

Test-Driven-Development

DI Dr. Martin Vasko

Test-Driven Development

01.02.2022

Vormittag

09:00 - 10:00: Grundlagen des Test-Driven Development

10:00 - 11:00: Designprinzipien von Testfällen

11:00 - 12:00: Hands-On Gruppenarbeit

Mittagspause

Test-Driven Development

01.02.2022

Nachmittag

13:00 - 14:00: Test-Driven Development mit Spring Boot

14:00 - 16:00: API Endpoints testen und entwickeln

16:00 - 17:00: Hands-On Gruppenarbeit

Test-Driven Development

02.02.2022

Vormittag

09:00 - 10:00: Mocking - aber mit Sinn!

10:00 - 11:00: BDD - Behavior Driven Design

11:00 - 12:00: Hands-On Gruppenarbeit

Mittagspause

Test-Driven Development

02.02.2022

Nachmittag

13:00 - 14:00: Endpoints testen mit Postman

14:00 - 16:00: Grundlagen des Security Testings

16:00 - 17:00: Hands-On Gruppenarbeit

Wer bin ich?

DI Dr. Martin Vasko

- An der TU Wien Informatik studiert
- Mehr als 10 Jahre in diversen Projekten als Software-Entwickler gearbeitet
- Banken, Versicherungen, Maut-Systeme, Ordinationssoftware, Sicherheitssoftware
- Bevorzugte Technologien:
React & TypeScript
(Frontend)
Java, Spring Boot
(Backend)



Wer sind Sie?

Bitte
abstimmen



Grundlagen des Test-Driven Development

“Warum soll ich Zeit in eine automatisierte Testinfrastruktur investieren?”

“Die Zeit die ich in die Wartung und Entwicklung der Testinfrastruktur stecke kann ich auch für Feature-Entwicklung nutzen”

Grundlagen des Test-Driven Development

Qualitätskriterien für eine Anwendung in Entwicklung und Produktion:

- Fehlerfrei
- Für alle Entwickler gut verständlich
- Hohe Produktivität der Entwickler gewährleisten
- Einfach um neue Features erweiterbar
- Langfristige Wartbarkeit sicherstellen

Grundlagen des Test-Driven Development

Zur Erreichung dieser Kriterien ist ein initiales Investment notwendig:

- Fehlerfrei
- Für alle Entwickler gut verständlich
- Hohe Produktivität der Entwickler gewährleisten
- Einfach um neue Features erweiterbar
- Langfristige Wartbarkeit sicherstellen
- Zerlegung und Transformation des Problems in ein Anwendungsdesign
- Auswahl der richtigen API
- Aufbau der Entwicklungsinfrastruktur
- Entwicklungsprozess um Test-Driven Development erweitern (inkl. Code-Reviews und Pair Programming)

Grundlagen des Test-Driven Development

- Dieses Investment zahlt sich erfahrungsgemäß relativ schnell aus
siehe dazu auch Martin Fowler:
[“Is high quality software worth the cost?”](#)
- Besonders automatisierte Test zahlen sich “schneller” aus, da damit manueller Aufwand eingespart wird

Rule of Ten

Je nach Projektphase kostet es das 10-fache Fehler in einer Anwendung zu finden und zu beheben:

- | | |
|--|-----------|
| - Fehler in Unit-Testphase: | 100 € |
| - Fehler in System-/Integrationstestphase: | 1.000 € |
| - Fehler in Akzeptanztestphase: | 10.000 € |
| - Fehler in Produktion: | 100.000 € |

Grundlagen des Test-Driven Development

Die grundlegende Idee des Test-Driven Development:

*Gegen Testfälle zu entwickeln, also zuvor definierte Testfälle von **rot** auf **grün** zu wechseln*

Aber was hindert uns im Projektalltag daran dieser Idee zu folgen und Teil des Arbeitsflusses zu machen?

Grundlagen des Test-Driven Development

(Viele) Antworten:

- Projektstress
- Nicht genügend Freiräume, neue Ideen weiter zu verfolgen
- Strenge Vorgaben innerhalb des Projekts lassen neue Ansätze nicht zu
- Umfangreiche Planungskonzepte (komplexer) Software sehen dieses Konzept nicht vor
- Umfangreiche Planung widerspricht der Natur des Entwicklers - eher eine Frau oder ein Mann der Tat

Grundlagen des Test-Driven Development

(Viele) Antworten:

- Projektstress
- ~~Nicht genügend Freiräume, neue Ideen weiter zu verfolgen~~
- ~~Strenge Vorgaben innerhalb des Projekts lassen neue Ansätze nicht zu~~
- ~~Umfangreiche Planungskonzepte (komplexer) Software sehen dieses Konzept nicht vor~~
- Umfangreiche Planung widerspricht der Natur des Entwicklers - eher eine Frau oder ein Mann der Tat

Grundlagen des Test-Driven Development

(Viele) Antworten:

- Projektstress
- ~~Nicht genügend Freiräume, neue Ideen weiter zu verfolgen~~
- ~~Strenge Vorgaben innerhalb des Projekts lassen neue Ansätze nicht zu~~
- ~~Umfangreiche Planungskonzepte (komplexer) Software sehen dieses Konzept nicht vor~~
- **Umfangreiche Planung widerspricht der Natur des Entwicklers - eher eine Frau oder ein Mann der Tat**

Grundlagen des Test-Driven Development

Das Beispiel:

1. Eine Anforderung wurde ausreichend spezifiziert und von der Projektleitung zur Umsetzung freigegeben
2. Es soll ein Email-Client mit Java Mail API entwickelt werden, der es der Gesamtanwendung ermöglicht Emails zu versenden
3. Grundlage des Beispiels ist das Open-Source Projekt TP-Core
<https://github.com/ElmarDott/TP-CORE>

Grundlagen des Test-Driven Development

Die Grundfunktionen:

- Verbindung mit einem SMTP-Server zum Versenden der Emails
- Erstellen einer Email-Nachricht mit Betreff, Absender, Inhalt und möglichen Anhängen
- Einlesen des Empfängers
- Mail für den Versand vorbereiten

Hinweis 🙌

Die Funktionalität lässt sich beliebig erweitern - darauf werden wir im weiteren Verlauf angemessen reagieren

