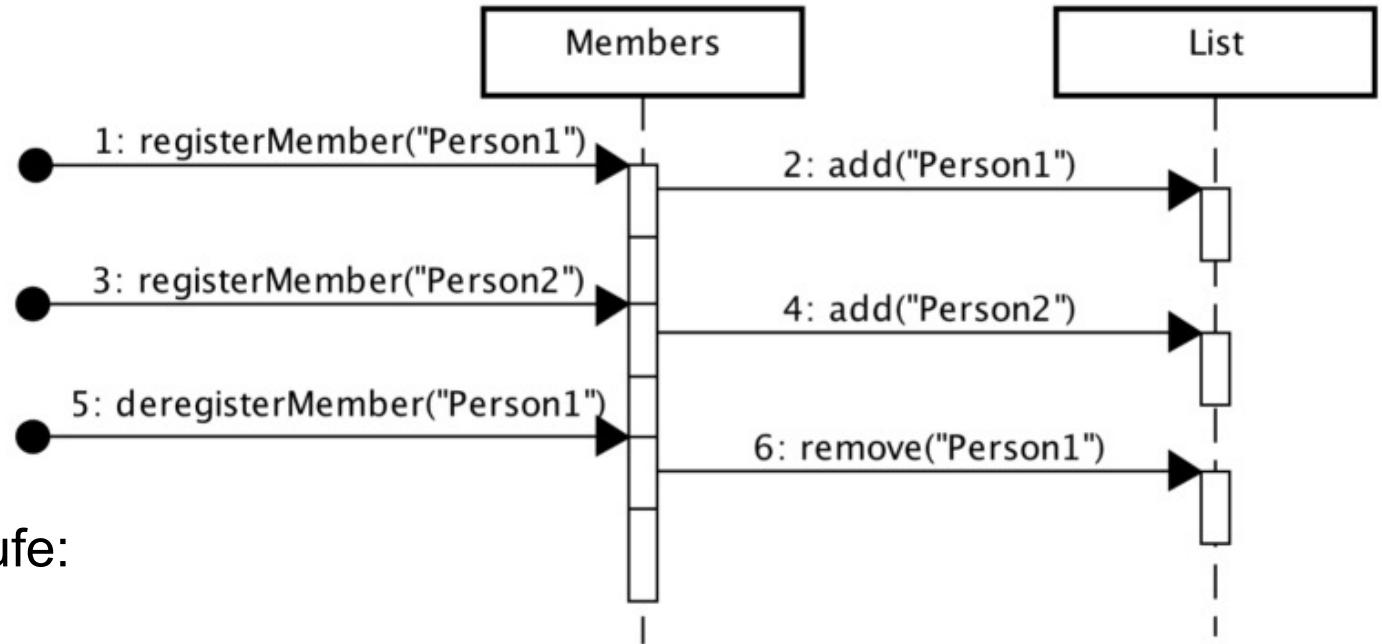


- Beim verhaltensbasierten Testen betrachtet man die **Interaktionen**, also die **aufgerufenen Methoden**, statt Veränderungen im Objektzustand.
- Demnach ist man beispielsweise nicht daran interessiert, ob sich nach einem Aufruf der Methode `registerMember (String)` die Anzahl der gespeicherten Elemente 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");

