

Praktiken und Werkzeuge für die Software Entwicklung Testframeworks

Alessia Bäcker, Touni Arar, und Marie Kastning
Seminararbeit

Praktiken und Werkzeuge für die
Softwareentwicklung SoSe 2021

23. August 2021

Prüfer:

Prof. Dr.-Ing. C. Bockisch
AG Programmiersprachen
und -werkzeuge

Betreuer:

M.Sc. Stefan Schulz
AG Programmiersprachen
und -werkzeuge



Testframeworks

Institut

Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Arbeitsgruppe Programmiersprachen und -werkzeuge
Hans-Meerwein-Str. 6
35043 Marburg
Deutschland

Lizenz

Dieses Werk ist lizenziert unter einer Creative Commons “Namensnennung – Nicht-kommerziell – Weitergabe unter gleichen Bedingungen 4.0 International” Lizenz.



Klassifikation (ACM CCS 2012)

- **Applied computing** Education; Document preparation;
- *Software and its engineering* Software notations and tools;

Schlüsselwörter

JUnit, TestNG, Selenium, Testframeworks, Testarten

Zusammenfassung

Testen ist von großer Relevanz um langfristig Kosten in der Softwareentwicklung zu reduzieren und Qualität zu wahren. In dieser Ausarbeitung werden drei Testframeworks vorgestellt. Zu Beginn wird das Framework JUnit vorgestellt und im weiteren Verlauf mit TestNg erweitert, indem neue Features wie Multithreading, DataProvider und Gruppierungen von Tests eingeführt werden. Abgerundet wird das Ganze mit Selenium, welches dem Testen von Browserapplikationen dient und zudem mit JUnit und TestNG zusammenarbeiten kann, indem es diese zur Evaluation nutzt. Abschließend werden die Unterschiede und Gemeinsamkeiten der drei Testframeworks herausgearbeitet. Dabei wird verdeutlicht, dass TestNg und JUnit sich sehr ähneln, während Selenium sich für ein anderes Anwendungsgebiet eignet.

Inhaltsverzeichnis

1	Einleitung	2
1.1	Funktionales und nicht-funktionales Testen	2
1.2	White-Box und Black-Box Testing	2
1.3	Agile Softwareentwicklung	3
2	JUnit	4
2.1	Einleitung	4
2.2	JUnit 5	5
2.3	Fazit	17
3	TestNG	18
3.1	Einleitung	18
3.2	Neue Funktionalitäten	18
3.3	Beispiele	21
3.4	Fazit	25
4	Selenium	26
4.1	Einleitung	26
4.2	Überblick	27
4.3	Testautomatisierung mit Selenium	27
4.4	Tests	37
4.5	Fazit	42
5	Gemeinsame Tests	44
6	Vergleich und Fazit	45
6.1	JUnit verglichen mit TestNG	45
6.2	JUnit und TestNG verglichen mit Selenium	45
6.3	Fazit	45
7	Eidesstattliche Erklärung	46
8	Literatur	47

1 Einleitung

In der Softwareentwicklung zählt das Testen zu einem der wichtigsten Bereiche. Es dient der Qualitätssicherung, Sicherheit und höherer Kundenzufriedenheit, wodurch außerdem langfristig Kosten gespart werden.

Eines der bekanntesten Tools um Java-Code zu testen ist JUnit, welches neben TestNG und Selenium in dieser Arbeit genauer vorgestellt wird. Diese drei Testframeworks bieten ein Grundgerüst samt Funktionen und Klassen, mithilfe derer man automatisierte Tests entwickeln kann.[Tal18]

1.1 Funktionales und nicht-funktionales Testen

Verschiedene Aspekte des Softwaresystems werden mit verschiedenen Tests überprüft, die in funktionale und nicht-funktionale unterteilt werden können.

Funktionales Testen Funktionale Tests werden nahe am Code entwickelt, um die Qualität und Korrektheit der Funktionsumsetzung zu testen. Außerdem dienen sie dazu Fehler frühzeitig zu erkennen.

Nicht-funktionales Testen Nicht-funktionale Tests behandeln die verbleibenden Aspekte, wie Performance, Benutzbarkeit und Zuverlässigkeit. Diese werden oft anhand eines User-Interfaces getestet, indem man Interaktionen eines Users simuliert. [BN16]

1.2 White-Box und Black-Box Testing

Bei dem Testen von Software unterscheidet man zwischen zwei wesentlichen Teststilen, Black-Box und White-Box Testing.

Blackbox Testing: Ein Teststil, bei dem dem Entwickler der Testfälle keine Kenntnisse über interne Strukturen, interne Prozesse und der genauen Implementierung vorliegen. Bei nicht-funktionalen Tests, wie bei der Entwicklung mit Selenium, wird überwiegend Blackbox-Testing verwendet.

Whitebox Testing: Ein Teststil, der die Eingaben, Ausgaben und die zwischen ihnen angegebenen Beziehungen zusammen mit der Kenntnis der internen Struktur, des internen Prozesses und der genauen Implementierung berücksichtigt. Bei funktionalen Tests, wie bei der Entwicklung mit JUnit und TestNG, wird überwiegend Whitebox-Testing verwendet. [BN16]

1.3 Agile Softwareentwicklung

Durch JUnit, TestNG und Selenium werden Frameworks für die Entwicklung von automatisierten Tests bereitgestellt, welches wesentlich für die agile Softwareentwicklung ist. Denn bei der agilen Softwareentwicklung versucht man so schnell wie möglich eine lauffähige Software zu realisieren und diese dann stetig weiterzuentwickeln. Gerade bei dieser Art von Softwareentwicklung bieten sich automatisierte Tests an, da man immer wieder den gleichen Test auf unterschiedliche Software ausführt. Somit gestaltet man den Entwicklungsprozess der Software sehr flexibel und es ist üblich, dass sich im Laufe des Projekts Ziele und Umfeld ändern.

Die Grundprinzipien für agile Tests sind schnelles Feedback, ein hoher Automatisierungsgrad und geringer Aufwand. [Wol]

2 JUnit

2.1 Einleitung

Unter dem Begriff Code-Unit versteht man einen logisch trennbareren Teil eines Computerprogramms. Im Sinne eines Java-Programms ist dies beispielsweise eine Package, Methode oder Klasse.[iee90] [Vog21a] Softwaretests sind ein signifikanter Teil des Entwicklungsprozesses eines Softwaresystems. Darunter steht das sogenannte Unit-Testing, was in den nächsten Abschnitten neben JUnit 5 als das Hauptthema behandelt wird.