Grundlagen des Test-Driven Development

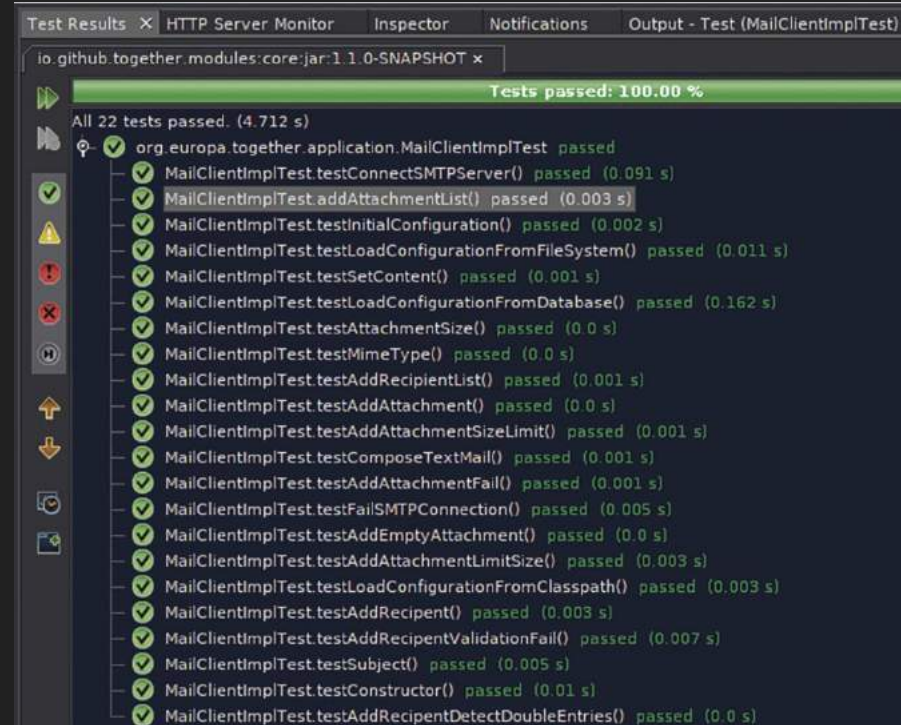
Die Randbedingungen:

- JUnit 5 für Unit-Tests
- JGiven für BDD-Testszenarios
- BeanMatchers zum Testen der Konstruktoren, etc.

Grundlagen des Test-Driven Development

Die Testarchitektur:

- Die Klasse [JavaMailClient.java](#) implementiert das Interface [MailClient.java](#)
- Sämtliche Testfälle sind in der Klasse [JavaMailClientTest.java](#) zusammengefasst



Grundlagen des Test-Driven Development

Wie so oft gilt:

KISS (Keep it simple, short)

Ist ein Testfall komplex und schlägt fehl, muss man ihn erst verstehen um ihn zu beheben zu können

```
@Test
void loadConfigurationFromDatabase() {
    LOGGER.log("TEST CASE: loadConfigurationFromDatabase", LogLevel.DEBUG);

    mailClient.populateDbConfiguration(SQL_FILE);
    assertTrue(mailClient.loadConfigurationFromDatabase());

    assertEquals("smtp.sample.org", mailClient.getDebugActiveConfiguration().get("mailer.host"));
    assertEquals("465", mailClient.getDebugActiveConfiguration().get("mailer.port"));
    assertEquals("send.from@mail.me", mailClient.getDebugActiveConfiguration().get("mailer.sender"));
    assertEquals("JohnDoe", mailClient.getDebugActiveConfiguration().get("mailer.user"));
    assertEquals("s3cr3t", mailClient.getDebugActiveConfiguration().get("mailer.password"));
    assertEquals("true", mailClient.getDebugActiveConfiguration().get("mailer.ssl"));
    assertEquals("true", mailClient.getDebugActiveConfiguration().get("mailer.tls"));
    assertEquals("false", mailClient.getDebugActiveConfiguration().get("mailer.debug"));
    assertEquals("1", mailClient.getDebugActiveConfiguration().get("mailer.count"));
    assertEquals("0", mailClient.getDebugActiveConfiguration().get("mailer.wait"));
}
```

Grundlagen des Test-Driven Development

Der logische Aufbau eines Tests:

Arrange (Vorbereiten)

Act (Ausführen)

Assert (Überprüfen)

Das AAA - Prinzip

```
@Test
void loadConfigurationFromDatabase() {
    LOGGER.log("TEST CASE: loadConfigurationFromDatabase", LogLevel.DEBUG);

    mailClient.populateDbConfiguration(SQL_FILE);
    assertTrue(mailClient.loadConfigurationFromDatabase());

    assertEquals("smtp.sample.org", mailClient.getDebugActiveConfiguration().get("mailer.host"));
    assertEquals("465", mailClient.getDebugActiveConfiguration().get("mailer.port"));
    assertEquals("send.from@mail.me", mailClient.getDebugActiveConfiguration().get("mailer.sender"));
    assertEquals("JohnDoe", mailClient.getDebugActiveConfiguration().get("mailer.user"));
    assertEquals("s3cr3t", mailClient.getDebugActiveConfiguration().get("mailer.password"));
    assertEquals("true", mailClient.getDebugActiveConfiguration().get("mailer.ssl"));
    assertEquals("true", mailClient.getDebugActiveConfiguration().get("mailer.tls"));
    assertEquals("false", mailClient.getDebugActiveConfiguration().get("mailer.debug"));
    assertEquals("1", mailClient.getDebugActiveConfiguration().get("mailer.count"));
    assertEquals("0", mailClient.getDebugActiveConfiguration().get("mailer.wait"));
}
```

Grundlagen des Test-Driven Development

Wichtig sind auch “false-positive” Konstellationen:

Hier zB. failConnectSMTPServer()

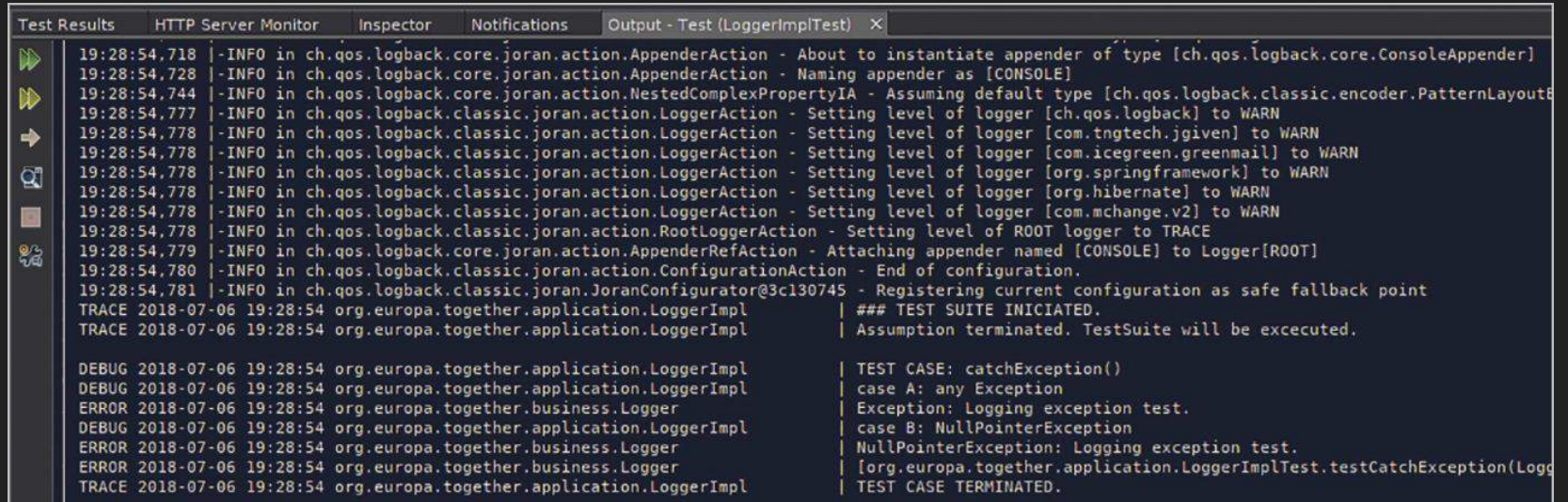
```
@Test
void failConnetSMTPServer() throws Exception {
    LOGGER.log("TEST CASE: failConnectSMTPServer", LogLevel.DEBUG);

    mailClient.clearConfiguration();
    assertThrows(Exception.class, () -> {
        mailClient.getSession();
    });
}
```

