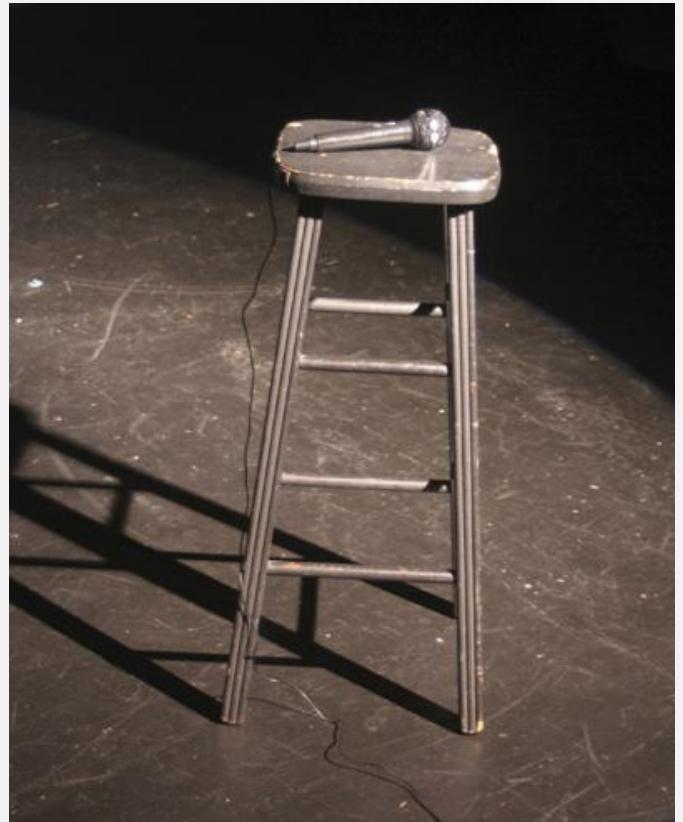
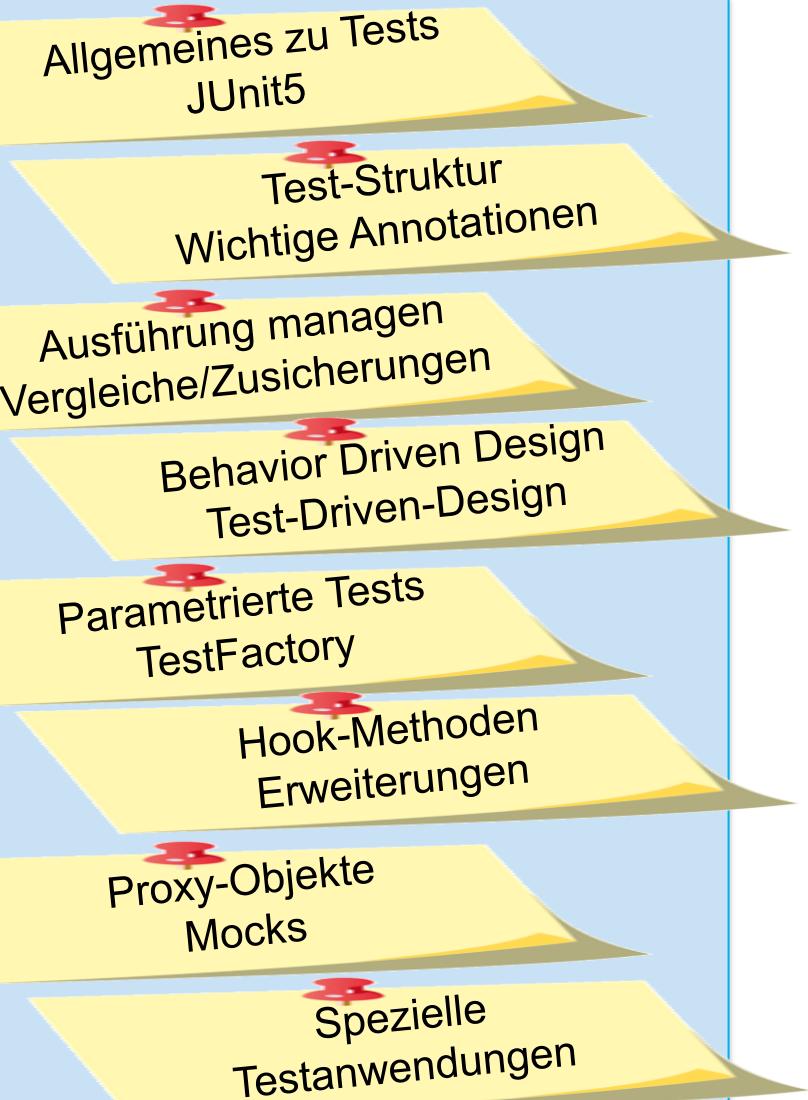




JUNIT

... testen Sie noch?
... oder entwickeln Sie schon?



Vorstellung

Name
Warum hier?

Erfahrung?
Frust? Freude?
Wünsche?



A photograph of two brown bears swimming in a river. One bear is facing right, and the other is facing left, seemingly interacting or competing for something. The water is brownish-green, and the background shows a grassy bank.

Gibt's noch was?

Unterrichtszeiten? (heute, morgen, überhaupt)

Pausenzeiten? (Raucher unter Ihnen?)

Toilette ... Snacks ...

...

Allgemeines zu Tests JUnit5

Test-Struktur
Wichtige Annotationen

Ausführung managen
Vergleiche/Zusicherungen

Behavior Driven Design
Test-Driven-Design

Parametrierte Tests
TestFactory

Hook-Methoden
Erweiterungen

Proxy-Objekte
Mocks

Spezielle
Testanwendungen



Testen ist Wichtig ...

... ein Entwickler mit Compilergen nach 30min Codereview einer kleinen Klasse ...

Ich habe mir die Parameter angesehen. In Zeile 5
wird es einen Compilerfehler geben.
Der Parameterwert stimmt so nicht.

... Antwort des Projektleiters

Schon gewusst?:
JAVA-Compiler sind kostenlos.

... kurz vor dem Livegang ...



Wir haben doch diese Regel, dass es keine roten
JUnit-Tests geben darf.
Schaust du dir mal die roten Tests an. Am besten,
du schaust auf das echte Ergebnis und trägst es
dann unter „erwartetes Ergebnis ein“

... Wochen später während einer problematischen Einführungsphase:

OH!
Das Programm läuft ja falsch. Ist das überhaupt
jemals richtig gelaufen?

... der selbe Entwickler nach einer Produktionsstörung
nach einem kleinen Refactoring ...

Wir haben uns das zu zweit angesehen und
waren der Meinung, dass das so richtig ist.
Aber mal so: Der Code vorher war aber auch total
kompliziert.

... Antwort des Projektleiters:

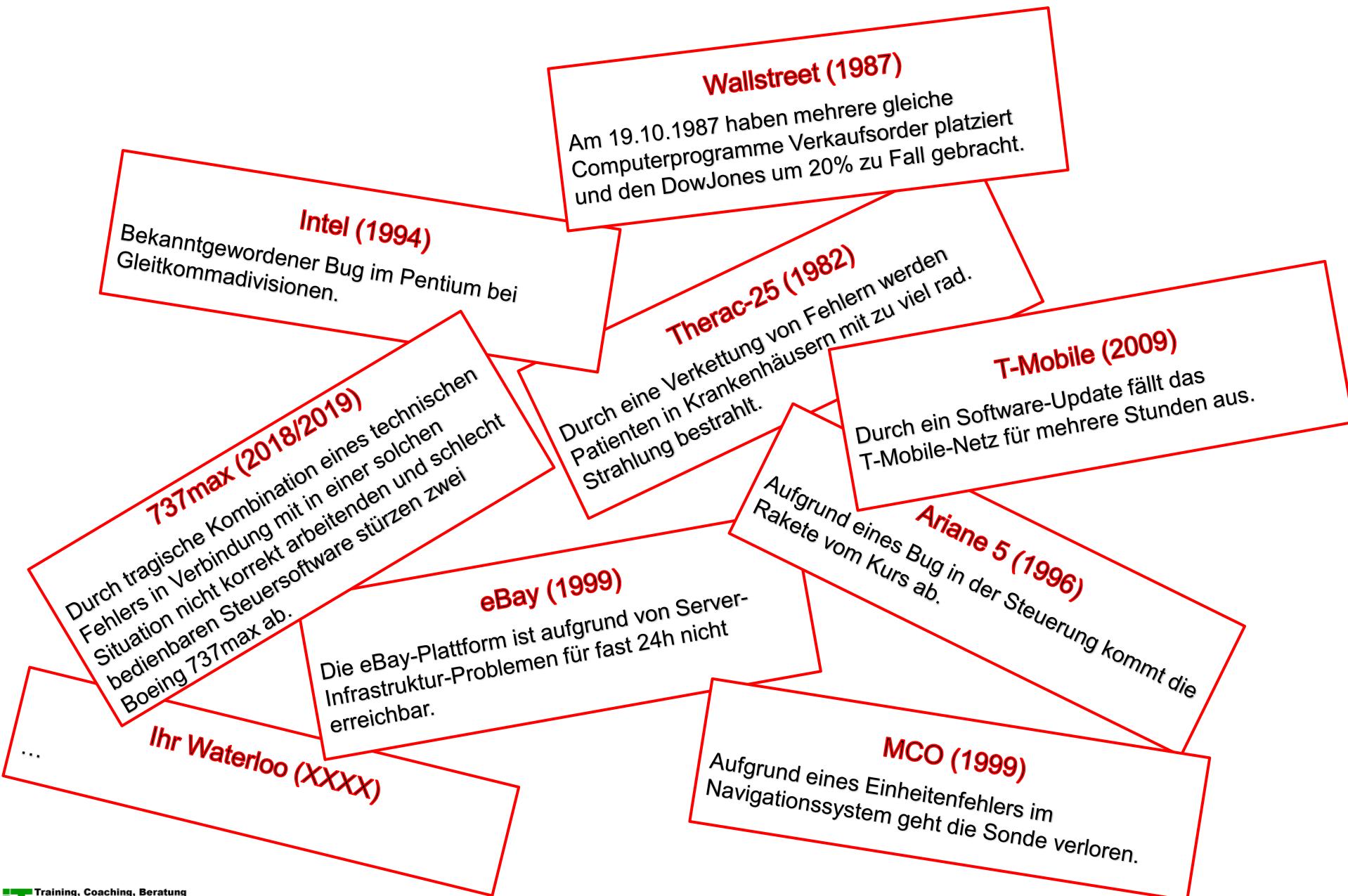
Kompliziert? Stimmt!
Aber dafür war der Code vorher noch
richtig!!!



Tests:

- Tests stellen die Qualität sicher
- Tests sind sinnvoll
- Tests müssen aber auch richtig/sinnvoll durchgeführt werden

Können Fehler passieren?



Qualität ?

Viele Menschen, viele Antworten ...