2.1.1 Was ist Unit-Testing?

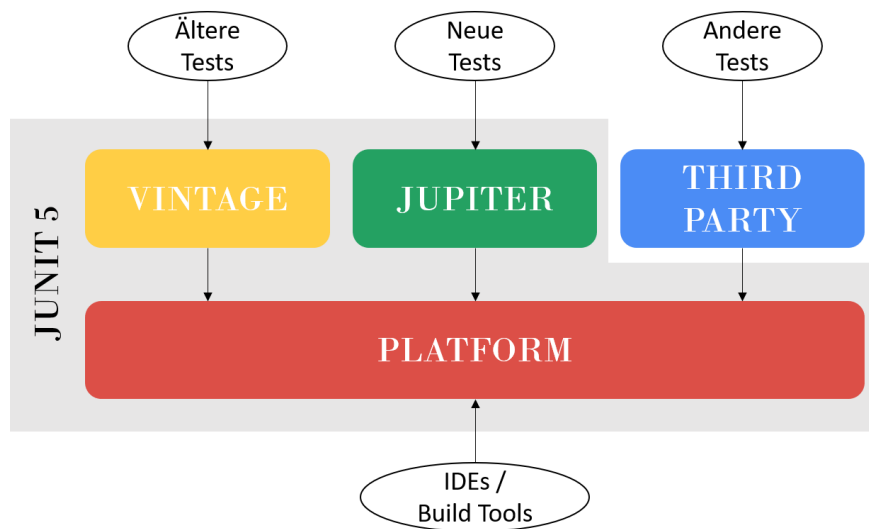
Unit-Testing ist eine Testtechnik, bei der einzelne Units getestet werden, um festzustellen, ob der Entwickler beim Schreiben der Unit Fehlern begangen ist. Es geht um die Korrektheit der Funktionalität einer Unit. Jede Unit des Systems wird isoliert, um die Fehler zu identifizieren, analysieren und beheben.[BN16] [TP221] Eine Test-Framework führt dabei eine Code-Unit aus, wobei erwartet wird, dass sie sich auf eine bestimmte Weise verhält. Diese Framework stellt dann sicher, ob dies der Fall ist.[Vog21a] [Vog21b]

2.1.2 Ziel des Unit Testings

Mit Unit-Tests stellt man sicher, dass bestimmte Teile einer Software wie erwartet funktionieren. Diese Tests werden normalerweise automatisch über ein Build-System ausgeführt und helfen dem Entwickler, den vorhandenen Code während der Entwicklung nicht zu beschädigen. Durch das automatische Ausführen von Tests können Software-Regressionen identifiziert werden (wenn ein Feature nicht mehr in der neuen Version einer Software funktioniert [TYo8]), die durch Änderungen im Quellcode verursacht werden. Durch eine hohe Testabdeckung des Codes kann man Funktionen und/oder Features weiterentwickeln, ohne viele manuelle Tests durchführen zu müssen. Man sollte Unit-Tests für die kritischen und komplexen Teile einer Anwendung schreiben.[Vog21b]

2.1.3 Vorteile von Unit Testing [Nov20]

- Agileres Entwickeln und sicheres Refactoring: *Einer der Hauptvorteile von Unit-Tests besteht darin, dass der Coding-Prozess agiler wird. Wenn einer Software immer mehr Funktionen hinzugefügt werden müssen, muss man manchmal altes Design und alten Code ändern. Das Ändern von bereits getestetem Code ist jedoch sowohl riskant als auch kostspielig. Wenn Unit-Tests vorhanden sind, können Entwickler sicher mit dem Refactoring fortfahren.*
- Frühere und bessere Fehlererkennung: *Fehler im Code werden frühzeitig erkannt. Da Unit-Tests von Entwicklern durchgeführt werden, die eigenen Code vor der Integration testen (bevor Entwickler Subsysteme zusammenführen [Wik21a]), können Probleme sehr früh erkannt und dann behoben werden, ohne die anderen Teile des Codes zu beeinträchtigen. Dies umfasst sowohl Fehler in der Implementierung als auch fehlende Teile der Spezifikation einer Unit.*
- Kostenreduktion: *Da die Fehler frühzeitig erkannt werden, können durch Unit-Tests die Kosten für die Fehlerbehebung gesenkt werden. Die Kosten der Behebung eines Fehlers, der in späteren Entwicklungsphasen festgestellt wurde, sind sehr hoch. Logischerweise sind Fehler, die früher erkannt werden, einfacher zu beheben, da später erkannte Fehler normalerweise die Folge vieler Änderungen sind und man nicht wirklich weiß, welche Änderung den Fehler verursacht hat.*



■ **Abbildung 1** JUnit 5 - allgemeine Struktur [MH20]

2.2 JUnit 5

In den folgenden Unterabschnitten wird JUnit vorgestellt, eine Unit-Testing Framework für Java. Es werden auch Codebeispiele eingeführt, die für die Praxis relevant sind, wenn Entwickler Unit-Tests schreiben. Wir werden uns hier mit dem neusten Standard von JUnit 5 beschäftigen.

2.2.1 Die Struktur von JUnit 5 [dRS21]

JUnit 5 besteht aus mehreren verschiedenen Modulen aus drei verschiedenen Teilprojekten:

- **JUnit-Plattform:** dient als Grundlage für das Starten von Test-Frameworks auf der Java Virtual Machine (JVM). Außerdem wird die Test-Engine API zum Entwickeln von Testframeworks bereitgestellt, die auf der Plattform ausgeführt werden können. Darüber hinaus bietet die Plattform einen Console-Launcher zum Starten der Plattform über die Kommandozeile und einen JUnit 4-basierten Runner zum Ausführen einer beliebigen Test-Engine auf der Plattform in einer JUnit 4-basierten Umgebung. Unterstützung für die JUnit-Plattform gibt es auch in gängigen IDEs (wie IntelliJ IDEA, Eclipse, NetBeans und Visual Studio Code) und Build-Tools (wie Gradle, Maven und Ant).
- **JUnit-Jupiter:** ist die Kombination aus dem neuen Programmiermodell und dem Erweiterungsmodell zum Schreiben von Unit-Tests und Erweiterungen in JUnit 5. Das Jupiter-Unterprojekt bietet eine TestEngine zum Ausführen von Jupiter-basierten Tests auf der Plattform.
- **JUnit-Vintage:** bietet eine TestEngine zum Ausführen von JUnit 3- und JUnit 4-basierten Tests auf der Plattform.