Grundlagen des Test-Driven Development

Logging:

In der Klasse [LogbackLoggerTest.java](#) wird der Logging-Wrapper selbst getestet



The screenshot shows the 'Test Results' window of an IDE, specifically the 'Output - Test (LoggerImplTest)' tab. The window displays a series of log messages and test execution details. The logs include information about configuring Logback, such as instantiating appenders, setting levels for various loggers (WARN, TRACE), and attaching the console appender. The test execution part shows the start of a test suite, followed by a test case 'catchException()' which includes sub-cases for 'any Exception' and 'NullPointerException'. The test suite ends with 'Assumption terminated. TestSuite will be executed.' and 'TEST CASE TERMINATED.'

```
Test Results HTTP Server Monitor Inspector Notifications Output - Test (LoggerImplTest) X
19:28:54,718 |-INFO in ch.qos.logback.core.joran.action.AppenderAction - About to instantiate appender of type [ch.qos.logback.core.ConsoleAppender]
19:28:54,728 |-INFO in ch.qos.logback.core.joran.action.AppenderAction - Naming appender as [CONSOLE]
19:28:54,744 |-INFO in ch.qos.logback.core.joran.action.NestedComplexPropertyIA - Assuming default type [ch.qos.logback.classic.encoder.PatternLayoutEncoder]
19:28:54,777 |-INFO in ch.qos.logback.classic.joran.action.LoggerAction - Setting level of logger [ch.qos.logback] to WARN
19:28:54,778 |-INFO in ch.qos.logback.classic.joran.action.LoggerAction - Setting level of logger [com.tngtech.jgiven] to WARN
19:28:54,778 |-INFO in ch.qos.logback.classic.joran.action.LoggerAction - Setting level of logger [com.icegreen.greenmail] to WARN
19:28:54,778 |-INFO in ch.qos.logback.classic.joran.action.LoggerAction - Setting level of logger [org.springframework] to WARN
19:28:54,778 |-INFO in ch.qos.logback.classic.joran.action.LoggerAction - Setting level of logger [org.hibernate] to WARN
19:28:54,778 |-INFO in ch.qos.logback.classic.joran.action.LoggerAction - Setting level of logger [com.mchange.v2] to WARN
19:28:54,778 |-INFO in ch.qos.logback.classic.joran.action.RootLoggerAction - Setting level of ROOT logger to TRACE
19:28:54,779 |-INFO in ch.qos.logback.core.joran.action.AppenderRefAction - Attaching appender named [CONSOLE] to Logger[ROOT]
19:28:54,780 |-INFO in ch.qos.logback.classic.joran.action.ConfigurationAction - End of configuration.
19:28:54,781 |-INFO in ch.qos.logback.classic.joran.JoranConfigurator@3c130745 - Registering current configuration as safe fallback point
TRACE 2018-07-06 19:28:54 org.europa.together.application.LoggerImpl | ### TEST SUITE INITIATED.
TRACE 2018-07-06 19:28:54 org.europa.together.application.LoggerImpl | Assumption terminated. TestSuite will be executed.

DEBUG 2018-07-06 19:28:54 org.europa.together.application.LoggerImpl | TEST CASE: catchException()
DEBUG 2018-07-06 19:28:54 org.europa.together.application.LoggerImpl | case A: any Exception
ERROR 2018-07-06 19:28:54 org.europa.together.business.Logger | Exception: Logging exception test.
DEBUG 2018-07-06 19:28:54 org.europa.together.application.LoggerImpl | case B: NullPointerException
ERROR 2018-07-06 19:28:54 org.europa.together.business.Logger | NullPointerException: Logging exception test.
ERROR 2018-07-06 19:28:54 org.europa.together.business.Logger | [org.europa.together.application.LoggerImplTest.testCatchException(Logg
TRACE 2018-07-06 19:28:54 org.europa.together.application.LoggerImpl | TEST CASE TERMINATED.
```

Designprinzipien von Testfällen

Hinweis 🙌

Der Rahmen für sinnvolle Testfälle muss schon beim Design der Implementierung definiert werden

- Immer nur einen Methodeneintrittspunkt und einen Austrittspunkt festlegen (wie oft wird return verwendet?)
- Methoden ohne Rückgabewert erschweren das Testdesign => sind nur dann testbar wenn Auswirkungen auf Objekte oder Aktionen gesetzt werden

Achtung 💥

Eines der zentralen Paradigmen Funktionaler Programmierung ist es "Seiteneffekte" zu minimieren.

Designprinzipien von Testfällen

Reporting

- Tiefgreifende Erkenntnisse über das Design der Applikation gewinnt man nur im direkten Umgang mit Code
- Aussagekraft der Testfälle kann schnell und einfach über Reports erlangt werden
 - Aus den Schwächen die durch Reports sichtbar werden lassen sich Maßnahmen zur Verbesserung in einem iterativen Vorgehen ableiten

Designprinzipien von Testfällen

Beispiel: [BeanMatcher](#)