Erweiterbar

Korrekt

Robust

Leicht erlernbar

effizient

Leicht
benutzbar

Portierbar

Hohe
Sicherheit

Aktuell

effektiv

wiederverwend
bar

geprüft

Qualität

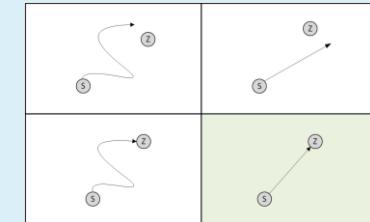
Wiederverwendbarkeit



Robustheit



Effektivität Effizienz



Qualität

Qualität ist die Abwesenheit von Fehlern

- › Defekt
- › Fehler
- › Bug

- › Tests sind erstellt zum **Auffinden von Fehlern**
- › Nur ein zunächst **fehlerhafter Test** ist ein guter Hinweis auf dessen **Verlässlichkeit**
- › Beim **Auffinden eines Fehlers** ist der Test **beendet**
 - › Dann beginnt das „Debugging“



Dijkstra (1930-2002):

- Tests can only show the presence of errors, never their absence

Testvarianten



Blackbox-Test

- › Verhaltensorientiert **ohne Kenntnis der Implementierung / des Aufbaus**
 - Kontrolle von Output bei bestimmtem Input
 - Basiert auf die Spezifikation / dem Pflichtenheft



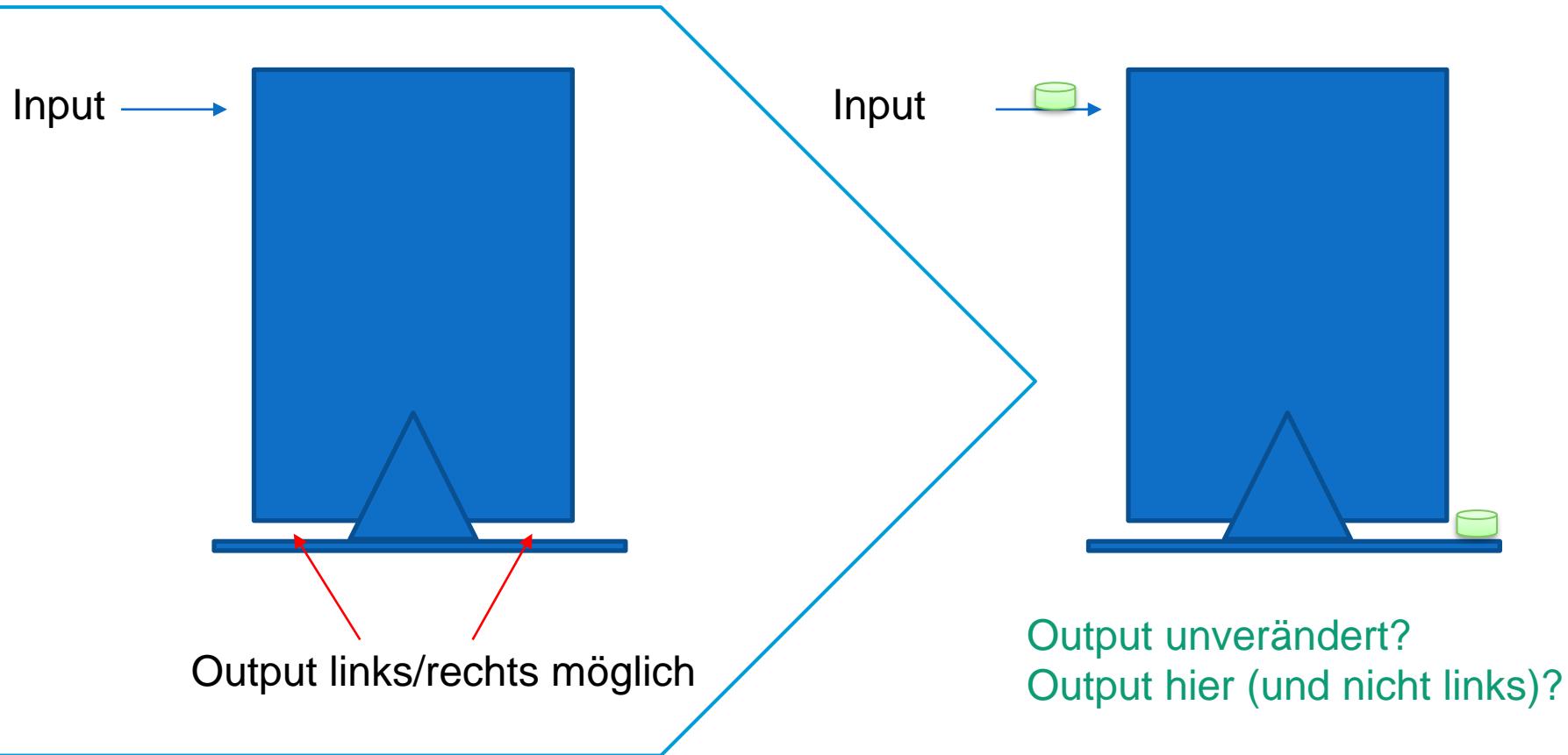
Whitebox-Test

- › Orientierung am Aufbau, der **Implementierung**
 - Setzen Kenntnis voraus, was ein Programm genau gemacht
 - Basiert auf Sourcecode

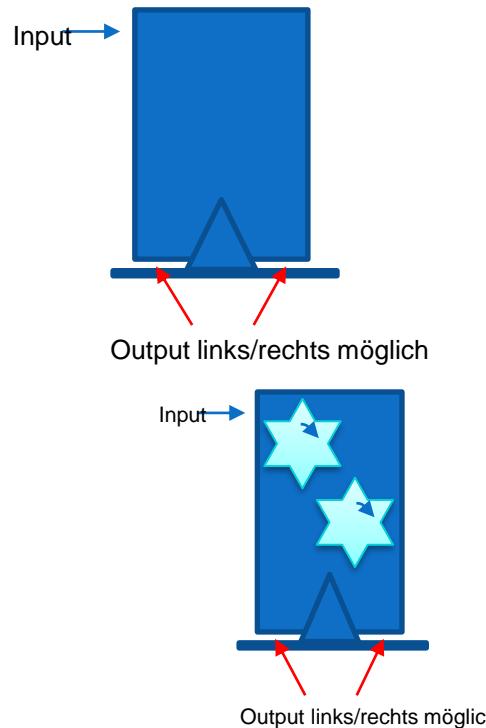


- Mit JUnit können beide Testvarianten durchgeführt werden

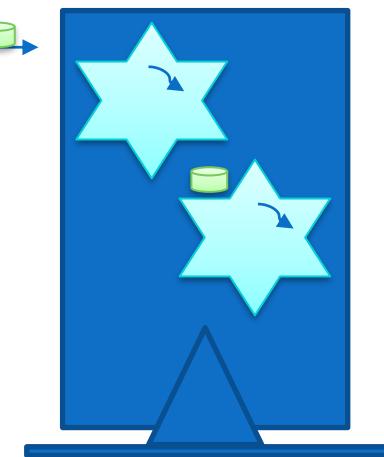
Black-box Test



White-box Test



Trifft Input zweites Zahnrad?



Oder:

- Trifft Input zwei Zahnräder, bevor zum Ausgang kommt?
- Dreht Input immer rechts?

Testvarianten



Whitebox-Test

- › Code-Coverage
- › Fault-Injection
- › Mutation-Tests



Blackbox-Test

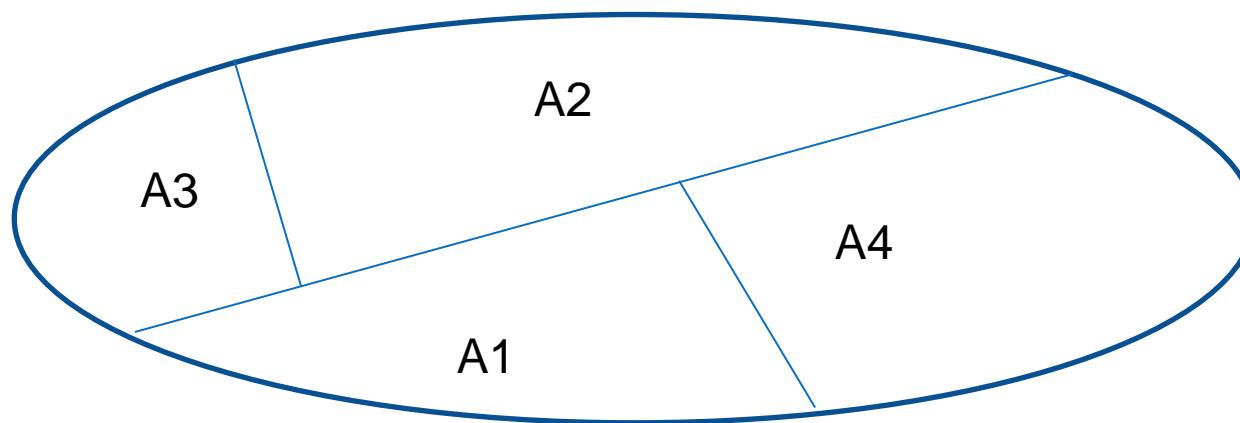
- › Systematische Tests
- › Zufalls-Tests
- › GUI-Tests
- › Modellbasierte Tests
- › Smoke-Tests
- › Sanity-Tests

Die richtigen Inputs wollen gefunden werden ...

```
public class FinancialAccount {  
  
    private Long amount;  
  
    public FinancialAccount(Long amount) {  
        this.amount = amount;  
    }  
  
    public Long getAmount() {  
        return amount;  
    }  
  
    public boolean withdraw(Long money) {  
        Long balance = amount - money;  
        if (balance > 0) {  
            amount = balance;  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

- › Man will ja nicht alle möglichen/denkbarer
Inputs testen ...
- › ABER:
 - › Wir benötigen realistische Inputs
 - › Wir benötigen die aussagefähigen Inputs

Input: Äquivalenzklassen



- › Der Test ist für ein Input einer **Äquivalenzklasse** (Partition) grün, dann ist er auch für alle anderen Werte dieser Partition grün.
- › Falls der Test für einen Wert der Partition rot ist, wird er auch für alle anderen Werte dieser Partition rot sein

Input: Partitionen

```
public class FinancialAccount {  
  
    private Long amount;  
  
    public FinancialAccount(Long amount) {  
        this.amount = amount;  
    }  
  
    public Long getAmount() {  
        return amount;  
    }  
  
    public boolean withdraw(Long money) {  
        Long balance = amount - money;  
        if (balance > 0) {  
            amount = balance;  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

- › Grenzen (MAX, MIN)
 - › Maximal fachlich sinnvoller Betrag
 - › Wert über/unter Grenze (unabhängig vom Sinn)
- › Werte in der „Mitte“
- › Typische relevante Werte
- › Auch sinnlose Werte
- › Werte, in denen Besonderes/Abweichendes erwartet wird
 - › „null“
 - › Leeres Array (Länge = 0)
 - › Nicht initialisierte Variablen

Beispiel: Finden von Inputs

```
public int calculateInterestDays(  
    Long startYear,  
    Long startMonth,  
    Long startDay,  
    int finalYear,  
    int finalMonth,  
    int finalDay) {  
  
    return 0; // that isn't correct ;)  
}
```

Beispiel: Finden von Inputs

```
public int calculateInterestDays(  
    Long startYear,  
    Long startMonth,  
    Long startDay,  
    int finalYear,  
    int finalMonth,  
    int finalDay) {  
  
    return 0; // that isn't correct ;)  
}
```

Monat:

1, 12, 6
-1, 13, MAX_LONG
null

Jahr:

0, 2000, 9999
2100, -9999
2020 (Schaltjahr)
2021 (kein Schaltjahr)
null

Tag:

1, 15, 31
-1, 0, 32, MAX_LONG
null

Kombinationen:

Monat 1, Tag 0, 15, 31, 32
Monat 2, Tag 0, 14, 28

Schaltjahr, Monat 2, Tag 28, 29

Kein Schaltjahr, Monat 2, Tag 28, 29

Gleicher Tag >>> Monat 1, Tag 1, Jahr 2100 für erstes und zweites Datum

Zweites Datum < Erstes Datum



Übung 01

› Ressourcen

de.hegmanns.training.junit5.practice.task01

Die Klasse „CalculatorForDevide“ bietet einen Service an, mit dem ermittelt werden kann, ob eine vorhandene Menge so in zwei Teile aufgeteilt werden kann, dass dabei jeweils eine gerade Zahl heraus kommt.

Beispiel:

Menge 8, könnte aufgeteilt werden in $4 + 4$ und damit ist eine Aufteilung in zahlenmäßig zwei gerade Mengen möglich.

Menge 5, könnte aufgeteilt werden in $2 + 3$, $1 + 4$ und damit ist eine Aufteilung in zahlenmäßig zwei gerade Mengen nicht möglich.

1. Überlegen Sie sich sinnvoll Test-Inputs, so dass mit großer Sicherheit über die Korrektheit der Implementierung entschieden werden kann.
In die Testklasse „EvenDividableTest“ ist schon ein zu testender Input eingetragen.
Ergänzen Sie die Liste der zu testenden Inputs im Java-DOC. (Schreiben Sie noch keine Tests!)