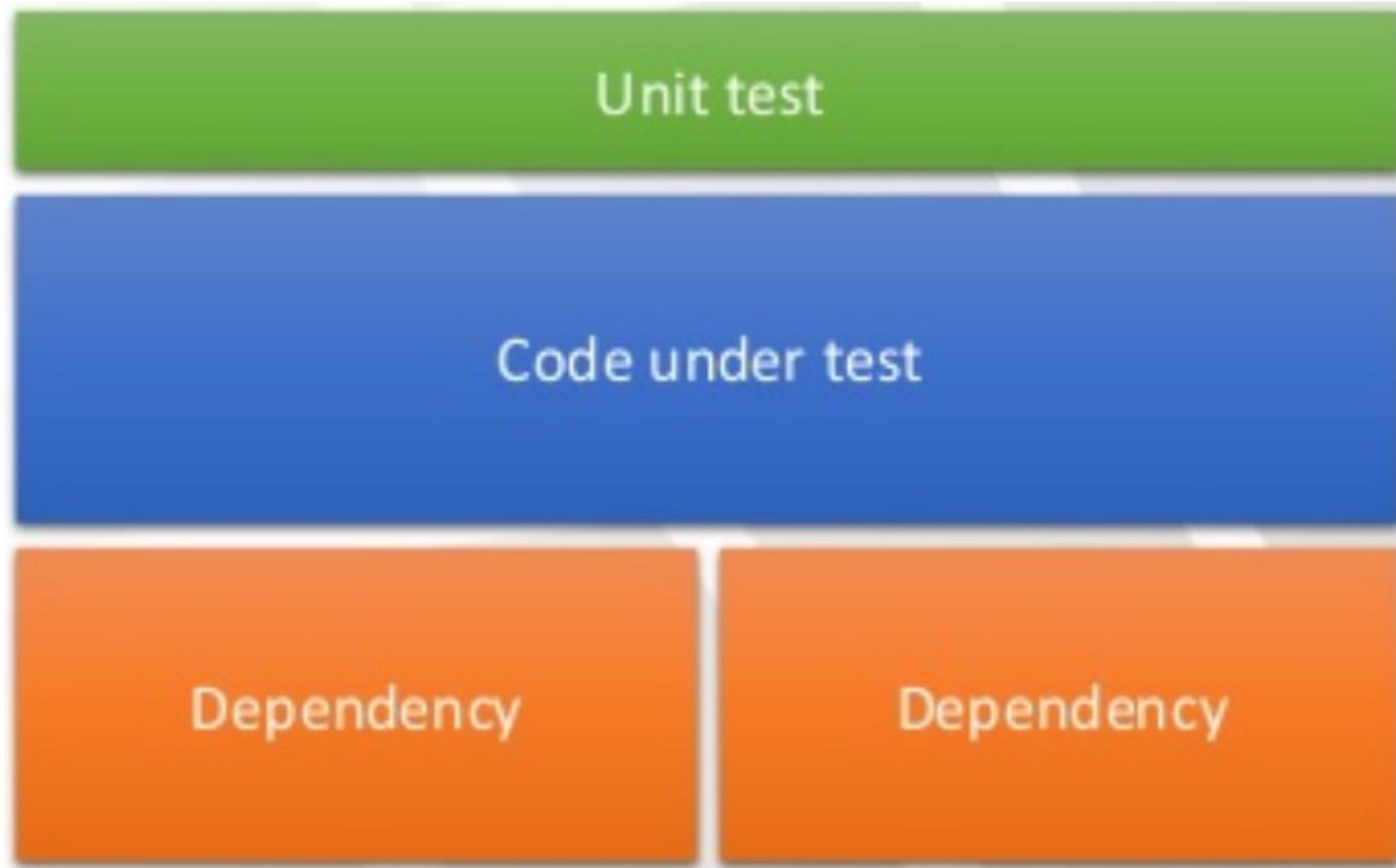
// Assert
verify(mockedList).add("Person1");
verify(mockedList).add("Person2");
verify(mockedList).remove("Person1");
```



Stellvertreter



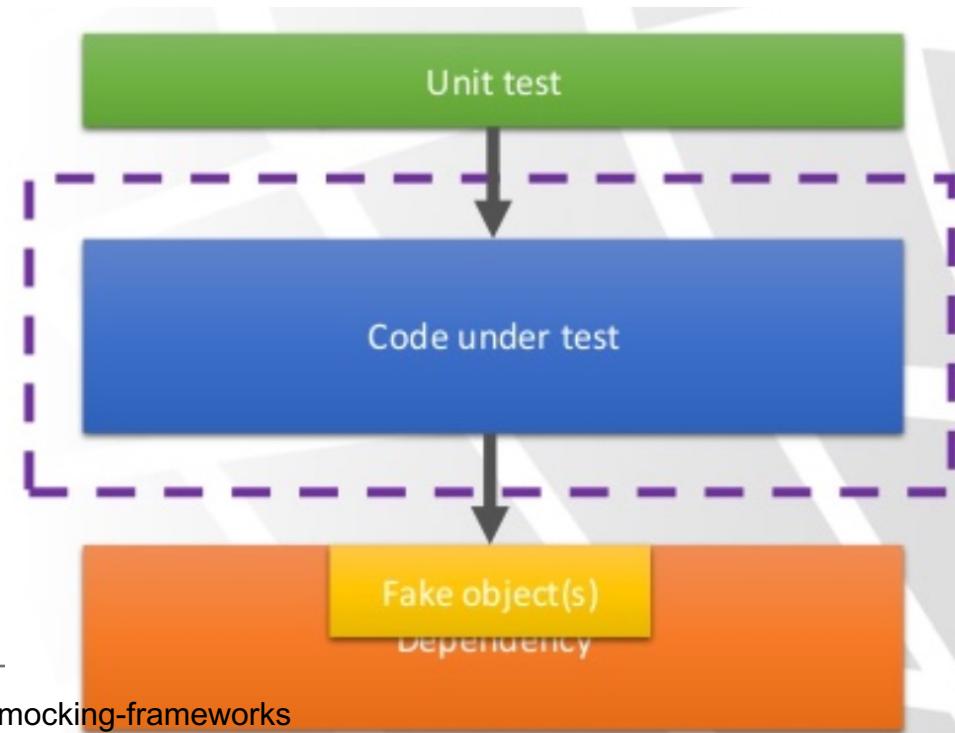
Stellvertreterobjekte – Warum sind sie nötig?





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 bereitzustellen zu müssen.**



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 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 AssertionException
    {
        PreConditions.checkState(displayMsgCalled,
            "method 'displayMsg' has not been called");
    }
}
```