- Mit wenigen Zeilen lassen sich einfach
Getter, Setter, toString
oder Konstruktoren testen

```
@Test
void constructor() {
    LOGGER.log("TEST CASE: constructor", LogLevel.DEBUG);

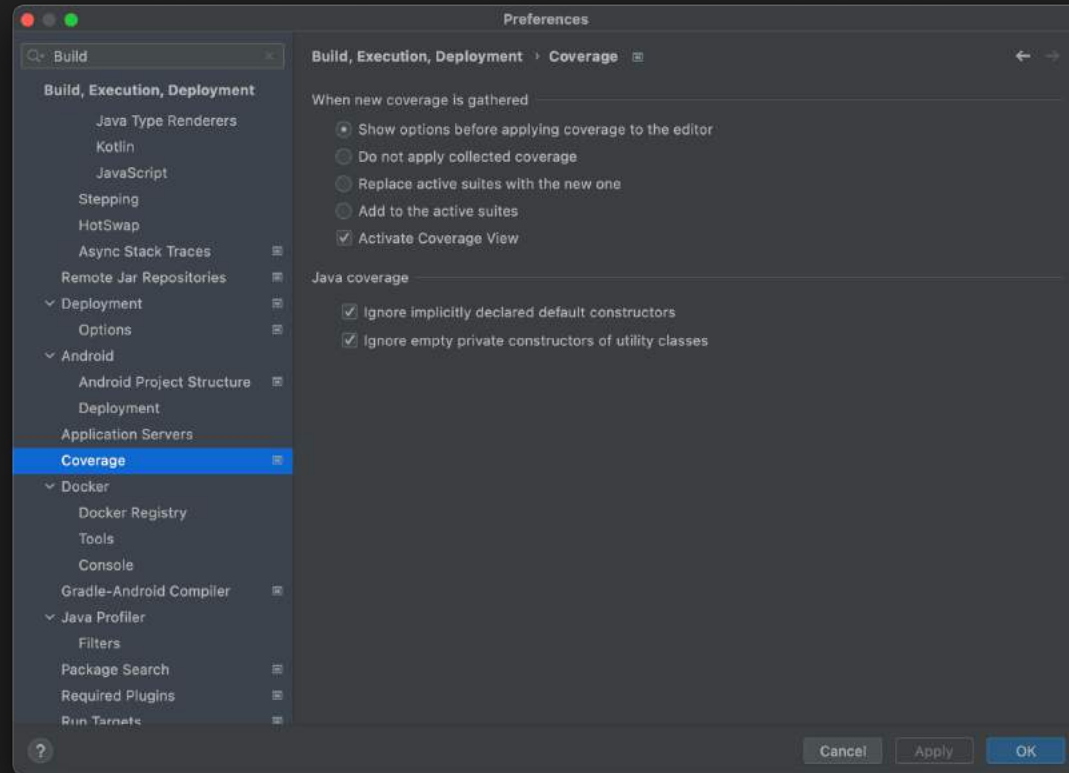
    assertThat(JavaMailClient.class, hasValidBeanConstructor());
}
```

Designprinzipien von Testfällen

Beispiel:

Code Coverage IntelliJ

- + Bereits in IntelliJ integriert
- + Kann als Webseite generiert werden
- + Integriert sich gut in Coverage Suites (zB JaCoCo)



Designprinzipien von Testfällen

Beispiel:

Code Coverage IntelliJ

Coverage: PalindromeTest.whenEmptyString_thenAccept × ⚙ —

↑ 50% classes, 42% lines covered in 'all classes in scope'

⌵

| Element | Class, % | Method, % | Line, % |
|----------|-----------|------------|-----------|
| 📁 at.etc | 50% (1/2) | 100% (1/1) | 42% (3/7) |
| | | | |

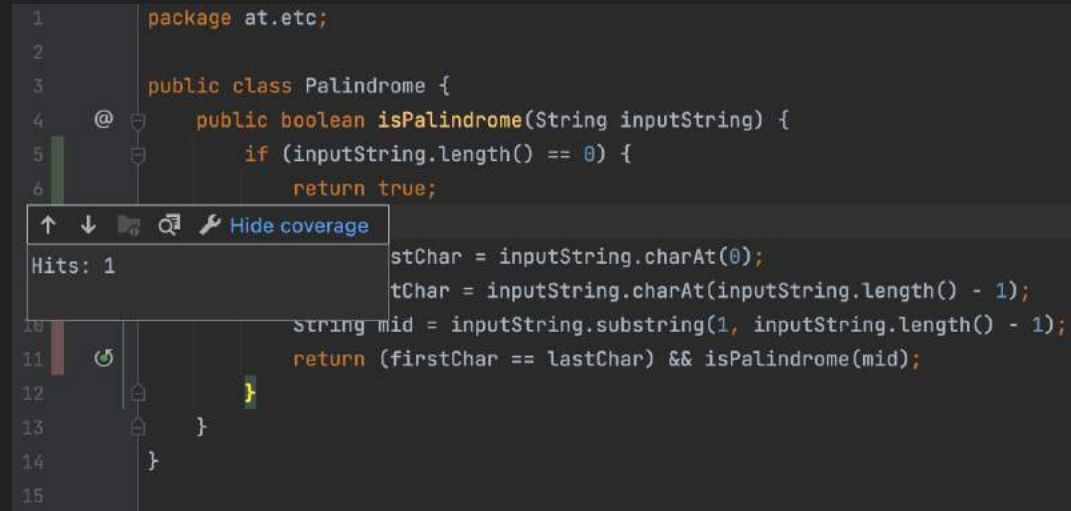
⌵

Designprinzipien von Testfällen

Beispiel:

Code Coverage IntelliJ

- Grün: Zeilen wurden während der Simulation ausgeführt
- Rot: Zeilen wurden nicht während der Simulation ausgeführt
- Gelb: Zeilen sind vom *Conditions Tracing Modus* abgedeckt



```
1 package at.etc;
2
3 public class Palindrome {
4     @ public boolean isPalindrome(String inputString) {
5         if (inputString.length() == 0) {
6             return true;
7
8         stChar = inputString.charAt(0);
9         tChar = inputString.charAt(inputString.length() - 1);
10        String mid = inputString.substring(1, inputString.length() - 1);
11        return (firstChar == lastChar) && isPalindrome(mid);
12    }
13 }
14 }
15
```

↑ ↓ 🔍 ⚙️ Hide coverage

Hits: 1

Designprinzipien von Testfällen

BDD - Behavior Driven Development:

- BDD Testfälle operieren eine Ebene über den Unit Testfällen
- Am Beispiel von TP-Core:
 - Die App ist in einer Schichtenarchitektur aufgebaut
 - Das Package `business` stellt die funktionale API dar und enthält ausschließlich Interfaces
 - Das Package `application` umfasst sämtliche Implementierungsklassen zu den Interfaces aus `business`
 - `services` komponieren diese Implementierungsklassen und werden direkt vom GUI aufgerufen
 - Das bedeutet für unseren `MailClient`:
Die gesamte Funktionalität ist im Application-Layer zu finden
- und genau da setzt BDD an

Designprinzipien von Testfällen

BDD - Behavior Driven Development:

- Auf Basis der Unit-Tests können wir von einer hochwertigen Implementierung des Application-Layers ausgehen
- Jetzt muss sichergestellt werden, dass der gesamte Vorgang eine Email zu versenden ebenfalls möglich ist
- Dazu werden die folgenden Szenarien erstellt:
 - `sendSingleEmail()`
 - `updateDatabaseConfiguration()`

Designprinzipien von Testfällen

BDD - Behavior Driven Development:

- Auch hier setzen wir auf das AAA-Prinzip:
 - Im Kontext von JGiven werden Vorbedingungen - **Arrange** - als `given()` definiert
 - **Act** definiert in JGiven das `when()` und
 - das abschließende **Assert** prüft als Post-Condition in JGiven mit der `then()` Methode das Ergebnis

Insgesamt also:

Given, Action, Outcome

Designprinzipien von Testfällen

BDD - Behavior Driven Development:

- JGiven Szenarien basieren grundsätzlich auf JUnit
- Allerdings definieren die Szenarien bereits fachliche Use Cases

```
@Test
void scenario_sendSingleEmail() {
    Client client = new Client();
    // PreCondition
    given().email_get_configuration(client)
        .and().smtp_server_is_available()
        .and().email_has_recipient(client)
        .and().email_contains_attachment(client)
        .and().email_is_composed(client);

    // Invariant
    when().smtp_server_is_available()
        .and().send_email(client);

    //PostCondition
    then().email_is_arrived();
}
```