Allgemeine Empfehlungen für Tests

- › Einfache Tests
- › Nicht einfach sinnfreie Tests
- › Leicht lesbare Tests
- › Schnelle Tests
- › Nur eine Angelegenheit pro Test
- › Testfokus liegt auf Daten
- › Wiederholbare Tests
- › Gut- und Schlechtwetter – Tests
- › Nicht auf Testabdeckung gerichtet Tests erstellen

JUNI5? Warum jetzt ein komplett neues JUNIT?



JUNIT4 hat Schwächen



JUNIT4 kann schon viel

- › Monolithischer Aufbau
- › Keine Integration von JAVA8-Features
 - › Lambdas
 - › Streams
- › Erweiterungskonzepte haben Einschränkungen
 - › Sie wirken vorwiegend immer auf die gesamte Klasse
 - › Lassen sich nicht immer kombinieren (beispielsweise ist nur ein TestRunner möglich)

- › Definiert Annotationen
- › Es existieren Konzepte für viele Test-Anwendungsfälle
- › Besitzt Erweiterungskonzepte
 - › TestRunner
 - › TestRules

JUNIT5: Architektur

JUnit4-Tests

JUnit5-Tests

Andere Tests

Vintage

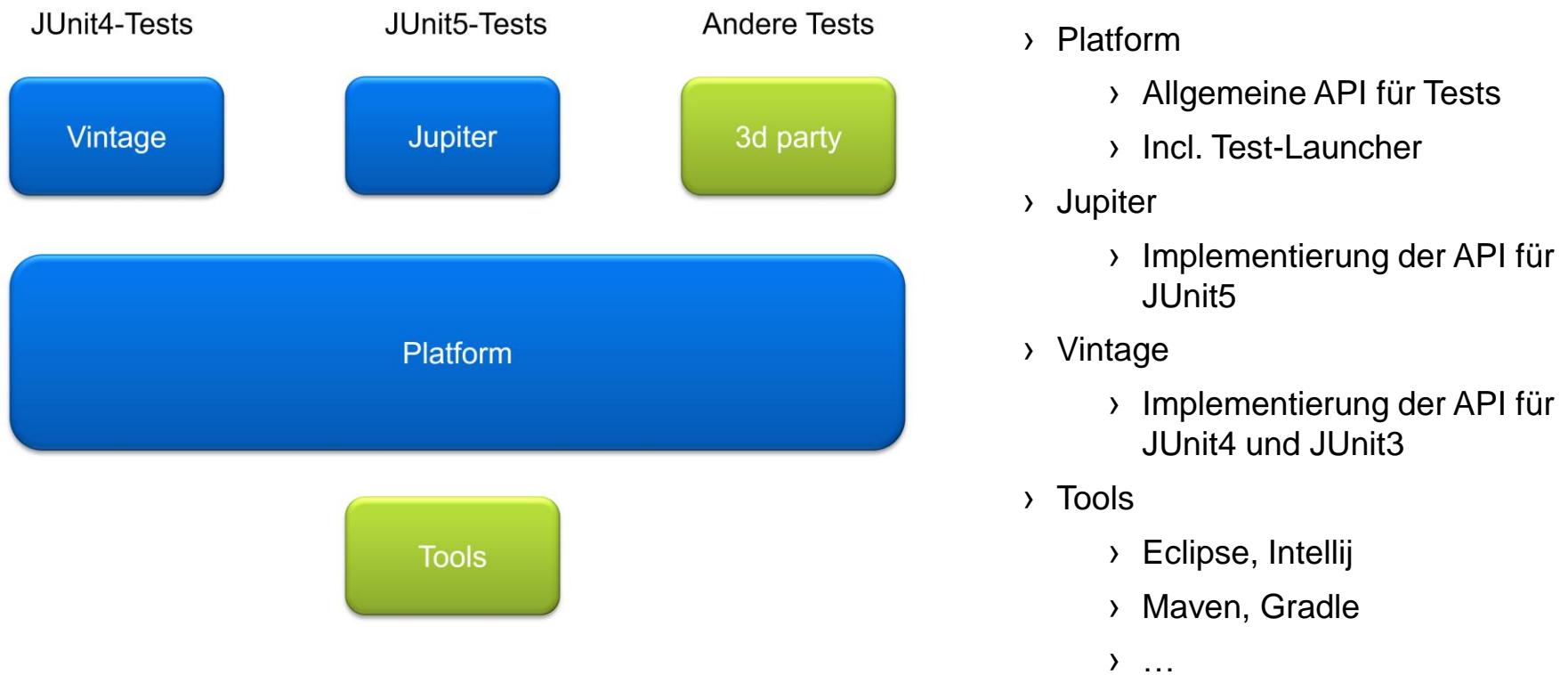
Jupiter

3d party

Platform

Tools

JUnit5 : Modularer Aufbau



Beispiele für PlatformErweiterungen

› JQwik

- › Framework für dynamische Tests mit Properties

› Specsy

- › Framework für Behavior Driven Tests
- › Ermöglicht selbstbeschreibene Tests

Test-Struktur
Wichtige Annotationen

Allgemeines zu Tests
JUnit5

Ausführung managen
Vergleiche/Zusicherungen

Behavior Driven Design
Test-Driven-Design

Parametrierte Tests
TestFactory

Hook-Methoden
Erweiterungen

Proxy-Objekte
Mocks

Spezielle
Testanwendungen



Kennzeichen der Test-Methode

- › Test-Annotation

```
@org.junit.jupiter.api.Test  
public void yourTestMethod(){  
    ....  
}
```

- › Mit @org.junit.jupiter.api.Test annotierte Methode wird von der Engine ausgeführt
 - › Es gibt noch einige andere Annotationen, die von der Engine ausgeführt werden
- › Ausführung / Ergebnis kann durch verschiedene Annotationen beeinflusst werden

Testmethode: Annotation

› Test-Annotation

```
@org.junit.jupiter.api.Test  
public void yourTestMethod(){  
    ....  
}
```

In der Test-Methode wird eine Exception vom Typ „AssertionError“ geworfen

Failure

In der Test-Methode wird eine Exception vom Typ „TestAbortedException“ geworfen

Abbruch

In der Test-Methode wird eine Exception anderen Typs als geworfen

Error

Die Test-Methode wird „normal“ fehlerfrei (ohne Exception) beendet.

OK

Tests ohne Inhalt sind immer grün

```
public class TestResultTest {

    @Test
    public void doNothingAndIsGreen() {

    }

    @Test
    public void doAnythingWithoutErrorAndIsGreen() {
        int i = 5;
        System.out.println("value = " + 5); // note: output in test is a silly idea
    }

    @Test
    public void thisMethodThrowsAnExceptionAndFails() {
        throw new IllegalStateException("hating this");
    }

    @Test
    public void thisMethodThrowsAnSpecialExceptionAndFails() {
        throw new AssertionError( detailMessage: "hating this" );
    }
}
```

```
▼ ⓘ TestResultTest
  ✓ doAnythingWithoutErrorAndIsGreen()
  ✘ thisMethodThrowsAnSpecialExceptionAndFails()
  ✓ doNothingAndIsGreen()
  ⓘ thisMethodThrowsAnExceptionAndFails()
```

Allgemeines Ausschalten von Tests

- › Tests können ausgeschaltet (disabled) werden
 - › Annotation: `@org.junit.jupiter.api.Disabled`
 - › Mögliche Angabe einer Dokumentation (Grund, Task, ...)
 - › Möglich an Testmethode oder Testklasse

```
@Test  
@Disabled  
public void disabledTestWithoutReason() {  
}  
  
@Test  
@Disabled("the test goes red, but I don't know the reason ...")  
public void disabledTestBySillyReason() {  
}
```

```
DisabledTest  
disabledTestWithoutReason()  
disabledTestBySillyReason()
```

Hier erst mal nur informativ (weil einige Testbeispiele ausgeschaltet sind)
Später mehr hierzu ...

Zusicherungs-Fehler und andere Fehler

```
public class FailuresAndErrorsTest {  
  
    public void thisMethodIsNoTestMethodBecauseItIsNotAnnotated() {  
    }  
  
    @Test  
    public void thisTestMethodWillRunGreen() {  
    }  
  
    @Test  
    public void thisTestMethodWillRunRedByFailure() {  
        throw new AssertionError(detailMessage: "this is an example failed test-method");  
    }  
  
    @Test  
    public void thisTestMethodWillRunRedByError() {  
        throw new RuntimeException("this is an example for an error in test-method");  
    }  
}
```

→ Test-Annotation

```
@org.junit.jupiter.api.Test  
public void yourTestMethod(){  
    ....  
}
```

In der Test-Methode wird eine Exception vom Typ „AssertionError“ geworfen

Failure

In der Test-Methode wird eine Exception vom Typ „TestAbortedException“ geworfen

Abbruch

In der Test-Methode wird eine Exception anderen Typs als geworfen

Error

OK

Die Test-Methode wird „normal“ fehlerfrei (ohne Exception) beendet.

Maven-Build-Test

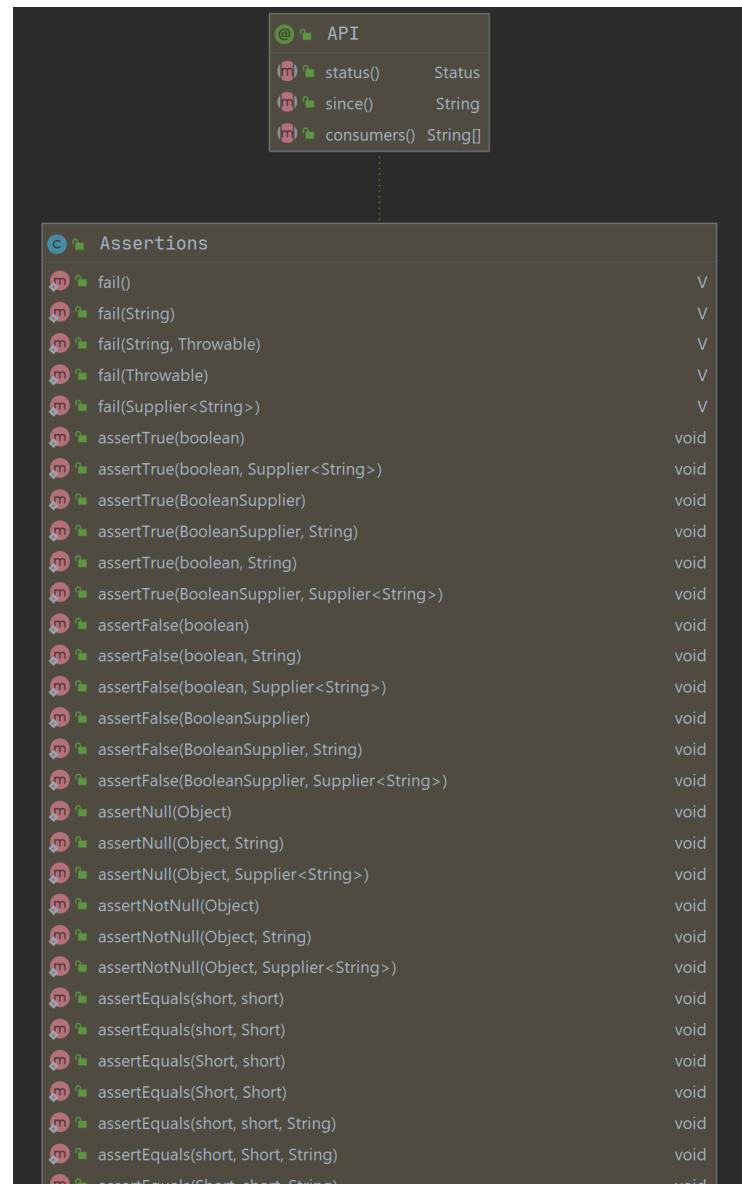
```
[INFO] Running de.hegmanns.training.junit5.example001.demo001.FailuresAndErrorsTest  
[ERROR] Tests run: 3, Failures: 1, Errors: 1, Skipped: 0, Time elapsed: 0.008 s <<< FAILURE! - in de.hegmanns.training.junit5.example001.demo001.FailuresAndErrorsTest  
[ERROR] thisTestMethodWillRunRedByError Time elapsed: 0 s <<< ERROR!  
java.lang.RuntimeException: this is an example for an error in test-method  
    at de.hegmanns.training.junit5.example001.demo001.FailuresAndErrorsTest.thisTestMethodWillRunRedByError(FailuresAndErrorsTest.java:24)  
  
[ERROR] thisTestMethodWillRunRedByFailure Time elapsed: 0 s <<< FAILURE!  
java.lang.AssertionError: this is an example failed test-method  
    at de.hegmanns.training.junit5.example001.demo001.FailuresAndErrorsTest.thisTestMethodWillRunRedByFailure(FailuresAndErrorsTest.java:19)
```