Abbildung 1 zeigt eine allgemeine Übersicht der Unterprojekte von JUnit 5.

Testframeworks

■ **Tabelle 1** JUnit Annotations [dRS21] [Vog21a]

Annotation	Beschreibung
@Test	Definiert eine Methode als Testmethode.
@Disabled("reason")	Deaktiviert eine Testmethode mit einem optionalen Grund.
@BeforeEach	Wird vor jedem Test ausgeführt. Wird verwendet, um die Testumgebung vorzubereiten, z. B. um die Felder in der Testklasse zu initialisieren, die Umgebung zu konfigurieren usw.
@AfterEach	Wird nach jedem Test ausgeführt. Wird verwendet, um die Testumgebung zu bereinigen, z. B. temporäre Daten zu löschen, Standardeinstellungen wiederherzustellen und teure Speicherstrukturen zu bereinigen.
@DisplayName("<Name>")	Name, der vom Test-Runner angezeigt wird. Im Gegensatz zu Methodennamen kann der Name Leerzeichen enthalten, um die Lesbarkeit zu verbessern.
@RepeatedTest(<Number>)	Ähnlich wie @Test, wiederholt denselben Test jedoch <Number> Mal.
@BeforeAll	Definiert eine Methode, die vor Beginn aller Tests einmal ausgeführt wird. Sie wird normalerweise verwendet, um zeitintensive Aktivitäten auszuführen, z. B. um eine Verbindung zu einer Datenbank herzustellen.
@AfterAll	Definiert eine Methode, die einmal ausgeführt wird, nachdem alle Tests abgeschlossen sind. Sie wird verwendet, um Bereinigungen durchzuführen, z. B. um die Verbindung zu einer Datenbank zu trennen.
@TestFactory	Identifiziert eine Methode, die eine "Fabrik" zum Erstellen von dynamischen Tests ist.
@Tag("<TagName>")	Versieht eine Testmethode. Tests in JUnit 5 können nach Tags gefiltert werden, beispielsweise werden nur Tests ausgeführt, die mit dem Tag "schnell" versehen sind.
@Timeout(<value>, <unit>)	Wird verwendet, um einen Test zum Scheitern zu bringen, wenn seine Ausführung eine bestimmte Dauer überschreitet.
@ParameterizedTest	Bezeichnet, dass eine Methode ein parametrisierter Test ist.

2.2.2 JUnit 5 Annotations

JUnit-Jupiter unterstützt Annotations zum Konfigurieren von Tests und zum Erweitern des Frameworks. [dRS21] In Tabelle 1 finden Sie einige bedeutsamen Annotations mit einer Beschreibung ihrer Funktion.

2.2.3 Vorgehen beim Schreiben von Unit-Tests

In Listing 1 ist eine zu testende Klasse. Unit-Tests haben immer die gleiche Struktur und bestehen aus drei Hauptteilen. Zuerst wird ein sogenanntes Szenario vorbereitet (Listing 2, Zeilen 6-9), dann wird die zu testende Methode oder Methodenkombination aufgerufen (Listing 2, Zeilen 13-14), und zuletzt wird mit der API der Testframework überprüft, ob der ausgeführte Code genau das erwartete Verhalten aufgezeigt hat (Listing 2, Zeilen 16-17). Dafür sind bei JUnit die assert-Methoden zuständig, die das gewünschte Ergebnis des Aufrufs mit dem tatsächlichen vergleichen und bei Abweichung einen Fehler melden. Der Teil, der Tests ausführt und solche Fehler meldet, ist der Test-Runner.[Ull16]

■ Listing 1 Einfache zu testende Klasse

```

1 public class Calculator {
2
3     // addiert zwei Zahlen a und b und gibt die Summe zurueck
4     public int add(int a, int b) {
5         return a + b;
6     }
7
8 }
```

■ Listing 2 Einblick in die Mindestanforderungen für einen Test

```

1 import static org.junit.jupiter.api.Assertions.*;
2 import org.junit.jupiter.api.*;
3
4 class CalculatorTest {
5
6     // Szenario aufbauen
7     int a = 1;
8     int b = 1;
9     private final Calculator calculator = new Calculator();
10
11     @Test
12     public void addTest() {
13         // Methode aufrufen und tatsaechliches Ergebnis speichern
14         int actualResult = calculator.add(a, b);
15
16         // Ueberpruefung mit einer JUnit Assertion
17         assertEquals(2, actualResult);
18     }
19
20 }
```

Es wird also eine separate Testklasse (Listing 2) für die zu testende Klasse (Listing 1) erstellt.

■ Einstellen von JUnit mit einem IDE

Wir werden für die Beispiele IntelliJ 2020.3 mit JUnit 5.7.2 nutzen. Für mehr Informationen über den Einstellungsprozess besuchen Sie bitte folgende Links:

<https://junit.org/junit5/docs/current/user-guide/#running-tests-ide-intellij-idea>

<https://www.jetbrains.com/help/idea/testing.html>

Testframeworks

■ **Tabelle 2** JUnit Assertions: “nonFloatingPoint” steht für nicht-Fließkommazahlen. “primitive” steht für primitive Datentypen. Es werden bei assertEquals() bzw. assertNotEquals() immer zwei Parameter gleicher Art verglichen.[Api] [Ull16] [Vog21a]