PART 5: Design For Testability





Sollbruchstellen





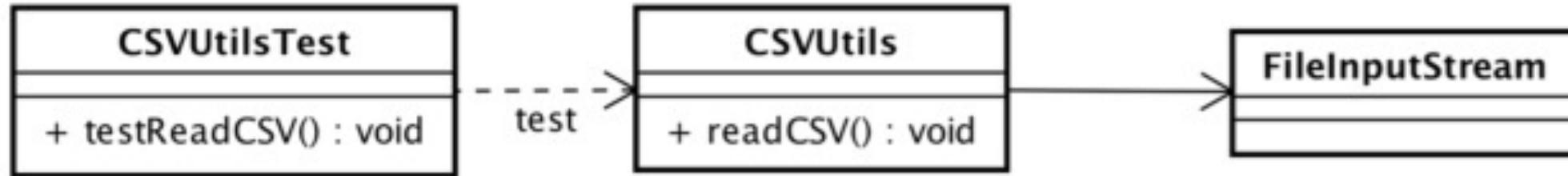
Sollbruchstellen – Injection Points

- 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)

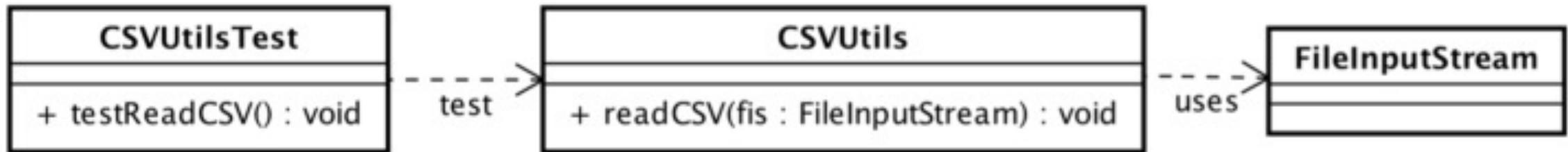


Sollbruchstellen – Injection Points

- Oftmals erschweren direkte Abhangigkeiten das Testen



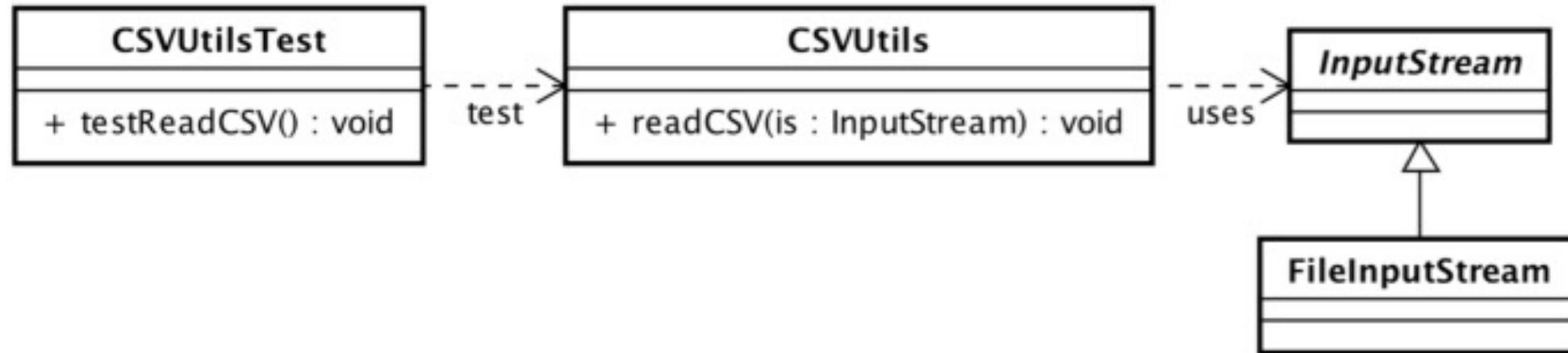
- Indirektion nutzen: Method Injection



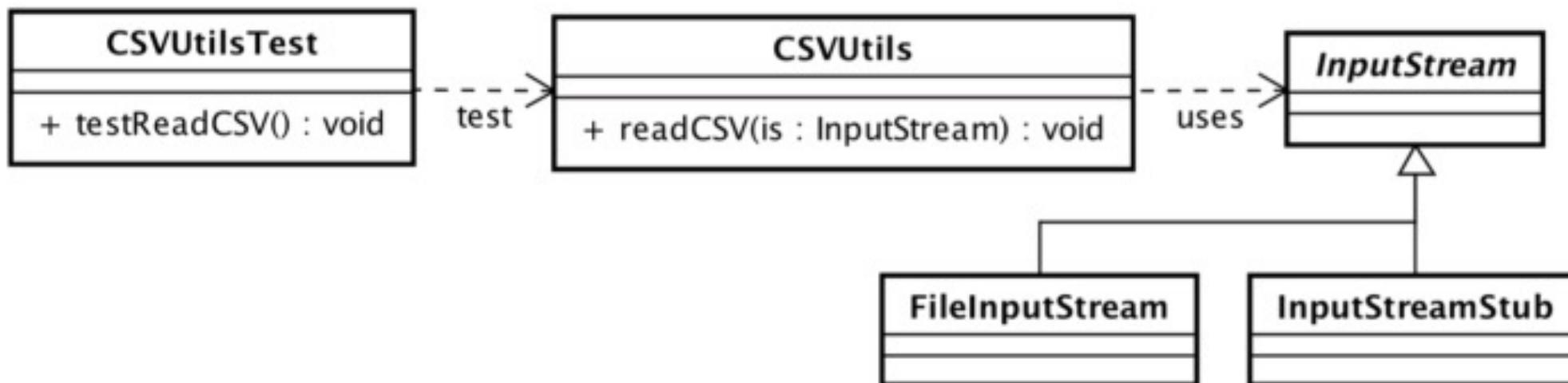


Sollbruchstellen – Injection Points

- Abstraktion nutzen



- Stub zur Testbarkeit nutzen







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

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>3.11.2</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>3.11.2</version>
    <scope>test</scope>
</dependency>
```

```
dependencies {
    testCompile 'junit:junit:4.13'
    // Mockito
    testCompile "org.mockito:mockito-core:3.11.2"
    testCompile "org.mockito:mockito-junit-jupiter:3.11.2"
}
```

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); ← das durch mock()  
    when(greeting.greet()).thenReturn("Changed by Mockito"); ← erstellte Test-Double kein  
    // Act  
    final String result = greeting.greet();  
    // Assert  
    assertEquals("Changed by Mockito", result); ← Zustandsbasiertes Testen  