Test-Runner
(IntelliJ)

▼ FailuresAndErrorsTest
 ✓ thisTestMethodWillRunGreen()
 ⚠ thisTestMethodWillRunRedByError()
 ✗ thisTestMethodWillRunRedByFailure()

Klasse zum Ausdrücken von Erwartungshaltungen: Assertions

- › Enthält einfache Wert-Vergleiche mit jeweils paarweise identischen Typen (int, short, ...)
 - › Sowohl Check auf Gleichheit als auch Check auf Ungleichheit
- › Enthält auch komplexere Vergleiche mit Collections
- › Enthält auch Checks auf Basis von Lambdas
- › Kann Lambdas ausführen
- › Kann mehrere Checks ausführen
- › Es finden sich nun asserts, die in JUNIT4 noch TestRules waren
 - › Erwartung einer Exception
 - › Maximale Ausführungszeit
 - › Kontrolle mehrerer Erwartungshaltungen, ohne direkt einen Fehler auszulösen
- › Die Assertions-Klasse ist eine Fassade und delegiert häufig zu anderen Klassen
- › Grundsätzlich dürfen auch andere Vergleichsklassen verwendet werden



Assertions-Klasse: Fassade für Vergleiche

```
public class ContainedCheckClassesTest {

    private final Integer expectedResult = 5; // try with 5

    @Test
    public void makeCheckWithoutContainedCheckClass() {
        int result = 2 + 2;

        if (expectedResult != result) {
            throw new AssertionError( detailMessage: "expected " + expectedResult + " but was " + result);
        }
    }

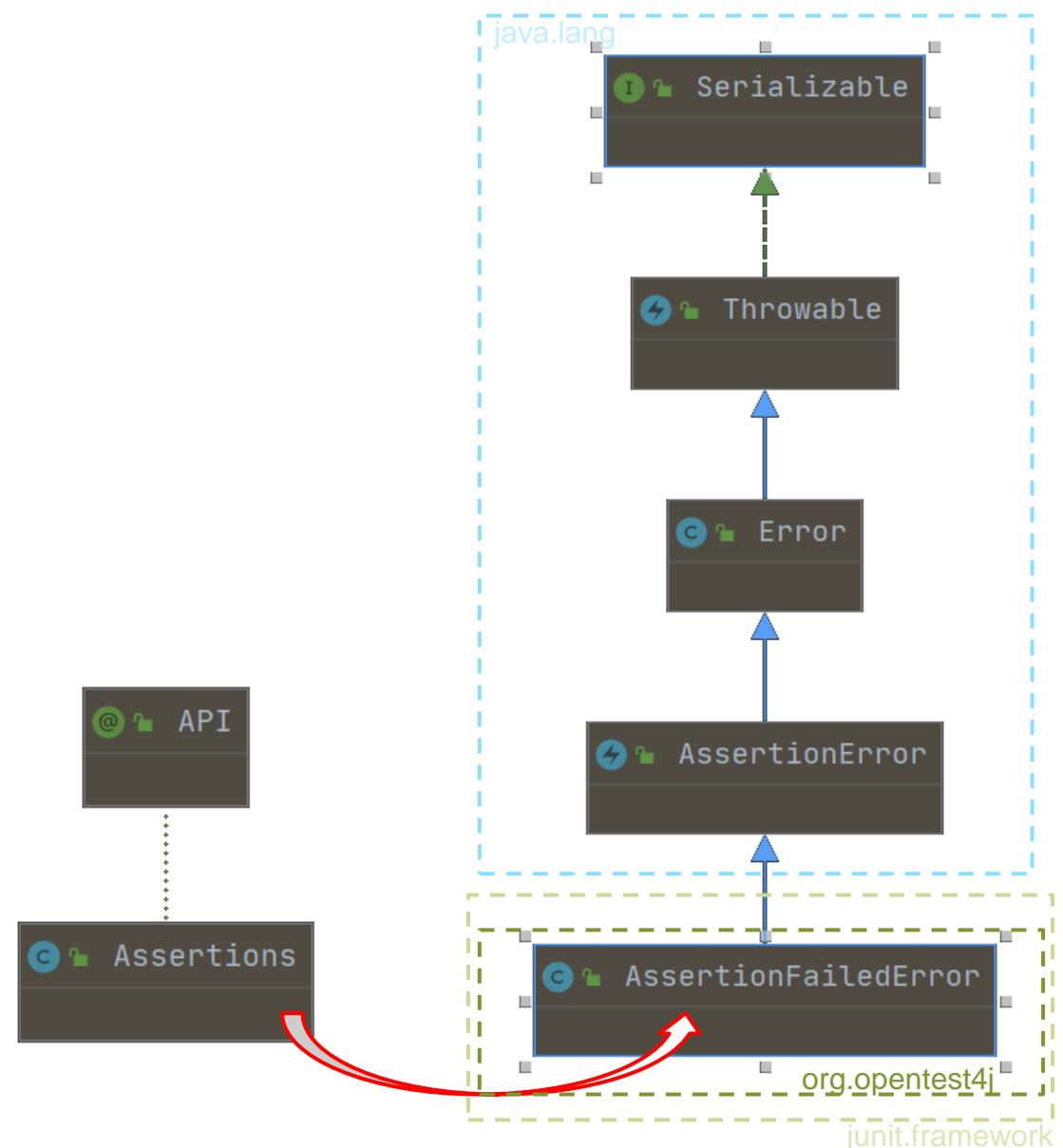
    @Test
    public void makeCheckWithContainedAssertionsClass() {
        int result = 2+2;

        Assertions.assertEquals(expectedResult, result, message: "problem adding two numbers");
    }
}
```

The screenshot shows a Java IDE interface with three main sections:

- Left Panel:** A tree view of test classes and methods. It shows a single test class, `ContainedCheckClassesTest`, which contains two methods: `makeCheckWithoutContainedCheckClass()` and `makeCheckWithContainedAssertionsClass()`. The first method has a duration of 35 ms and failed, while the second has a duration of 8 ms and failed.
- Middle Panel:** A stack trace of the failed assertion. It starts with `java.lang.AssertionError: expected 5 but was 4` and continues to the source code location at `ContainedCheckClassesTest.java:15`.
- Right Panel:** A detailed view of the failed assertion. It shows the error message: `org.opentest4j.AssertionFailedError: problem adding two numbers ==>`, followed by the expected value: `Expected :5` and the actual value: `Actual :4`. There is also a link: `<Click to see differences>`.

JUNIT5-Assertions werfen bei Failure AssertionFailedError-Instanzen



Assertions: „assertEquals(...)" / „assertNotEquals(...)" / „assertArrayEquals"



Absicherung von Gleichheit / Ungleichheit

- › Einzelwert-Überprüfung auf Gleichheit/Ungleichheit
 - › Mit String, einfachen Datentypen, Wrapper, Kombination Wrapper<->einfacher Datentyp
 - › Allgemein Object (dann über die equals-Methode)
- › Array-Überprüfung auf Gleichheit/Ungleichheit
 - › Länge
 - › Einzel-Werte
 - › Bei Object-Array über equals-Methode
- › Iterable-Überprüfung auf Gleichheit
 - › Einzelwerte
 - › Bei Object-Iterable über equals-Methode
 - › Länge
- › String-List-Überprüfung auf Matching (assertLinesMatch)
- › Object-Überprüfung auf Identität/Nicht-Identität
- › Mit und Ohne Nachricht im Fehlerfall
 - › Einfacher String
 - › Supplier<String>

Assertions: „assertTrue(...)" / „assertFalse(...)" / „fail(...)"



Absicherung einer/keiner Exception

- › Zusicherung einer bestimmten Exception
 - › Die Exception wird zurück gegeben zwecks weiterer Auswertung
- › Zusicherung keiner Exception
 - › Der Ergebnisausdruck wird zurück gegeben
- › Im Zuge von Exception-Zusicherungen kommt es häufig zu mehreren Checks
 - › Die Exception (entweder Zusicherung/Ausschluss)
 - › **Weitere Auswertung der Exception**
 - › **Weitere Auswertung des Ergebnisses**

Test-Methode: Beim ersten ungetrappen Error wird beendet

- › Beim ersten (ungecatchen) Fehler wird die Ausführung der Testmethode **abgebrochen**
- › Möglichst **nur einen Check** pro Testmethode
 - › Gefahr des „Verdeckens“ von Fehlern
 - › Ein Testdurchlauf soll direkt alle Fehler aufzeigen

```
@Test
public void noFailureSoAllIsProceeded() {
    int firstNumber = 1;
    int secondNumber = 2;

    int sum = firstNumber + secondNumber;
    int expectedSum = 3;

    Assertions.assertEquals(expectedSum, sum);
    System.out.println("This Message is printed for this test-method");
}

@Test
public void failureStopsProceedingTheRest() {
    int firstNumber = 1;
    int secondNumber = 2;

    int sum = firstNumber + secondNumber;
    int expectedSum = 5;

    Assertions.assertEquals(expectedSum, sum);
    // the following code isn't proceeded

    System.out.println("This messages isn't printed for this test-method");
    Assertions.assertEquals( expected: 10, firstNumber, message: "it's wrong, but this failure isn't displayed ...");
}

@Test
public void onlyNonCatchedErrorsStopProceeding() {
    int firstNumber = 1;
    int secondNumber = 2;

    int sum = firstNumber + secondNumber;
    int expectedSum = 5;

    try {
        Assertions.assertEquals(expectedSum, sum);
        // the following code inside this block isn't proceeded
        System.out.println("wouldn't be printed");
    } catch (AssertionError error) {
        // cached and forgot ...
    }

    System.out.println("This messages is printed for this test-method");
    Assertions.assertEquals( expected: 10, firstNumber, message: "it's wrong, and now it's displayed");
}
```

Textur eines Tests: AAA-Pattern

Arrange

- › Initiale Situation für den Test herstellen (Testvorbereitung)
 - › Nötige Instanzen erzeugen
 - › Konfiguration herstellen
 - › Ggf. Server starten, Datenbank starten, ...

Act

- › Aktion ausführen, die zum erwarteten Ergebnis führt.
 - › Sollte im Allgemeinen die Ausführung einer Methode sein
 - › Rückgabewerte einer Methode
 - › Betrachtung des Objekts, an dem die Methode ausgeführt wurde

Assert

- › Erwartetes Ergebnis gegen erhaltenes Ergebnis kontrollieren
 - › Rückgabewert, Instanz-Eigenschaft
- › Wenn die Erwartung zutrifft, soll sich ein Test möglichst unauffällig verhalten
- › Wenn die Erwartung nicht zutrifft, soll dies angezeigt werden
 - › ... und zwar so, dass möglichst ohne Code-Einsicht das Problem sofort erkannt werden kann ...

Textur eines Tests: AAA-Pattern

Arrange

- › Initiale Situation für den Test herstellen (Testvorbereitung)

Nötige Instanzen erzeugen

```
@Test  
public void withdrawOneReducesBalanceByOne(){  
    Account anyAccount = new Account( accountNumber: "1", balance: 10);  
    anyAccount.withdraw( money: 1);  
    MatcherAssert.assertThat(anyAccount.getBalance(), Matchers.is( value: 9));  
}
```

- › Sollte im Allgemeinen

- › Rückgabewert€ einer Methode

Act

Ein analoges Pattern:

Given

When

Then

- › Ein Test besteht aus diesen Phasen
- Rückgabewert, Instanz Eigenschaften
- › Wenn die Erwartung zutrifft, soll sich ein Test möglichst unauffällig verhalten
 - › Wenn die Erwartung nicht zutrifft, soll dies angezeigt werden
 - › ... und zwar so, dass möglichst ohne Code-Einsicht das Problem sofort erkannt werden kann ...

Assert

Code ausgeführt wurde

kontrollieren

Styleguides für Tests, Testmethoden, Testklassen ...

