



# JUnit 5 Workshop

**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](mailto:michael_inden@hotmail.com)

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

Kurse: Bitte sprecht mich an!

[https://github.com/Michaeli71/JUnit5\\_Workshop\\_2Days](https://github.com/Michaeli71/JUnit5_Workshop_2Days)





# 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
  - Repeated 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

# 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



# 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 aufzudecken 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**
-



## External Quality entspricht Benutzersicht

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





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

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



# Warum testen wir? Was ist Qualität?

---

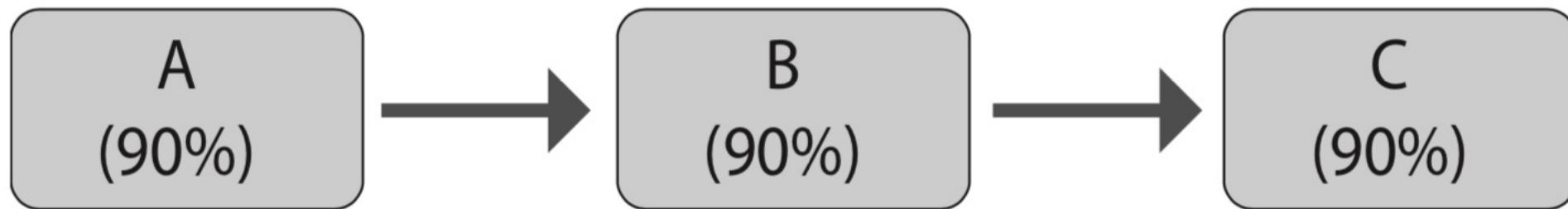


- Unter **Qualität** versteht jeder leicht etwas anderes:
  - gute Benutzbarkeit oder
  - umfangreiche Funktionalität.
  - Immer jedoch hohes Maß an **Erwartungskonformität** und **Zuverlässigkeit** gewünscht
  - Im richtigen Leben erwarten wir fast immer eine Qualität von **nahezu 95 – 100 %**.
- **Wieso geben wir uns bei Software oftmals mit weniger zufrieden?** Was könnten die Ursachen nicht optimaler Qualität sein?
  - Komplexität ist in Software häufig recht hoch
  - Einzelteile sind zum Teil nicht gut getestet
  - Informatik hat noch nicht den Ingenieursgrad wie Autoindustrie oder Maschinenbau



## Einfluss der Qualität der Einzelteile

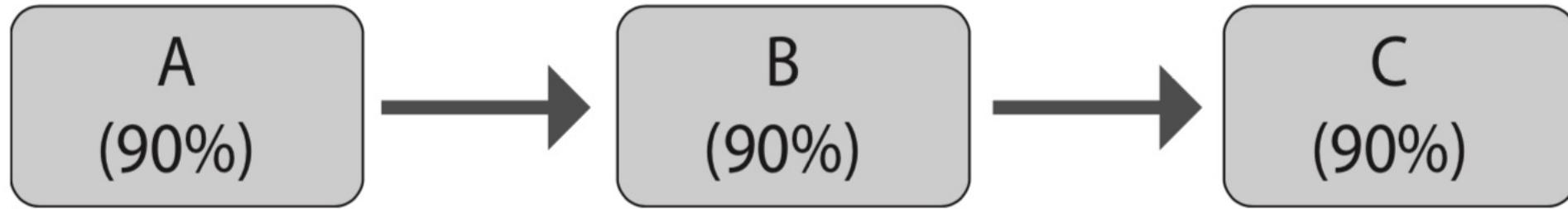
- Die Industrie besitzt **Normen** und **Standards**. Qualität sowie Interoperabilität ist gegeben.
- Je höher die Qualität der Einzelteile, desto höher ist auch die Qualität des Gesamtprodukts. Schauen wir auf ein einfaches System mit drei Bausteinen:



- Wie hoch ist die Gesamtqualität?



## Einfluss der Qualität der Einzelteile



- Laut **Systemtheorie**: Produkt der Einzelqualitäten

$$0.9 * 0.9 * 0.9 = 0.73 \Rightarrow 73 \%$$

- Auf Software übertragen, haben wir dabei folgende Probleme:
  - Wer hat schon einmal Systeme mit nur 3 Klassen oder Bestandteilen gebaut?
  - Normale Anwendungen bestehen nicht nur aus einer Vielzahl an Klassen und Objekten, sondern diese besitzen insbesondere auch komplizierte, teils verzwickte Abhängigkeiten.

## Wieso erstellen wir Unit Tests?

---



- Wir betreiben damit **Qualitätssicherung auf Kleinteilebene**.
  - Enge Integration in den Entwicklungsprozess => **schnelles Feedback**
  - Meistens lassen sich die Fehler auch leicht korrigieren.
- 
- **Kommen wir nochmal aus Standardisierung und Qualität zurück: Jeder Einzelne von uns kann ein wenig zur Verbesserung beitragen, indem man sich an Coding Conventions hält, gute Unit Tests schreibt und auf ein sauberes Design achtet.**

## Was macht einen guten Unit Test aus?



A good Unit Test should be:



Easy  
to write

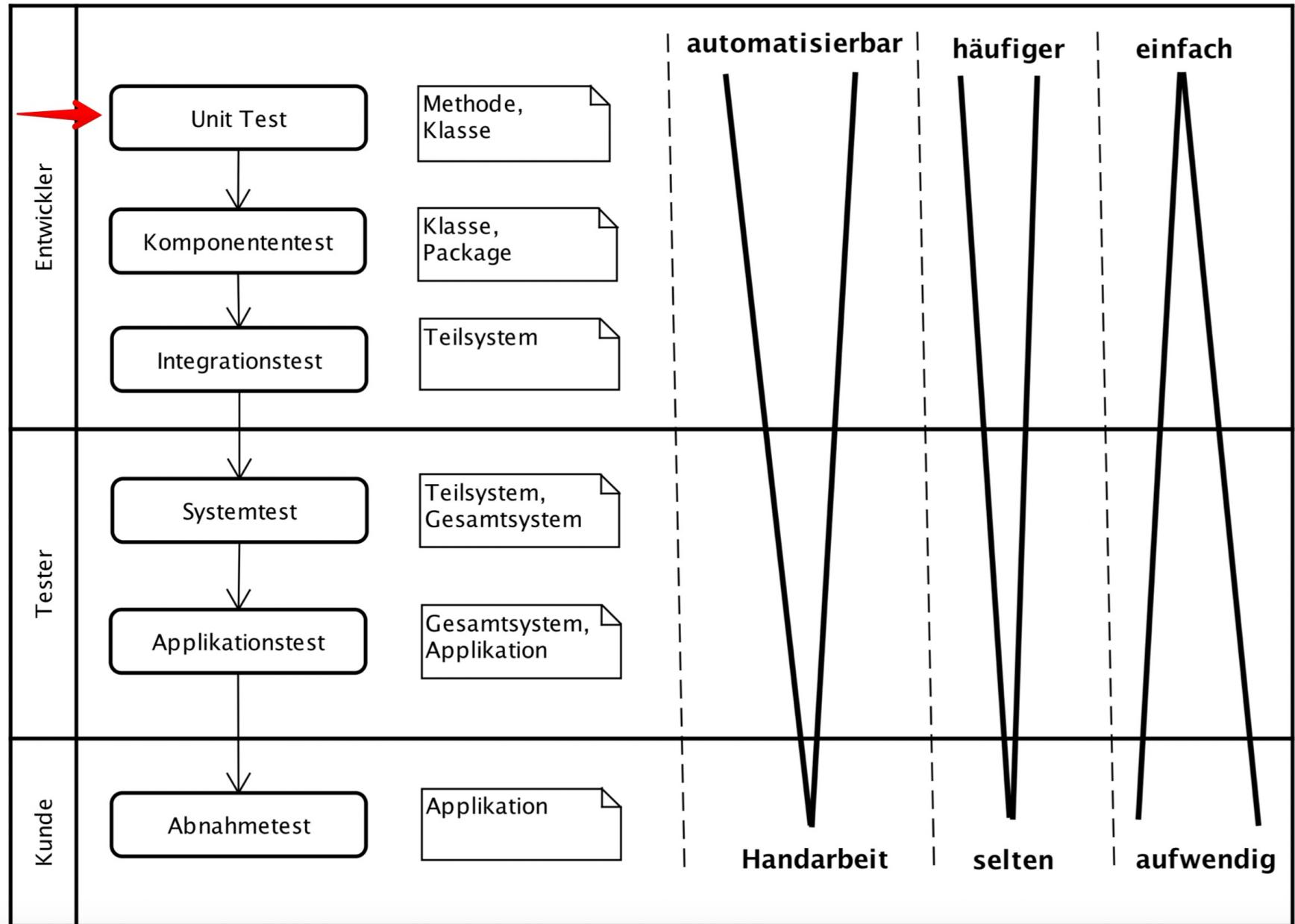


Simple  
to read

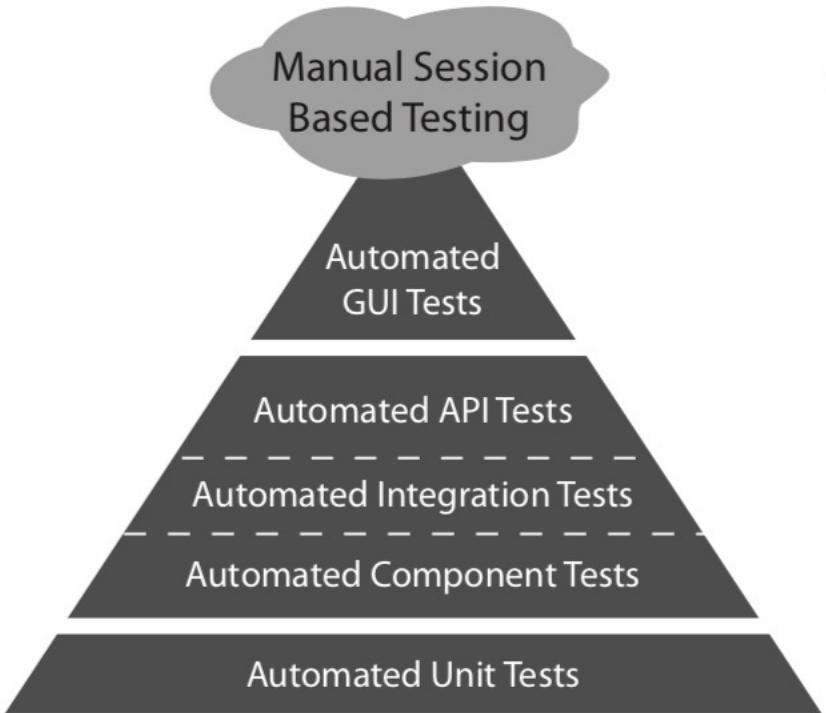


Trivial  
to maintain

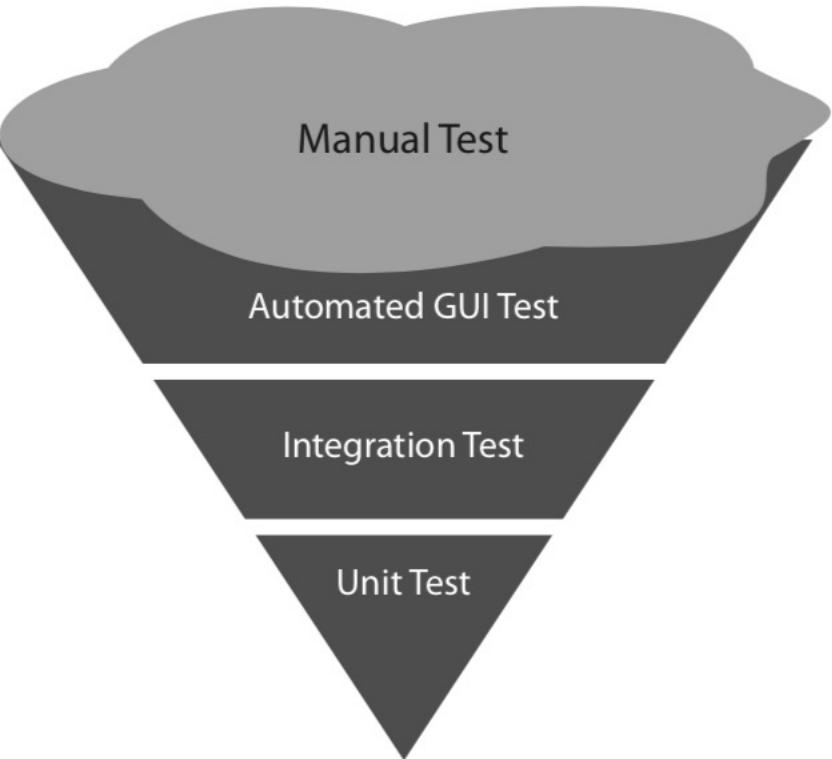
# Arten von Tests



# Testpyramide



The Ideal Testing  
Automation Pyramid



The Non-Ideal Testing  
Automation Inverted Pyramid



---

# Was erschwert uns, Tests zu schreiben?

hands  
on



## Schwierigkeiten beim Unit Testen

---

- Was erschwert es uns, gute Unit Tests zu schreiben? – Es sind es fast dieselben Sachen, die schon das Entwickeln erschweren können.
  - Man sollte **VORAB** wissen, was zu realisieren ist und wie es funktionieren soll.
  - Wenn man zu den Anforderungen die passende Software entwickelt, dann sind eine gelungene **Namensgebung** und ein **problemangepasstes Design**, insbesondere **klare Zuständigkeiten**.
  - Oftmals sieht die Realität anders aus und Klassen bieten zu viel Funktionalität. Dadurch existieren meistens zu viele Abhängigkeiten auf andere Klassen.
  - Das wiederum führt zu Fragilität, ändert man an einer Stelle, so zerbricht etwas an anderer.

## Schwierigkeiten beim Unit Testen

---



- Für Unit Tests helfen uns Informationen zur gewünschten Funktionalität, um adäquate Testfälle erstellen zu können und nicht nur Pseudotests für triviale get() / set() -Methoden zu schreiben.
- gelungene, **aussagekräftige Namen** der Testmethoden helfen das Wesentliche des Testfalls erkennen zu können
- Um jeden Testfall möglichst knackig formulieren zu können, sollten **nicht zu viele Abhängigkeiten** von anderen Klassen existieren.
- Abhilfe bieten die im später vorgestellten **Test Doubles**. Der Name ist plakativ und ähnelt dem **Stunt Double** aus Filmen: Ein Test Double ist ein Stellvertreter für ein Anwendungsobjekt, um Abhängigkeiten aufzulösen und eine Unit besser testbar zu machen.



# Gute Angewohnheiten



# (Eclipse) Plugin MoreUnit



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

**Eclipse Marketplace**

Select solutions to install. Press Install Now to proceed with installation.  
Press the "more info" link to learn more about a solution.



Search Recent Popular Favorites Installed  Giving IoT an Edge

Find:  × All Markets All Categories Go

**MoreUnit 3.3.0**

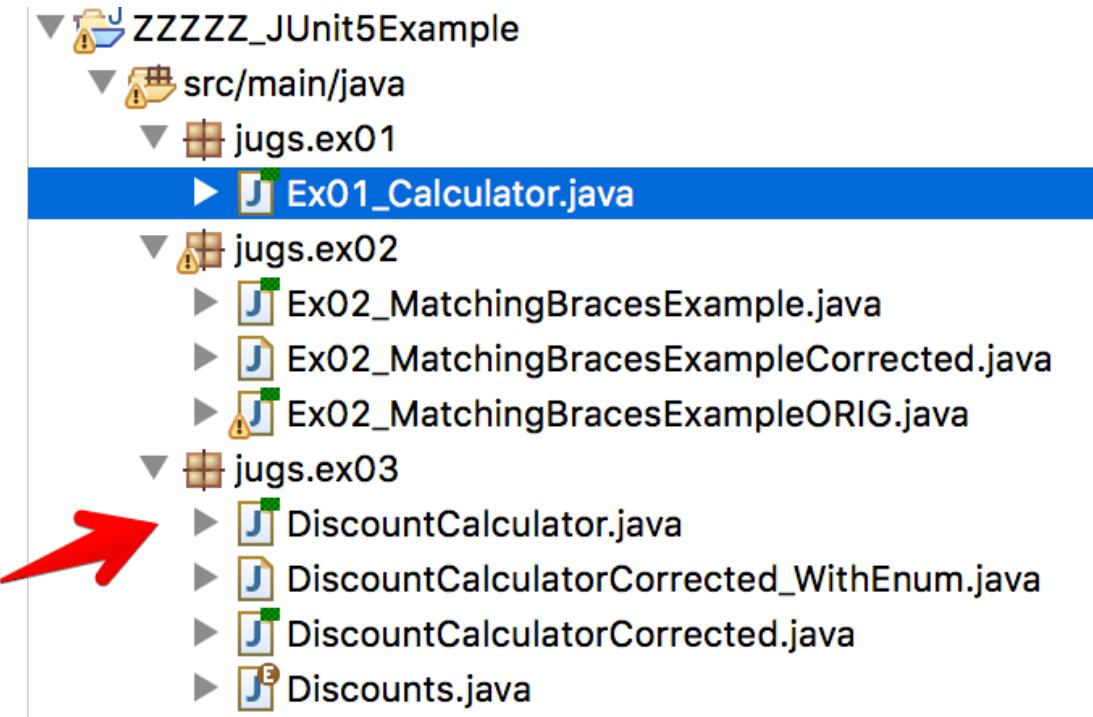
MoreUnit is an Eclipse plugin that should assist you in writing more unit tests. It supports all programming languages (switching between tests and classes under... [more info](#))  
by [\\_](#), EPL  
[test](#) [Favorite](#) [junit](#) [testing](#) [mock](#)

 561  Installs: **128K** (646 last month) Installed

## (Eclipse) Plugin MoreUnit



- Tastaturkürzel zum Ausführen (CTRL+R) und zum Wechseln zwischen Klasse und Test (CTRL+J).
- Icon-Dekoration im Package Explorer:  
grünen Punkt zeigt, ob zu einer Klasse ein Test existiert.
- Refactorings: Klassen und korrespondierende Testklassen werden synchron zueinander verschoben oder umbenannt.



