



JUNIT

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

Allgemeines zu Tests
JUnit5

Test-Struktur
Wichtige Annotationen

Ausführung managen
Vergleiche/Zusicherungen

Behavior Driven Design
Test-Driven-Design

Parametrisierte Tests
TestFactory

Hook-Methoden
Erweiterungen

Proxy-Objekte
Mocks

Spezielle
Testanwendungen



A photograph of three brown bears swimming in a river. The bears are in the water, with their heads and shoulders visible. They appear to be interacting, with one bear on the left, one in the middle, and one on the right. The water is a murky brown color. In the background, there is a riverbank with some green plants and dry sticks.

Gibt's noch was?

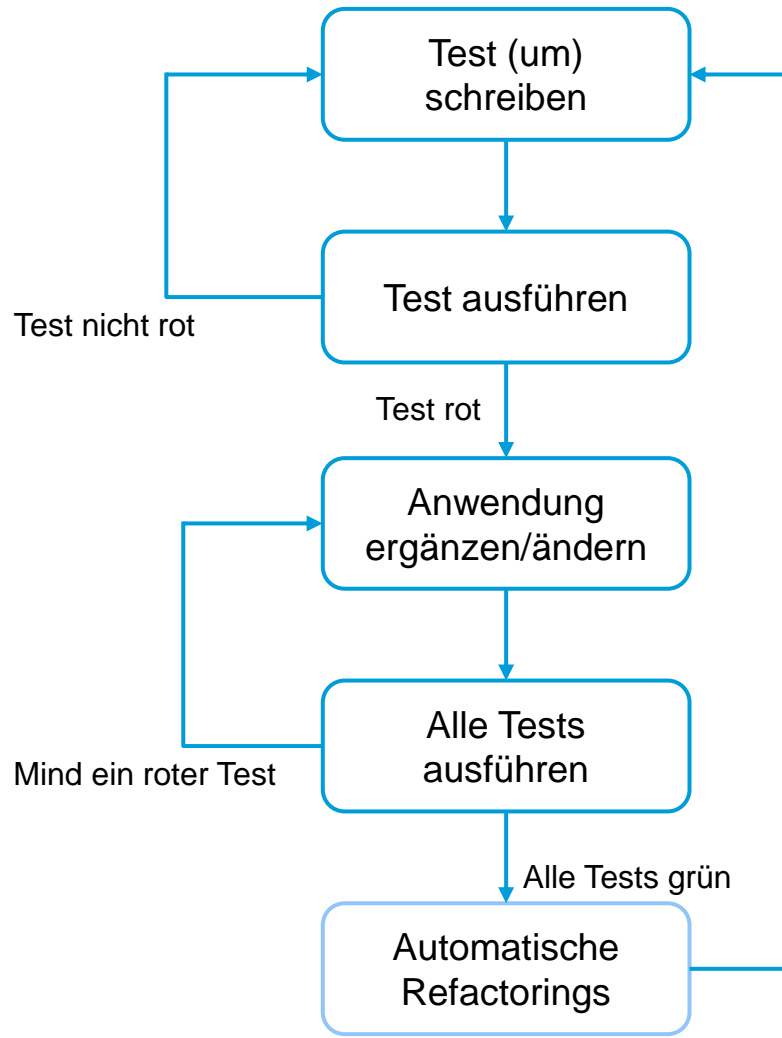
Unterrichtszeiten? (heute, morgen, überhaupt)

Pausenzeiten? (Raucher unter Ihnen?)

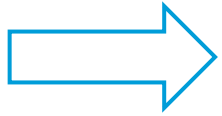
Toilette ... Snacks ...

...

Grundprinzip von TDD

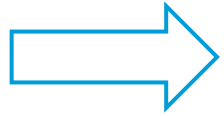






Ein leerer String gibt die Zahl 0 zurück



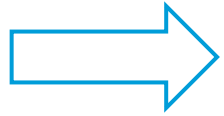


Ein leerer String gibt die Zahl 0 zurück



Ein String mit nur einer Zahl gibt diese Zahl zurück





Ein leerer String gibt die Zahl 0 zurück

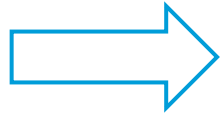


Ein String mit nur einer Zahl gibt diese Zahl zurück

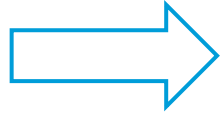


Ein String mit zwei Komma-separierten Zahlen gibt die Summe dieser Zahlen zurück





Ein leerer String gibt die Zahl 0 zurück



Ein String mit nur einer Zahl gibt diese Zahl zurück

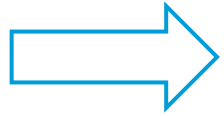


Ein String mit zwei Komma-separierten Zahlen gibt die Summe dieser Zahlen zurück



Ein String mit zwei Zahlen in jeweils einer separaten Zeile gibt die Summe der Zahlen zurück