}
```

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

zustandsbasiertes Testen

Mockito Basics



- Mocking nutzt man für **Kollaboratoren**, 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

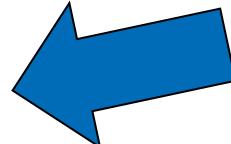


```
@Test
public void testGreetingReturnValue()
{
    // Arrange
    final Greeting greeting = Mockito.mock(Greeting.class);

    // 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());
}
```



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 übergebenen Mocks auf
    InOrder inOrder = Mockito.inOrder(firstMock, secondMock);

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

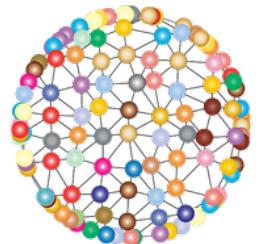
    // Prüfe, dass die Aufrufe in der richtigen Reihenfolge erfolgen
    inOrder.verify(firstMock).methodOne();
    inOrder.verify(secondMock).methodTwo();
    inOrder.verify(firstMock).methodThree();
}
```

Mockito: Auf die Dosis kommt es an ☺



FUN





**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();  
}
```

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



Questions?

Hilfe





- **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 \(Hamcrest vs. AssertJ\)](https://de.slideshare.net/tsveronese/assert-j-techtalk (Hamcrest vs. AssertJ))



Thank You