# (Eclipse) Plugin MoreUnit



New JUnit Test Case

**JUnit Test Case**

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

JUnit 3  JUnit 4  JUnit 5  Spock  TestNG

Source folder: ZZZZ\_JUnit5Refresher/src/test/java

Package: newinrefresher.parameterized

Name: Ex01\_LeapYearTest

Superclass:

Which method stubs would you like to create?

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

Do you want to add comments? (Configure templates and default value [here](#))  
 Generate comments

Class under test: newinrefresher.parameterized.Ex01\_LeapYear

New JUnit Test Case

**Test Methods**

Select methods for which test method stubs should be created.

Available methods:

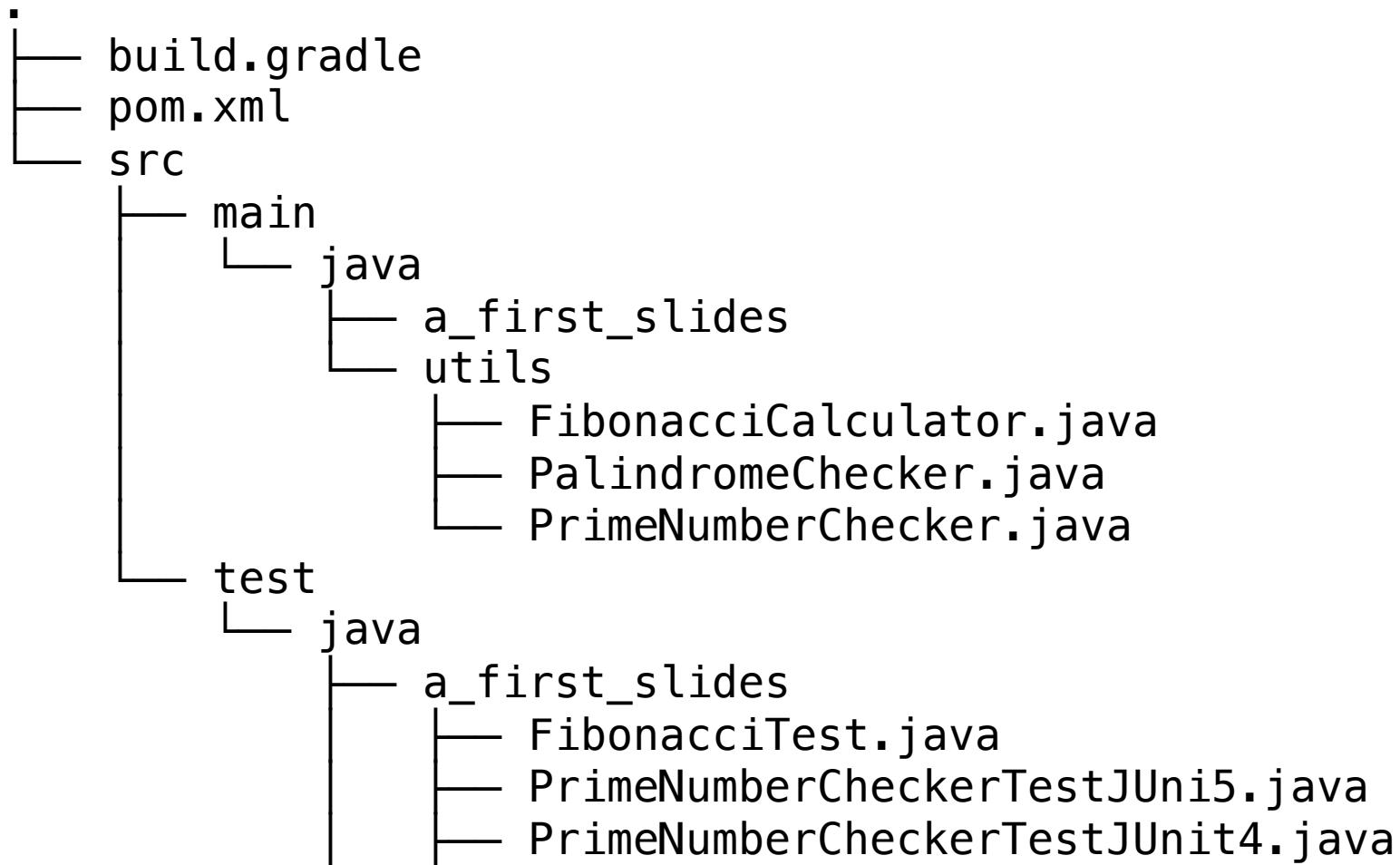
- ▼ C Ex01\_LeapYear  
  ● S main(String[])  
  △ S isLeap(int)  
  ▼ C Object  
    ● C Object()  
    ● NF getClass()  
    ● N hashCode()  
    ● equals(Object)  
    ● N clone()  
    ● toString()  
    ● NF notify()  
    ● NF notifyAll()  
    ● F wait()  
    ● NF wait(long)  
    ● F wait(long, int)  
    ● finalize()

1 method selected.

Create final method stubs  
 Create tasks for generated test methods

# 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()**



## 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?**



## Testfälle definieren

---

- **JUnit** ist ein in Java geschriebenes Framework, das beim Schreiben und bei der Automatisierung von Testfällen auf Klassenebene unterstützt.
- Zum Testen einer Applikationsklasse wird normalerweise eine korrespondierende **Testklasse** geschrieben, in Methoden werden Testbehauptungen aufgestellt und ausgewertet
- Häufig beginnt man zur **Absicherung wichtiger Funktionalität** eigener Klassen damit, **zunächst einige zentrale Methoden durch Tests zu überprüfen**.
- Dies kann anschließend **schrittweise ausgeweitet** werden.



## Beispiel: Ein erster Unit Test mit JUnit 5

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

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

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

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



- **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

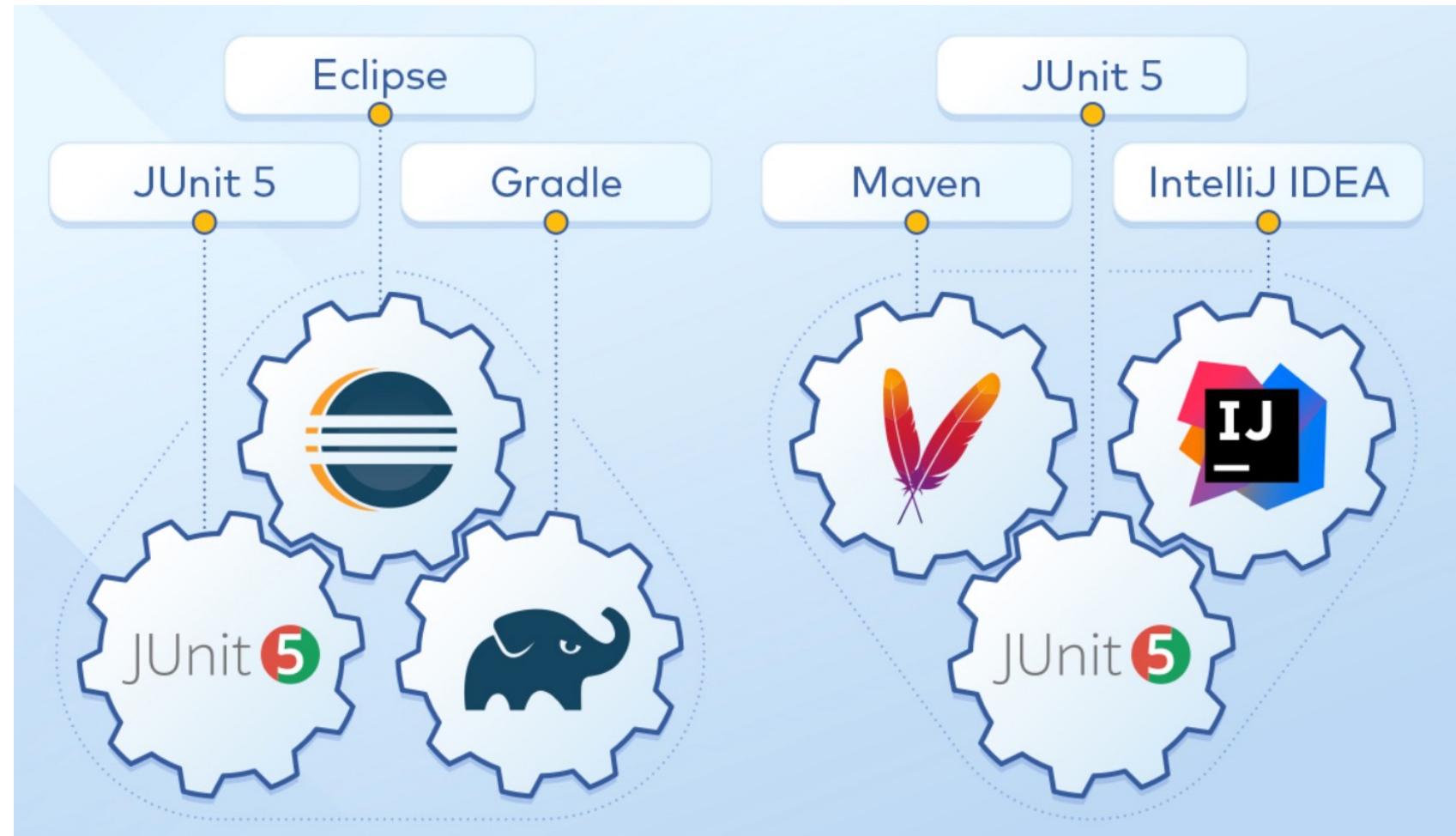
---

- Qualität der Einzelbausteine hat einen großen Einfluss auf die Qualität des Gesamtsystems.
- Unit Tests = Qualitätssicherung auf Kleinteilebene und erhöht die innere Qualität.
- Dabei helfen folgende Eigenschaften von Unit Tests:
  - **Klares Ergebnis** – bestanden (Grün) oder Fehler (Rot).
  - **Messbar** – Die Anzahl der Testfälle und die damit geprüften Programmteile (Testabdeckung) lassen sich leicht auswerten.
  - **Implementierungsnahe und fokussiert** – Wenn man Unit Tests parallel zur Implementierung erstellt und ausführt, erhält man schnell Feedback und oftmals lassen sich Fehler auch leicht korrigieren.
  - **Know-how-Gewinn** – Man beschäftigt sich automatisch intensiver mit den Anforderungen und den bereits bestehenden Implementierungen. Es baut sich dadurch kontinuierlich ein gutes Verständnis auf.
  - **Wiederholbar** – Nach Änderungen im Sourcecode können Tests erneut ausgeführt werden. Bei positiver Ausführung erhöht dies die Sicherheit, keine Defekte eingefügt zu haben.
- Die obigen Punkte verdeutlichen: Von gut formulierten Unit Tests profitiert man am meisten, wenn man diese regelmäßig ausführt.



## Tests (regelmässig) ausführen

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



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

## 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.



# Exercises Part 1

[https://github.com/Michaeli71/JUnit5 Workshop](https://github.com/Michaeli71/JUnit5_Workshop)





# 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

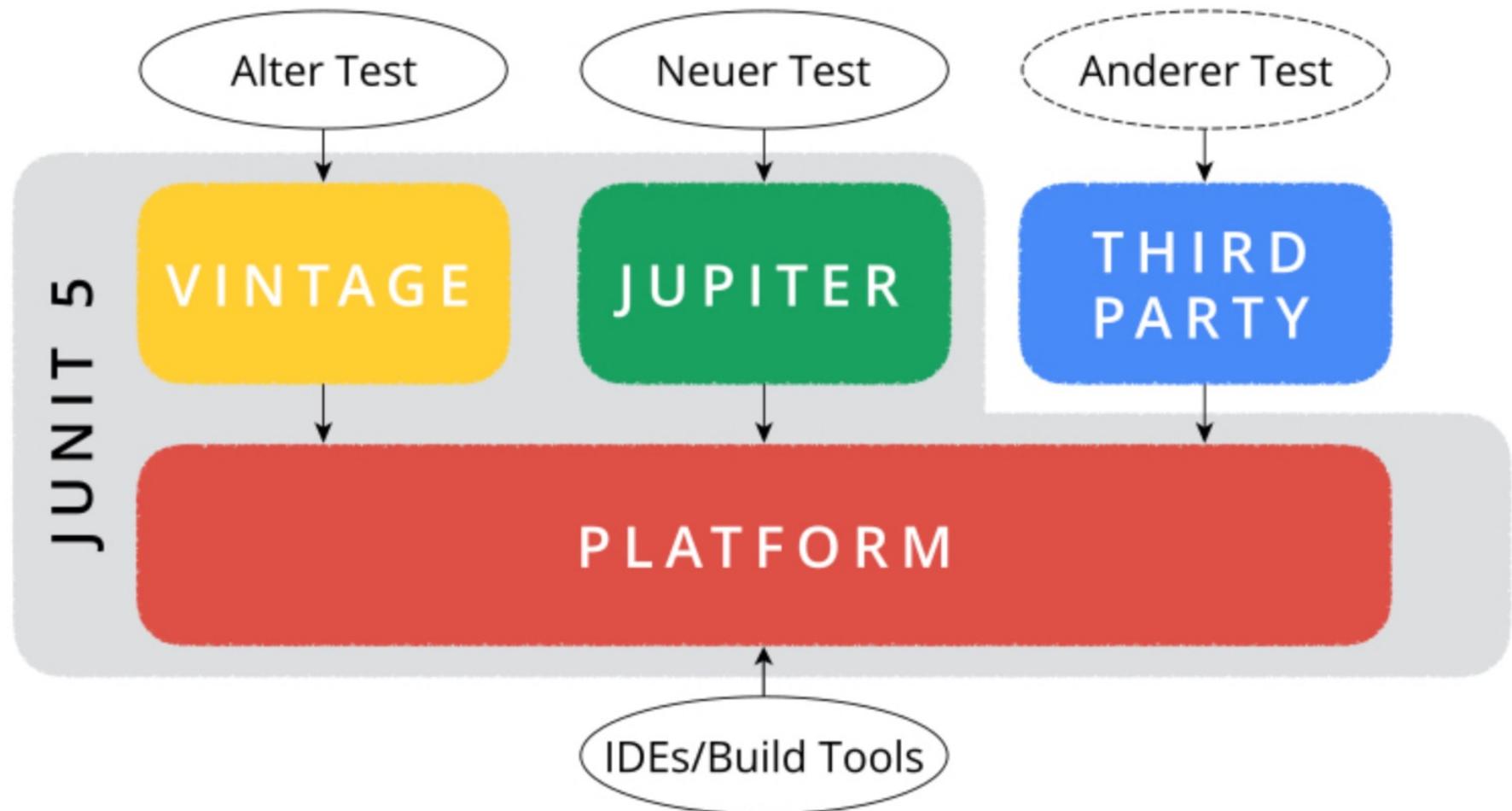
Q & A

Support JUnit



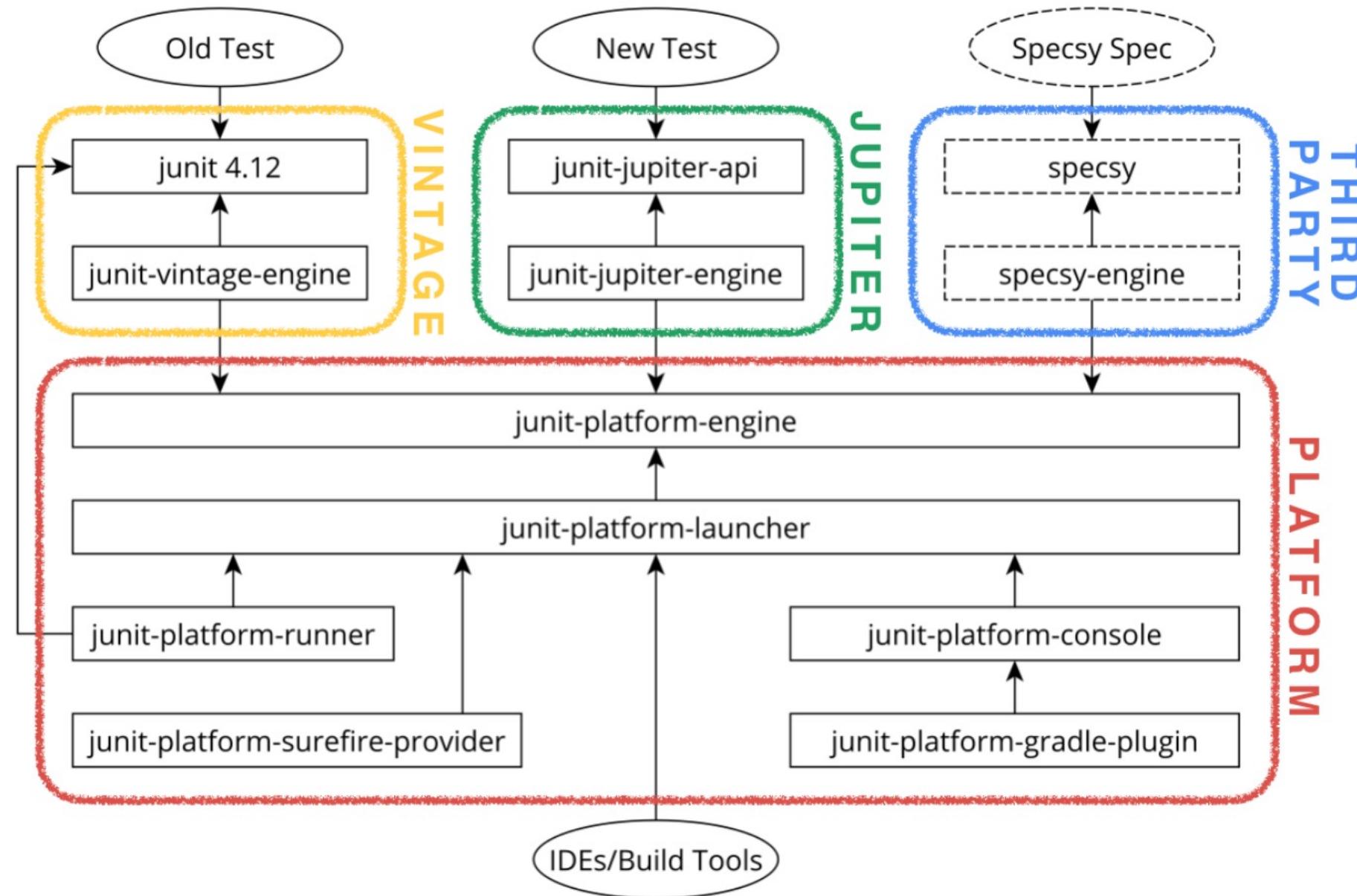
## Architektur

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



# 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");

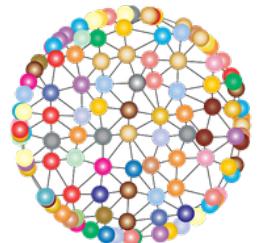
    assertFalse(false);
    assertNull(null);
    assertNotNull(new Object());
    assertSame(null, null);
    assertNotSame(null, new Object());
}
```



## 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;  
}
```