Methode	Beschreibung
static void assertTrue(boolean condition) static void assertFalse(boolean condition)	Überprüft jeweils, ob der Parameter condition wahr bzw. falsch ist.
static void assertNull(Object object) static void assertNotNull(Object object)	Überprüft jeweils, ob der Parameter object null bzw. nicht null ist.
static void assertSame(Object expected, Object actual) static void assertNotSame(Object unexpected, Object actual)	Überprüft jeweils, ob sich expected/unexpected und actual übereinstimmen (haben die gleiche Referenz auf das gleiche Objekt im Speicher) bzw. nicht übereinstimmen.
static void assertEquals(Object expected, Object actual) static void assertEquals(nonFloatingPoint expected, nonFloatingPoint actual) static void assertEquals(float/double expected, float/double actual, float/double delta)	Die erste Methode überprüft, ob zwei Objekte gleich sind, indem sie expected.equals(actual) aufruft, die zweite prüft, ob zwei Zahlen gleich sind, und die dritte, ob zwei float bzw. double Zahlen den gleichen Wert haben oder sich eventuell um eine Differenz delta unterscheiden.
static void assertEquals(primitive[] expecteds, primitive[] actuals) static void assertEquals(Object[] expecteds, Object[] actuals)	Hat die gleiche Aufgabe wie assertEquals, erledigt diese aber für jedes Element in den Arrays expecteds und actuals und vergleicht mit Beachtung der Reihenfolge.
static void assertNotEquals(nonFloatingPoint unexpected, nonFloatingPoint actual) static void assertNotEquals(float/double unexpected, float/double actual, float/double delta) static void assertNotEquals(Object unexpected, Object actual)	Die negierte Version von assertEquals.
assertThrows(Class<T> expectedType, Executable executable)	Überprüft, ob die Ausführung der übergebenen Executable eine Exception des erwarteten Typs auslöst und diese zurückgibt (nicht behandelt).

2.2.4 Die assert-Methoden

Zur Überprüfung, ob tatsächliche Ergebnisse eines Tests mit den erwarteten übereinstimmen, bietet JUnit die sogenannten assert-Methoden. Diese Methoden befinden sich in der Assertions Klasse in der Package org.junit.jupiter.api und müssen in die Testklasse importiert werden.[Api] [Ull16] In Tabelle 2 sind einige wichtigen assert-Methoden.

2.2.5 Code-Beispiele

In diesem Unterabschnitt werden Code-Beispiele gegeben, um Annotations (Tabelle 1) und assert-Methoden (Tabelle 2) zu verdeutlichen. Die Kommentare im Code enthalten auch wichtige Informationen. Sei wiederum die Klasse in Listing 1 die zu testende Klasse. Wir haben diese erweitert um noch mehr Funktionalität in Listing 3 und möchten gerne die Methoden testen und dabei uns anschauen, wie viel Testabdeckung wir erreicht haben:

■ Listing 3 Erweiterte Klasse

```

1 public class Calculator {
2
3     double ans = Double.NaN;
4     double memory = Double.NaN;
5
6     // speichert das Ergebnis einer Operation im Speicher
7     public void memorize(){
8         memory = ans;
9     }
10
11    // löscht den Wert im Speicher
12    public void forget(){
13        this.memory = Double.NaN;
14    }
15
16    // Addition
17    public double add(double a, double b) {
18        return ans = a + b;
19    }
20
21    // Multiplikation
22    public double mul(double a, double b){
23        return ans = a*b;
24    }
25
26    // Subtraktion
27    public double sub(double a, double b){
28        return ans = a-b;
29    }
30
31    // Division
32    public double div(double a, double b) throws Exception {
33        if (b == 0){
34            throw new ArithmeticException("Undefined");
35        }
36        return ans = a/b;
37    }
38
39    // Logarithmus zur Basis 2
40    public double log2(double a) {
41        if (a <= 0){
42            throw new ArithmeticException("Undefined");
43        }
44        return ans = Math.log(a)/Math.log(2);
45    }
46
47    // Potenzfunktion

```

Testframeworks

```
48 public double pow(double a, double b){
49     double fraction = b - Math.floor(b);
50     double result = 1;
51     for (int i = 1; i <= (int) b; i++){
52         result = result*a;
53     }
54     if (fraction > 0){
55         result = result*Math.pow(10,Math.log10(a)*fraction);
56     }
57     return ans = result;
58 }
59
60 // n-te Fibonacci Zahl, naive Implementierung
61 public int fib(int n) {
62     if(n <= 1) {
63         return n;
64     }
65     return (int) (ans = fib(n-1) + fib(n-2));
66 }
67
68 // n-te Fibonacci Zahl, schnelle rekursive Implementierung
69 public int fastFib(int n){
70     return (int) (ans = fib_aux(n,0,1));
71 }
72
73 // Hilfsfunktion fuer fastFib
74 private int fib_aux(int n, int acc_o, int acc_1) {
75     if (n == 0){
76         return acc_o;
77     }
78     int temp = acc_1;
79     acc_1 = acc_o + acc_1;
80     acc_o = temp;
81     return fib_aux(n-1, acc_o, acc_1);
82 }
83
84 }
```

Wir sehen uns jetzt die Testklasse einen Test nach dem anderen an und erklären diese:

In Listing 4 ist ein Beispiel für die Nutzung der Annotations `@BeforeAll`, `@AfterAll`, `@BeforeEach` und `@AfterEach` (siehe Tabelle 1).

■ **Listing 4** Anfang der Testklasse mit `@BeforeAll`, `@AfterAll`, `@BeforeEach` und `@AfterEach`

```
1 import org.junit.jupiter.api.*;
2 import org.junit.jupiter.params.ParameterizedTest;
3 import org.junit.jupiter.params.provider.Arguments;
4 import org.junit.jupiter.params.provider.MethodSource;
5 import java.time.Duration;
6 import java.util.concurrent.TimeUnit;
7 import java.util.stream.Stream;
8 import static org.junit.jupiter.api.Assertions.*;
9
10 class CalculatorTest {
11
12     private static Calculator calculator;
```

```

13
14 // initialisiert calculator und meldet den Start der Tests
15 @BeforeAll
16 public static void start(){
17     calculator = new Calculator();
18     System.out.println("started testing...");
19 }
20
21 // meldet das Ende der Tests
22 @AfterAll
23 public static void finish(){
24     System.out.println("ended testing...");
25 }
26
27 // vergisst das Ergebnis der letzten Operation und meldet den Wert
28 @BeforeEach
29 public void preparation(){
30     System.out.println("forgot: " + calculator.memory);
31     calculator.forget();
32 }
33
34 // merkt sich das Ergebnis der letzten Operation und meldet den Wert
35 @AfterEach
36 public void finalization(){
37     calculator.memorize();
38     System.out.println("remembered: " + calculator.memory);
39 }

```

Zu jedem Test gehört ein Testname in der JUnit-Engine. Dies ist standardmäßig der Name der Methode, man kann diesen aber beliebig ändern mit der Annotation `@DisplayName`. [dRS21] Ein Beispiel stellt der Code-Ausschnitt in Listing 5 dar. `@Test` wird hier außerdem benutzt, um den Test zu definieren.