Name der Testmethode

- › Testname sollte die Testsituation ausdrücken
 - › Nicht/wenig zurückhalten in der Länge des Namens
- › Testname sollte positiv formuliert werden
- › Dadurch werden fehlerhafte (rote) Tests selbstsprechend

Die zu testende Instanz/Methode

- › Die zu testende Instanz sollte im Zweifel unter den Variablen schnell auffindbar sein
- › Möglichst klaren und ggf. allgemeinen Namen verwenden
 - › Tipp: „subject under test“, also „sut“
 - › „object under test“, also „out“

Andere Instanzen, Konfiguration, Werte

- › Möglichst Hinweis auf die Art mit geben
- › Falls passend, in dem Variablennamen auch die Fachlichkeit zum Ausdruck bringen
- › Beispiele:
 - › any, anyLowNumber, anyHighNumber, zero, oneHigherThanMaxInt, anyNegativeNumber, ...
 - › disruptedAnswerFromRestService, unparsableContentFromService, ...

Der Name macht ...



```
GoodNamesAndBadNamesTest
  ✓ checkMath01()
  ✓ checkMath02()
  ✗ checkMath03()
```

Andere können auch schlechte Namen ...

```
{org.apache.commons.lang3.ValidateTest}
```

```
-----  
@Test  
void testIsTrue1() {  
    Validate.isTrue(true);  
    try {  
        Validate.isTrue(false);  
        fail("Expecting IllegalArgumentException");  
    } catch (final IllegalArgumentException ex) {  
        assertEquals("The validated expression is false", ex.getMessage());  
    }  
}  
  
-----  
@Test  
void testIsTrue2() {  
    Validate.isTrue(true, "MSG");  
    try {  
        Validate.isTrue(false, "MSG");  
        fail("Expecting IllegalArgumentException");  
    } catch (final IllegalArgumentException ex) {  
        assertEquals("MSG", ex.getMessage());  
    }  
}  
  
-----  
@Test  
void testIsTrue3() {  
    Validate.isTrue(true, "MSG", 6);  
    try {  
        Validate.isTrue(false, "MSG", 6);  
        fail("Expecting IllegalArgumentException");  
    } catch (final IllegalArgumentException ex) {  
        assertEquals("MSG", ex.getMessage());  
    }  
}
```



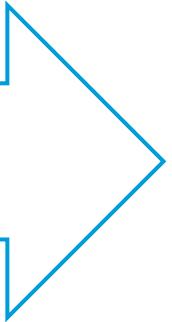
... und als schlechtes Beispiel dienen.

Noch etwas zu Namen in Tests allgemein

- › Namen in Tests haben typischerweise eine andere Motivation als Applikationscode
 - › Testklassen, Testmethoden, TestRessourcen gehören nicht zum Applikationscode
- › Daher im Team überlegen, ob Namen in Tests (Klasse, Methode, Variablen) den typischen Styleguides entsprechen müssen
 - › Vielleicht ist auch ein „eigener Test-Styleguide“ sinnvoll
 - › Vielleicht ist es auch sinnvoll, den Styleguide für Tests auszuschließen
- › An Stelle von Camel-Case Unterstriche zwischen Worten verwenden
 - › Also an Stelle: withdrawOneReducesBalanceByOne
eher: withdraw_one_reduces_balance_by_one
- › Bei Tests an einer konkreten Methode Methodename und den Testcase mit Unterstrich trennen
 - › Also an Stelle:
eher:
 - › Keine nichtssagenden Namen verwenden
 - › Gleiche Sorgfalt wie für Applikationscode verwenden
 - › Gute Namen entscheiden bei roten Tests über das Verständnis und schnelle Lösung
 - › Nicht einfach Tests/Namen durchnummernieren

Noch ein paar weitere Tipps zur Vorgehensweise

Fehler bei der Quellcodedurchsicht gefunden?

- 
- › NICHT sofort korrigieren.
 - › Besser auf jeden Fall einen Test hierzu schreiben (der dann rot wird)
 - › Denn:
 - › Vielleicht gibt's noch weitere Fehler, die so spontan gar nicht auffallen
 - › Der Fehler ist, wenn der Test dann grün ist, auf jeden Fall korrekt gelöst

Suchen Sie mögliche Testkonstellationen/Inputs?

- 
- › NICHT aus dem Quellcode Konstellationen ermittelt.
 - › Das lenkt zu sehr ab und schränkt zu sehr ein
 - › Besser allgemein aus Typ der Input-Werte kombiniert mit Fachlichkeit suchen (Fachlichkeit ist aber eher unwichtig)
 - › Denn:
 - › Sie erhalten ggf. unnötigen Code signalisiert
 - › Tests auf Basis der Implementierung sind selbst-prophezeiend



Kleiner Rückblick

- › Für eine Test-Methode: @Test-Annotation (auf Jupiter-Test achten)
 - › Für Ausführung in Maven/Gradle auf Dateipattern achten
- › Assertions-Klasse enthält diverse Zusicherungsprüfungen
- › Test-Styleguide zu Eigen machen
- › Implementieren Sie Tests gemäß des AAA-Pattern
- › Möglichst einen guten Testumfang finden
 - › 100% Abdeckung mit möglichst wenig Testfälle



Übung 01 (Fortsetzung)

› Ressourcen

de.hegmanns.training.junit5.practice.task01

Die Klasse „CalculatorForDivide“ bietet einen Service an, mit dem ermittelt werden kann, ob eine vorhandene Menge so in zwei Teile aufgeteilt werden kann, dass dabei jeweils eine gerade Zahl heraus kommt.

Beispiel:

Menge 8, könnte aufgeteilt werden in $4 + 4$ und damit ist eine Aufteilung in zahlenmäßig zwei gerade Mengen möglich.

Menge 5, könnte aufgeteilt werden in $2 + 3$, $1 + 4$ und damit ist eine Aufteilung in zahlenmäßig zwei gerade Mengen nicht möglich.

1. Überlegen Sie sich sinnvoll Test-Inputs, so dass mit großer Sicherheit über die Korrektheit der Implementierung entschieden werden kann.
In die Testklasse „EvenDividableTest“ ist schon ein zu testender Input eingetragen.
Ergänzen Sie die Liste der zu testenden Inputs im Java-DOC. (Schreiben Sie noch keine Tests!)
2. Erstellen Sie vier zusätzliche Testmethoden, so dass mit den Tests eine Aussage über die Korrektheit der Funktion getroffen werden kann.
Verwenden Sie die bereits vorab erstellte Testklasse „EvenDividableTest“.
Verwenden Sie gute Methodennamen.
Falls hilfreich, dokumentieren Sie in jeder Testmethode jeweils mit Hilfe des AAA-Pattern.

Falls nötig, korrigieren Sie die Implementierung der Klasse CalculatorForDivide.

Assertions: „assertThrows(...)" / „assertDoesNotThrow(...)"



Absicherung einer/keiner Exception

- › Zusicherung einer bestimmten Exception
 - › Die Exception wird zurück gegeben zwecks weiterer Auswertung
- › Zusicherung keiner Exception
 - › Der Ergebnisausdruck wird zurück gegeben
- › Im Zuge von Exception-Zusicherungen kommt es häufig zu mehreren Checks
 - › Die Exception (entweder Zusicherung/Ausschluss)
 - › Weitere Auswertung der Exception
 - › Weitere Auswertung des Ergebnisses

```
@Test
public void expectedException() {
    NullPointerException expectedException = Assertions.assertThrows(NullPointerException.class, () -> {
        Object o = null;
        o.toString();
    });

    // here you can work with get exception
}
```

Assertions: „assertIterableEquals(...)"



Gleichheit von Mengen

- › Zusicherung von Gleichheit an konkreter Position
 - › Zusicherung der gleichen Länge
- › Durchlaufen der Collection über Iterator

```
@Test
public void expectedIterable() {
    List<String> lines = Stream.of("hello", "great", "world").collect(Collectors.toList());

    Assertions.assertIterableEquals(Arrays.asList("hello", "great", "world"), lines);
}
```

Assertions: „assertSame(...)" / „assertNotSame(...)"



Identität von Instanzen

- › Referenzen verweisen auf die gleiche Instanz

```
@Test  
public void expectedSame() {  
    Integer a = 12;  
    Integer b = a;  
  
    Assertions.assertSame(b, a);  
}
```

Assertions: „assertLinesMatch(...)"



Gleichheit von String-Listen

- › Gleicher Inhalt des Strings am jeweiligen Index
- › Gleiche Länge der Listen

```
@Test
public void expectedLines() {
    List<String> lines = Stream.of("hello", "great", "world").collect(Collectors.toList());

    Assertions.assertLinesMatch(NSArray.asList("hello", "great", "world"), lines);
}
```

Assertions: „`assertNull(...)`“ / „`assertNotNull(...)`“



Erwartung von null

- › Variable ist null zugewiesen / bzw. noch nicht initialisiert

```
@Test
public void expectedNull() {
    Integer a = null;
    Integer b = 12;

    Assertions.assertNull(a);
    Assertions.assertNotNull(b, message: "not null expected");
    Assertions.assertNotNull(b, () -> "not null expected");
}
```



Übung 02

› Ressourcen

[de.hegmanns.training.junit5.practice.task02](#)

Die Klasse „ShortTermCalculator“ bietet einen Service an, mit dem überlange Wörter in einer abgekürzten Weise dargestellt werden können.

Überlang ist ein Wort immer dann, wenn es sich aus mehr als 10 Zeichen zusammen setzt. Die abgekürzte Schreibweise besteht dann aus dem ersten Zeichen, gefolgt von der Anzahl der Zeichen zwischen dem ersten und dem letzten Zeichen und schließlich dem letzten Zeichen.

Es wird immer davon ausgegangen, dass der Input aus genau einem Wort besteht.

Beispiel:

Das Wort „internationalization“ würde als abkürzende Schreibweise „i18n“ enthalten.

1. Überlegen Sie sich sinnvoll Test-Inputs, so dass mit großer Sicherheit über die Korrektheit der Implementierung entschieden werden kann.
2. Die bereits vorhandene Methode sichert das Werfen einer „NumberFormatException“ auf Junit3/4-Art. Stellen Sie die Methode unter Zuhilfenahme von Junit5-Features um.
3. Erstellen Sie fünf Testmethoden.
Verwenden Sie die bereits vorab erstellte Testklasse „ShortTest“
Falls Fehler gefunden werden, bitte den ShortTermCalculator entsprechend korrigieren.

Assertions: Timeout



Erwartung einer maximalen Ausführungszeitdauer

- › Bei Überschreiten der maximal erwarteten Zeit wird der Test rot
- › Zwei Varianten
 - › Abbrechen der Ausführung (assertTimeout(...))
 - › Warten auf das Ende der Ausführung (assertTimeoutPreemptively(...))



Timeout ohne Abbruch ...

- › Gibt einen Hinweis auf die wirkliche Ausführungszeitdauer
- › Verlängert allerdings die Testausführung
- › Möglichst keine weiteren Checks
 - › Hierdurch kann das Timeout versteckt werden

```
@Test  
public void timeoutWaiting() {  
    Assertions.assertTimeout(Duration.ofSeconds(1), () -> {  
        Thread.sleep(1L 2200);  
        System.out.println("proceed");  
    });  
}  
  
@Test  
public void timeoutNotWaiting() {  
    Assertions.assertTimeoutPreemptively(Duration.ofSeconds(1), () -> {  
        Thread.sleep(1L 2000);  
        System.out.println("proceed");  
    });  
}
```

✗ timeoutWaiting()
✗ timeoutNotWaiting()

2 s 238 ms
1 s 13 ms

Assertions mit timeout



JUNIT4 hat Schwächen



JUNIT4 kann schon viel

- › Gehören eigentlich nicht in reine Unit-Tests
- › Verlängern die Testausführung
- › Testen häufig die Plattform und Konfiguration mit
- › Sollten nicht im allgemeiner Testausführung rein
 - › Ggf. in eigenem Profil
 - › Ggf. in

- › Geben einen Aufschluss über Performance, Skalierbarkeit

Assertions: assertAll(...), Ausführung mehrerer Checks



Ausführung mehrerer Checks mit assertAll(...)