Designprinzipien von Testfällen

Lessons learned:

- Tests können (leider) nicht garantieren dass die gesamte Applikation fehlerfrei ist
- Als Entwickler kann man aber unterschiedliche Situationen (von granular Unit-Tests bis fachlich Szenarien) abbilden und simulieren
- Continuous Integration und Delivery stellen sicher dass Regressionen frühzeitig erkannt werden
- Zeit die zum Setup investiert wurde amortisiert sich im Laufe des Projekts rasch
- Für TDD ist keine umfassende Planung notwendig - das Konzept lässt sich sehr gut iterativ in bestehende und neue Projekte integrieren
- Entwickler werden “gezwungen” ihren Code selbst zu verwenden - in Testfällen

Hands-On Gruppenarbeit

1. Bilden Sie ein Team aus zwei bis drei TeilnehmerInnen
2. Konzeptionieren Sie Testfälle für eine Applikation aus Ihrem Projektalltag und beantworten Sie folgende Fragen:
 - a. Wie beurteilen Sie die Test-Coverage?
 - b. Welche fachlichen Use-Case Szenarien machen Sinn?
 - c. Konzeptionieren Sie diese fachlichen Use Cases in JGiven Szenarien

Test-Driven Development mit Spring Boot

- JUnit5 (and beyond)
- Unit Testen mit Spring Boot

JUnit 5 (and beyond)

JUnit 5 (and beyond)

Eine (ganz) kurze Einführung in Software Testen:

- Ein Software-Test führt eine Component aus und überprüft deren Verhalten auf unterschiedliche Ein- und Ausgaben
- Software-Tests werden typischerweise als Teil des Build-Prozesses automatisiert
- Eine hohe Test-Coverage erhöht das Auffinden von Regressionen
- “Don’t test the framework”

JUnit 5 (and beyond)

Eine (ganz) kurze Einführung in Software Testen - Begriffe:

- Die Klasse oder allgemein der Code der getestet wird: *code under test*
- Die Applikation die getestet wird: *application under test*
- Um dynamische Aspekte eines Objekts zu Isolieren, ist es manchmal notwendig den Objekt-State zu fixieren: *Test Fixture*
- **Unit Test:** Testet einen kleinen Codeteil - eine Methode oder Klasse.
Externe Abhängigkeiten sollten entfernt werden oder durch Mocks ersetzt werden

JUnit 5 (and beyond)

Eine (ganz) kurze Einführung in Software Testen - Begriffe:

- **Integration Test:** Testet das Verhalten einer Component oder die Integration einer Component mit anderen Components.
- **Performance Tests:** Benchmarken die Ausführung von Components kontinuierlich

JUnit 5 (and beyond)

Eine kurze Geschichte in Java Testen:

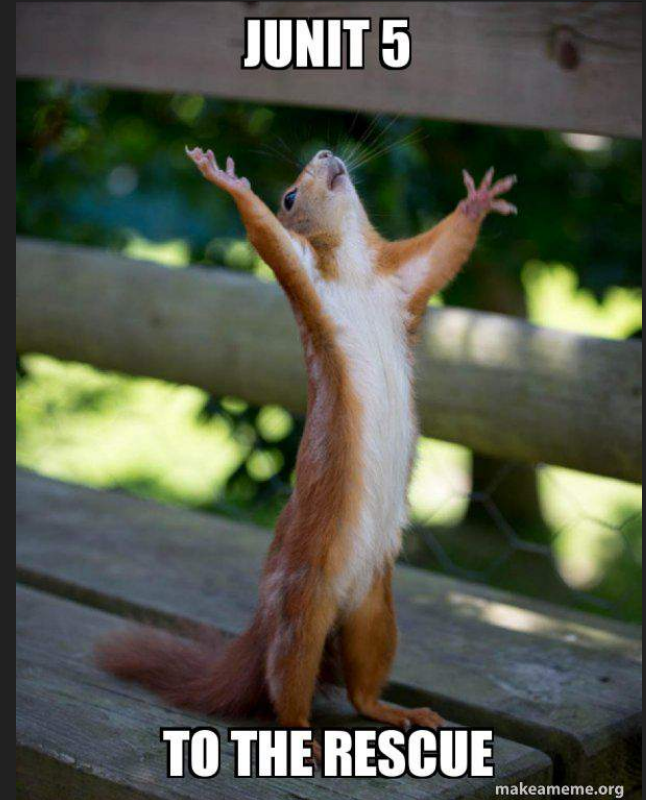
JUnit 5 ist eine Weiterentwicklung von JUnit 4

JUnit 4 ist/war eine gute Lösung, hatte aber im praktischen Einsatz folgende Limitierungen:

- ein einziges JAR
- nur ein TestRunner kann Tests ausführen
- JUnit 4 ist limitiert auf Java 7 und bietet (noch) keine Unterstützung für Java 8

JUnit 5 (and beyond)

- + JUnit 5 bietet mehr Granularität und ermöglicht damit den Import lediglich der benötigten Bibliotheken (und reduziert damit die Programmgröße)
- + JUnit 5 ermöglicht die Ausführung mehrerer TestRunner simultan (und erhöht damit die Test-Performance)
- + JUnit 5 unterstützt Java 8 Features



JUnit 5 (and beyond)

Clonen Sie das Beispiel-Repository:
<https://github.com/martinvasko/etc-tdd-2022>
und installieren Sie alle Abhängigkeiten

Übung: 15 Minuten

JUnit 5 (and beyond)

Ergänzen Sie folgende Methoden in der Klasse AppTest:

```
/**
 * Tear all things up
 */
@BeforeAll
public void setUp() {
    log.info("@BeforeAll - executes once before all test methods in this class");
}

/**
 * Tear all things up - before each test
 */
@BeforeEach
void init() {
    log.info("@BeforeEach - executes before each test method in this class");
}
```

Was passiert?

Übung: 5 Minuten

JUnit 5 (and beyond)

Der wahrscheinlich einfachste Test:

```
@Test
```

```
Run Test | Debug Test
```

```
public void shouldAnswerWithTrue() {  
    |    assertTrue(true);  
}
```

Übung: 5 Minuten

JUnit 5 (and beyond)

Passend dazu gibt es
natürlich auch `tearDown`
`@Annotations`

```
@AfterEach
void tearDown() {
    log.info("@AfterEach - executed after each test method.");
}

@AfterAll
static void done() {
    System.out.println("@AfterAll - executed after all test methods.");
}
```

JUnit 5 (and beyond)

JUnit 5 unterstützt auch Lambda-Expressions:

@Test

Run Test | Debug Test

```
void lambdaExpressions() {  
    assertTrue(Stream.of(1, 2, 3).mapToInt(i -> i).sum() > 5, () -> "Sum should be greater than 5");  
}
```

assertAll():

@Test

Run Test | Debug Test

```
void groupAssertions() {  
    int[] numbers = { 0, 1, 2, 3, 4 };  
    assertAll("numbers", () -> assertEquals(numbers[0], 0), () -> assertEquals(numbers[3], 3),  
        () -> assertEquals(numbers[4], 4));  
}
```

JUnit 5 (and beyond)

Assumptions in JUnit 5

Assumptions ermöglichen das Ausführen von Tests wenn bestimmte “Vorbedingungen” erfüllt sind.