Ein leerer String gibt die Zahl 0 zurück



Ein String mit nur einer Zahl gibt diese Zahl zurück



Ein String mit zwei Komma-separierten Zahlen gibt die Summe dieser Zahlen zurück



Ein String mit zwei Zahlen in jeweils einer separaten Zeile gibt die Summe der Zahlen zurück



Ein String mit drei Zahlen jeweils separiert wie c) und d) gibt die Summe dieser Zahlen zurück





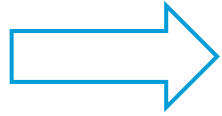
Ein leerer String gibt die Zahl 0 zurück



Ein String mit nur einer Zahl gibt diese Zahl zurück



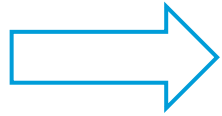
Ein String mit zwei Komma-separierten Zahlen gibt die Summe dieser Zahlen zurück



Ein String mit zwei Zahlen in jeweils einer separaten Zeile gibt die Summe der Zahlen zurück



Ein String mit drei Zahlen jeweils separiert wie c) und d) gibt die Summe dieser Zahlen zurück



Ein String mit mindestens einer negativen Zahl wirft eine Exception



Ein leerer String gibt die Zahl 0 zurück

Ein String mit nur einer Zahl gibt diese Zahl zurück

Ein String mit zwei Komma-separierten Zahlen gibt die Summe dieser Zahlen zurück

Ein String mit zwei Zahlen in jeweils einer separaten Zeile gibt die Summe der Zahlen zurück

Ein String mit drei Zahlen jeweils separiert wie c) und d) gibt die Summe dieser Zahlen zurück

Ein String mit mindestens einer negativen Zahl wirft eine Exception

Eine Zahl größer 1000 wird ignoriert
Was bedeutet dies? Muss ggf. genauer formuliert werden.
Hier: Eine einzelne Zahl größer 1000 zählt als Summand 0



Katas

- › Kleine wiederkehrende Übungen
 - Zur Perfektionssteigerung
 - Sicherung von Handgriffen
- › Teilnehmer
 - Alleine
 - In Gruppe
- › Falls Kata mehrschrittig ist:
 - Immer nur den jeweiligen Schritt lesen und durchführen
- › Im nur volle Konzentration auf den aktuellen Schritt bzw. die jeweilige Aufgabe
- › Ein Mitglied kommt nach vorne
 - Entweder auf Zeit
 - Oder für eine Teilaufgabe
- › Arbeit an dem Kata / dem Kata-Schritt
 - Beschreibt seine Arbeit
 - Wenn man nicht mehr weiter kommt, kann man sich helfen lassen
 - Nach Fertigstellung / Zum Ende wird das Ergebnis diskutiert

Quelle für Katas und Übungen:

<https://codeforces.org/>

- › Viele Übungen, meist mit dem Hintergrund der Erstellung optimierten Codes
 - Ziel ist die Erzeugung effizienter Algorithmen

Weitere Quelle für Katas: <https://adventofcode.com/>

- › Ein wenig Zeitvertreib zu jeder Jahreszeit
 - › <https://adventofcode.com/2015>
 - › <https://adventofcode.com/2016>
 - › <https://adventofcode.com/2017>
 - › <https://adventofcode.com/2018>
 - › <https://adventofcode.com/2019>
-
- › Es sind keine JAVA-Katas, aber Aufgaben, die mit Hilfe kleiner Programmen gelöst werden können



Übung 15

› Ressourcen

de.hegmanns.training.junit5.practice15

Entwickelt Sie den Kennwortchecker für „gute“ Kennwörter:

Ein "gutes" Kennwort liegt dann vor, wenn der Passwortchecker keinen Fehler wirft.

Folgende Bedingungen müssen für ein "gutes" Kennwort vorliegen:

- Kennwort muss aus mehr als 8 Zeichen bestehen
- Kennwort muss aus mindestens einem groß geschriebenen Buchstaben bestehen
- Kennwort muss aus mindestens einem klein geschriebenen Buchstaben bestehen
- Kennwort muss aus mindestens einer Ziffer bestehen
- Kennwort muss aus mindestens einem der folgenden Zeichen bestehen: Leerzeichen, Komma, Punkt, Semikolon, Ausrufezeichen
- Falls nach einem groß geschriebenen Buchstaben noch ein Buchstabe kommt, muss dieser Buchstabe klein geschrieben sein
- Wenn nur die Buchstaben (egal ob klein oder groß) aus dem Kennwort extrahiert werden, darf Nicht das Wort "java" herauskommen.
(Klein- und Großschreibung soll hier ignoriert werden)