- › Auftretende Fehler (Error) werden gesammelt
- › Am Ende erfolgt eine Zusammenfassung der Fehler
- › Der Test ist grün, wenn alle Executables grün sind.



Einsatz gut überlegen

- › Testmethode soll weiterhin den gleichen Kontext prüfen, also sparsam einsetzen
 - › z.B. die Attribute einer Instanz
- › Test-Recovery-Tools könnten an ihre Grenzen geraten

```
@Test
public void andWhatAboutErrors() {
    Long oneValue = null;

    // you get all failures and errors displayed :
    Executable[] checkExecutables = {
        () -> {Assertions.assertNotNull(oneValue); },
        () -> {Assertions.assertTrue( condition: oneValue < 5); },
        () -> {throw new RuntimeException("what a chrime"); },
        () -> {Assertions.assertNull(oneValue, message: "you see only problems in the summary, so not this (green) check"); },
        () -> {Assertions.assertEquals( expected: 2, oneValue); }
    };

    Assertions.assertAll( heading: "now " + checkExecutables.length + " tests are proceeded",
        checkExecutables
    );
}
```

```
org.opentest4j.MultipleFailuresError: now 5 tests are proceeded (4 failures)
  expected: not <null>
  <no message> in java.lang.NullPointerException
  what a chrime
  expected: <2> but was: <null>
```

Assert und Fehlertexte

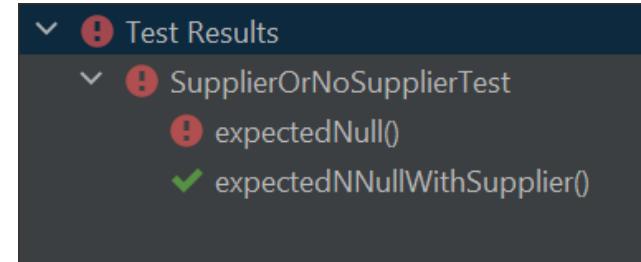
› Typische Methoden von Assertions

- › Ohne Fehlertext
- › Mit Fehlertext als String
- › Mit Fehlertext über Supplier

Supplier

- › Unterstützung lazy initialization
- › Fehlerwert wird erst bei negativer Auslösung der Erwartung erzeugt

```
public class SupplierOrNoSupplierTest {  
  
    @Test  
    public void expectedNull() {  
        Integer a = null;  
  
        Assertions.assertNull(a, message: "value = " + a.intValue());  
    }  
  
    @Test  
    public void expectedNNullWithSupplier() {  
        Integer a = null;  
  
        Assertions.assertNull(a, () -> "value = " + a.intValue());  
    }  
}
```





Kleiner Rückblick

- › Für eine Test-Methode: @Test-Annotation (auf Jupiter-Test achten)
 - › Für Ausführung in Maven/Gradle auf Dateipattern achten
- › Assertions-Klasse enthält diverse Zusicherungsprüfungen
- › Test-Styleguide zu Eigen machen
- › Implementieren Sie Tests gemäß des AAA-Pattern
- › Möglichst einen guten Testumfang finden
 - › 100% Abdeckung mit möglichst wenig Testfälle



Übung 03

› Ressourcen

de.hegmanns.training.junit5.practice.task03

Die Klasse „LongTimeService“ ermittelt nach einer gewissen zufällig liegenden Gedenkzeit die Summe zweier Zahlen.

Es gibt auch im Test-Bereich die Testklasse „LongtimeServiceTest“, die einerseits die maximale Ausführungszeit sicherstellen soll und andererseits auch, dass die Ausführungszeit zwischen einem Minimum- und Maximum-Wert liegt.

1. Die beiden Test verwenden allerdings keine JUnit5-Assertions zur Zeit-Sicherstellung.
Refactoren Sie die beiden Testmethoden der gestalt, dass die entsprechenden Zusicherungsmethoden der Assertions-Klasse verwendet werden.

2. Die im LongtimeService durch Math.random() dargestellte Gedenkzeit könnte in realen Anwendungsfällen durch einen zusätzlichen Serveraufruf auftreten, ggf. in Verbindung mit dem Aufbau einer DB-Session.
Im Testfall verbrennt man allerdings nur unnötig Zeit, wenn man beispielsweise nur das richtige Ergebnis sicherstellen möchte. Refactoren Sie die LongTimeService-Klasse so, dass die korrekte Berechnung sicher und schnell getestet werden kann. Realisieren sie auch einen Test zur Überprüfung der Berechnung

Initialisierungen und Nacharbeiten

Einmalig vor allen Tests

@org.junit.jupiter.api.BeforeAll

Methode muss static sein!

› Grundinitialisierungen

- › Z.B. Datenbank-Verbindung
- › Z.B. Frameworks (Spring...)
- › Z.B. Server starten

Jeweils vor JEDEM Test

@org.junit.jupiter.api.BeforeEach

› Initialisierungen

- › Z.B. Testdaten updaten
- › Z.B. DB-Transaktion starten
- › Z.B. Zeitmessung beginnen

Jeweils nach JEDEM Test

@org.junit.jupiter.api.AfterEach

› Aufräumen nach dem Test

- › Z.B. Zeitnahme
- › Z.B. DB-Transaktion rollback
- ›

Einmalig nach allen Tests

@org.junit.jupiter.api.AfterAll

Methode muss static sein!

› Aufräumen nach allen Tests

- › Z.B. Datenbank-Verbindung beenden
- › Z.B. Server beenden
- › Z.B.

Initialisierungen und Nacharbeiten

Zur Erinnerung JUNIT 4

Einmalig zu allen Tests

JUNIT 5

@org.junit.jupiter.api.....

Jeweils vor JEDEM Test

Test

AfterAll

AfterEach

BeforeAll

BeforeEach

Jeweils nach jedem Test

Einmalig nach allen Tests

@org.junit.jupiter.api.BeforeAll

@org.junit.jupiter.api.AfterEach

@org.junit.jupiter.api.AfterAll

Methode muss static sein!

› Grundinitialisierungen

› Z.B. Datenbank-Verbindung

› Z.B. Frameworks (Spring...)

› Z.B. Server starten

JUNIT 4

› Initialisierungen

@org.junit....

› Z.B. Daten update

› Z.B. DB-Transaktion starten

Test

› Z.B. Zeitmessung beginnen

AfterClass

After

› Aufräumen nach dem Test

› Z.B. Zeithnahme

BeforeClass

Before

› Z.B. DB-Transaktion rollback

›

› Aufräumen nach allen Tests

› Z.B. Datenbank-Verbindung beenden

› Z.B. Server beenden

› Z.B.

Mehrere Lifecycle-Methoden?

- › Grundsätzlich sind auch mehrere Lifecycle-Methoden mit gleicher Annotation erlaubt
 - › Es werden alle Lifecycle-Methoden ausgeführt
 - › Reihenfolge ist ähnlich „zufällig“ wie Ausführung der Testmethoden



Kleiner Rückblick

- › Für eine Test-Methode: @Test-Annotation (auf Jupiter-Test achten)
 - › Für Ausführung in Maven/Gradle auf Dateipattern achten
- › Assertions-Klasse enthält diverse Zusicherungsprüfungen
- › Test-Styleguide zu Eigen machen
- › Implementieren Sie Tests gemäß des AAA-Pattern
- › Möglichst einen guten Testumfang finden
 - › 100% Abdeckung mit möglichst wenig Testfälle



Übung 04

› Ressourcen

[de.hegmanns.training.junit5.practice.task04](#)

Stellen Sie die Testklasse „BoundedCounterTest“ so um,
dass unnötige Code-Douplizierungen vermieden werden können.

HINWEIS:

Diese Vorgehensweise ist nicht untypisch.

Im Allgemeinen werden zunächst diverse Testfälle / Testmethoden erstellt, um zwischendurch die Testfälle selbst zu refactoren und übersichtlicher/einfacher zu gestalten.

Im ersten Schritt kommt es auf die Erstellung der Testfälle an, im zweiten Schritt auf die Verbesserung des Codes.

Ähnlichkeiten im Setup

› Ähnliche / Gleiche Setups können in den Before-Lifecycle-Methoden realisiert werden

› ABER:

› Testklasse sollte testgetrieben sein

› Use-Case-getrieben

› Service-getrieben

› Input-getrieben

Mehrere Klassen

› Durch die Idee der sinnvollen Testklassenaufteilung entstehen mehrere Testklassen mit prinzipiell gleichen Setups

Lösung

› Testklassen-Hierarchien
› Eingebettete Klassen
› Test-Interface
› Test-Factory

Hierarchien

- › Konstruktor-Aufruf wie üblich
- › Auch die Test-Methoden der Oberklasse gehören zur Unterkorrekte, werden also mit ausgeführt
 - › Ggf. werden die Testmethoden der Oberklasse also mehrfach ausgeführt
- › Instanz-Lifecycle-Methoden der Oberklasse werden grundsätzlich in die Unterkorrekte vererbt
 - › ABER Achtung: überschriebene Methoden werden überdeckt!
- › Static-Lifecycle-Methoden (@BeforeAll, @AfterAll) werden auch übernommen
 - › Auch hier mit Ausnahme von reimplementierten Methoden

Und was ist mit Oberklassen/Subklassen?

- › Die „Before“-Methoden werden von den Oberklassen jeweils als Erstes ausgeführt
 - › Also ähnlich wie Konstruktor-Aufrufe von der Oberklasse die Hierarchie runter
- › Die „After“-Methoden werden von den Oberklassen jeweils als Letztes ausgeführt
 - › Also ähnlich wie Destruktor-Aufrufe von der Klasse zur Oberklasse die Hierarchie rau
- › Annotationen an Oberklassen werden mitvererbt
- › Test-Methoden (mit @Test annotiert) werden in Unterklassen mit vererbt
 - › Tests aus Oberklassen werden also ggf. in einem Testlauf öfter ausgeführt ...



NoDos



Gründe für Oberklassen

- › Aufteilung von Tests, die eigentlich in eine Klasse gehören, weil sonst die Testklasse zu groß wäre
 - › Das wäre zumindest ein Punkt für eine zu große Fachklasse

- › Allgemeine Konfigurationen in auch für mehrere Tests wiederkehrende Setups unterbringen
 - › Spring-Konfiguration laden
 - › DB-Verbindung laden
- › Alternativen überdenken, z.B. eine Extension

Testtemplate

- › Generische Klasse mit Testmethoden
 - › Und ggf. noch weiteren Parametrierungen
- › Jede Instanz der Template-Klasse ergeben neue Tests mit neuen Konstellationen
 - › Verschiedene Parametrierung
 - › Verschiedene Konditionen (Versionen, System-Variablen, ...)
 - › Verschiedenes Verhalten beteiligter Systeme
- › Tests sind im TestRunner erkennbar

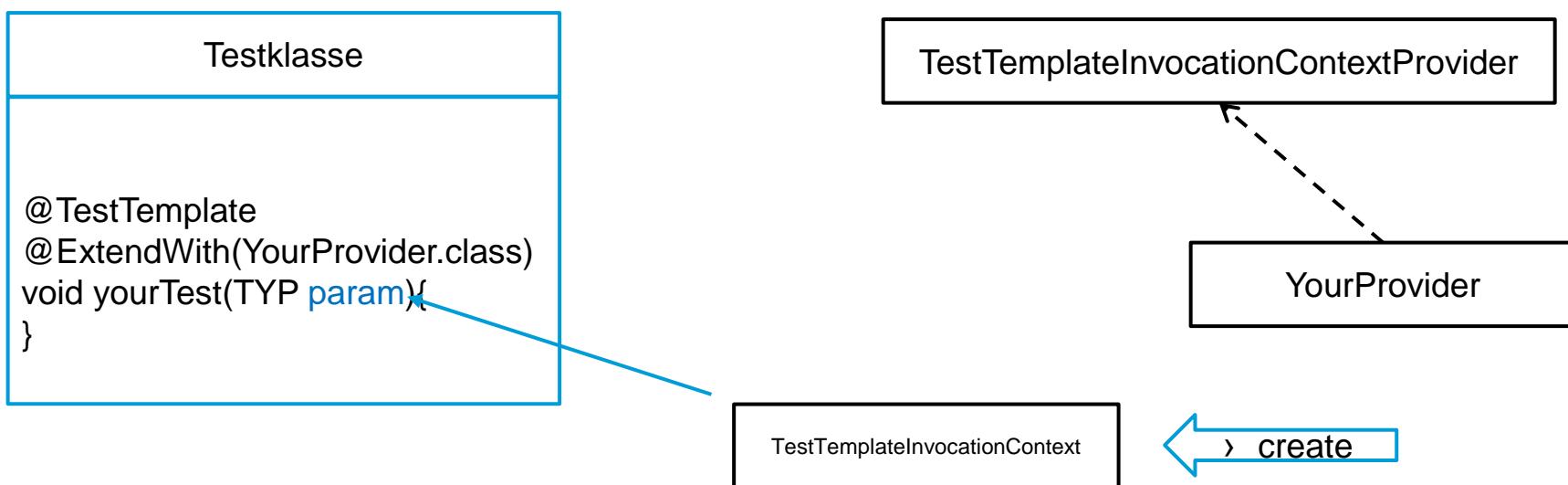
Werden von einem Provider
geliefert



```
    ▼ ✓ Test Results
      ▼ ✓ NumberCheckWithTemplateTest
        ▼ ✓ checkIsPreferred(NumberCheckTestCase)
          ✓ for odd preferred NumberCheck, number 5 should be preferred
          ✓ for odd preferred NumberCheck, number 120 should be not preferred
```