@Test

Run Test | Debug Test

```
void trueAssumption() {  
    assertTrue(5 > 1);  
    assertEquals(5 + 2, 7);  
}
```

@Test

Run Test | Debug Test

```
void falseAssumption() {  
    assertFalse(5 < 1);  
    assertEquals(5 + 2, 7);  
}
```

@Test

Run Test | Debug Test

```
void assumptionThat() {  
    String someString = "Just a string";  
    assumingThat(someString.equals("Just a string"),  
        () -> assertEquals(2 + 2, 4));  
}
```


JUnit 5 (and beyond)

Exceptions Testen:

Mit `assertThrows()` kann getestet werden, ob eine cut (Class under test) eine Exception wirft - oder nicht.

```
@Test
```

```
Run Test | Debug Test
```

```
void shouldThrowException() {  
    Throwable exception = assertThrows(UnsupportedOperationException.class, () -> {  
        throw new UnsupportedOperationException("Not supported");  
    });  
    assertEquals(exception.getMessage(), "Not supported");  
}
```

```
@Test
```

```
Run Test | Debug Test
```

```
void assertThrowsException() {  
    String str = null;  
    assertThrows(IllegalArgumentException.class, () -> {  
        Integer.valueOf(str);  
    });  
}
```

JUnit 5 (and beyond)

1. Implementieren Sie eine einfache Klasse:
2. Erstellen Sie eine Test-Klasse
3. Testen Sie die Methode multiply mit geeigneten Werten

```
public class Calculator {  
  
    public int multiply(int a, int b) {  
        return a * b;  
    }  
}
```

JUnit 5 (and beyond)

JUnit 5 asserts erweitern mit Hamcrest

Hamcrest ist ein Assertion framework das JUnit um Matcher-Klassen erweitert

Zentral ist das `assertThat()` statement

```
import org.junit.jupiter.api.Test;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.equalTo;
import static org.hamcrest.Matchers.is;
import static org.hamcrest.Matchers.containsString;
import static org.hamcrest.Matchers.anyOf;
```

Run Test | Debug Test | ?

```
public class HamcrestSampleTest {
```

```
    @Test
```

Run Test | Debug Test | ✓

```
    void test() {
```

```
        boolean a = true;
```

```
        boolean b = true;
```

```
        assertThat(a, equalTo(b));
```

```
        assertThat(a, is(equalTo(b)));
```

```
        assertThat(a, is(b));
```

```
    }
```

```
    @Test
```

Run Test | Debug Test

```
    void extendedTest() {
```

```
        assertThat("test", anyOf(is("testing"), containsString("est")));
```

```
    }
```

```
}
```

JUnit 5 (and beyond)

JUnit 5 asserts erweitern mit Hamcrest

JUnit assert statements im Vergleich zu Hamcrest assert statements

```
// JUnit for equals check  
assertEquals(expected, actual);  
// Hamcrest for equals check  
assertThat(actual, is(equalTo(expected)));  
  
// JUnit for not equals check  
assertNotEquals(expected, actual)  
// Hamcrest for not equals check  
assertThat(actual, is(not(equalTo(expected))));
```

JUnit 5 (and beyond)

JUnit 5 erweitern mit Hamcrest

Hamcrest Matcher für Listen

```
@Test
```

```
Run Test | Debug Test
```

```
public void listShouldInitiallyBeEmpty() {  
    List<Integer> list = Arrays.asList(5, 2, 4);  
  
    assertThat(list, hasSize(3));  
  
    // ensure the order is correct  
    assertThat(list, contains(5, 2, 4));  
  
    assertThat(list, containsInAnyOrder(2, 4, 5));  
  
    assertThat(list, everyItem(greaterThan(1)));  
}
```

JUnit 5 (and beyond)

JUnit 5 erweitern mit Hamcrest

```
List<Integer> list = Arrays.asList(5, 2, 4);
```

Stellen Sie mit unterschiedlichen Testfällen sicher, dass

1. die Liste nur 3 Element hat
2. die Liste die Elemente 2, 4, 5 in beliebiger Reihenfolge beinhaltet
3. jedes einzelne Element größer als 1 ist

JUnit 5 (and beyond)

JUnit 5 erweitern mit Hamcrest

```
Integer[] ints = new Integer[] {7, 5, 12, 16};
```

Stellen Sie mit unterschiedlichen Testfällen sicher, dass

1. das Array 4 Elemente hat
2. die Elemente 7, 5, 12, 16 in genau der Reihenfolge beinhaltet

JUnit 5 (and beyond)

JUnit 5 erweitern mit Mockito

Manchmal ist es notwendig bestimmte Funktionalitäten isoliert zu testen: Side-Effects müssen vermieden werden, bestimmte Vorbedingungen erfüllt sein, etc.

All das bieten Mock-Frameworks die hierfür Mock-Objects zur Verfügung stellen.

Klassisches Beispiel ist ein Dataprovider:

In produktiven Umgebungen wird die Implementierung mit einer Datenbank verbunden.

In Testumgebungen wird für die Datenbank ein Mock-Object verwendet.

JUnit 5 (and beyond)

JUnit 5 erweitern mit Mockito

```
public class Database {  
  
    public boolean isAvailable() {  
        // TODO implement the access to the database  
        return false;  
    }  
  
    public int getUniqueId() {  
        return 42;  
    }  
}
```

```
public class Service {  
  
    private Database database;  
  
    public Service(Database database) {  
        this.database = database;  
    }  
  
    public boolean query(String query) {  
        return database.isAvailable();  
    }  
  
    @Override  
    public String toString() {  
        return "Using database with id: " + String.valueOf(database.getUniqueId());  
    }  
}
```

JUnit 5 (and beyond)

JUnit 5 erweitern mit Mockito

Startet Mockito

Erstellt ein Mock-Object der
Database-Klasse

Konfiguriert das Mock-Object

Führt Test-Code aus

```
import static org.junit.jupiter.api.Assertions.assertNotNull;  
import static org.junit.jupiter.api.Assertions.assertTrue;  
import static org.mockito.Mockito.when;
```

```
import org.junit.jupiter.api.Test;  
import org.junit.jupiter.api.extension.ExtendWith;  
import org.mockito.Mock;  
import org.mockito.junit.jupiter.MockitoExtension;
```

→ `@ExtendWith(MockitoExtension.class)`

Run Test | Debug Test

```
class ServiceTest {
```

→ `@Mock`

```
    Database databaseMock;
```

`@Test`

Run Test | Debug Test

```
    public void testQuery() {
```

```
        assertNotNull(databaseMock);
```

→ `when(databaseMock.isAvailable()).thenReturn(true);`

→ `Service t = new Service(databaseMock);`

```
        boolean check = t.query("* from t");
```

```
        assertTrue(check);
```