Übung 16

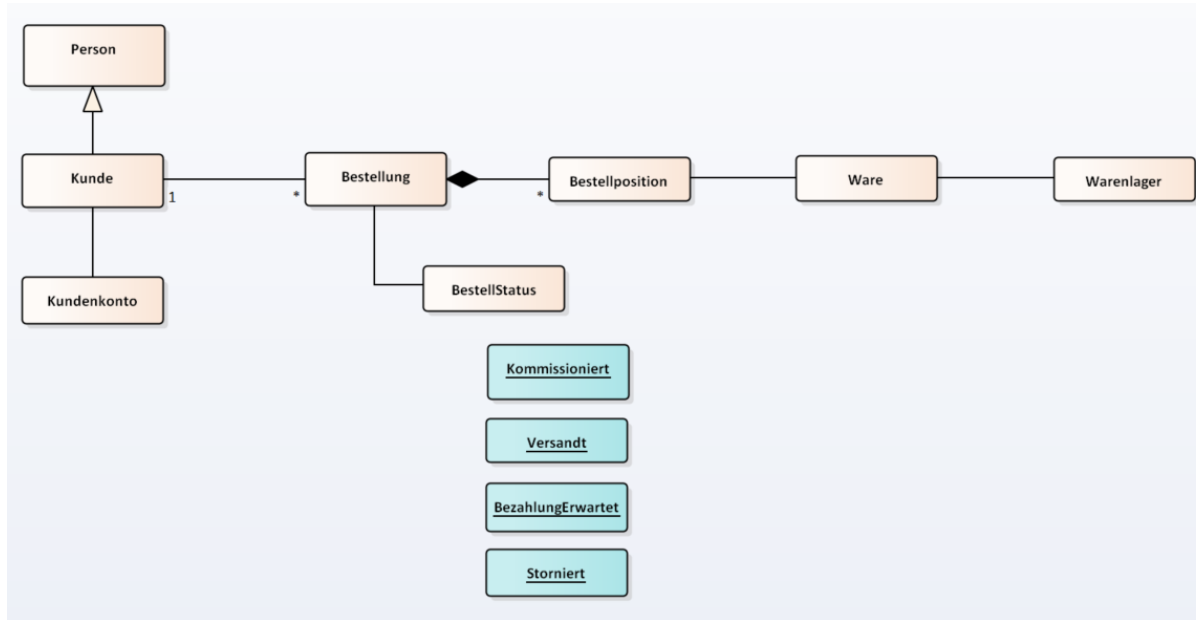
› Ressourcen

[de.hegmanns.training.junit5.practice16](https://de.hegmanns.training/junit5/practice16)

Erstellen Sie einen Währungsrechner.

Die Übung sollen Sie im Plenum fertig stellen. Im Rahmen der gemeinsamen Übung werden wir an wichtigen Stellen halten, um das ein und Andere Thema zu vertiefen.

Fokus im UNIT-Test



› Lieferung nur nach Vorkasse

› Wie muss ein Junit-Test hier aussehen?

- › Anlage der Bestellung
- › Sicherstellung der nötigen Ware im Warenlager
- › Simulieren des Geldtransfers und Einzahlen auf Kundenkonto