■ **Listing 5** Beispiel: `@DisplayName`

```

1 @Test
2 @DisplayName("Subtraction Test")
3 public void subTest(){
4     assertEquals(0, calculator.sub(1,1));
5 }

```

Mit `@ParameterizedTest` anstelle von `@Test` schreibt man einen Test, der mit verschiedenen Parametern (bzw. Eingaben) wiederholt wird. Dabei ist jede Wiederholung wie ein eigenständiger Test, also die `@BeforeEach` und `@AfterEach` Methoden werden für jede Wiederholung aufgerufen. Jeder Test hat einen Namen, der als Standardwert aus seinem Index (Ausführungsreihenfolge) und den Parametern (Reihenfolge im Methodenkopf) besteht, allerdings kann man dies anpassen. Ein parametrisierter Test muss die Parameter aus einer Quelle empfangen, dafür gibt es verschiedene Wege, aber wir zeigen in Listing 6 die Möglichkeit, die sowohl mit primitiven als auch benutzerdefinierten Datentypen und Enums funktioniert, nämlich `@MethodSource("<Methodenname>")`. Diese holt die Parameter aus einer Methode, die einen Stream von Arguments zurückgibt. [Par18b]

Testframeworks

■ Listing 6 Beispiel: @ParametrizedTest

```
1  @ParameterizedTest(name = "result of {0} x {1} should be {2}")
2  @MethodSource("createMulArgs")
3  public void mulTest(double a, double b, double expectedAnswer) {
4      assertEquals(expectedAnswer, calculator.mul(a,b));
5      assertEquals(expectedAnswer, calculator.ans);
6  }
7
8  // stellt die Parameter fuer den parametrisierten Test mulTest bereit
9  private static Stream<Arguments> createMulArgs() {
10     return Stream.of(
11         Arguments.of(5, 0, 0),
12         Arguments.of(7, 4, 28),
13         Arguments.of(3, 2, 6)
14     );
15 }
```

Statt @Test kann man auch @RepeatedTest(<Anzahl>) nutzen. Das ist bloß der gleiche Test, nur <Anzahl> Mal wiederholt. [Api] Ein Beispiel dazu ist in Listing 7 zu sehen.

■ Listing 7 Beispiel: @RepeatedTest

```
1  @RepeatedTest(5)
2  public void addTest() {
3      assertEquals(2, calculator.add(1,1));
4      assertEquals(2, calculator.ans);
5  }
```

JUnit zeigt immer auf der Konsole welche Tests fehlgeschlagen sind und warum. Tests können auch aufgrund unbehandelter Exceptions scheitern, nicht nur wegen unerwartetem Ergebnis. Listing 8 zeigt die Methode assertThrows(), damit kann man durch die Eingabe einer Exception-Klasse und eines Executables prüfen, ob der Code im Executable eine Exception wirft. Wenn keine Exception oder eine andere Art als die erwartete geworfen wird, scheitert der Test. [Api] [Vog21a]

■ Listing 8 Beispiel: assertThrows()

```
1  @Test
2  public void divTest() throws Exception {
3      assertThrows(ArithmeticException.class, () -> { calculator.div(0,0); });
4      assertThrows(ArithmeticException.class, () -> { calculator.div(1,0); });
5      assertEquals(1, calculator.div(1,1));
6  }
```

Im Folgenden ist ein Beispiel für die Deaktivierung eines bestimmten Tests, wenn alle Tests ausgeführt werden. @Disabled("<Grund>") tut genau dies in Listing 9 und schreibt den Grund "fails due to design flaws" auf der Konsole. [dRS21] Dieser Test ist auch ein gutes Beispiel für die frühzeitige Erkennung von Implementierungsfehlern.

■ Listing 9 Beispiel: @Disabled

```
1  // gutes Beispiel fuer schlechtes Design
2  @Test
3  @Disabled("fails due to design flaws")
4  public void powTest(){
5      // korrekt
6      assertEquals(1448.15, calculator.pow(2,10.5), 0.01);
7      // falsches Ergebnis, negative Exponente nicht betrachtet im Code
```

```

8     assertEquals(0.5, calculator.pow(2,-1));
9 }

```

Listing 10 zeigt, wie man eine Methode mit `@Tag` versehen kann (siehe Tabelle 1). Der Test ist auch ein gutes Beispiel für White-Box Testing. Informationen zur Konfiguration von Tags in IntelliJ findet man unter: <https://www.jetbrains.com/help/idea/run-debug-configuration-junit.html#configTab>.

■ **Listing 10** Beispiel: `@Tag`

```

1  // gutes Beispiel fuer einen White-Box Test
2  @Test
3  @Tag("slow")
4  public void log2Test(){
5      // typische Faelle
6      assertEquals(3,calculator.log2(8));
7      assertEquals(3, calculator.ans);
8      assertEquals(10, calculator.log2(1024));
9      assertEquals(10, calculator.ans);
10     // Randfaelle
11     assertEquals(0, calculator.log2(1));
12     assertEquals(0, calculator.ans);
13     assertThrows(ArithmeticException.class, () -> calculator.log2(0));
14     assertThrows(ArithmeticException.class, () -> calculator.log2(-8));
15 }

```

Folgende Tests sind die letzten in der Klasse `CalculatorTest`. `@Timeout` wird benutzt, um einen Test zum Scheitern zu bringen, falls seine Ausführung eine bestimmte Dauer überschreitet (Listing 11 Zeilen 3-10). `assertTimeoutPreemptively()` und `assertTimeout()` sind Alternativen zu `@Timeout` in JUnit 5 (Listing 11 Zeilen 17-24). Sie nehmen eine `Duration` (Dauer) und einen `Executable` (Test-Code) als Parameter. Diese beiden Methoden verfolgen das gleiche Ziel, jedoch mit einem feinen Unterschied. `assertTimeoutPreemptively()` bricht die Ausführung des `Executables` direkt ab, wenn die Zeitüberschreitung auftritt und als Folge scheitert der Test, während `assertTimeout()` dies nicht tut, sondern sie lässt ihn erst zu Ende laufen, dann scheitert der Test (verhält sich wie `@Timeout`). `assertTimeoutPreemptively()` führt den `Executable` in einem anderen Thread als der des aufrufenden Codes aus, aufgrund dessen könnten aber Probleme auftauchen, mehr dazu unter: [https://junit.org/junit5/docs/current/user-guide/#writing-tests-assertions\[dRS21\]](https://junit.org/junit5/docs/current/user-guide/#writing-tests-assertions[dRS21]) [Vog21a] [Api]