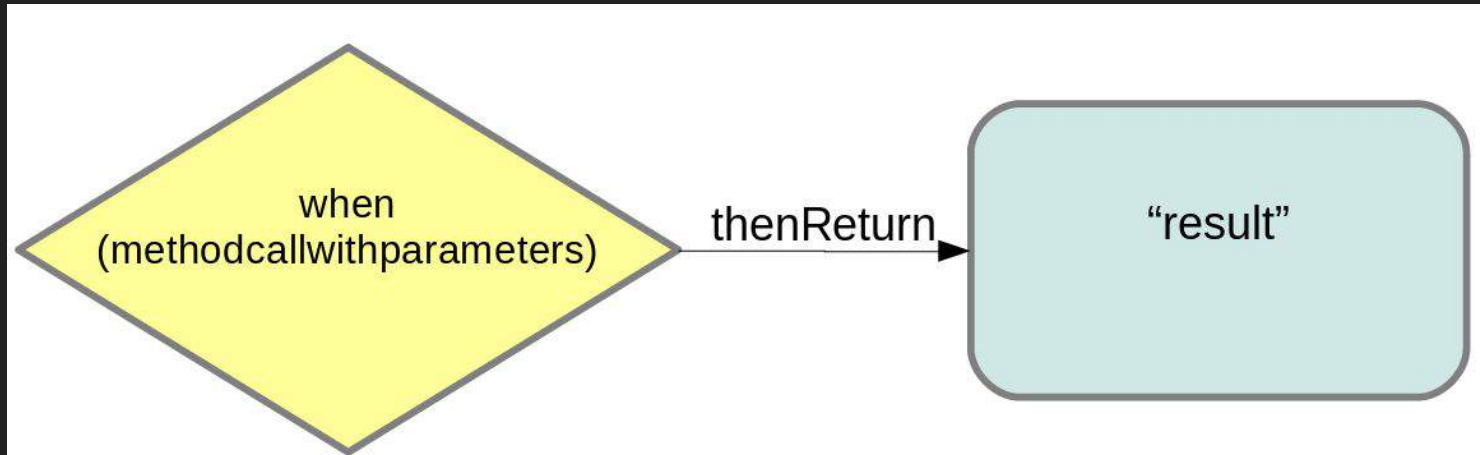
```
    }
```

```
}
```

JUnit 5 (and beyond)

JUnit 5 erweitern mit Mockito

Mit Mockito ist es möglich die return-Werte der Mock-Objects mit einer eigenen API zu kontrollieren:



JUnit 5 (and beyond)

JUnit 5 erweitern mit Mockito

Weitere Beispiele für

`when().thenReturn()`

```
@Test
```

```
Run Test | Debug Test
```

```
public void ensureMockitoReturnsTheConfiguredValue() {
```

```
    // define return value for method getId()
    when(databaseMock.getId()).thenReturn(42);
```

```
    Service service = new Service(databaseMock);
```

```
    // use mock in test....
```

```
    assertEquals(service.toString(), "Using database with id: 42");
```

```
}
```

JUnit 5 (and beyond)

JUnit 5 erweitern mit Mockito

Weitere Beispiele für

`when().thenReturn()`

```
@ExtendWith(MockitoExtension.class)
```

Run Test | Debug Test

```
public class MockitoSampleTest {
```

```
    @Mock
```

```
    Iterator<String> i;
```

```
    Comparable<String> c;
```

```
    // demonstrates the return of multiple values
```

```
    @Test
```

Run Test | Debug Test

```
    public void testMoreThanOneReturnValue() {
```

```
        when(i.next()).thenReturn("Mockito").thenReturn("rocks");
```

```
        String result = i.next() + " " + i.next();
```

```
        // assert
```

```
        assertEquals("Mockito rocks", result);
```

```
    }
```

```
    // this test demonstrates how to return values based on the input
```

```
    // and that @Mock can also be used for a method parameter
```

```
    @Test
```

Run Test | Debug Test

```
    public void testReturnValueDependentOnMethodParameter(@Mock Comparable<String> c) {
```

```
        when(c.compareTo("Mockito")).thenReturn(1);
```

```
        when(c.compareTo("Eclipse")).thenReturn(2);
```

```
        // assert
```

```
        assertEquals(1, c.compareTo("Mockito"));
```

```
        assertEquals(2, c.compareTo("Eclipse"));
```

```
    }
```

```
    // return a value based on the type of the provide parameter
```

```
    @Test
```

Run Test | Debug Test

```
    public void testReturnValueIndependentOnMethodParameter2(@Mock Comparable<Integer> c) {
```

```
        when(c.compareTo(isA(Integer.class))).thenReturn(0);
```

```
        // assert
```

```
        assertEquals(0, c.compareTo(Integer.valueOf(4)));
```

```
    }
```

```
}
```

JUnit 5 (and beyond)

JUnit 5 erweitern mit Mockito

@Spy oder spy() kann verwendet werden um reale Objekte zu mocken.

In diesem Beispiel wird jeder Aufruf an das Object weitergeleitet - wenn nicht anders angegeben.

```
public class MockitoSpyTest {  
    @Test  
    Run Test | Debug Test  
    public void testLinkedListSpyCorrect() {  
        // Lets mock a LinkedList  
        List<String> list = new LinkedList<>();  
        List<String> spy = spy(list);  
        doReturn("foo").when(spy).get(0);  
  
        assertEquals("foo", spy.get(0));  
    }  
}
```

JUnit 5 (and beyond)

JUnit 5 erweitern mit Mockito

Mockito ermöglicht die Überprüfung, ob eine Methode mit bestimmten Parametern aufgerufen wurde mittels `verify()`-Methode.

```
@ExtendWith(MockitoExtension.class)
```

```
Run Test | Debug Test
```

```
public class MockVerifyTest {
```

```
    @Test
```

```
Run Test | Debug Test
```

```
    public void testVerify(@Mock Database database) {
```

```
        when(database.getUniqueId()).thenReturn(43);
```

```
        database.setUniqueId(12);
```

```
        database.getUniqueId();
```

```
        database.getUniqueId();
```

```
        verify(database).setUniqueId(ArgumentMatchers.eq(12));
```

```
        verify(database, times(2)).getUniqueId();
```

```
        verify(database, never()).isAvailable();
```

```
        verify(database, never()).setUniqueId(13);
```

```
        verify(database, atLeastOnce()).setUniqueId(12);
```

```
        verify(database, atLeast(2)).getUniqueId();
```

```
        verifyNoMoreInteractions(database);
```

```
    }
```

```
}
```

Unit Testen mit Spring Boot

Inversion of Control

Das Prinzip von Inversion of Control (IoC) liegt darin, die Kontrolle eines Objekts oder Teile einer Applikation auf ein Framework zu übertragen.

Damit übernimmt das (auch) Framework den Kontrollfluss der Applikation im Gegensatz zu herkömmlichen Programmiermodellen in denen der Kontrollfluss durch Code und Logik kontrolliert wird.