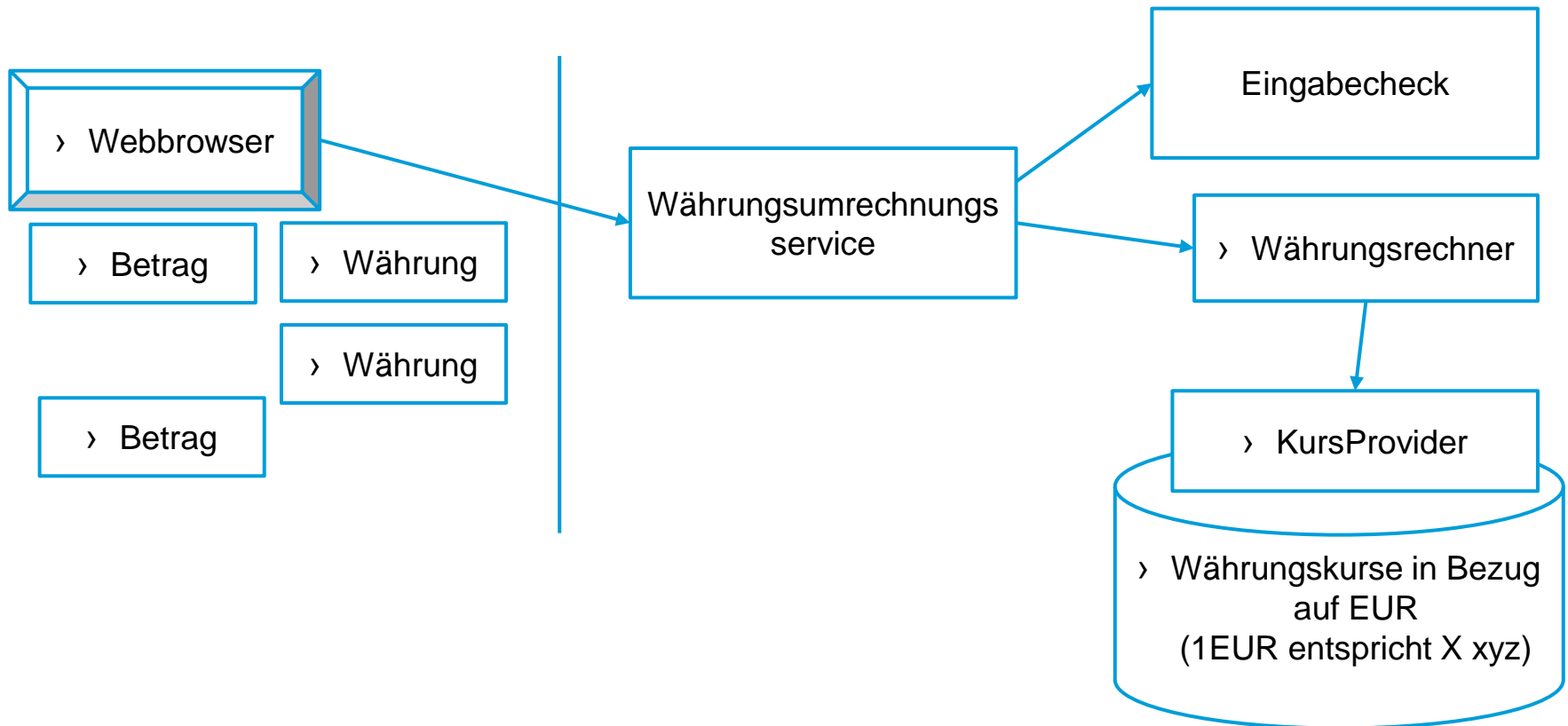
Übung 16.1

› Ressourcen

de.hegmanns.training.junit5.practice16

Erstellen Sie einen Währungsrechner.

Die Übung sollen Sie im Plenum fertig stellen. Im Rahmen der gemeinsamen Übung werden wir an wichtigen Stellen halten, um das ein und Andere Thema zu vertiefen.





› *Beispiel: Lagerlogistik*

- › Ein Ladungsträger wird erfasst, vermessen und dann im Hochregal platziert
- › Es wird zunächst die erste Bahn komplett befüllt, bevor die zweite Bahn befüllt wird
- › Wie sieht ein sinnvoller Unit-Test aus, zur Kontrolle, ob ein Ladungsträger auch wirklich zur zweiten Bahn adressiert wird?
 - › Müssen zunächst x Ladungsträger erfasst werden (x = Kapazität der ersten Bahn), um dann zu kontrollieren, ob der $x+1$ -te Ladungsträger zur zweiten Bahn gefahren wird?


› *Beispiel: Online-Kaufhaus*

- › Die ersten x Bestellungen müssen per Vorkasse beglichen werden.
- › Wie sieht ein sinnvoller Unit-Test aus, zur Kontrolle, dass nach x Bestellungen keine Vorkasse mehr nötig ist?
 - › Müssen zunächst x Bestellungen komplett abgewickelt werden, incl. Sicherstellung von genug Ware im Lage, Einbuchungen von Geld aufs Kundenkonto?

› *Beispiel: Online-Kaufhaus*

- › Der Kunde erhält nach 10 Bestellungen mit einem Bestellwert über 100EUR ein Warengutschein.
- › Müssen nun Bestellungen von Kunden erzeugt werden und komplett abgewickelt werden?

Isoliertes Testen ist angesagt

- 
- › Ausführung nur des nötigen bestimmenden Schritts (ohne vorherige Schritte)
 - › Alles Andere dauert ggf. zu lang
 - › Alles Andere kann recht kompliziert werden (Prozessketten sind mitunter voneinander abhängig)
 - › Verwenden nur des zu testenden Objekts, möglichst ohne weitere Objekte
 - › Alle andere kann ggf. komplex werden (wg. diverser Constraints)
 - › Die anderen Objekte können variierende Ergebnisse verursachen
 - › Möglichst allgemein ohne spezielle Konfiguration
 - › Weil Tests sonst für andere Konfigurationen angepasst werden müssen

Stellvertreter

- › Assoziierte Objekte / Komposite sollen gegen einfachere Objekte ausgetauscht werden
 - › Klares Verhalten zwecks einfachem Check der zu sichernden Komponente
 - › Unterdrücken für den speziellen Testfall unnötige Komplexität
 - › Durch langen Prozess
 - › Durch komplexen Prozess
 - › Durch komplexe Klassen-Struktur