Testtemplate

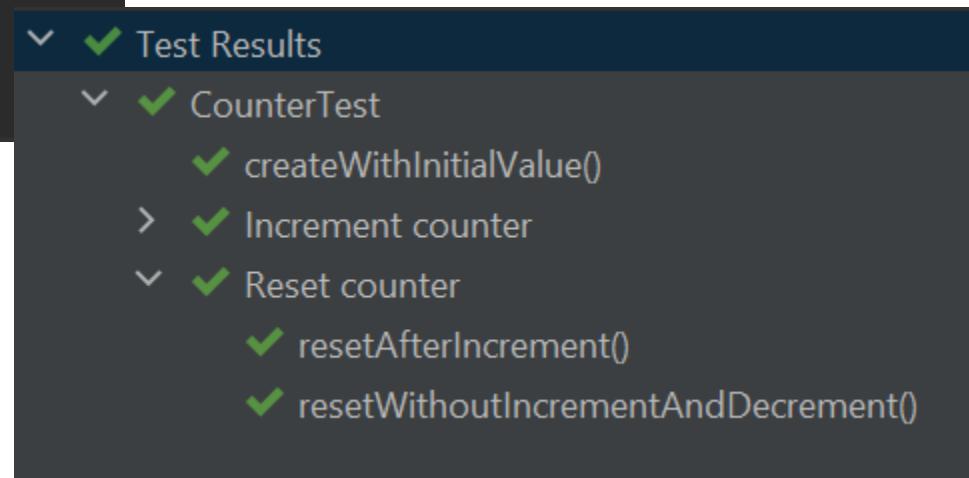
- › Generische Klasse mit Testmethoden
 - › Und ggf. noch weiteren Parametrierungen
- › Jede Instanz der Template-Klasse ergeben neue Tests mit neuen Konstellationen
 - › Verschiedene Parametrierung
 - › Verschiedene Konditionen (Versionen, System-Variablen, ...)
 - › Verschiedenes Verhalten beteiligter Systeme
- › Tests sind im TestRunner erkennbar



Eingebettete Tests (nested tests)

- › Sind technisch in eine Klasse eingebettete Klassen
- › Heben Verwandtschaft deutlicher hervor
 - › Alle Tests in gleicher Source-Datei
 - › Bieten aber dennoch eine Struktur und Code/Sach-Trennung

```
public class CounterTest {  
  
    private Counter counter;  
    private static final int INITIAL_VALUE = 5;  
  
    @BeforeEach  
    public void beforeAnyTest() { counter = new Counter(INITIAL_VALUE); }  
  
    @Test  
    public void createWithInitialValue() { Assertions.assertEquals(INITIAL_VALUE, counter.getValue()); }  
  
    @Nested  
    @DisplayName("Reset counter")  
    class Resetting {...}  
  
    @Nested  
    @DisplayName("Increment counter")  
    class Incrementing{...}  
}
```





Übung 05

› Ressourcen

[de.hegmanns.training.junit5.practice.task05](#)

Erstellen Sie die Test-Oberklasse „AbstractFileHandlingTest“, die bei Verwendung eine zu definierende Datei leer anlegt und zum Ende eines jeden Tests wieder löscht.

Implementieren Sie auch den Konstruktor und die noch nicht vollständige Methode.

HINWEIS:

JUNIT5 bietet so etwas ähnliches bereits als Feature. Das werden wir später kennen lernen.



Übung 06

› Ressourcen

de.hegmanns.training.junit5.practice.task06

Im source-Folder findet sich die Klasse „Cube“, die mittels Parameter in den „unfair“-Modus betrieben werden kann, so dass als Ergebnis immer die Zahl 6 gewürfelt wird.

a)

Erstellen Sie einen Test mit Hilfe von Template, mit dem Abhängig vom festgelegten Parameter das Verhalten des Cube getestet werden kann.

b)

Erstellen Sie hierfür eine neue Annotation, so dass für die Testmethode nur noch eine Annotation verwendet werden muss.

Ausführung managen
Vergleiche/Zusicherungen

Allgemeines zu Tests
JUnit5

Test-Struktur
Wichtige Annotationen

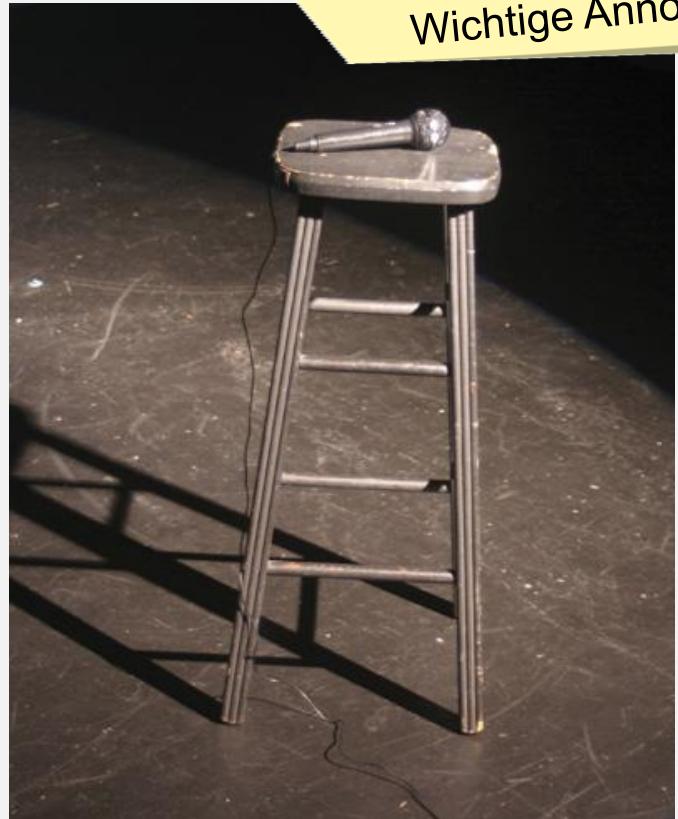
Behavior Driven Design
Test-Driven-Design

Parametrierte Tests
TestFactory

Hook-Methoden
Erweiterungen

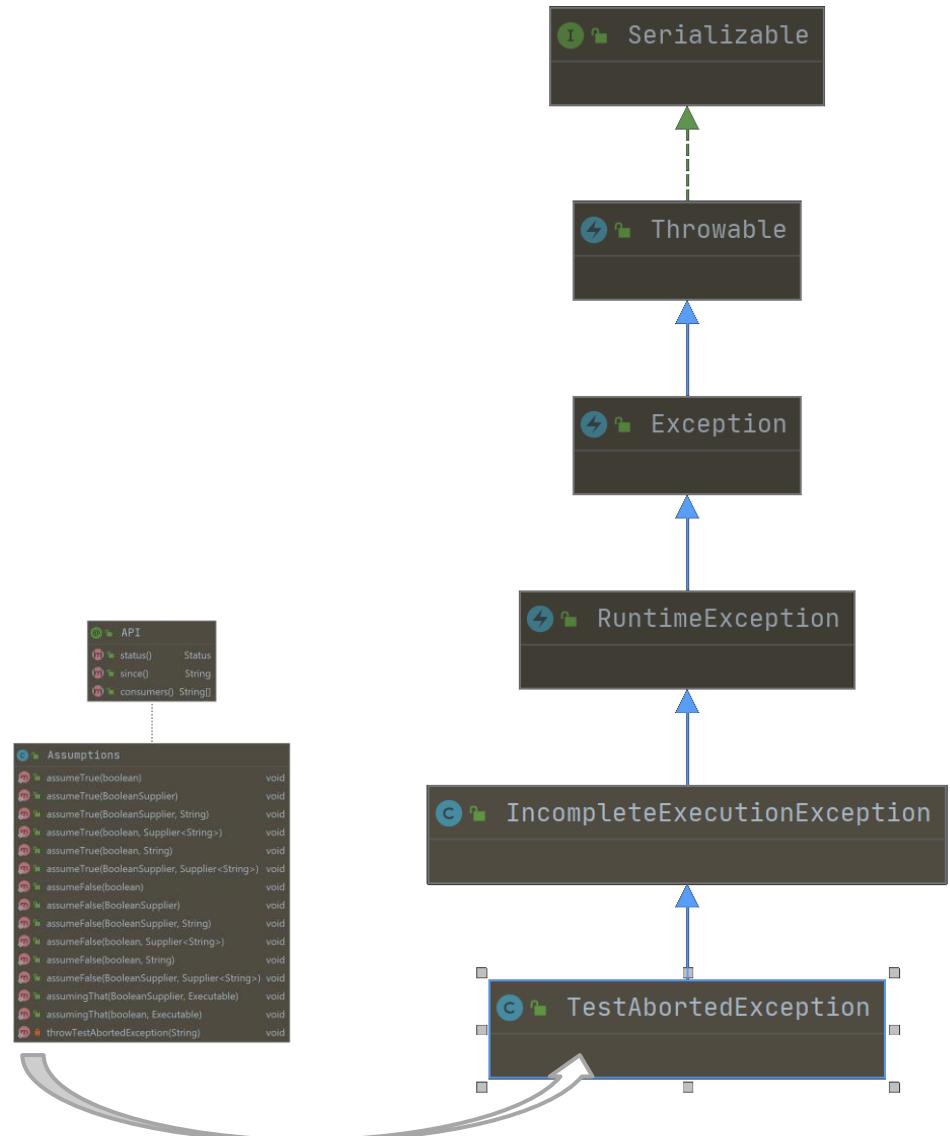
Proxy-Objekte
Mocks

Spezielle
Testanwendungen



Zusicherung, die zu Abbruch ohne Fehler führt: Assumptions

- › Hierdurch können Vorbedingungen zugesichert werden
 - › Vereinfachung von Teststrukturen



Achtung bei Assumptions

```
public class AssumptionTest {  
  
    @Test  
    public void disabledWithFollowingCode() {  
        Assumptions.assumeTrue( assumption: false, message: "assume is working");  
        // not proceed  
        Assertions.assertTrue( condition: false, message: "this will not fail ;)");  
    }  
  
    @Test  
    @Disabled  
    public void disabledWithoutProceedingAnyCode() {  
        Assertions.assertTrue( condition: false); // complete test is disabled  
    }  
  
    @Test  
    public void disabledByAssumingWithImportantTest() {  
        int anyFirst = 1;  
        int anySecond = 2;  
        Integer sum = add(anyFirst, anySecond);  
        Assumptions.assumeTrue( assumption: sum!=null);  
        Assertions.assertEquals( expected: 3, sum);  
    }  
  
    // only for training placed here  
    private Integer add(Integer firstSummand, Integer secondSummand) {  
        return null;  
    }  
}
```

- Wirkung ist ähnlich ausgeschalteter Tests (vielleicht sogar schlimmer, Ausschalten geschieht programmatisch ...)
- Immer gut überlegen, ob ein Test nicht besser rot werden sollte