Unit Testen mit Spring Boot

Inversion of Control

Vorteile dieses Ansatzes:

- + Decoupling der Ausführung der Tasks von der Implementierung
- + Einfacher Wechsel zwischen verschiedenen Implementierungen
- + Höhere Modularität der Applikation
- + Einfachere Testbarkeit durch Isolation der Components

Unit Testen mit Spring Boot

Inversion of Control

Nachteile dieses Ansatzes:

- Abhängigkeit vom Framework wird erhöht
- Vendor-Lock-in

Unit Testen mit Spring Boot

Dependency Injection

Dependency Injection (DI) ist eine Implementierung von Inversion of Control.

DI ist ein Paradigma das beim Zusammenbau von Objektnetzen eine möglichst hohe Kohäsion und eine möglichst lose Kopplung von Objekten ermöglicht.

Kohäsion: Ein Maß für den logischen Zusammenhang der Daten und Methoden einer Klasse

Kopplung: Ein Maß der Abhängigkeit der einzelnen Komponenten

Unit Testen mit Spring Boot

Dependency Injection

Dependency Injection bildet die unterste Ebene der Infrastruktur einer Spring-basierten Anwendung - sie ist eine mögliche Art der Steuerungsumkehr:

Objekte instanziiieren oder referenzieren benötigte Kollaborateure nicht selber, sondern diese werden von außen zur Verfügung gestellt.

Unit Testen mit Spring Boot

Constructor-based Dependency Injection

In Constructor-based DI werden alle Abhängigkeiten die die Klasse benötigt als Argumente im Constructor definiert.

Seit Spring 4.3 ist die `@Autowired` Annotation beim Constructor dafür optional - wenn es nur einen Constructor in der Klasse gibt

```
public class UserService {  
  
    UserRepository userRepository;  
  
    @Autowired  
    public UserService(UserRepository userRepository) {  
        this.userRepository = userRepository;  
    }  
  
    public int getUsersCount() {  
        return userRepository.getUsersCount();  
    }  
  
    public List<String> getUsers() {  
        return userRepository.getUsers();  
    }  
}
```

Unit Testen mit Spring Boot

Setter Injection

In der Setter Injection werden für die benötigten Abhängigkeiten setter Methoden definiert die mit der `@Autowired` Annotation versehen sind

```
public class UserService {  
  
    private UserRepository userRepository;  
  
    @Autowired  
    public void setUserRepository(UserRepository userRepository) {  
        this.userRepository = userRepository;  
    }  
  
    public int getUsersCount() {  
        return userRepository.getUsersCount();  
    }  
  
    public List<String> getUsers() {  
        return userRepository.getUsers();  
    }  
}
```

Unit Testen mit Spring Boot

Field Injection

Mit der Field Injection werden die Abhängigkeiten direkt in den Variablen aufgelöst die mit der `@Autowired` Annotation versehen sind

```
public class UserService {  
  
    @Autowired  
    private UserRepository userRepository;  
  
    public int getUsersCount() {  
        return userRepository.getUsersCount();  
    }  
  
    public List<String> getUsers() {  
        return userRepository.getUsers();  
    }  
}
```

Unit Testen mit Spring Boot

Constructor-based Dependency Injection

Wieso die Constructor-based Dependency Injection bevorzugt werden sollte:

- Alle benötigten Abhängigkeiten sind zur Initialisierung der Instanz vorhanden
- Identifizierung von Code-Smells:
Wenn der Constructor zu viele Argumente übernimmt ist das ein Zeichen dafür dass die Klasse zu viel Verantwortung übernimmt
- Einfachere Umsetzung von Tests
- Immutability wird (einfacher) erreicht

```
public class UserService {  
  
    UserRepository userRepository;  
  
    @Autowired  
    public UserService(UserRepository userRepository) {  
        this.userRepository = userRepository;  
    }  
  
    public int getUsersCount() {  
        return userRepository.getUsersCount();  
    }  
  
    public List<String> getUsers() {  
        return userRepository.getUsers();  
    }  
}
```


Unit Testen mit Spring Boot

Dependency Injection

Mittels `@SpringBootTest` Annotation lässt sich die gesamte DI auch innerhalb der Testfälle für Spring Boot Applikationen nutzen.

```
@SpringBootTest
```

```
Run Test | Debug Test
```

```
class DemoApplicationTests {
```

```
    @Autowired
```

```
    UserRepository userRepository;
```

```
    @Test
```

```
Run Test | Debug Test
```

```
    void contextLoads() {
```

```
        assertEquals(2, userRepository getUsersCount());
```

```
    }
```

```
}
```

Unit Testen mit Spring Boot

```
12  @ExtendWith(SpringExtension.class)
13  @SpringBootTest
14  class SampleUnitTest {
15
16      @Autowired
17      private RegisterUseCase registerUseCase;
18
19      @Test
20      void savedUserHasRegistrationDate() {
21          User user = new User( userName: "steve", email: "steve@apple.com");
22          User savedUser = registerUseCase.registerUser(user);
23          assertThat(savedUser.getRegistrationDate()).isNotNull();
24      }
```

Achtung 💣

Dieser “Unit”-Test benötigt ein paar Sekunden zum Durchlaufen: Der Spring-Context muss geladen werden. Ein guter TDD-Test benötigt lediglich ein paar Millisekunden um den “test / code / test” - Flow nicht zu unterbrechen.

Unit Testen mit Spring Boot

Field Injection ist
problematisch

```
12  @ExtendWith(SpringExtension.class)
13  @SpringBootTest
14  class SampleUnitTest {
15
16      @Autowired
17      private RegisterUseCase registerUseCase;
18
19      @Test
20      void savedUserHasRegistrationDate() {
21          User user = new User( userName: "steve", email: "steve@apple.com");
22          User savedUser = registerUseCase.registerUser(user);
23          assertThat(savedUser.getRegistrationDate()).isNotNull();
24      }
```

Unit Testen mit Spring Boot

Constructor Injection ist
viel eleganter



```
5  @Service
6  public class RegisterUseCase {
7
8      private final UserRepository userRepository;
9
10     public RegisterUseCase(UserRepository userRepository) {
11         this.userRepository = userRepository;
12     }
13
14     public User registerUser(User user) {
15         return userRepository.save(user);
16     }
17
18 }
```

Unit Testen mit Spring Boot

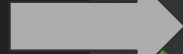
UserRepository ist jetzt
final



```
5  @Service
6  public class RegisterUseCase {
7
8
9
10 public RegisterUseCase(UserRepository userRepository) {
11     this.userRepository = userRepository;
12 }
13
14 public User registerUser(User user) {
15     return userRepository.save(user);
16 }
17
18 }
```

Unit Testen mit Spring Boot

Boilerplate Code
reduzieren mit Lombok



```
8      @Service
9      @RequiredArgsConstructor
10     public class RegisterUseCase {
11
12         private final UserRepository userRepository;
13
14         @
15         public User registerUser(User user) {
16             user.setRegistrationDate(LocalDate.now());
17             return userRepository.save(user);
18         }
19     }
```