■ **Listing 11** Beispiel: Zeitüberschreitung

```

1  @Test
2  @Tag("fast")
3  @Timeout(value = 5, unit = TimeUnit.MILLISECONDS)
4  public void fastFibTest(){
5      assertEquals(832040, calculator.fastFib(30));
6      assertEquals(832040, calculator.ans);
7      // sehr schnell
8      assertEquals(102334155, calculator.fastFib(40));
9      assertEquals(102334155, calculator.ans);
10 }
11
12
13 // gutes Beispiel fuer einen Black-Box Test
14 @Test
15 @Tag("very_slow")

```

Testframeworks

```
16 public void fibTest(){
17     // oder assertTimeoutPreemptively nutzen
18     assertTimeout(Duration.ofMillis(10),() -> {
19         assertEquals(832040, calculator.fib(30));
20         assertEquals(832040, calculator.ans);
21         // sehr langsam
22         assertEquals(102334155, calculator.fib(40));
23         assertEquals(102334155, calculator.ans);
24     });
25 }
26 }
```

■ Hierarchische Tests:

Mit den dynamischen Tests von JUnit 5 ist es möglich, Tests zur Laufzeit zu definieren. Auf diese Weise können Tests aus Parametern, externen Datenquellen oder einfachen Lambda-Ausdrücken erstellt werden. Diese Tests eignen sich besonders für hierarchische Daten, unterstützen aber `@BeforeEach` und `@AfterEach` nicht.

Es gibt `DynamicTest` und `DynamicContainer`, beide implementieren das Interface `DynamicNode`. Erstere ist eine Wrapper-Klasse für einen einzelnen Test und letztere ein Container für mehrere dynamische Tests. `@TestFactory` annotiert Methoden, die eine einzige `DynamicNode` oder einen Iterator, Iterable oder Stream von `DynamicNode` zurückgeben müssen. Wenn wir eine hierarchische Datenstruktur haben und für jedes Element in dieser Struktur einen Test generieren möchten, dann nennen wir die einzelnen Elemente Knoten. Beim Durchlaufen der Struktur müssen wir allgemein für jeden Knoten drei Schritte ausführen:

1. `DynamicTest`-Instanzen erstellen, um das Verhalten des Knotens zu testen.
2. Einen `DynamicContainer` für jedes Kind des Knotens erstellen.
3. Einen `DynamicContainer` für den Knoten selbst erstellen, um die zuvor erstellten Tests und Container zusammenzufügen.

Auf diese Art, wenn das korrekte Verhalten eines Knotens vom korrekten Verhalten seiner Kinder abhängt, kann man den Weg fehlgeschlagener Tests bis zur Ursache verfolgen. [dRS21] [Api] [Par18a]

Betrachten wir nun die Klassen und das Interface in Listing 12, Listing 13 und Listing 14, dann können wir hierarchische Tests wie in Listing 15 schreiben.

■ Listing 12 Repräsentiert Zahlen

```
1 public class Number implements ArithmeticExpression{
2
3     private final int value;
4
5     public Number(int value){
6         this.value = value;
7     }
8
9     @Override
10    public int evaluate() {
11        return this.value;
12    }
13
14 }
```


■ Listing 13 Repräsentiert Addition

```

1 public class Addition implements ArithmeticExpression{
2
3     ArithmeticExpression exp1;
4     ArithmeticExpression exp2;
5
6     public ArithmeticExpression getExp1() {
7         return exp1;
8     }
9
10    public ArithmeticExpression getExp2() {
11        return exp2;
12    }
13
14    public Addition(ArithmeticExpression exp1, ArithmeticExpression exp2) {
15        this.exp1 = exp1;
16        this.exp2 = exp2;
17    }
18
19    @Override
20    public int evaluate() {
21        return exp1.evaluate() + exp2.evaluate();
22    }
23
24 }

```

■ Listing 14 Repräsentiert arithmetische Ausdrücke

```

1 public interface ArithmeticExpression {
2
3     int evaluate();
4
5 }

```

■ Listing 15 Testklasse für arithmetische Ausdrücke mit @TestFactory

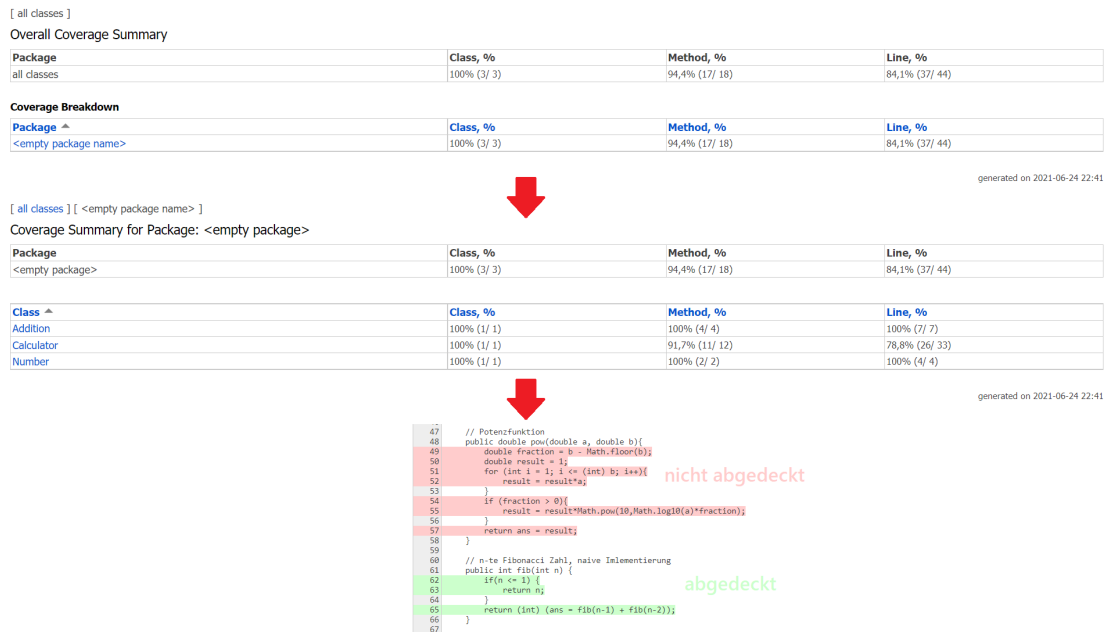
```

1 import org.junit.jupiter.api.DynamicContainer;
2 import org.junit.jupiter.api.DynamicTest;
3 import org.junit.jupiter.api.DynamicNode;
4 import org.junit.jupiter.api.TestFactory;
5 import java.util.List;
6 import java.util.stream.Stream;
7 import static org.junit.jupiter.api.Assertions.*;
8
9 class ArithmeticExpressionTest {
10
11     // (1 + (2 + 2))
12     ArithmeticExpression testExpression_0 = new Addition(new Number(1), new Addition(new
13         ↪ Number(2), new Number(2)));
14     // (2 + (2 + 2))
15     ArithmeticExpression testExpression_1 = new Addition(new Number(2), new Addition(new
16         ↪ Number(2), new Number(2)));
17
18     // flache Listen der Bäume der erwarteten Ergebnisse von evaluate()
19     List<Integer> expecteds_0 = List.of(5, 1, 4, 2, 2);
20     List<Integer> expecteds_1 = List.of(5, 1, 4, 2, 2);