Verwenden von Proxy-Objekten

›

Beispiele von Proxy-Objekten

- › Datenbank-Proxy (kann umfangreiche Datenbank ersparen)
 - › Gleiche Schnittstelle (z.B. Connection); Die Proxy-Klasse kontrolliert aber nur, ob ein bestimmtes SQL im Statement übergeben wurde
- › Andere (für die Testfall) unwichtige Domänenobjekte
- › Ein eigentlich verwendeter externer Service
 - › Ein externer Service steht nicht immer zur Verfügung
 - › Ein externer Service verursacht nicht immer das gleiche Ergebnis
 - › Ein Proxy kann gesichertes Verhalten erzeugen (auch seltene Fehlerfälle)

Simple Anwendungsbeispiele

- › Ein Begrüßungsservice, der abhängig von der Tageszeit den User mit „Guten Morgen“, „Guten Tag“, „Guten Abend“ begrüßt.
 - › Hier wäre ein Zeit-Proxy sehr gut, weil die echte aktuelle Uhrzeit die Ausführung aller Testcase zur gleichen Zeit verhindert
- › Verwendung eines weiteren Service
- › Unsichere Ausführungen externer Services
- › Testservices sind nicht vorhanden
- › Lange Ausführungszeiten

```
public class GreetService {  
  
    public static String sayHello() { return "hello" + LocalTime.now().getHour(); }  
  
    public String greet(String name) {  
        int hour = LocalTime.now().getHour();  
  
        String greetString = "";  
        if (hour < 6) {  
            greetString = "Good Night";  
        }else{  
            if (hour < 12) {  
                greetString = "Good Morning";  
            }else{  
                if (hour < 18) {  
                    greetString = "Good Day";  
                }else{  
                    if (hour < 21) {  
                        greetString = "Good Evening";  
                    }else{  
                        greetString = "Good Night";  
                    }  
                }  
            }  
        }  
        return greetString + ", " + name;  
    }  
}
```

› Benefit

› Verwendung eines Proxies ergibt häufig ein gutes Design

Testen unter Verwendung von Mocks: Fallstricke

- › Es muss das gesamte Spektrum des realen Objekts / Services abgebildet werden können
 - › Sonst wird nicht die Gesamtheit der möglichen Inputs/Verhaltens getestet
- › Es wird das Mock selbst getestet
 - › Es wird sich im Test quasi im Kreis gedreht

Beispiele

- › Ein Service, der Werte mit einer String-Repräsentation addiert
 - › Der Check ist zwar wichtig, muss aber bei der eigentlichen Kontrolle der Korrektheit der Addition nicht mit kontrolliert werden
 - › Gleiches gilt für die String->Integer-Konvertierung

Vorteile

- › Test bezieht sich nur noch auf ein Detail
 - › Einzelnen Aspekte können isoliert getestet werden
- › Test gibt Hinweise auf zu starke Zuständigkeit

Mocks selbstgebastelt



Untestbarer Code

GreetService

- Direkt LocalTime ermitteln
- In Abhängigkeit Begrüßung ermitteln

- › Gar nicht / schlecht testbar
- › Lösung: Das Problem delegieren

GreetService

- In Abhängigkeit Begrüßung ermitteln

CurrentHourProvider

- Aktuelle Stunde ermitteln

TEST

TestCurrentHourProvider

- Konfigurieren der „aktuellen Stunde“

Mocks selbstgebastelt



Untestbarer Code

GreetService

- Direkt LocalTime ermitteln
- In Abhängigkeit Begrüßung ermitteln

- › Gar nicht / schlecht testbar
- › Lösung: Das Problem delegieren

GreetService

- In Abhängigkeit Begrüßung ermitteln

TEST

```
public class GreetService01 {  
  
    public String greet(String name) {  
        int hour = LocalTime.now().getHour();  
        if (hourForTestcase != null) {  
            hour = hourForTestcase;  
        }  
        String greetString = "";  
        if (hour < 6) {  
            greetString = "Good Night";  
        } else {  
            if (hour < 12) {  
                greetString = "Good Morning";  
            } else {  
                if (hour < 18) {  
                    greetString = "Good Day";  
                } else {  
                    if (hour < 21) {  
                        greetString = "Good Evening";  
                    } else {  
                        greetString = "Good Night";  
                    }  
                }  
            }  
        }  
        return greetString + ", " + name;  
    }  
  
    private Integer hourForTestcase;  
    public void setHourForTestcase(int hour) { hourForTestcase = hour; }  
  
    public void resetTestcase() { hourForTestcase = null; }  
}
```