▼ B\_DelayedMsgCreationTest [Runner: JUnit 5] (1.993 s)  
  x withMessageSimple() (1.993 s)

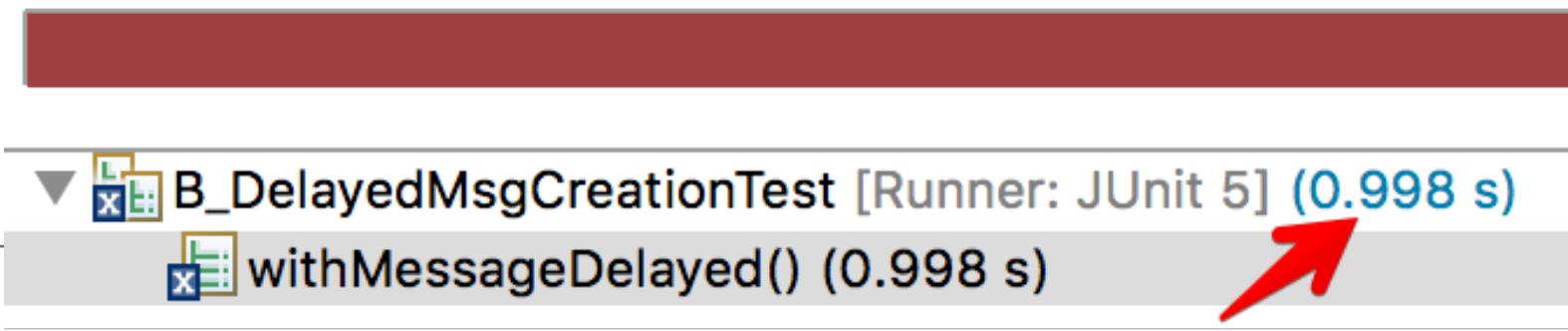


**Was ist daran  
unschön?**

## 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"));  
}
```

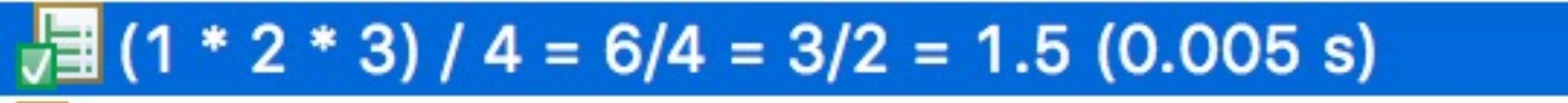




## Spezielle Testnamen

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

▼ 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 Klassifikationen

---

- Spezielle Klassifikationen mit der Annotation `@Tag`

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



## Kontextinfos TestInfo

---

- TestInfo dient als Ersatz für die JUnit 4 Rule TestName
- **Dazu: Parameter in Testmethoden möglich!**

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

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

# Tag basierte Ausführung

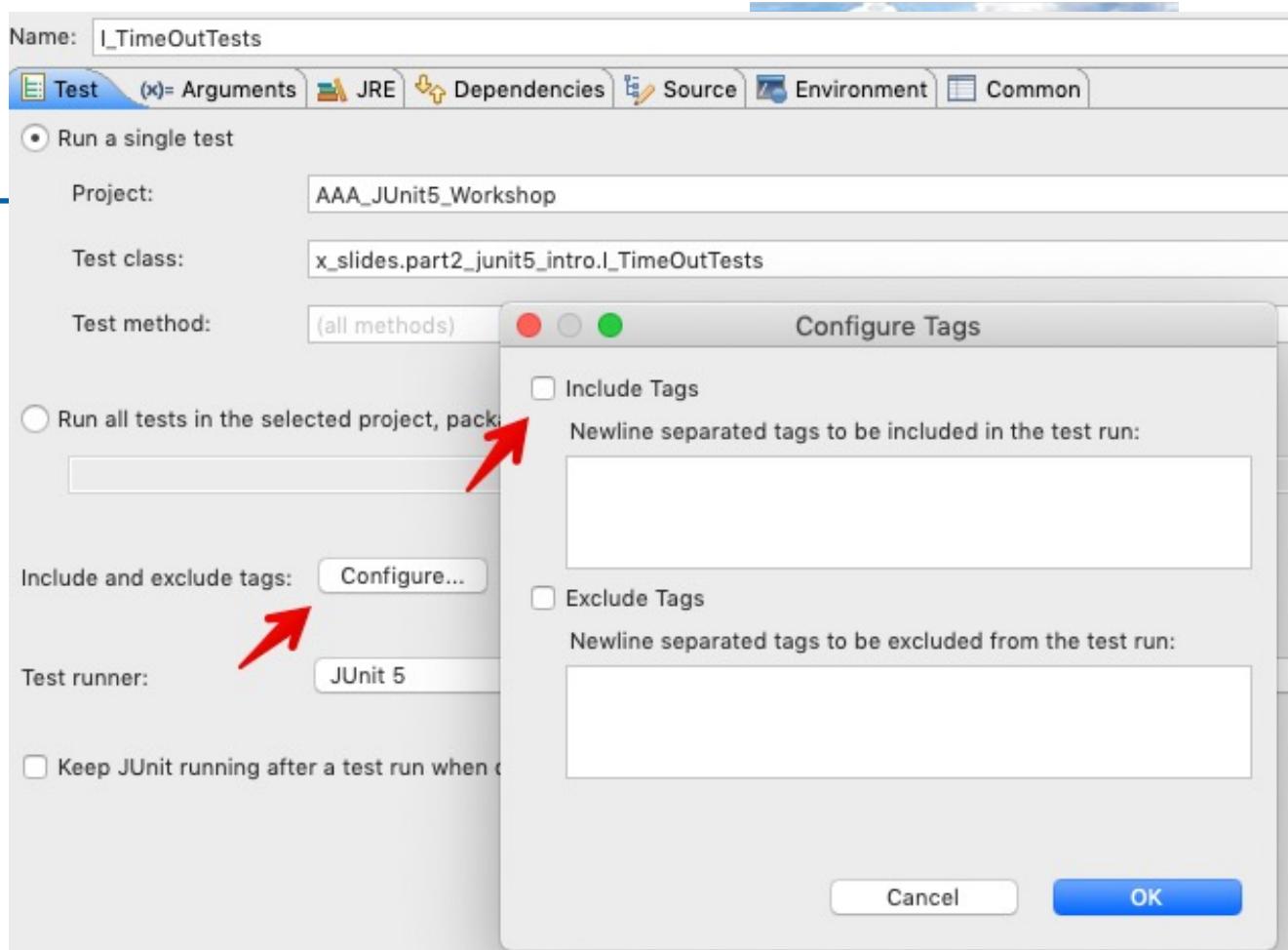
- **Gradle**

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

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

- **Maven**

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





# Spezielle Assertions





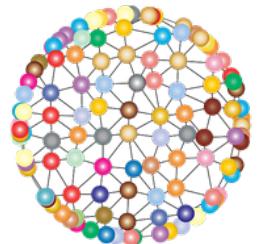
## Multiple Asserts

---

```
@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)

---

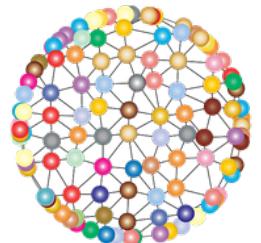
```
@Test
void multipleAssertsforOneTopic_Diff2() {
    Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");
    assertAll(() -> assertEquals("Tim", mike.name),
              () -> assertEquals(LocalDate.of(1971, 3, 27), mike.dateOfBirth),
              () -> assertEquals("Kiel", mike.homeTown));
}
```

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?



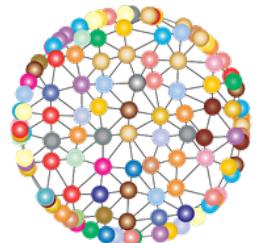
# Spezielles Handling bei Gleitkommazahlen



```
@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)  
     π = 3.1415 (with four digit precision) (0.000 s)



# 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);  
}
```

▼	G_ArrayEqualsWrong [Runner: JUnit 5] (0.001 s)
	arraysCompare() (0.000 s)
	nestedArraysCompare() (0.000 s)

# Arrays RICHTIG vergleichen

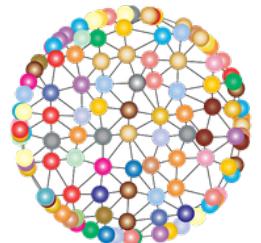


```
@Test  
void arraysCompare()  
{  
    final String[] words = { "Word1", "Word2" };  
    final String[] expected = { "Word1", "Word2" };  
  
    assertEquals(expected, words);  
}
```

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

▼ G\_ArrayEquals [Runner: JUnit 5] (0.000 s)

- arraysCompare() (0.000 s)
- nestedArraysCompare() (0.000 s)



**Wie testen wir  
Collections?**

# Collections vergleichen – problemlos?!



```
@Test  
void listSetCompareSameCollectionType()  
{  
    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);  
}
```



▼	ListTest [Runner: JUnit 5] (0,022 s)
	listSetCompareSameCollectionType() (0,006 s)
	listSetCompareCorrected() (0,010 s)

# Collections vergleichen – Was tun bei abweichendem Typ



```
@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]>

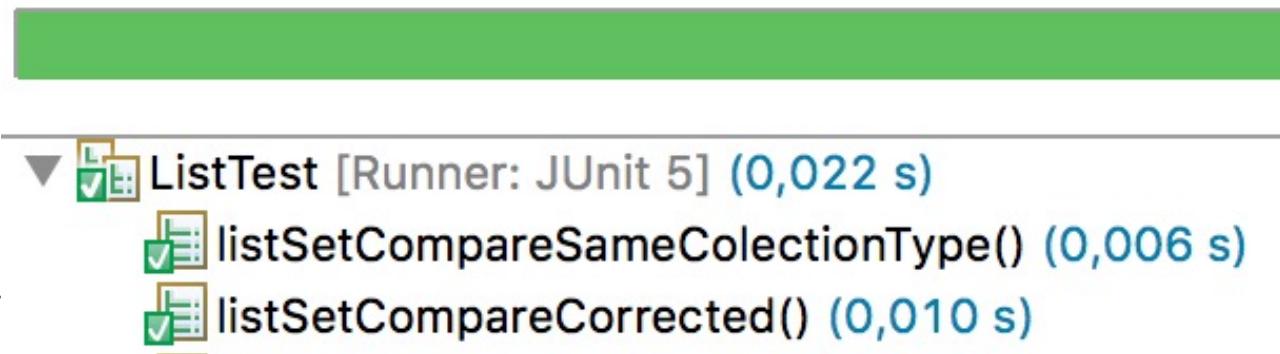
Runs: 3/3 ✘ Errors: 0 ✘ Failures: 1

▼	ListTest [Runner: JUnit 5] (0,010 s)
	listSetCompareSameColectionType() (0,003 s)
	listSetCompareCorrected() (0,002 s)
	listSetCompareWrong() (0,005 s)

# Collections SICHERER vergleichen



```
@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
}
```



## Exceptions prüfen

- Seit JUnit 4 kann eine bei der Testausführung erwartete Exception in der Annotation **@Test** als Parameter **expected** angegeben werden:

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

- Problematisch: Wenn man den Message-Text auswerten möchte
- Auch kein AAA-Stil

## JUnit Rules: Exceptions prüfen

---



- Besser JUnit Rule **ExpectedException**

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

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



## 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");
}
```



---

# Timeout Assertions



## JUnit Rules: Time-outs prüfen (JUnit 4)



- Manchmal soll ein Test nur eine maximale Zeit dauern => Time-out

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

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

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

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

## JUnit Rules: Time-outs prüfen (JUnit 5)



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

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

▼ B\_TimeOutTests [Runner: JUnit 5] (10.696 s)

- testFibRecWithBigNumber() (8.685 s)
- testFibRecWithBigNumber\_Timeout\_Preemptive() (2.011 s)

## JUnit Rules: Time-outs prüfen (JUnit 5)



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

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

B_TimeOutTests [Runner: JUnit 5] (19.489 s)		
✓	testFibRecWithBigNumber() (8.710 s)	
✗	testFibRecWithBigNumber_Timeout() (8.771 s)	
✗	testFibRecWithBigNumber_Timeout_Improved() (2.007 s)	

## JUnit Rules: Time-outs prüfen (JUnit 5)



```
@Test  
void testCalcFib30_With_Timeout_And_Result()  
{  
    // Provide result of calculation if within time out  
    ThrowingSupplier<Long> action = () -> FibonacciCalculator.fibRec(30);  
  
    Long value = assertTimeoutPreemptively(Duration.ofSeconds(1),  
                                         action);  
  
    assertEquals(832040, value);  
}
```

Runs: 4/4      ✘ Errors: 0      ✘ Failures: 2

I_TimeOutTests [Runner: JUnit 5] (20.700 s)		
✓	testFibRecWithBigNumber() (9.136 s)	
✓	<b>testCalcFib30_With_Timeout_And_Result() (0.010 s)</b>	
✗	testFibRecWithBigNumber_Timeout_Preemptive() (2.010 s)	
✗	testFibRecWithBigNumber_Timeout() (9.544 s)	



## JUnit Test (temporär) ausschalten

---

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



# Life Cycle



# JUnit 5 Life Cycle



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

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

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

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

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

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

setupServer  
prepareDatabase  
simpleTest  
cleanupDatabase  
prepareDatabase  
anotherTest  
cleanupDatabase  
tearDownServer



# Ausführungsreihenfolge



# JUnit 4 Test Methoden ordnen (@FixMethodOrder)



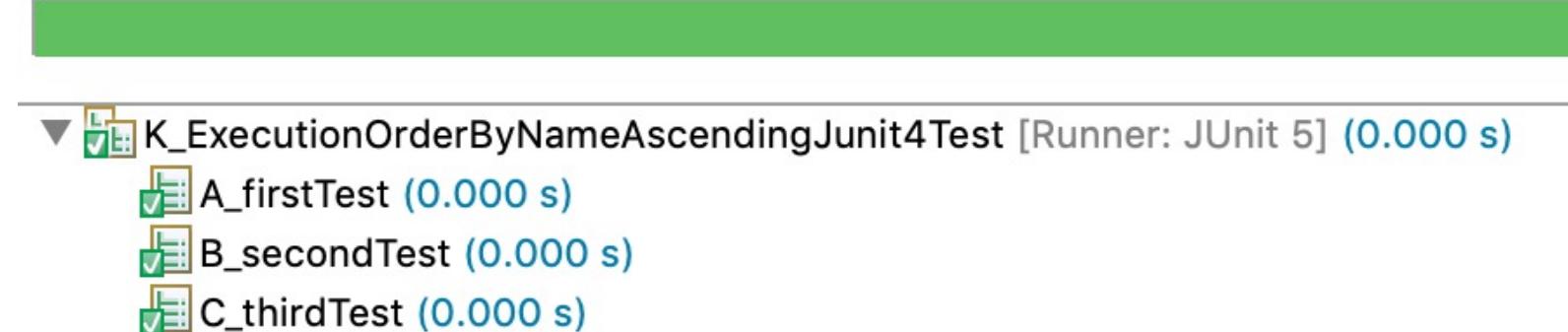
```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class K_ExecutionOrderByNameAscendingJUnit4Test
{
    private final static StringBuilder result = new StringBuilder("");

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

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

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

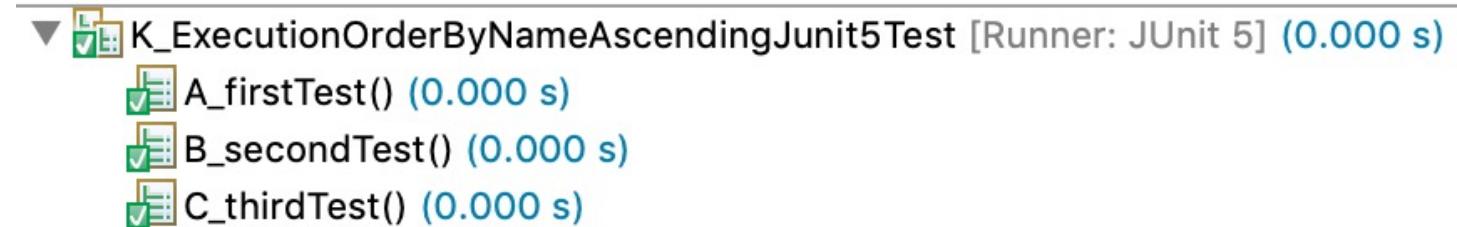
    @AfterClass
    public static void assertOutput()
    {
        assertEquals("abc", result.toString());
    }
}
```