```

Testframeworks

```
19
20 // aktuelle Position im Baum/in der Liste
21 int currentIndex = 0;
22
23 // der hierarchische Test
24 @TestFactory
25 public Stream<DynamicNode> test(){
26     DynamicNode dn_o = buildTestTree(testExpression_o, expecteds_o);
27     // Position zuruecksetzen
28     currentIndex = 0;
29     DynamicNode dn_1 = buildTestTree(testExpression_1, expecteds_1);
30     return Stream.of(dn_o,dn_1);
31 }
32
33 // baut den test Baum
34 public DynamicNode buildTestTree(ArithmeticExpression ae,
35                                 List<Integer> expecteds) {
36     int expected = expecteds.get(currentIndex++);
37     if (ae.getClass() == Addition.class) {
38         return DynamicContainer.dynamicContainer("addition",
39         Stream.of(DynamicTest.dynamicTest("result should be " + expected,
40         () -> {
41             int actual = ae.evaluate();
42             assertEquals(expected, actual);
43         })),
44         buildTestTree(((Addition) ae).getExp1(), expecteds),
45         buildTestTree(((Addition) ae).getExp2(), expecteds)));
46     }
47     return DynamicTest.dynamicTest("number should be " + expected, () -> {
48         int actual = ae.evaluate();
49         assertEquals(expected, actual);
50     });
51 }
52
53 }
```



■ **Abbildung 2** Testbericht

■ Testabdeckung

In Abbildung 2 ist ein Testbericht über die Testabdeckung, der mit IntelliJ erstellt wurde. Für mehr Information darüber, wie man den Bericht erstellt und liest, empfehlen wir folgende Quelle: <https://www.jetbrains.com/help/idea/code-coverage.html>

2.3 Fazit

JUnit ist an sich eine kompakte Testframework für Java, die den Entwicklern zahlreiche essenzielle Funktionalitäten beim Testen anbietet. Parametrisierte, sowie dynamische Tests sind dabei eine große Stärke von Jupiter. Sie ist außerdem sehr populär in der Praxis[ZM17], gut dokumentiert und kann in den am verbreitetsten IDEs integriert werden[dRS21], um Unit-Tests während des agilen Entwicklungsprozesses durchzuführen. Natürlich ist JUnit alleine aber nicht immer alles, was ein Entwickler von solch einer Framework erwartet, aber dafür ist sie jedoch erweiterbar, was heißt, dass Entwickler selbst eine andere Framework entwickeln können, die auf JUnit basiert und dabei neue Funktionen implementiert, die dann ihren Bedürfnissen besser dient, was einen großen Spielraum für Programmierer schafft.

3 TestNG

3.1 Einleitung

TestNG ist ein Open-Source Framework zum Testen von Java Programmen. Es baut auf bekannten Konzepten aus JUnit und NUnit auf und ergänzt diese durch neue Funktionalitäten. TestNG wurde unter anderen von Cédric Beust entwickelt um alle Kategorien von Tests abzudecken und ist besonders geeignet für automatisierte Unit-Tests einzelner Klassen und Methoden.

Das Testframework wird in allen wichtigen Java IDEs, wie Eclipse, IntelliJ IDEA und NetBeans IDE, entweder direkt oder durch Plugins unterstützt. Somit ist es sehr flexibel einsetzbar in verschiedenen Anwendungsbereichen und durch das Open-Source Konzept auch leicht zugänglich.

3.2 Neue Funktionalitäten

TestNG erweitert das sehr bekannte Framework JUnit um einige neue Funktionalitäten, die im folgenden Abschnitt genauer beschrieben werden. Dazu gehören verschiedene Annotations, die XML-Konfiguration, die Übergabe von Parametern an eine Testmethode mittels einem DataProvider, die Möglichkeit einzelne Testfälle zu gruppieren und Testfälle in einem oder mehreren Threads ablaufen zu lassen.

3.2.1 Annotations

TestNG bietet verschiedene Annotations um Methoden innerhalb der Testklasse zu definieren, die in Tabelle 3 genauer beschrieben werden.