Mocks selbstgebastelt



Untestbarer Code

- › Bei einfachen Konstellationen problemfrei möglich
- › Je komplexer das Verhalten wird, desto schwieriger
- › Beispiele
 - › Mock einer DB-Verbindung
 - › Mock eines Restful-Service

GreetService

- In Abhängigkeit Begrüßung ermitteln

CurrentHourProvider

- Aktuelle Stunde ermitteln

TEST

TestCurrentHourProvider

- Konfigurieren der „aktuellen Stunde“

Mocks über Frameworks

- › Frameworks bieten quasi Mocks von der Stange
- › Leichte Konfigurierbarkeit
- › Integration in Testframework (Jupiter)
 - › Verschiedene Varianten der Integration
- › Integration/Kompatibilität mit anderen Frameworks (AssertJ, Hamcrest)
- › Mock-Frameworks bieten:
 - › Unterstützung in schwer testbaren Konstellationen
 - › Erstellung von Delegates als Mock
 - › Unterstützung für White-Box-Tests
 - › Zusicherung von Verhalten (z.B. Ausführung konkreter Methoden)
 - › Untersuchung von internem Verhalten
 - › Zusicherung von Calls
- › Mockito
- › EasyMock
- › PowerMock
- › JMock

- › Direkte Mocks
 - › Vordefiniertem Verhalten (Defaultverhalten)
 - › Über Annotation @Mock
 - › DI in andere Objekte
- › Static Methoden
- › Spying
 - › Automatisiert von Mocks
 - › Für andere Instanzen hinzufügbare

Mockito: Mit Mock arbeiten

Mock erzeugen

TYPE `yourMock` = Mockito.mock(TYPE.class);

Mock konfigurieren

- › Definition von Output/Verhalten bei Methodenaufruf
 - › Es werden alle Methoden beschrieben, die vom Basisobjekt im Rahmen des Tests ausgeführt werden

Mockito.when(`yourMock.method(...)`).thenReturn(`result`);

```
HourProvider hourProvider = Mockito.mock(HourProvider.class);  
Mockito.when(hourProvider.getCurrentHour()).thenReturn(6);
```

Mock verwenden (lassen)

- › Mock in das Basisobjekt hinein konfigurieren
 - › Methode
 - › Attribut über setter
 - › Attribut über Konstruktor

```
GreetService03 greetService = new GreetService03();  
greetService.setHourProvider(hourProvider);
```

Mockito: Konfiguration der Parameter von Methodenausführungen

Konkrete Werte

Mockito.when(**yourMock.method(...)**).thenReturn(**result**);

```
GreetService greet = Mockito.mock(GreetService.class);
Mockito.when(greet.greet(name: "Bernd")).thenReturn("hello Hegi");
Mockito.when(greet.greet(name: "Franziska")).thenReturn("hello Franzi");
```

Vergleiche/Komparator

Mockito.when(**yourMock.method(Mockito.any...())**).thenReturn(**result**);

› Mockito besitzt umfangreiche Komparatoren zur Abfrage

```
Mockito.when(greet.greet(Mockito.anyString())).thenReturn("hello anybody");
Mockito.when(greet.greet(Mockito.contains("er"))).thenReturn("hey, are you Bernd?");
```


Mockito: erweiterte Konfiguration

- › Mit einem Namen belegen
- › Mit Default-Verhalten belegen
 - › Ideal für Klassen (an Stelle von reinen Interfaces) als Mock-Basis
 - › Aber auch in anderen Situationen können allgemeine Rückgabewerte definiert werden
- › Erweiterte Konfigurationen

Default/Antwort-Verhalten



- › RETURNS MOCKS
- › RETURNS_DEEP_STUBS
- › RETURNS_SMART_NULLS
- › CALLS_REAL_METHODS
- › RETURNS_SELF
- › RETURNS_DEFAULTS

Mockito: Noch einfacher mit der Mockito-Extension

- › Mock über Annotation definierbar
- › Basisobjekt auch über Annotation definierbar
 - › Automatische Injizierung aller nötigen Mocks
 - › Dies geht auch für private Attribute

```
@ExtendWith(MockitoExtension.class)
public class GreetService03Test {

    @Mock
    HourProvider hourProvider;

    @InjectMocks
    GreetService03 greetService;
}
```

```
@Test
public void sixHourResultsInGoodMorning_handling02() {
    shouldBeTimeAtHour(6);

    String greet = greetService.greet(name: "Bernd");
    MatcherAssert.assertThat(greet, Matchers.is(value: "Good Morning, Bernd"));
}

private void shouldBeTimeAtHour(int hour) {
    Mockito.when(hourProvider.getCurrentHour()).then((d) -> {
        try{Thread.sleep(1: 2000);}catch(Exception e){}
        return hour;
    });
}
```

Mockito: Spy-Funktion

- › Zusicherung von Service-Aufrufen
 - › Verwendung wird aufgezeichnet
 - › Verwendung kann nachträglich kontrolliert werden

Aufnahme in Aufzeichnung

TYPE `yourSpy` = Mockito.spy(instance);

HINWEIS: Für Mocks ist Spy automatisch vorgenommen ;)

Zusicherung der Verwendung

Mockito.verify(`spy`, `MODE`).`method(...)`;

- › Mode: Aussage über die Häufigkeit
 - › Mockito.never()
 - › Mockito.times(...)