# JUnit 5 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/JUnit5 Workshop](https://github.com/Michaeli71/JUnit5_Workshop)





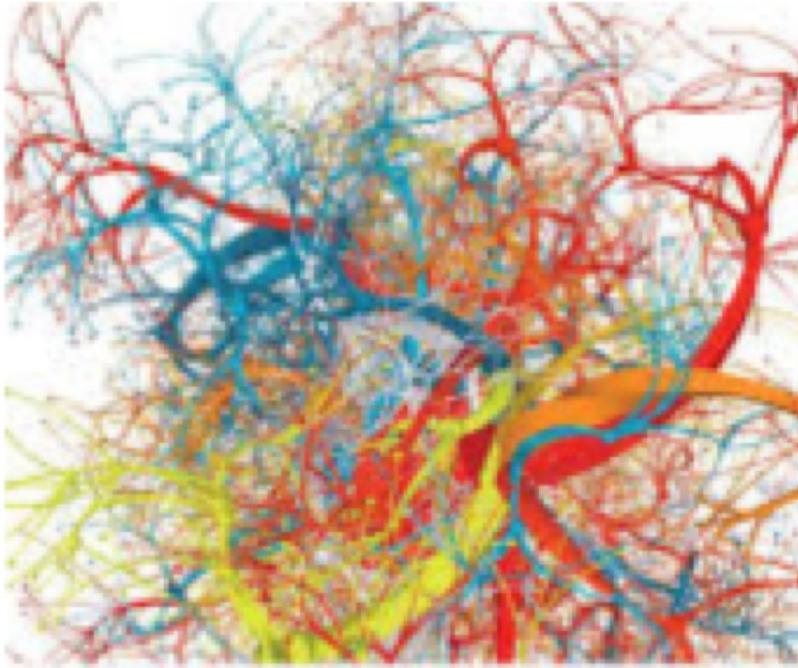
# PART 3

# 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");  
}
```

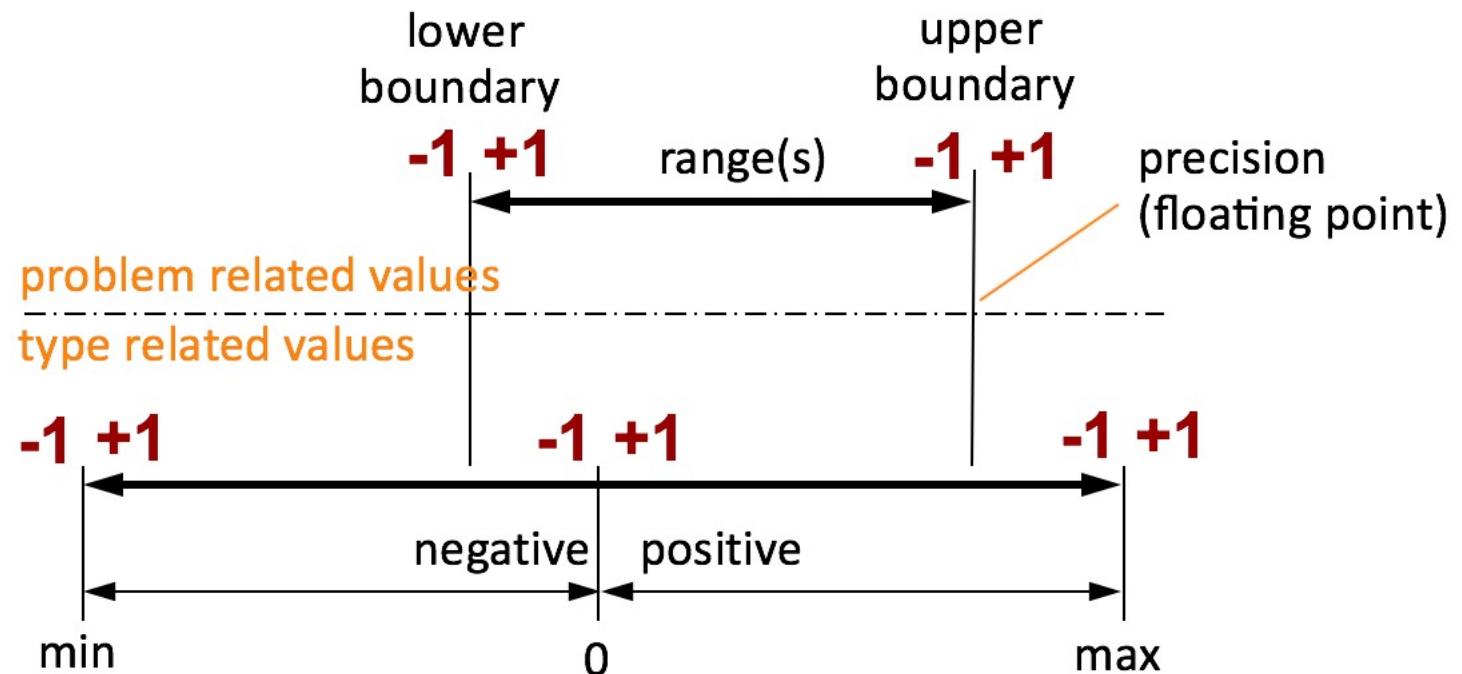
```
@Test  
public void testCalcDiscount_BigOrder_BigDiscount()  
{  
    final int bigAmount = 2000;  
    assertEquals(7, calculator.calcDiscount(bigAmount), "7 % discount");  
}
```

---



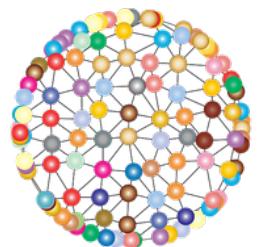
## Grenzwerttests

- **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));
    }
}
```



- Wie testen wir folgende Klasse Adder mit verschiedenen Wertekombinationen?

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

- Schreiben wir also eine Testklasse mit
  - `@RunWith(Parameterized.class)`
  - Einem Konstruktor und allen Eingaben und Expected
  - Einer statischen Methode zum Generieren der Testdaten.
  - Einer Testmethode

```

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

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

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

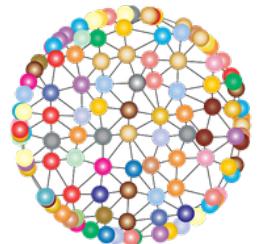
    @Test
    public void sum()
    {
        assertEquals(expected, Adder.add(first, second));
    }
}

```



▼ AdderJUnit4Test [Runner: JUnit 5] (0.000 s)

- ▶ [1 + 2 = 3] (0.000 s)
- ▶ [3 + -3 = 0] (0.000 s)
- ▶ [7 + 2 = 9] (0.000 s)
- ▶ [7 + -2 = 5] (0.000 s)

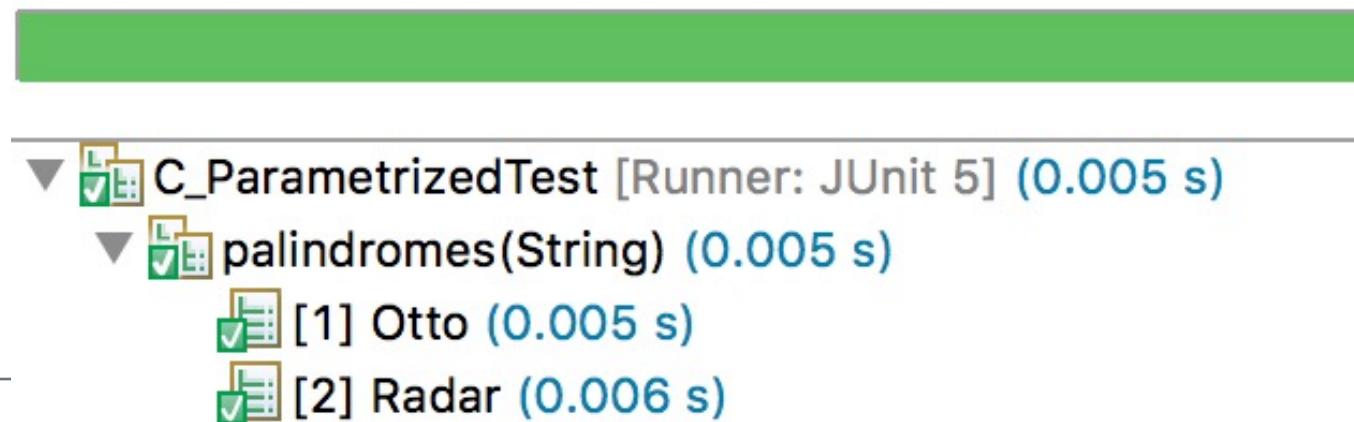


**Das kann es ja jetzt  
nicht wirklich sein?!**

## Parameterized Test – JUnit 5 @ParameterizedTest / @ValueSource



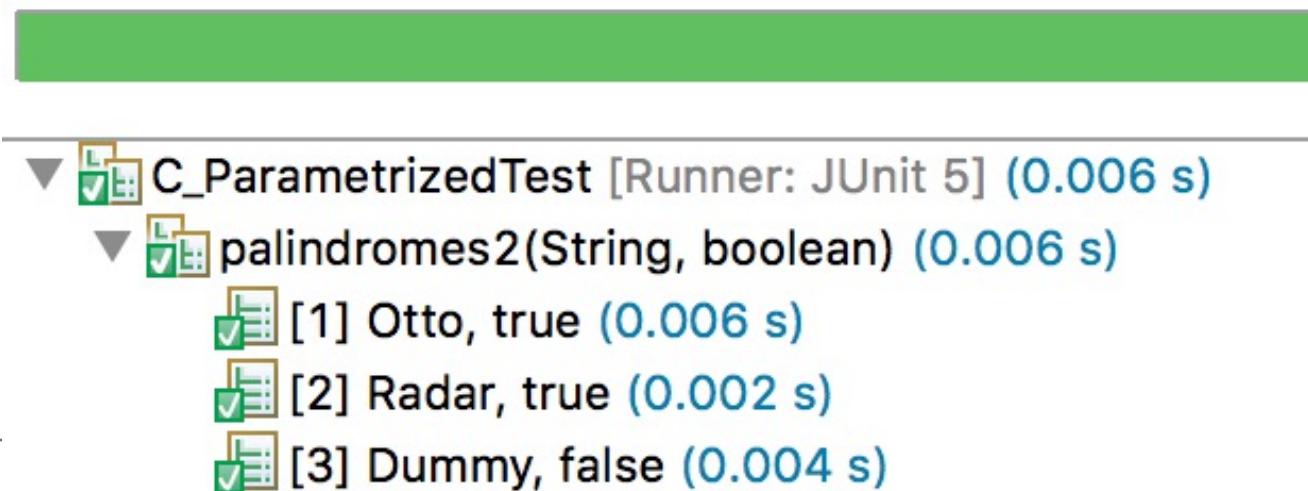
```
@ParameterizedTest  
@ValueSource(strings = { "Otto", "Radar" })  
void palindromes(String candidate)  
{  
    boolean isPalindrome = PalindromeChecker.isPalindrome(candidate));  
  
    assertTrue(isPalindrome);  
}
```



## Parameterized Test – @CsvSource



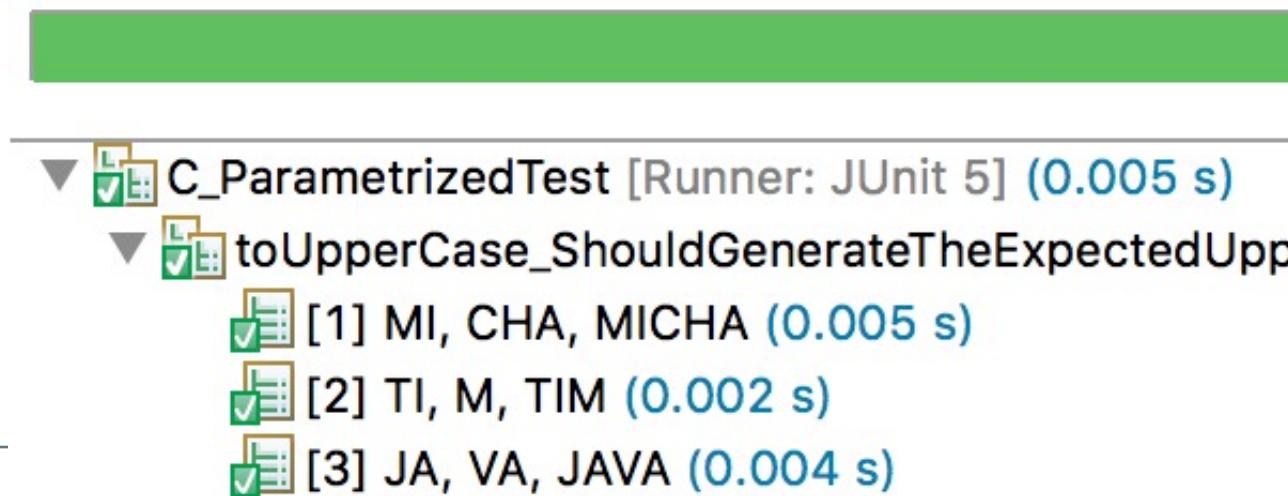
```
@ParameterizedTest  
@CsvSource({ "Otto,true", "Radar,true", "Dummy,false" })  
void palindromes2(String candidate, boolean expected)  
{  
    boolean isPalindrome = PalindromeChecker.isPalindrome(candidate));  
  
    assertEquals(expected, isPalindrome);  
}
```



## Parameterized Test – mehrere Eingaben



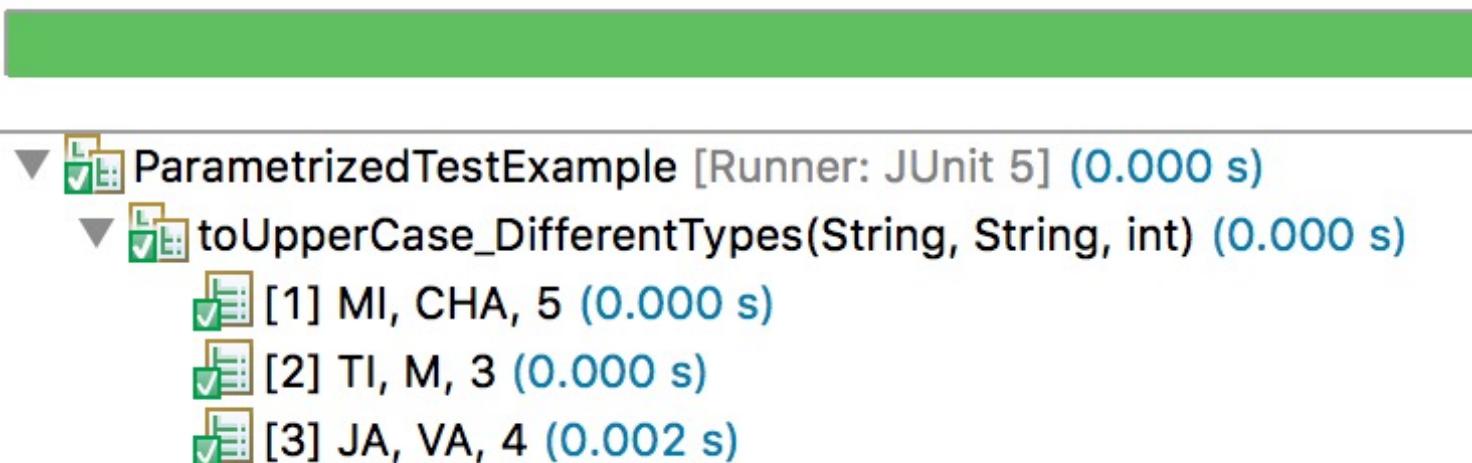
```
@ParameterizedTest  
@CsvSource({"MI,CHA,MICHA", "TI,M,TIM", "JA,VA,JAVA"})  
void toUpperCase_ShouldGenerateTheExpectedUppercaseValue(String input1,  
                                                       String input2,  
                                                       String expected) 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



# Parameterized Test – 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);  
}
```