▼ Ø AssumptionTest		14 ms
Ø	disabledWithFollowingCode()	13 ms
Ø	disabledByAssumingWithImportantTest()	1 ms
Ø	disabledWithoutProceedingAnyCode()	

Allgemeines Ausschalten von Tests

- › Tests können ausgeschaltet (disabled) werden
 - › Annotation: `@org.junit.jupiter.api.Disabled`
 - › Mögliche Angabe einer Dokumentation (Grund, Task, ...)
 - › Möglich an Testmethode oder Testklasse



Vorsicht bei allgemein ausgeschalteten Tests

- › Ausgeschaltete Tests fallen nicht immer sofort auf
- › Ausgeschaltete Tests verursachen keinen Fehler
- › Ausgeschaltete Tests bleiben häufig versehentlich ausgeschaltet
- › Jeder Test hat seinen Sinn
 - › Falls nicht, Test löschen und nicht disablen
 - › Bei einem Problem lieber sofort lösen
- › Niemals einen roten Test disablen



Ausschalten zur schnellen Arbeit

- › Während der Entwicklung macht es mitunter Sinn, langsame Tests auszublenden
- › Aber bei Abschluss sollten dann wieder alle Tests ausgeführt werden

```
@Test  
@Disabled  
public void disabledTestWithoutReason() {  
}  
  
@Test  
@Disabled("the test goes red, but I don't know the reason ...")  
public void disabledTestBySillyReason() {  
}
```

Allgemeines Ausschalten von Tests

- › Tests können ausgeschaltet (disabled) werden
 - › Annotation: `@org.junit.jupiter.api.Disabled`
 - › Mögliche Angabe einer Dokumentation (Grund, Task, ...)
 - › Möglich an Testmethode oder Testklasse

Idealerweise eine Lösung/Issue mit angeben

```
@Test  
@DisplayName("will be solved by issue #12")  
public void domainTest() {  
  
}
```



Ausschalten zur schnellen Arbeit

- › Während der Entwicklung macht es mitunter Sinn, langsame Tests auszublenden
- › Aber bei Abschluss sollten dann wieder alle Tests ausgeführt werden

Bedingtes Ausschalten von Tests

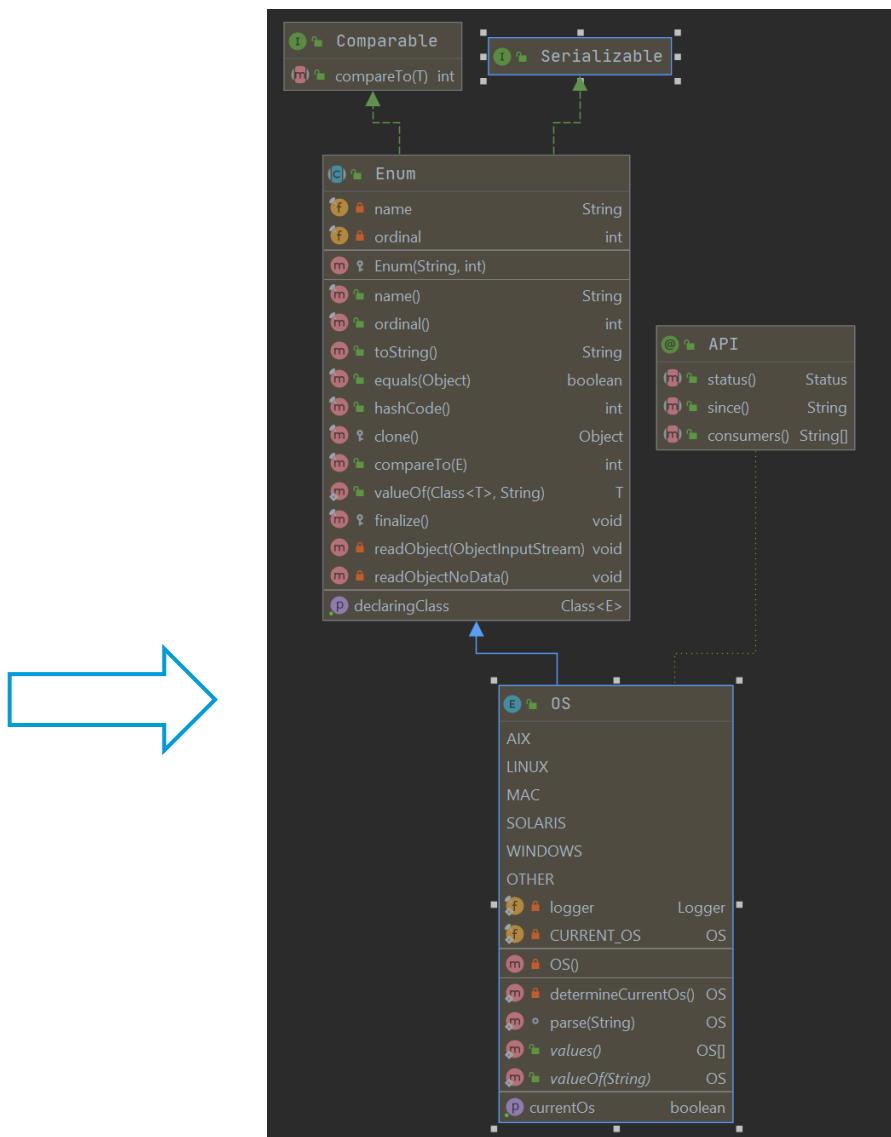
- › Tests können abhängig vom Vorliegen/nicht Vorliegen bestimmter Eigenschaften ein/ausgeschaltet werden
 - › Betriebssystem-Variante
 - › JRE-Version
 - › System-Variable(n)
 - › Komplett eigene Bedingungen
 - › Bedingungen aus Script/Methoden
- › Tests können auch unterteilt werden
 - › Gradle/Maven: Unterscheidung „Komponenten/Integrationstest“
 - › Auch sonst erweiterbar über Kategorien

```
@Test
@DisabledIfSystemProperty(named = "os.version", matches = ".*10.*")
void notRunOnlyOnWindows10() {
    System.out.println("not run this only on windows 10 version");
}

@Test
@EnabledIfSystemProperty(named = "os.version", matches = ".*10.*")
void runOnlyOnWindows10() {
    System.out.println("Run this only on WINDOWS OS 10 version");
}
```

Bedingte Ausführung: Abhängig vom Betriebssystem

→ @EnableOnOs(...)
@DisableOnOs(...)



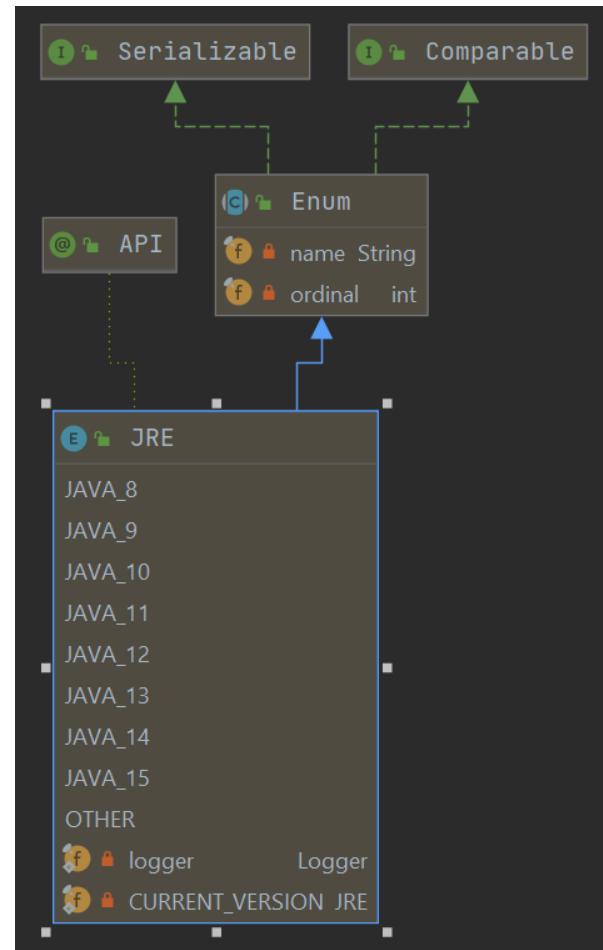
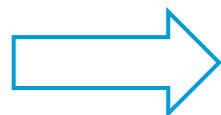
Bedingte Ausführung: Abhängig von JAVA-Plattform

@EnableOnJre(...)
@DisableOnJre(...)

@EnableForJreRange(min=..., max=...)
@DisableForJreRange(min=..., max=...)

@EnableForJreRange(min=...)
@DisableForJreRange(min=....)

@EnableForJreRange(max=...)
@DisableForJreRange(max=....)



Bedingte Ausführung: Abhängig von (JAVA)-System-Property

→ @EnabledIfSystemProperty(named=..., matches=...)

→ @DisabledIfSystemProperty(named=..., matches=...)

→ System.getProperties()

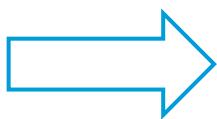
→ System.getProperty(...)

Bedingte Ausführung: Abhängig von (OS)-Environment



@EnabledIfEnvironmentVariable(named=..., matches=...)

@ DisabledIfEnvironmentVariable(named=..., matches=...)



Hintergrund zu bedingten Ausführung

- › Basis ist org.junit.jupiter.api.extension.ExecutionCondition
- › Erstellung einer Annotation
 - › Mit „@ExtendWith“ der erstellten ExecutionCondition

```
public class HogiSampleTestCondition implements ExecutionCondition {  
  
    @Override  
    public ConditionEvaluationResult evaluateExecutionCondition(ExtensionContext extensionContext) {  
        Optional<AnnotatedElement> element = extensionContext.getElement();  
  
        if (element.isPresent()) {  
            HogiCondition annotation = element.get().getAnnotation(HogiCondition.class);  
            if (annotation != null) {  
                String info = annotation.info();  
                return ConditionEvaluationResult.disabled("don't have time for proceeding this boring test: " + info);  
            }  
        }  
        return ConditionEvaluationResult.enabled(...);  
    }  
}  
  
{@Target({ElementType.TYPE, ElementType.METHOD})}  
@Retention(RetentionPolicy.RUNTIME)  
@ExtendWith(HogiSampleTestCondition.class)  
public @interface HogiCondition {  
    String info() default "";  
}
```



Kleiner Rückblick

- › Für eine Test-Methode: @Test-Annotation (auf Jupiter-Test achten)
 - › Für Ausführung in Maven/Gradle auf Dateipattern achten
- › Assertions-Klasse enthält diverse Zusicherungsprüfungen
- › Test-Styleguide zu Eigen machen
- › Implementieren Sie Tests gemäß des AAA-Pattern
- › Möglichst einen guten Testumfang finden
 - › 100% Abdeckung mit möglichst wenig Testfälle



Übung 07.1

› Resourcen

de.hegmanns.training.junit5.practice.task07

a)

Erstellen Sie eine Ausführungskondition, die mit einer bestimmten Annotation (@DisableOnTime) und Angabe eines Zeitraums (@DisableOnTime(from=„12:00“, until=„12:10“)) versehene Testmethoden disabled.

Es ist bereits eine entsprechende Test-Klasse vorbereitet mit zwei Methoden.

b)

Verknüpfen Sie die soeben erstellte Ausführungskondition mit der Environment-Eigenschaft „ENV“, so dass die markierten Tests nur in einem bestimmten Environment nicht ausgeführt werden.
(beispielsweise auf einem build-Server)

Hinweis:

So etwas kann sinnvoll sein, wenn es beispielsweise bei Integrationstest Zeiten gibt, in denen bestimmte Services (Datenbanken, etc) entweder nicht verfügbar sind, oder durch andere Tests stark blockiert sind.



Übung 07.2

→ Resourcen

KEINE

Erstellen Sie die Lösungsklassen im Package
de.hegmanns.training.junit5.practice.task07

Erstellen Sie eine Ausführungskondition, die abhängig vom Methodenname eine Methode nicht ausführt.
Sofern entweder im Anzeigename oder im Methodenname der Wortbestandteil „long“ vorkommt,
soll die Methode in Abhängigkeit des System-Property „no.long.tests=true“ gefiltert werden.