Übung 17

› Ressourcen

`de.hegmanns.training.junit5.practice.task17`

Wir wollen uns wieder mit dem BoundedCounter beschäftigen.

Der BoundedCounter besitzt ja obere und untere Grenzen.

Nun müsste – so wie ja in einigen Tests schon realisiert – zwecks Test, ob der BoundedCounter nicht über/unter die Grenzen zählt, dieser zunächst mittels `increment()/decrement()` in die Nähe der Grenzen gebracht werden ; oder der BoundedCounter direkt entsprechend initialisiert werden.

a)

Da aber Seiteneffekte nicht ausgeschlossen werden können, soll der BoundedCounter so umgestellt werden, dass wesentliche Teile simpel durch einen Mock ausgetauscht werden können, um beweisen zu können, dass der BoundedCounter wirklich nicht seine Grenzen unter/überschreitet.

b)

Stellen Sie zudem sicher, dass im Fall des bereits am „Anschlag“ stehenden BoundedCounter auch nicht mehr `incrementiert` wird.

Stellen Sie umgekehrt in einem zweiten Test sicher, dass andernfalls immer die `incrementiere`-Methode ausgeführt wird.



Übung 16.2

› Ressourcen

`de.hegmanns.training.junit5.practice16`

Entwickeln Sie den Währungsrechner weiter im Plenum.

Testabdeckung: glaube keiner Abdeckung, die du nicht selbst ...

- › Testabdeckung als Wert kann trügerisch sein
- › Grundsätzlich aber ein möglicher Indikator
- › Sinnvoll könnte ein Verlauf sein
 - › Z.B. niemals kleiner als letzter Commit



Tests auditen

- › Erstellen Sie sich ggf. einen Styleguide
- › Notieren Sie sich (ggf. im Team) (DoD), auf welche Testmerkmale Sie über einen bestimmten Zeitraum achten wollen
 - › Beispiele
 - › KEINE Integrationstests
 - › Namensgebung (sowohl Testmethoden, als auch Variablen)
 - › Stete Verwendung eines konkreten Vergleichsframeworks
 - › Mindestens ein Refactoring pro Issue
- › Schauen Sie auf die Testabdeckung
 - › Unterwerfen Sie sich ihr aber nicht ...
- › Reduktion der Ausführungszeit um ... Minuten
- › Mindestanzahl Tests pro Feature/Issue
- › Wurde nach TDD-Prinzipien entwickelt

Unit-Test als täglicher Begleiter während der Entwicklung

- › Gehen Sie nach den TDD-Regeln vor
 - › Machen Sie sich einen Fahrplan
 - › Halten Sie sich streng an die zu realisierenden Features
- › Bauen Sie Refactoring-Runden für den Applikationscode ein
 - › Diese ergeben sich häufig
 - › Halten Sie Ihre Klassen testbar
 - › Gut testbare Klassen haben auch ein gutes Design
- › Bauen Sie Refactoring-Runden für den Testcode ein
 - › Achten Sie auf gute Klassennamen, Methodennamen
 - › Achten Sie auf Einfachheit
 - › Verwenden Sie möglichst wenige integrative Tests
 - › Ganz kann auf integrative Tests häufig nicht verzichtet werden
 - › Erstellen Sie für Ihre Arbeit ein Testset zur möglichst häufigen Ausführung
 - › Dennoch müssen andere Tests auch regelmäßig ausgeführt werden
 - › Regel: Vor dem endgültigen commit/push immer alle Tests ausführen
 - › Regel: Während der Arbeit die beteiligten Testcase ausführen
 - › Regel: mindestens alle 10min alle nicht integrativen Tests ausführen
 - › Regel: mindestens alle 30min alle Tests ausführen

Ideen zur Unterstützung in JUnit-Tests

- › Testobjekte
 - › Typische Instanzen (die beispielsweise auch in einer Test-DB zu finden sind)
- › Objektgeneratoren/Factories
 - › Zur Erzeugung typischer fachlicher Konstellationen (ggf. noch über eine Fassade allgemein verfügbar)
- › Spezielle Mocks für spezielle fachliche/technische Situationen
 - › Z.B. defekte entfernte Schnittstelle
 - › Z.B. geregelter Nachbetrieb ohne aktive Schnittstelle
- › Für integrative Tests
 - › Herstellung bestimmter Konstellationen in der Datenbank
 - › Z.B. Kunde mit bestimmten Kontosaldo
 - › Z.B. volles Lager
 - › Z.B. gestörte Transportstrecke
- › Erstellung einfach nutzbarer, kompakter Zusicherungen/Assertions
 - › Ergebnis-Check eines Service-Aufrufs
 - › Zustands-Check