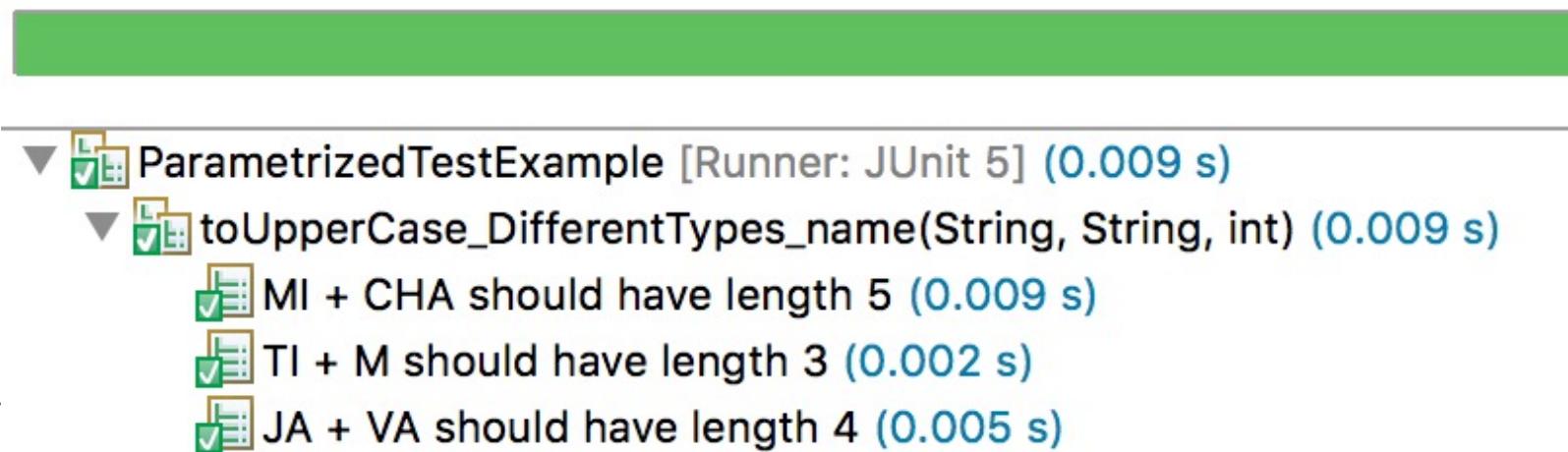


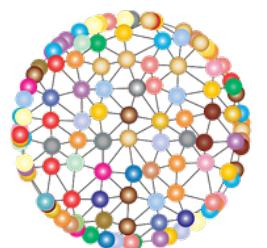
# Parameterized Test – bessere Benennung



```
@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);
}
```





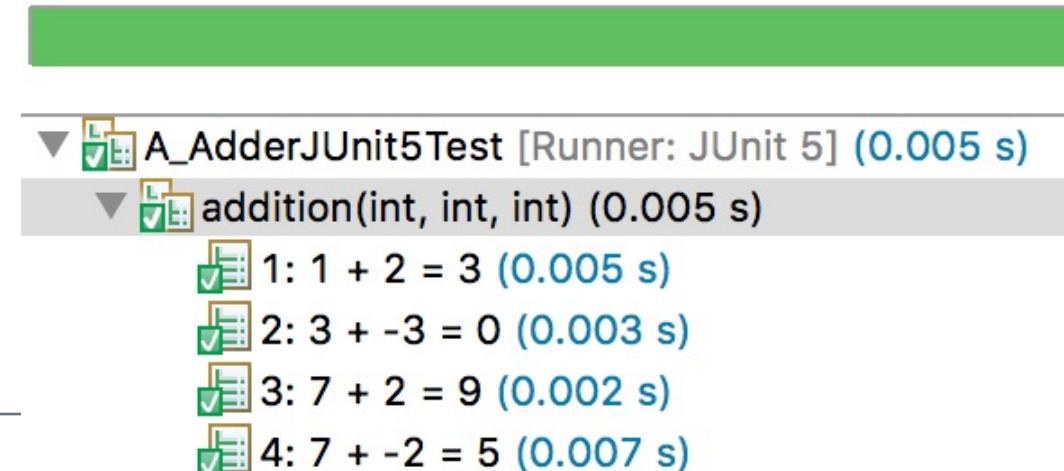
**Wie sieht denn nun mit  
JUnit 5 der Test für den  
Adder aus?**

# 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);
    }
}
```



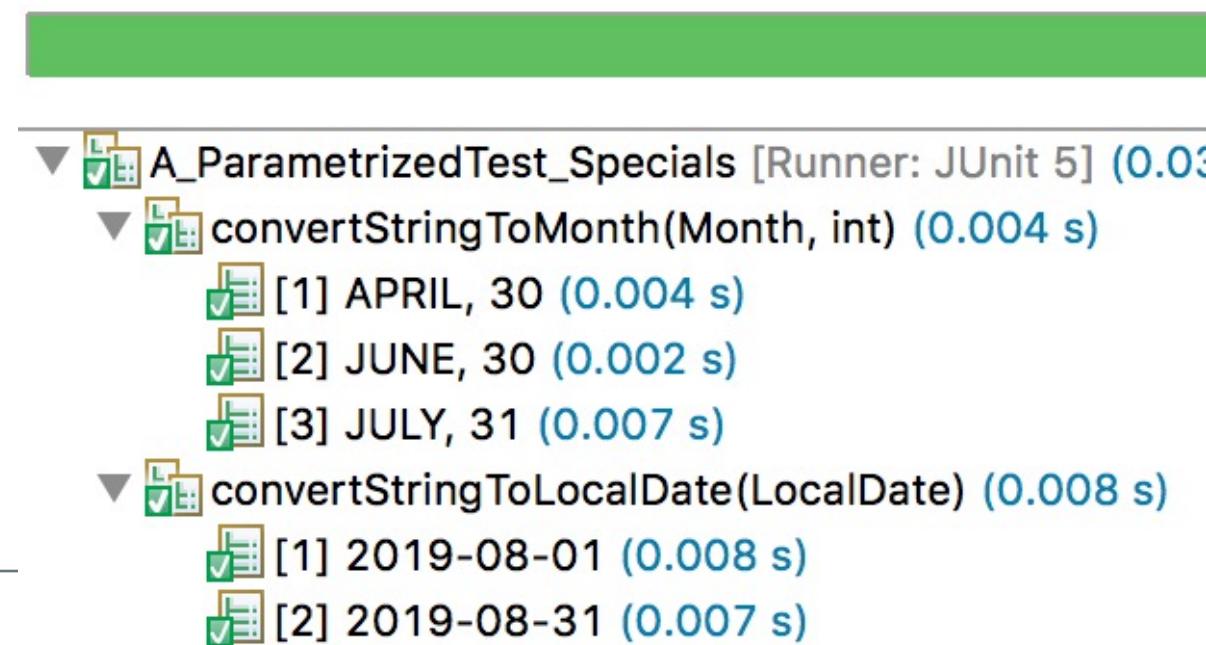
# Parameterized Test – Diverse Konvertierungen und Hilfen



- *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));
}
```

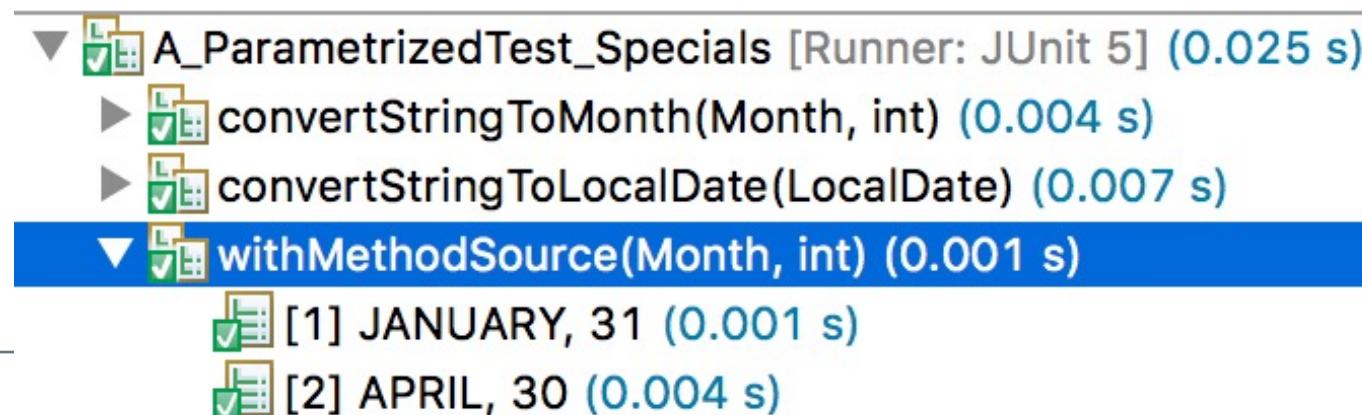


## Parameterized Test – @MethodSource



```
@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));
}
```

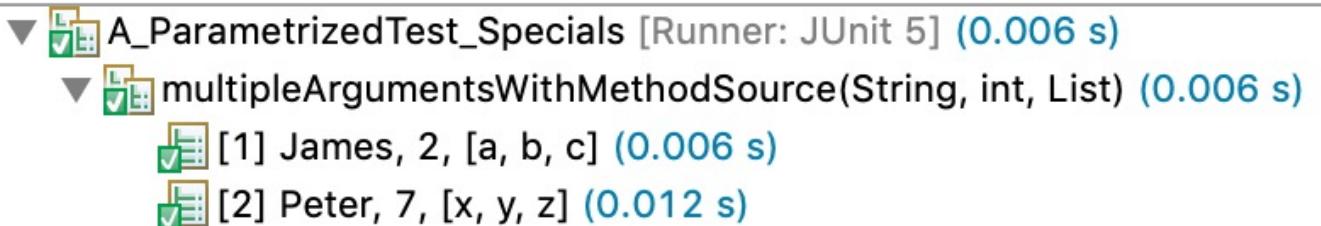


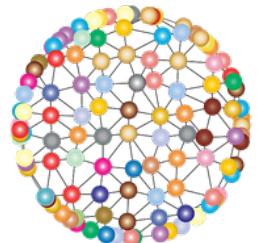
## Parameterized Test – @MethodSource



```
@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);
}

// 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



```
@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);
}
```

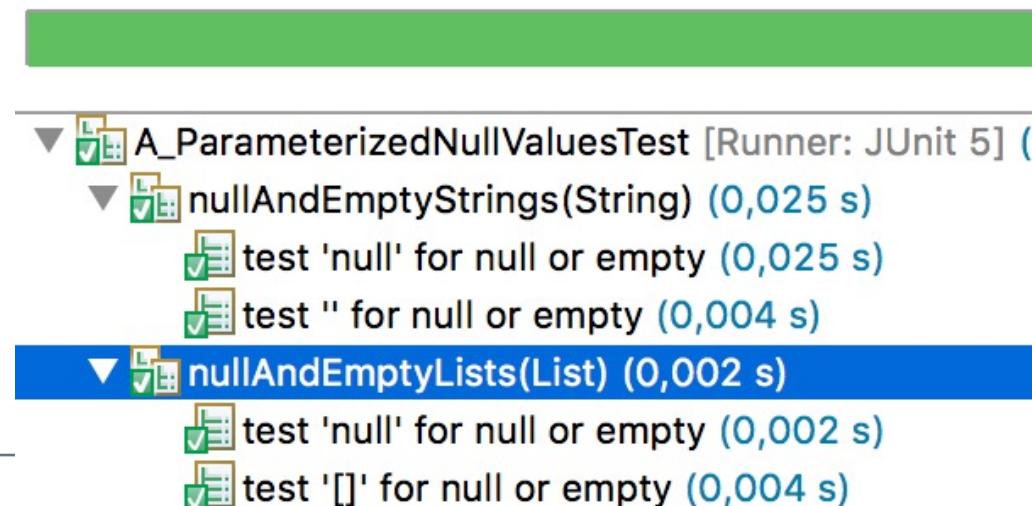
```
arabic,roman
1, I
2, II
3, III
4, IV
5, V
7, VII
9, IX
17, XVII
40, XL
90, XC
```

# 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 – Randfälle prüfen II

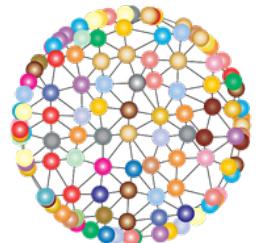


```
class DateRange
{
    // one of both can be null
    LocalDate from;
    LocalDate to;

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

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

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



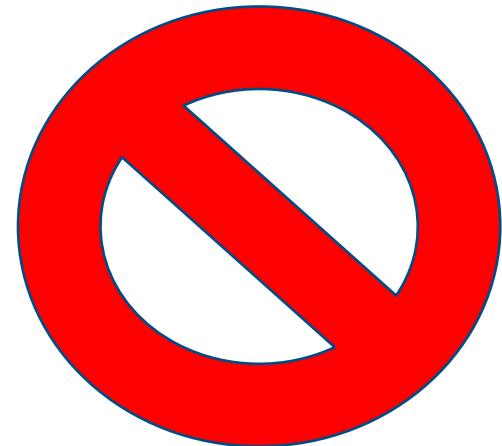
**Wie lässt sich das mit  
einem Parameterized Test  
machen?**

## Parameterized Test – Randfälle prüfen, @MethodSource



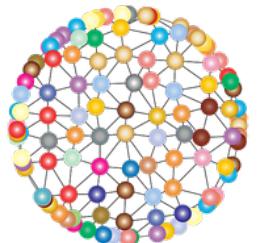
```
@ParameterizedTest(name = "validateDateRange({0} => {1})")
@MethodSource("allCombinations")
void validateDateRangeConstruction(LocalDate from, LocalDate to,
                                    Class<? extends Throwable> expectedException)
{
    // Mehr Infos später ...
    if (expectedException !=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

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





Ist das wirklich eine  
gute Idee?



## Parameterized Test – Randfälle prüfen, `@MethodSource`

---



- **NEIN**, zu übereifrig, weil durch Fallunterscheidungen unübersichtlich:
  - da zwei Fälle eine erlaubte Eingabe und
  - einer einen Fehlerfall darstellten.
- **Es ist natürlicher, diese Prüfung durch zwei Testmethoden ausführen zu lassen.**
- **Interessanterweise kann man dann auf den parametrisierten Test verzichten und das Ganze enorm vereinfachen -- auch beim Testen ist dies ein erstrebenswertes Ziel:**

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

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

---

## 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)));
}
```



# Fallstudie

## **@Test => @Parameterized Test**

## @Test => @Parameterized Test – z. B. für Listen als Expected



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

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

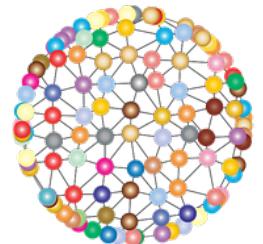
    final List<LocalDate> expectedSundays =
        Arrays.asList(LocalDate.parse("2020-01-05"),
                     LocalDate.parse("2020-01-12"), LocalDate.parse("2020-01-19"),
                     LocalDate.parse("2020-01-26"), LocalDate.parse("2020-02-02"),
                     LocalDate.parse("2020-02-09"),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

    assertEquals(expectedSundays, sundays);
}
```



---

Wie lösen wir die  
Schwierigkeiten mit  
Listen, wenn “Expected”  
aufwendiger zu  
konstruieren ist?



## @Test => @Parameterized Test



// Schritt 1: Logik vereinfachen, um weitere Extraktion vorzubereiten

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

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

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

    assertEquals(expectedSundays, sundays);
}
```

## @Test => @Parameterized Test



// Schritt 2: Konvertierung von String in LocalDate in Methode auslagern

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

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

    assertEquals(expectedSundays, sundays);
}

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

## @Test => @Parameterized Test



//Schritt 3: Bereitstellung Expected in Methode auslagern

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

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

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

    assertEquals(expectedSundays, sundays);
}

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

## @Test => @Parameterized Test



// Schritt 4: Datumsbereich als Parameter aufnehmen

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

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

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

    assertEquals(expected, result);
}

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

## @Test => @Parameterized Test



// Schritt 5: Umwandlung in Parameterized Test

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

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

    assertEquals(expected, result);
}

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

## @Test => @Parameterized Test



// Schritt 6: Weitere Test-Parametrierungen ergänzen

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

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

    assertEquals(expected, result);
}

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



**Geht es noch eleganter?  
Ja, ArgumentConverter!**

## @Parameterized Test – Trick: Argument Converter für «Expected»

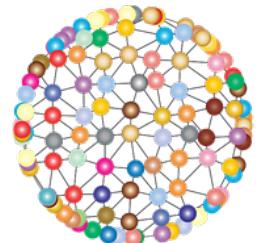


```
@ParameterizedTest(name = "sundays between {0} and {1} => {2}")
@MethodSource("startAndEndDateAndResults")
void sundaysBetween(LocalDate start, LocalDate end,
    @ConvertWith(ToLocalDateListConverter.class) List<LocalDate> expected)
{
    final List<LocalDate> result = SundayCalculator.allSundaysBetween(start,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

    assertEquals(expected, result);
}

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

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



**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



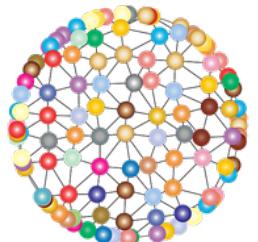
```
static class JsonToPerson extends SimpleArgumentConverter
{
    private static final Gson gson =
        new GsonBuilder()/* TODO IN EXERCISES */.create();
```

```
@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'
```



**Wie kann man  
Dynamic Tests erzeugen?**

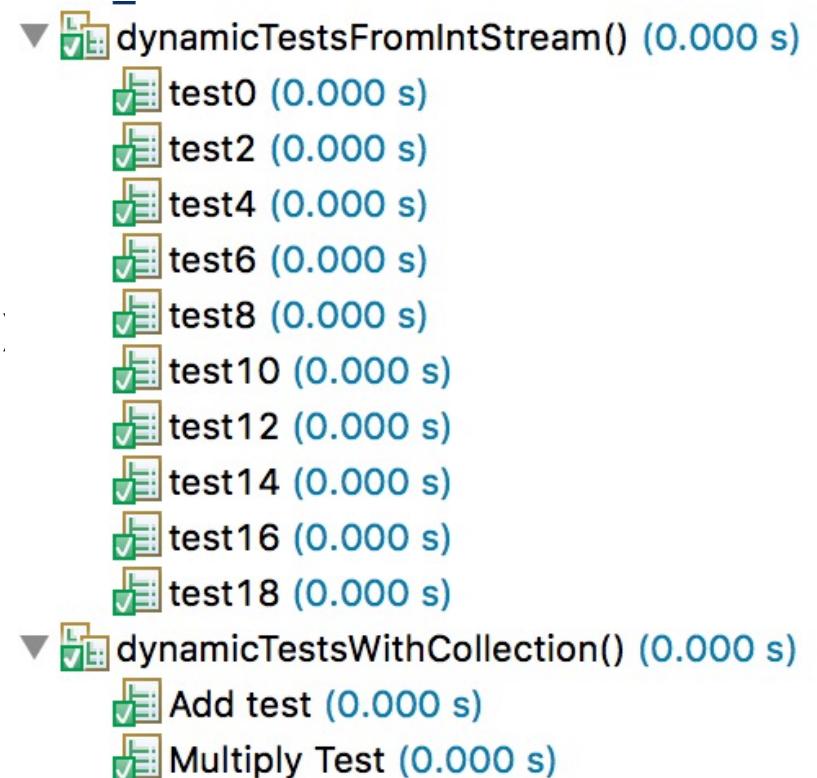


# Dynamic Tests



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

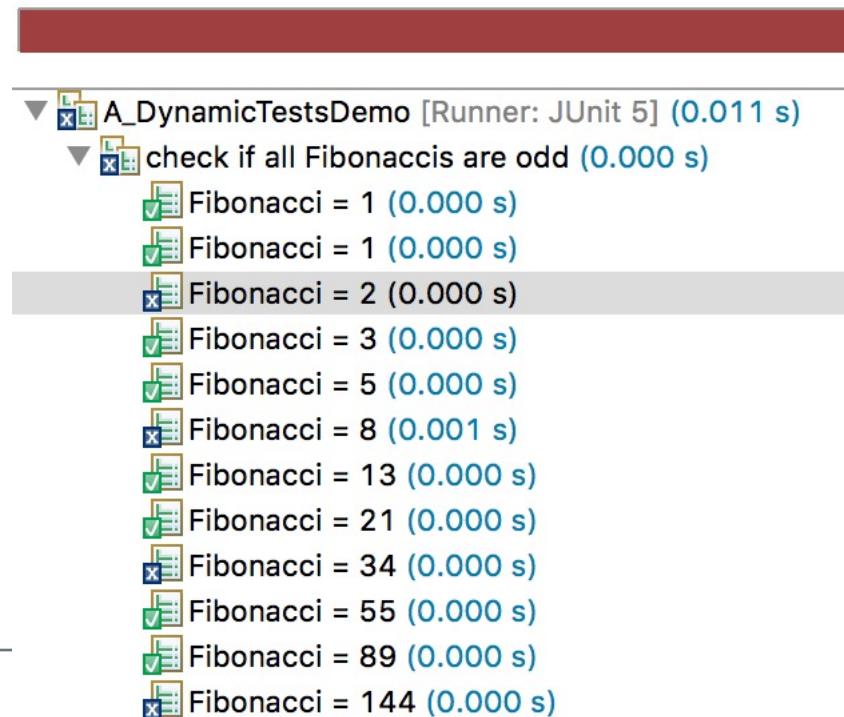
```
@TestFactory
Collection<DynamicTest> dynamicTestsWithCollection()
{
    return Arrays.asList(dynamicTest("Add test",
        () -> assertEquals(2, Math.addExact(1,
    dynamicTest("Multiply Test",
        () -> assertEquals(4, Math.multiplyExact(2, 2))));
```

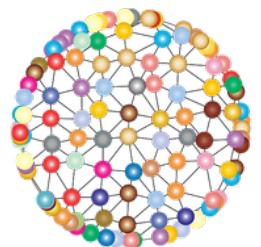




## Dynamic Tests

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





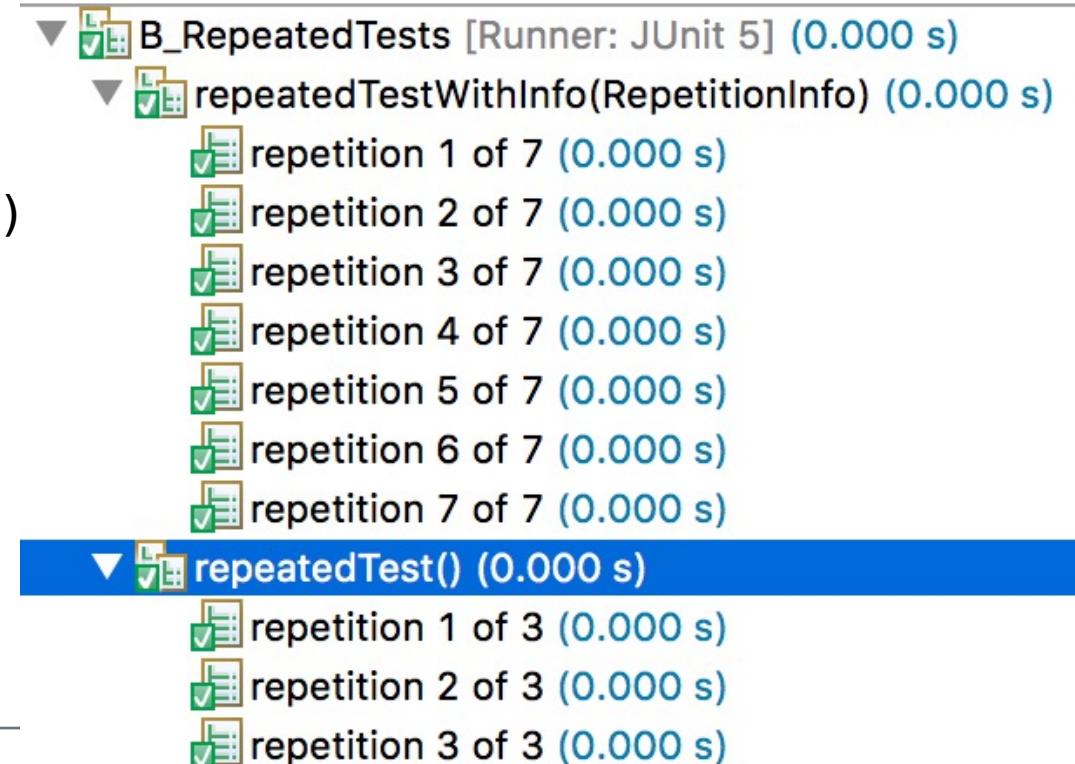
**Wie kann man Tests  
mehrmals ausführen und  
warum sollte man das  
wollen?**

# Repeated Tests – Grundlagen



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

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

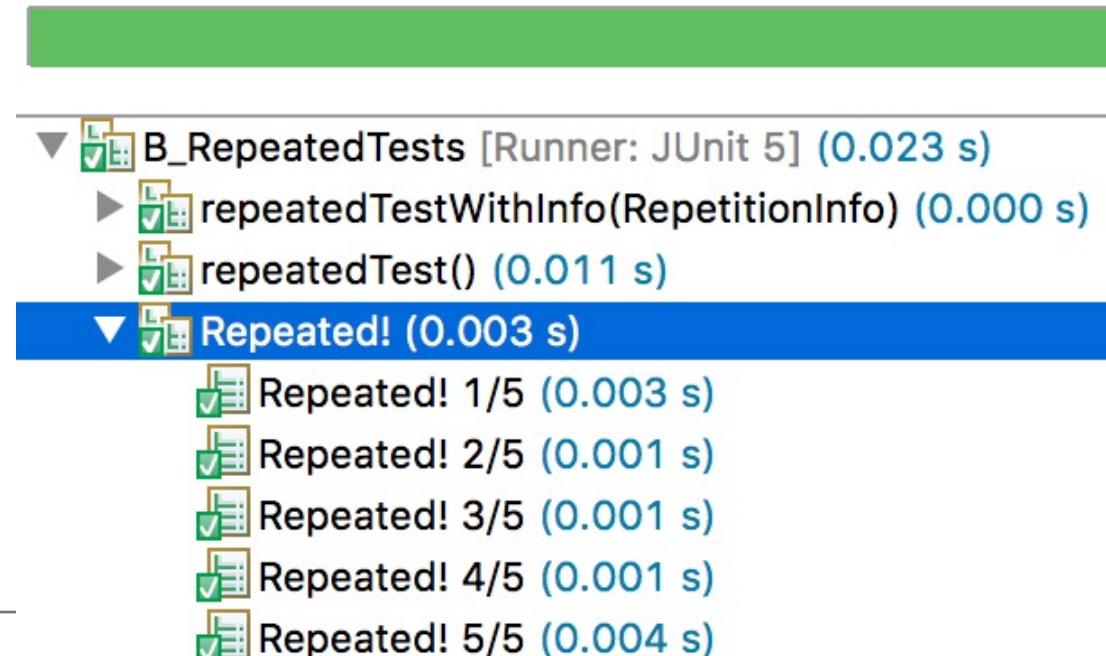


## Repeated Tests – Namensgebung und Infos



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

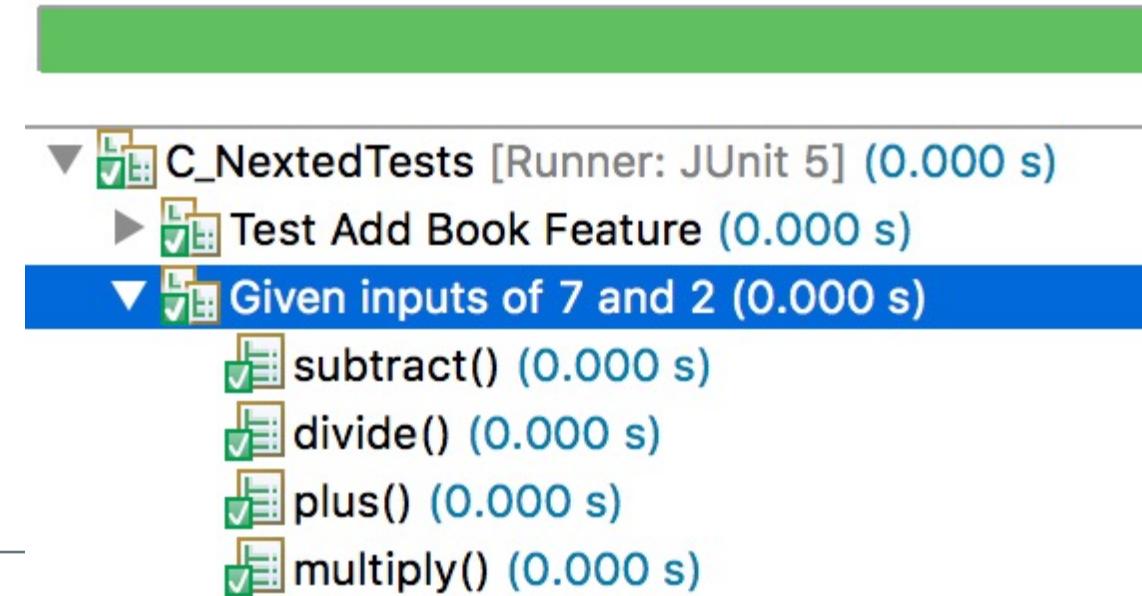
    assertEquals(testInfo.getDisplayName(),
                 "Repeated! " + current +
                 "/" + total);
}
```





## Nested Tests

```
@Nested  
@DisplayName("Given inputs of 7 and 2")  
class PredefinedValues  
{  
    final int input1 = 7;  
    final int input2 = 2;  
  
    @Test  
    void plus()  
    {  
        assertEquals(9, input1 + input2);  
    }  
  
    @Test  
    void subtract()  
    {  
        assertEquals(5, input1 - input2);  
    }  
    // ...  
}
```

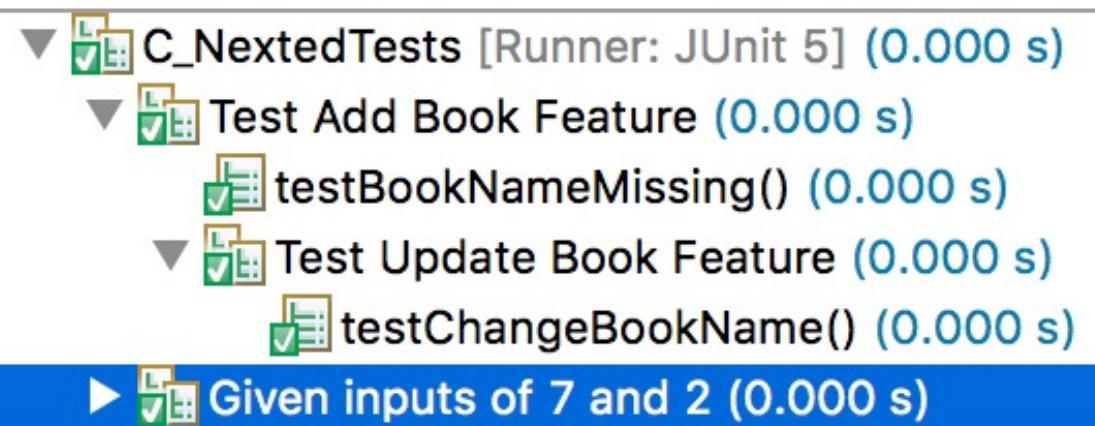




## Nested Tests

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

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





- ▼ **OptionalTest [Runner: JUnit 5] (0.021 s)**
  - ▼ **Testfälle für ein leeres Optional (0.011 s)**
    - Optional.isPresent() liefert false (0.005 s)**
    - Optional.get() liefert Exception (0.006 s)**
  - ▼ **Testfälle für ein vorhandenes Optional (0.010 s)**
    - Optional.isPresent() liefert true (0.002 s)**
    - Optional.get() liefert den Wert (0.008 s)**



# Simple Extensions





## Simple Extensions: Benchmarking

```
public class BenchmarkExtension implements BeforeTestExecutionCallback,  
                                         AfterTestExecutionCallback  
{  
    private long start;  
  
    @Override  
    public void beforeTestExecution(ExtensionContext ctx) throws Exception  
    {  
        start = System.currentTimeMillis();  
    }  
  
    @Override  
    public void afterTestExecution(ExtensionContext ctx) throws Exception  
    {  
        System.err.println("Test " + ctx.getDisplayName() + " took " +  
                           (System.currentTimeMillis() - start) + " ms");  
    }  
}
```



## Simple Extensions: Benchmarking

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

        assertEquals(2971215073L, value);
    }

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

Test testFibRecWithBigNumber() took 8675 ms  
Test testFibRecWithBigNumber\_Timeout() took 2012 ms



---

# PART 4: 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	@BeforeEach	Aktion <b>einmalig</b> vor allen Testmethoden
@Before	@BeforeEach	Aktion <b>jeweils</b> vor allen Testmethoden
@After	@AfterEach	Aktion <b>jeweils</b> nach allen Testmethoden
@AfterClass	@AfterAll	Aktion <b>einmalig</b> 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/JUnit5 Workshop](https://github.com/Michaeli71/JUnit5_Workshop)