Unit-Test als täglicher Begleiter während der Entwicklung

- › Testen Sie nie spezielle fachliche Konfigurationen
 - › Bauplan, Topologie
- › Verzichten Sie weitestgehend auf integrative Tests
 - › Sie werden nicht komplett drauf verzichten können
 - › Verbinden Sie niemals integrative Tests mit fachlichen Tests

Unit-Test als täglicher Begleiter während Erweiterung

- › Gehen Sie nach den TDD-Regeln vor
 - › Machen Sie sich einen Fahrplan
- › Falls andere Tests unerwartet mit rot werden, korrigieren Sie diese niemals einfach durch Austauschen der Werte ohne Nachdenken
 - › Bei Tests, die erwartet rot werden, gilt das Gleiche

Unit-Test als täglicher Begleitung während Bugfixing

- › Versuchen Sie immer als erstes den Bug mit einem Test nachzuweisen
 - › Als roter Test
 - › Isolieren Sie weitestgehend auf die konkrete Klasse / Methode
- › Nutzen Sie ggf. automatisierte Refactoring-Möglichkeiten
 - › Bei nicht automatisierten Refactorings IMMER Tests schreiben/kontrollieren

JUNIT 4

JUNIT 5

Runner

TestRule

MethodRule

Extension, Callbacks

- › Zugriff/Benachrichtigung während des Lifecycle
- › Wird als Interceptor während der Testausführung
 - › AOP-Konzept für Tests
- › Initialisierung von Instanzen
 - › Transaktion starten
 - › Server starten
- › Wiederkehrende Aufräumarbeiten/Endaktionen
 - › Server beenden
 - › Commit/rollback einer Transaktion

Lifecycle

TestInstancePostProcessor

BeforeAllCallback

@BeforeAll

BeforeEachCallback

@BeforeEach

BeforeTestExecutionCallback

@Test

TestExecutionExceptionHandler

@AfterEach

AfterTestExecutionCallback

@AfterAll

AfterEachCallback

AfterAllCallback

- › In den Lifecycle kann per Callback/Handler eingeklinkt werden
- › Querschnittsfunktionen
 - › Laden einer Konfiguration (z.B. Spring)
 - › Initialisieren von Objekten (Mocks)
 - › Öffnen/Schließen einer Transaktion
 - › Loggen von Informationen
- › Zusätzliche Überwachungen/Zusicherungen
 - › Überwachen von App-Logs
- › Bereitstellen von Framework-Instanzen
 - › Contexte (ejb, ...)
 - › DB-Connection
 - › EJBs
 - › ...



Übung 18

› Ressourcen

`de.hegmanns.training.junit5.practice.task17`

Erstellen Sie eine Extension „TestSummary“, mit der die Dauer der einzelnen Methoden ausgegeben werden kann (in ms).

TIPP:


Schauen Sie sich hierzu das Interface `InvocationInterceptor` an.

- › Spring bietet umfangreiche Testfallunterstützung
 - › AOP-Konzepte
 - › Transaktionshandling für Tests
 - › ApplicationContext für Tests
 - › Stand-Alone-Contexte
 - › Contexte in Verbindung mit Container

Spring-Unterstützung

- › Extension: SpringExtension.class
- › Verwendung der Konfigurationen
 - › SpringBoot
 - › Eigene Konfiguration für Test
- › Bereitstellung aller gewohnten Services

Arquillian: Test auf dem EJB-Container

- › Akzeptanztests
 - › GUI-Tests
 - › Komponententests
 - › EJB
 - › JPA / Persistenz
- 
- › Arbeitet mit Junit-Frameworks zusammen
 - › Arbeitet mit JEE-Container
 - › Führt Deployment / Undeployment für Tests durch

Ende des Tages ...



Zeit für einen Rückblick

- › Tests
- › Testmetriken
- › Jupiter-Annotationen
- › Assertions

Zeit für einen Rückblick

- › Parametrisierte Tests
- › Test-Frameworks
- › Test Driven Design
- › Zusicherungen

Zeit für einen Rückblick


- › Mocks
- › Spezialfälle
 - › Spring
 - › Datenbanken
 - › Docker

Zeit für ein Resümee

- › 3 Tage
- › Mehr als 222 Folien
- › 19 Übungen

Zeit für ein Resümee

>?????



Im Norden sagt man „Tschüss“

Zeit zu geh 'n