---

# PART 5: 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

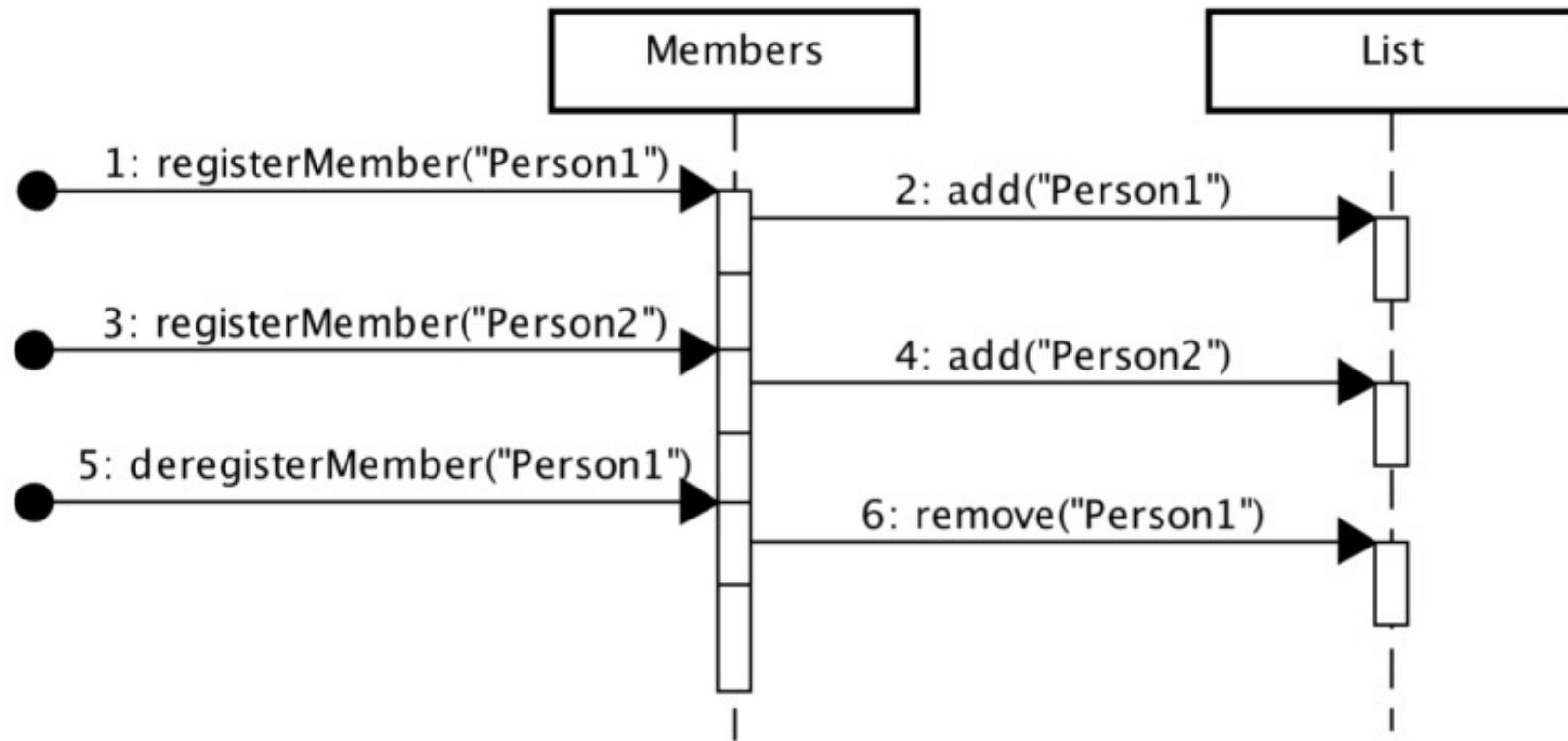
---



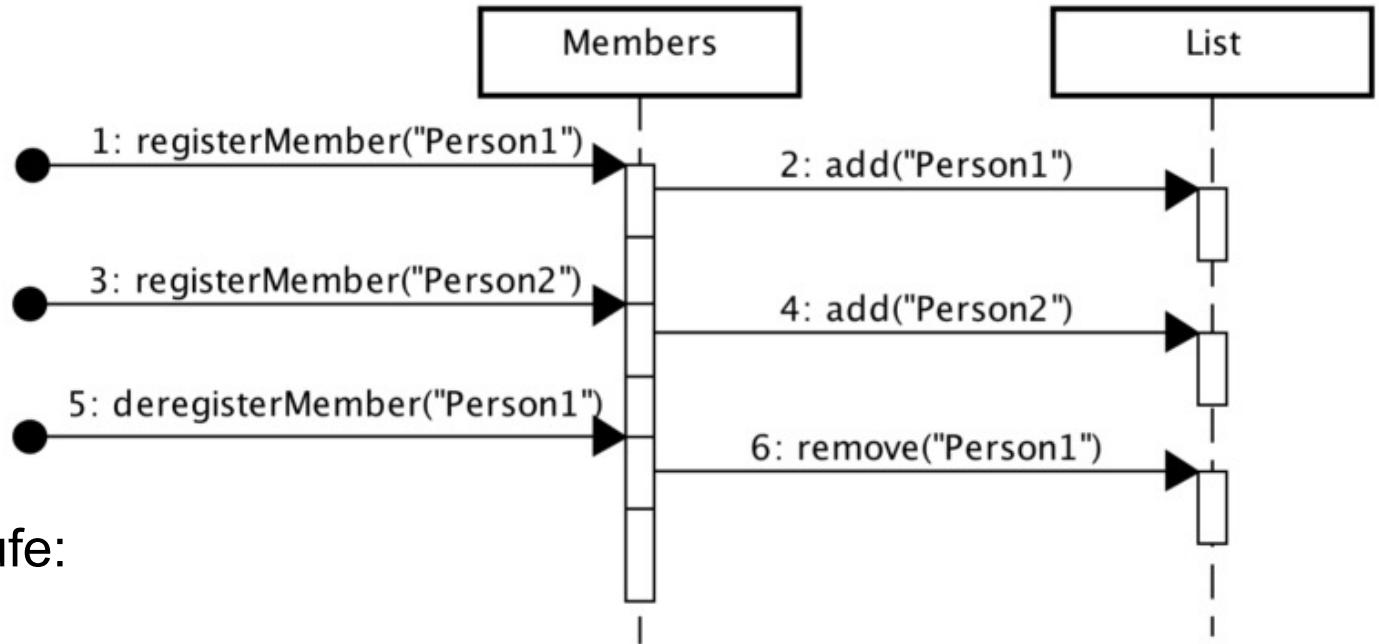
- Beim verhaltensbasierten Testen betrachtet man die **Interaktionen**, also die **aufgerufenen Methoden**, statt Veränderungen im Objektzustand.
- Demnach ist man beispielsweise nicht daran interessiert, ob sich nach einem Aufruf der Methode `registerMember (String)` die Anzahl der gespeicherten Elemente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

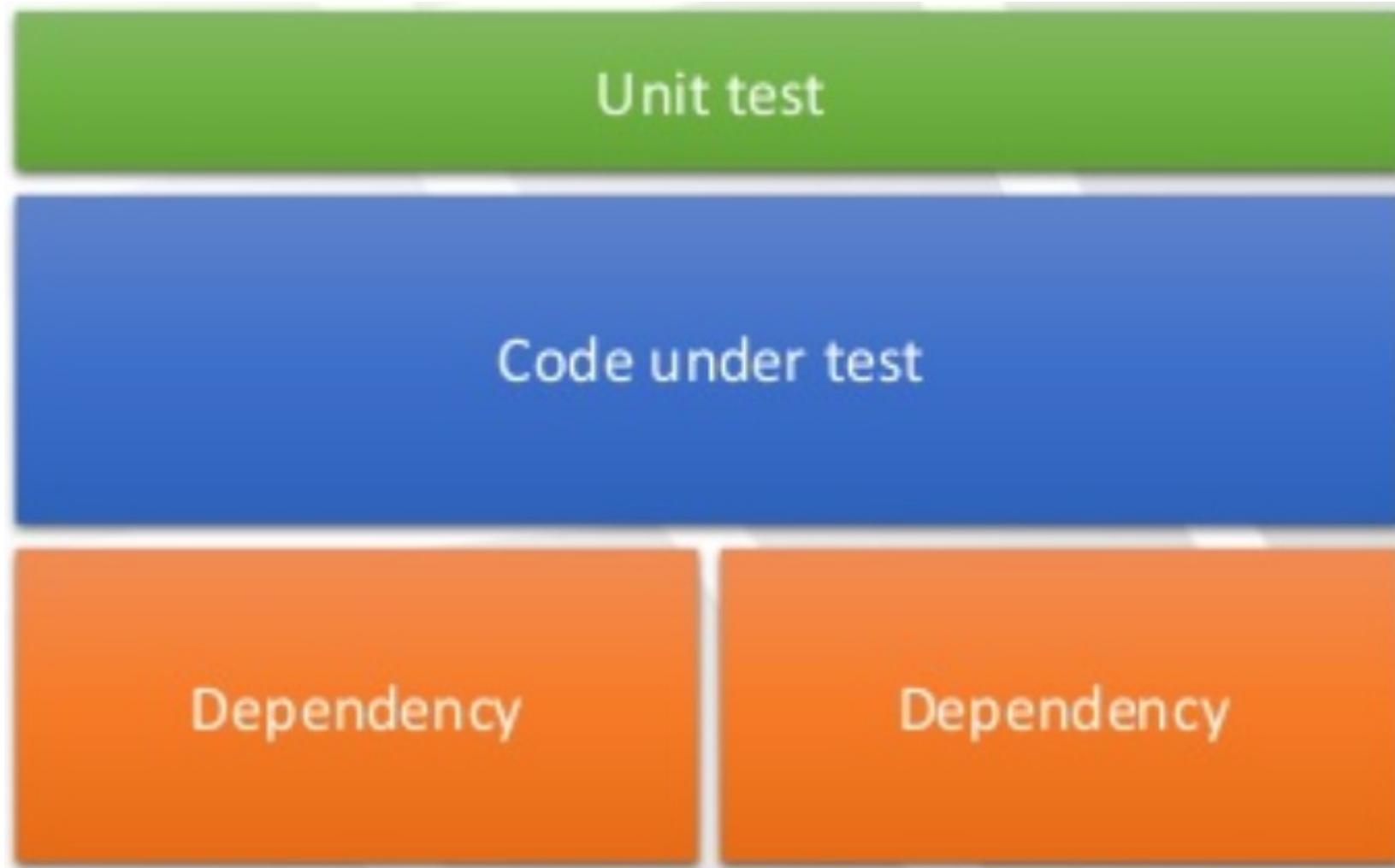


---

# Stellvertreter



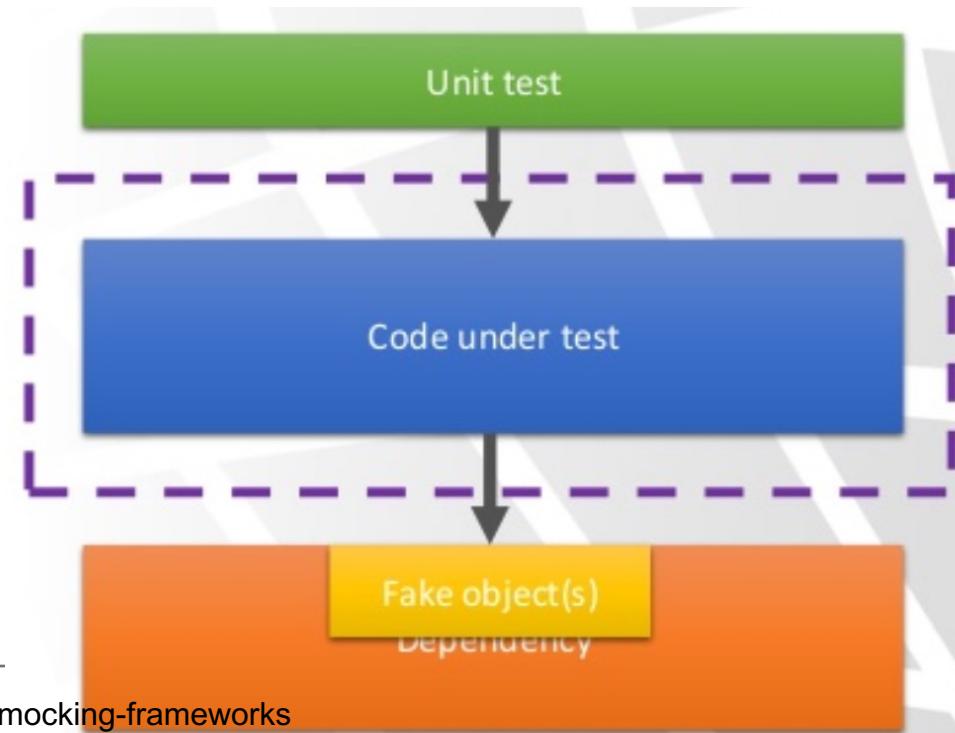
# Stellvertreterobjekte – Warum sind sie nötig?





## 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;
    }

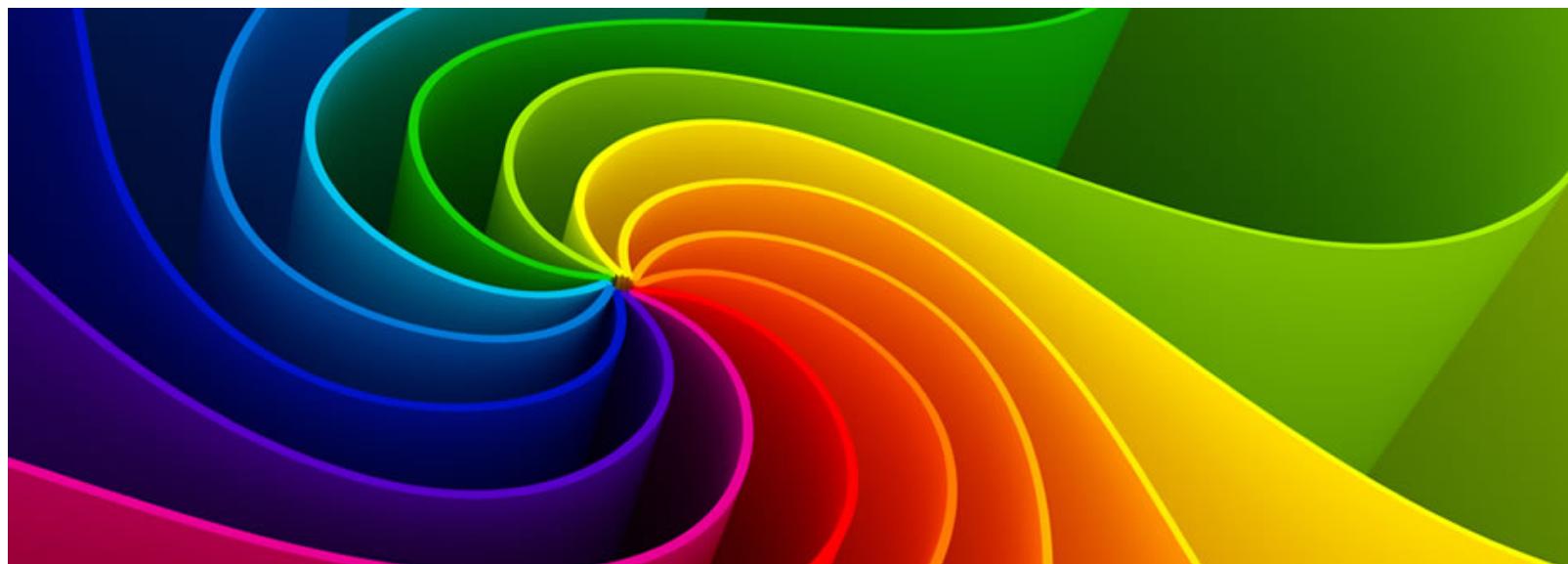
    // MOCK
    public void verifyDisplayMsgWasCalled() throws AssertionError
    {
        PreConditions.checkState(displayMsgCalled,
            "method 'displayMsg' has not been called");
    }
}
```



---

# PART 6:

# Design For Testability





---

# Sollbruchstellen





## Sollbruchstellen – Injection Points

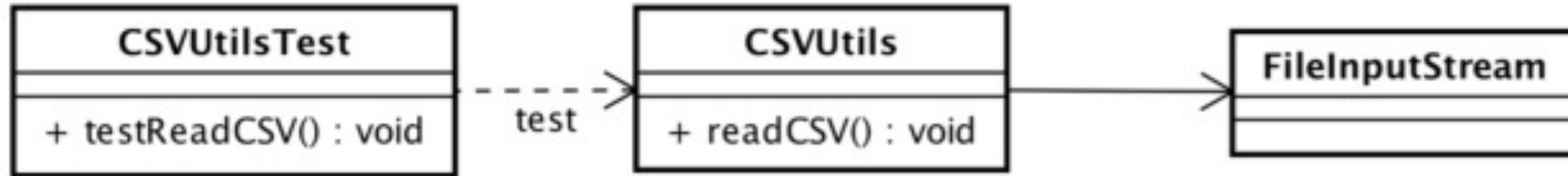
---

- 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)**

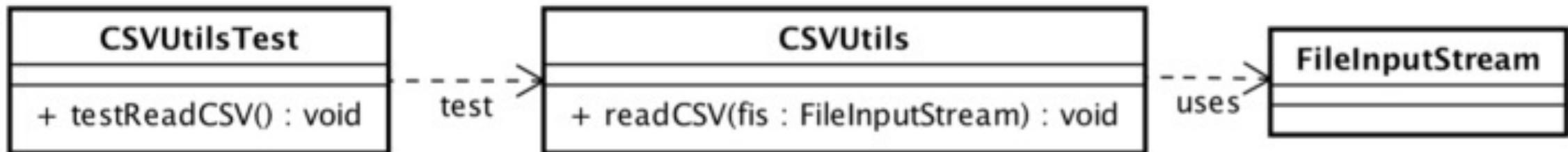


## Sollbruchstellen – Injection Points

- Oftmals erschweren direkte Abhangigkeiten das Testen



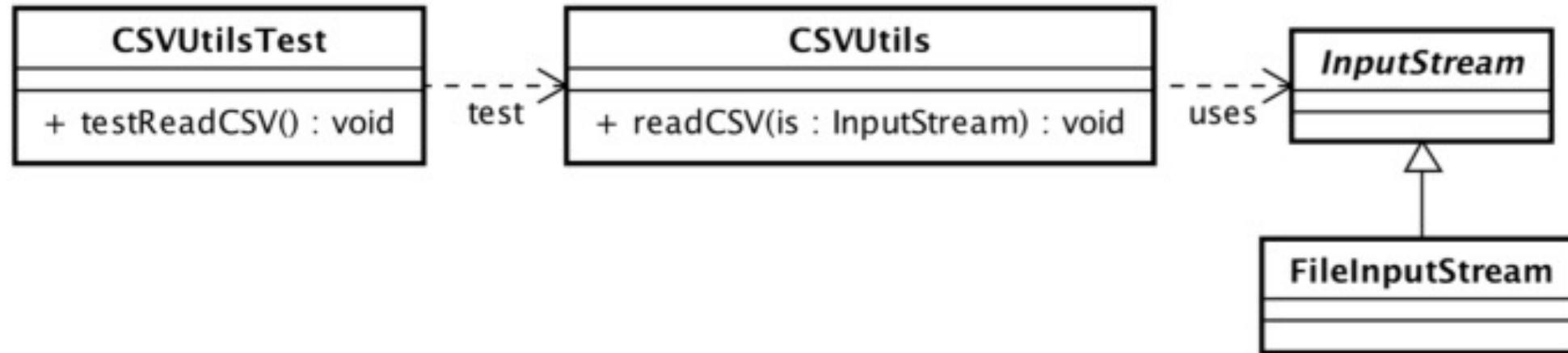
- Indirektion nutzen: Method Injection



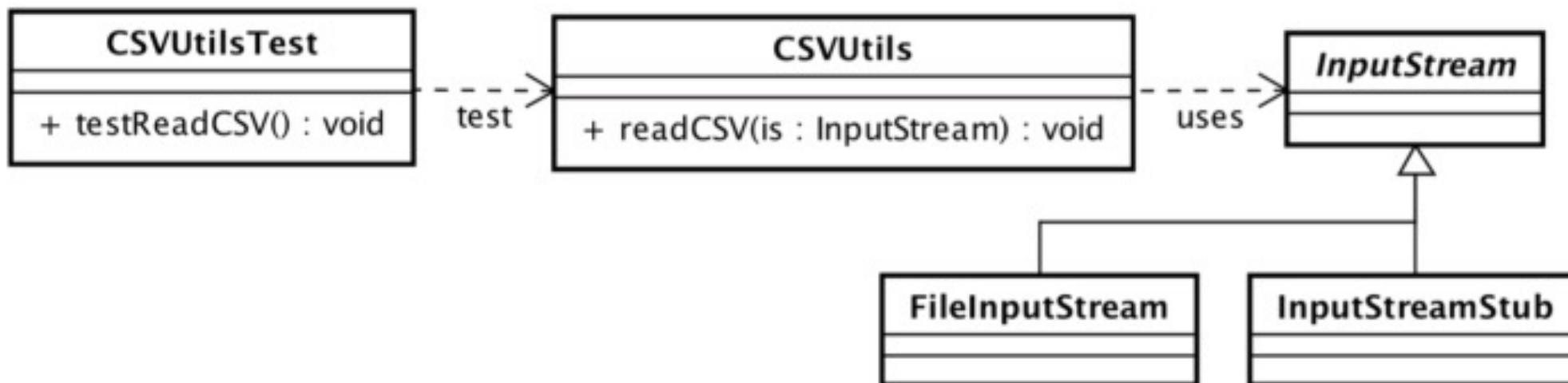


## Sollbruchstellen – Injection Points

- Abstraktion nutzen



- Stub zur Testbarkeit nutzen







## 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!



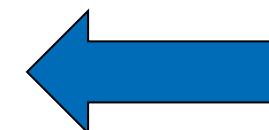
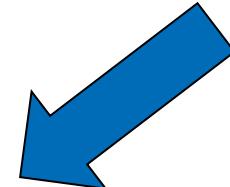
- **Benutzerinteraktion** ist beim manuellen Ausführen eines Unit Tests (z. B. in der IDE) schon **störend**
- **Showstopper** in einem automatischen Build auf Continuous-Integration-Server.
- **Also was nun? Überlegen wir kurz, was wir machen können.**



## 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");
        }
    }

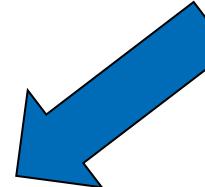
    protected void showWarning(final String message)
    {
        JOptionPane.showConfirmDialog(null, message);
    }
}
```





## 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?**





## 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>3.12.4</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>3.12.4</version>
    <scope>test</scope>
</dependency>
```

```
dependencies {
    testCompile 'junit:junit:4.13'
    // Mockito
    testCompile "org.mockito:mockito-core:3.12.4"
    testCompile "org.mockito:mockito-junit-jupiter:3.12.4"
}
```

---

# 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

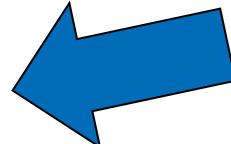


```
@Test
public void testGreetingReturnValue()
{
    // Arrange
    final Greeting greeting = Mockito.mock(Greeting.class);

    // 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();

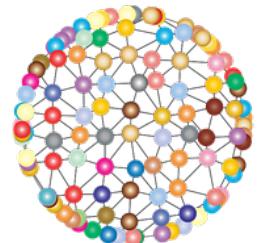
    // Prüfe, dass die Aufrufe in der richtigen Reihenfolge erfolgen
    inOrder.verify(firstMock).methodOne();
    inOrder.verify(secondMock).methodTwo();
    inOrder.verify(firstMock).methodThree();
}
```

# 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**



---

# Exercises Part 5 + 6

[https://github.com/Michaeli71/JUnit5 Workshop](https://github.com/Michaeli71/JUnit5_Workshop)





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

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



```
@Test
public void testFlightMileage_asKm2() throws Exception
{
    // setup fixture
    String validFlightNumber = "LX 857";
    // exercise constructor
    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());
}
```

## Test Smell: Falsche Nutzung von assertTrue()/assertFalse()



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

## Test Smell: Falsche Nutzung von assertTrue()/assertFalse()



```
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? 😞



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

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

@Test
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");
}
```

A large green arrow pointing downwards from the first code block to the second code block.

## 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



```
@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()); ! org.opentest4j.AssertionFailedError: expected: <1> but was: <2>  
    assertEquals("Tom", result.get(0)); // Wird nicht ausgeführt  
}
```



```
@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**

**Eclipse Marketplace**

Select solutions to install. Press Install Now to proceed with installation.  
Press the "more info" link to learn more about a solution.



Search Recent Popular Favorites Installed  Giving IoT an Edge

Find:  X All Markets ▼ All Categories ▼ Go

**EclEmma Java Code Coverage 3.1.3**

 EclEmma is a free Java code coverage tool for Eclipse, available under the Eclipse Public License. It brings code coverage analysis directly into the Eclipse... [more info](#)

by [Mountainminds GmbH & Co. KG](#), EPL 2.0  
[quality metrics](#) [code coverage](#) [fileExtension\\_exec](#)

★ 1112  Installs: **767K** (3'468 last month) Installed

# Test Coverage



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

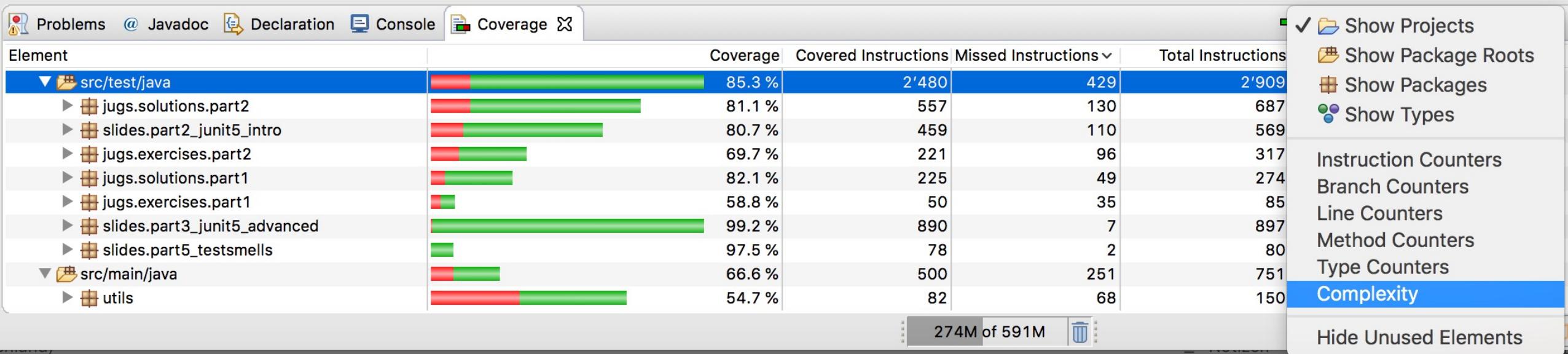


- Dadurch werden während der Programm- bzw. Testausführung entsprechende Informationen über ausgeführte Sourcecode-Teile gesammelt
- Das Ergebnis wird nach Ausführungsende automatisch in einer speziellen Coverage-View angezeigt
- Praktischerweise von Projektebene bis hinunter zu einzelnen Java-Methoden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

# EclEmma – Eclipse Plugin



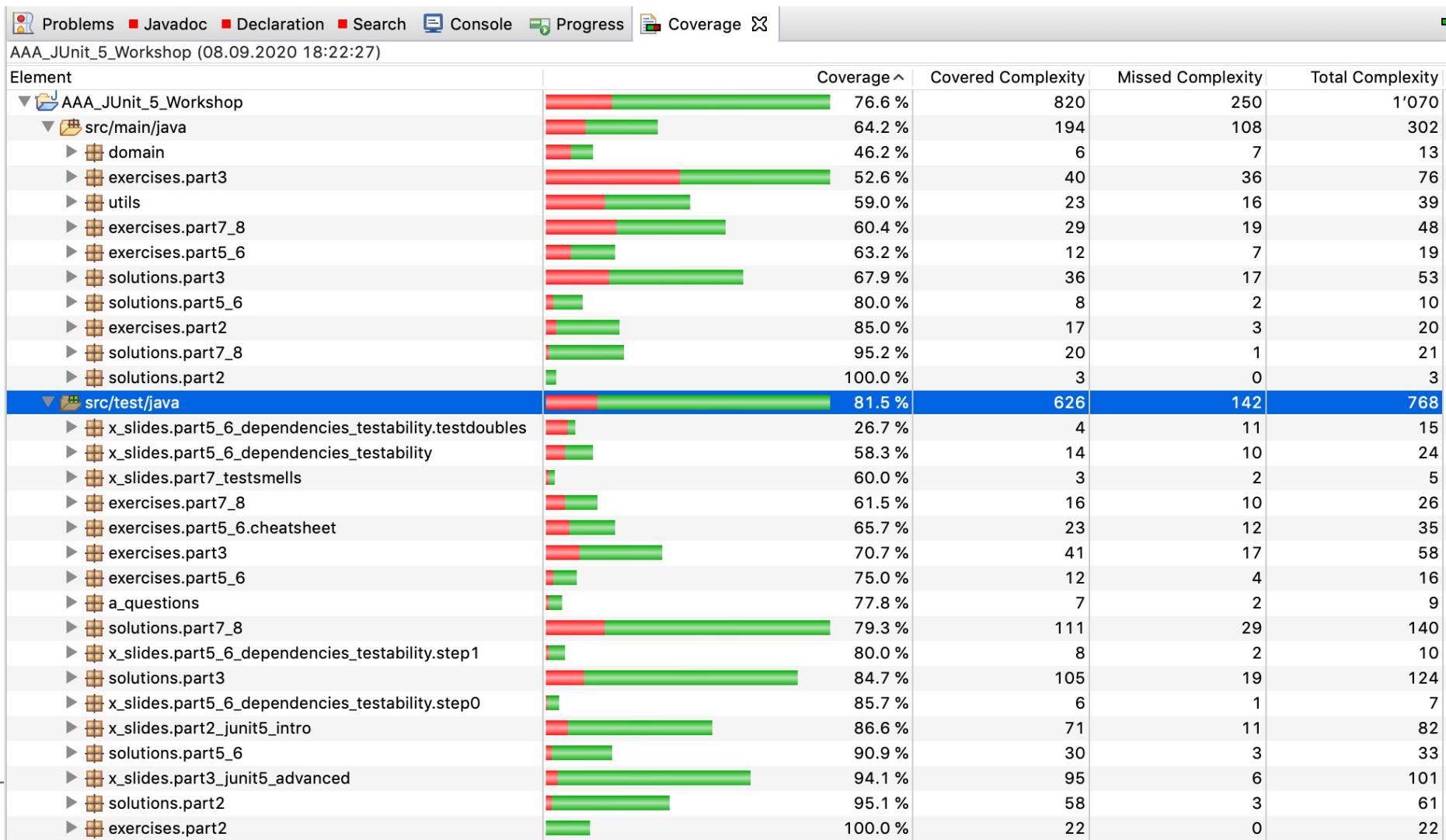
- Konfigurierbar



# EclEmma – Eclipse Plugin



- Einfach, schnell, grafisch, übersichtlich



# 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

The screenshot shows an IDE interface with several tabs at the top: DiscountCalcula, Ex01\_Calculator, Ex01\_Calculator, DynamicTests.ja, Ex02\_MatchingBr, Ex02\_MatchingBr, and Ex02\_Matchin. Below the tabs is a code editor window displaying Java code for a DiscountCalculator. The code uses if statements to calculate discounts based on the count of items. The code editor highlights specific lines with colors: green for lines 7, 8, 10, 11, 12, 13, 14, and 17; yellow for lines 12 and 14; and red for lines 17 and 20. A red arrow points from the status bar at the bottom to the 'Coverage' tab in the bottom navigation bar.

```
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

---

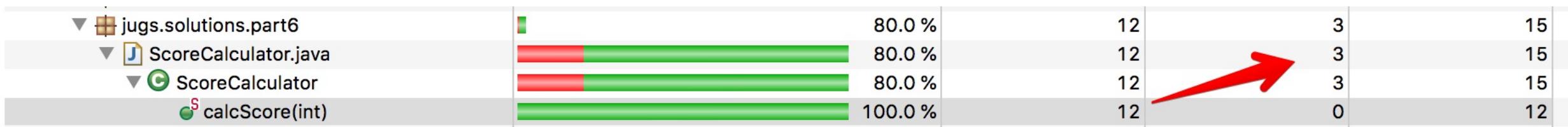


- Generell: Je kürzer und weniger Parameter und Verzweigungen eine Methode hat, desto besser lässt sich für diese eine hohe Testabdeckung erzielen
- Das erreicht man fast automatisch, wenn man Grundregeln guten OO-Designs befolgt:
  - Cyclomatic Complexity gering halten
  - SRP einhalten
- Wenn man die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

# Besonderheit I: Wieso 80% und nicht 100%?



```
J 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
```

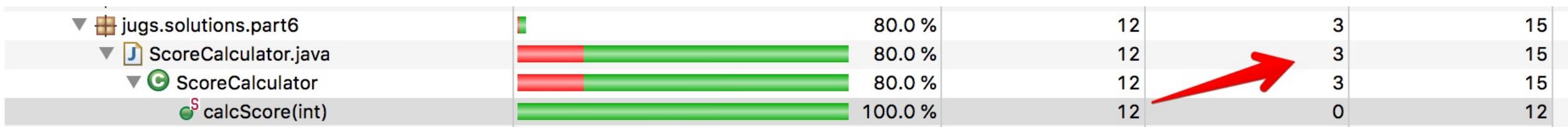


# Besonderheit I: Wieso nicht 100%?



```
J 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();
}
```



# Besonderheit I: Private Konstruktor => 100%



```
J 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	C	100.0 %	12	0	12
calcScore(int)	S	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;
    }
}
```

## 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



Safari Mail - Michael Ir 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\_OI ... ⌂ ⌂ ⌂

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	Cxt	Missed	Lines	Missed	Methods	Missed	Classes
<a href="#">exercises.part7_8</a>	<div style="width: 42%; background-color: red; height: 10px;"></div> <div style="width: 58%; background-color: green; height: 10px;"></div>	58%	<div style="width: 37%; background-color: red; height: 10px;"></div> <div style="width: 63%; background-color: green; height: 10px;"></div>	63%	19	48	31	96	9	26	1	7
<a href="#">exercises.part3</a>	<div style="width: 22%; background-color: red; height: 10px;"></div> <div style="width: 78%; background-color: green; height: 10px;"></div>	78%	<div style="width: 18%; background-color: red; height: 10px;"></div> <div style="width: 82%; background-color: green; height: 10px;"></div>	58%	36	76	36	154	8	26	1	9
<a href="#">utils</a>	<div style="width: 14%; background-color: red; height: 10px;"></div> <div style="width: 86%; background-color: green; height: 10px;"></div>	64%	<div style="width: 14%; background-color: red; height: 10px;"></div> <div style="width: 86%; background-color: green; height: 10px;"></div>	69%	16	39	25	56	10	21	0	7
<a href="#">solutions.part3</a>	<div style="width: 10%; background-color: red; height: 10px;"></div> <div style="width: 90%; background-color: green; height: 10px;"></div>	76%	<div style="width: 10%; background-color: red; height: 10px;"></div> <div style="width: 90%; background-color: green; height: 10px;"></div>	76%	17	53	17	66	4	14	1	5
<a href="#">domain</a>	<div style="width: 5%; background-color: red; height: 10px;"></div> <div style="width: 95%; background-color: green; height: 10px;"></div>	70%	<div style="width: 5%; background-color: red; height: 10px;"></div> <div style="width: 95%; background-color: green; height: 10px;"></div>	50%	7	13	4	19	1	7	0	1
<a href="#">exercises.part5_6</a>	<div style="width: 5%; background-color: red; height: 10px;"></div> <div style="width: 95%; background-color: green; height: 10px;"></div>	84%		n/a	7	19	7	35	7	19	3	9
<a href="#">exercises.part2</a>	<div style="width: 5%; background-color: red; height: 10px;"></div> <div style="width: 95%; background-color: green; height: 10px;"></div>	90%	<div style="width: 5%; background-color: red; height: 10px;"></div> <div style="width: 95%; background-color: green; height: 10px;"></div>	100%	3	20	4	34	3	18	0	5
<a href="#">solutions.part5_6</a>	<div style="width: 5%; background-color: red; height: 10px;"></div> <div style="width: 95%; background-color: green; height: 10px;"></div>	85%		n/a	2	10	2	18	2	10	0	3
<a href="#">solutions.part7_8</a>	<div style="width: 5%; background-color: red; height: 10px;"></div> <div style="width: 95%; background-color: green; height: 10px;"></div>	100%	<div style="width: 5%; background-color: red; height: 10px;"></div> <div style="width: 95%; background-color: green; height: 10px;"></div>	96%	1	21	0	36	0	6	0	4
<a href="#">solutions.part2</a>	<div style="width: 5%; background-color: red; height: 10px;"></div> <div style="width: 95%; background-color: green; height: 10px;"></div>	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/JUnit5 Workshop](https://github.com/Michaeli71/JUnit5_Workshop)





# 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://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](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