■ Tabelle 3

Annotation	Beschreibung
@Test	Definiert eine Methode als Testmethode.
@BeforeClass	Definiert eine Methode, die vor Beginn des ersten Testfalls in der aktuellen Klasse einmal ausgeführt wird.
@AfterClass	Definiert eine Methode, die nach allen Testfällen der aktuellen Klasse einmal ausgeführt wird.
@BeforeMethod	Definiert eine Methode, die vor jeder Testmethode ausgeführt wird.
@AfterMethod	Definiert eine Methode, die nach jeder Testmethode ausgeführt wird.
@BeforeTest	Definiert eine Methode, die vor jedem Testfall ausgeführt wird.
@AfterTest	Definiert eine Methode, die nach jedem Testfall ausgeführt wird.
@BeforeSuite	Definiert eine Methode, die vor allen Tests in der aktuellen Suite einmal ausgeführt wird.
@AfterSuite	Definiert eine Methode, die nach allen Tests in der aktuellen Suite einmal ausgeführt wird.
@BeforeGroup	Definiert eine Methode, die vor einem Test, der zu einer der angegebenen Gruppen gehört, einmal ausgeführt wird.
@AfterGroup	Definiert eine Methode, die nach dem letzten Test, der zu einer der angegebenen Gruppen gehört, einmal ausgeführt wird.

Hierbei ist anzumerken, dass die Annotations *@BeforeClass* und *@AfterClass* in JUnit *@BeforeAll* und *@AfterAll* entsprechen, sowie *@BeforeTest* und *@AfterTest* den Annotations *@BeforeEach* und *@AfterEach*. Weitere in der Tabelle genannten Annotations, wie *@BeforeSuite* oder *@BeforeGroup* sind eine Erweiterung und somit in dieser Art nicht bei JUnit vorhanden.

3.2.2 XML-Datei

Ein wesentlicher Bestandteil von TestNG ist die Konfiguration der Tests mittels einer XML-Datei. Sie dient dabei der Definition von Tests und sogenannten Suites, wobei ein Suite mehrere Testfälle enthalten kann. Viele der zuvor genannten Annotations können alternativ mithilfe von XML-Konfigurationen umgesetzt werden um Testfälle zu definieren, Parameter der Testmethode zu übergeben oder die Tests auf beliebig vielen Threads durchführen zu lassen. Zudem lässt sich hier die Gruppierung von unterschiedlichen Testfällen verwalten. In der XML-Datei kann zum Beispiel angegeben werden, dass nur eine bestimmte Gruppe von Tests durchgeführt werden soll und dabei kann eine weitere Gruppe ausgeschlossen werden. Außerdem kann die Anzahl an Threads auf denen die

Testframeworks

Testfälle ablaufen sollen, bestimmt werden.

Mehr zum genauen Aufbau der Datei wird im Abschnitt 3.3 Beispiele beschrieben.

3.2.3 DataProvider

Parametrisierte Tests sind bereits in JUnit eine wichtige Funktionalität. TestNG erweitert diese durch die Möglichkeit Parameter mittels einem DataProvider während der Laufzeit zu übergeben. Dieser wird mit der Annotation `@DataProvider(name = "dataProvider")` gekennzeichnet und es wird der Name `dataProvider` zugewiesen. Der Testmethode, die den DataProvider als Parameter übergeben bekommen soll, muss dieser Name mit `@Test(dataProvider = "dataProvider")` zugewiesen werden. Die Methode bekommt ein `Object[][]`-Objekt übergeben, wobei jedes `Object[]` eine Liste von Parametern für die Methode darstellt. Somit führt jede neue übergebene Liste von Parametern zu einem weiteren Testlauf während der Laufzeit. Das hat die Vorteile, dass die Datenabdeckung von Tests deutlich erhöht werden kann und bietet die Möglichkeit datengetriebene Testläufe durchzuführen.

3.2.4 Gruppierung von Tests

TestNG bietet im Gegensatz zu JUnit eine flexible Einteilung der Testmethoden in verschiedene Gruppen und Untergruppen mittels Annotations und XML-Konfigurationen. Dies bietet die Möglichkeit unterschiedliche Teststufen, wie Modultests, Integrationstests oder Akzeptanztests, voneinander abzugrenzen und separat voneinander durchzuführen. Bei der Ausführung der Testfälle ist es möglich in der XML-Datei gewisse Gruppen miteinzubeziehen und andere auszuschließen, also bestimmte Testfälle in einer Gruppe oder Untergruppe durchzuführen und andere nicht. Zudem wird es benötigt um auf Testmethoden in unterschiedlichen Klassen zugreifen zu können. Abschließend bietet TestNG hier eine sehr flexible Gruppierung von Testmethoden mit möglichen Untergruppen, was den großen Vorteil mit sich bringt, dass keine Neukompilierung benötigt wird, wenn man zwei verschiedene Mengen von Testfällen hintereinander ausführen möchte.

3.2.5 Multithread-Testing

Mit TestNG lassen sich bestimmte Teile einer Software darauf überprüfen, ob sie mehrere Aufgaben, Berechnungen, Anweisungen oder Befehle parallel ausgeführt werden können mit Multithread-Testing. Das ist ein neues Feature, was in JUnit so nicht mit enthalten ist. Sowohl über Annotations als auch mittels XML-Konfiguration lassen sich bestimmte Testmethoden oder Testgruppen in beliebig vielen Threads ausführen. In der XML-Datei ist es dabei auch möglich für ganze Suites eine Anzahl an Threads anzugeben, in denen die enthaltenen Testfälle ablaufen sollen. Dabei kann man die Anzahl der Threads variieren zwischen einem einzigen Thread, wobei alle Testfälle sequenziell ablaufen würden und mehreren Threads um eine parallele Ausführung der Tests zu erreichen. Multithread-Testing kann auch dafür benutzt werden mehrere Instanzen einer einzigen Testmethode in mehreren Threads parallel laufen zu lassen, was zum Beispiel bei parametrisierten Tests auftritt, wo mit einem DataProvider Daten während der Laufzeit übergeben werden.

3.3 Beispiele

Um die Erweiterungen von TestNG zu JUnit und deren neuen Funktionalitäten genauer zu beschreiben, werden nun einige Code-Beispiele zu Testfällen in TestNG vorgestellt. Hierfür wird die selbe Calculator-Klasse aus den Beispielen im Abschnitt von JUnit getestet.

3.3.1 Annotations

■ **Abbildung 3** Annotations

```

1  import org.testng.annotations.*;
2  import static org.testng.Assert.assertThrows;
3  import static org.testng.AssertJUnit.assertEquals;
4
5  class CalculatorTest {
6      private static Calculator calculator;
7      @BeforeClass
8      public static void start(){
9          calculator = new Calculator();
10         System.out.println("started testing...");
11     }
12     @AfterClass
13     public static void finish(){
14         System.out.println("ended testing...");
15     }
16     @BeforeMethod
17     public void preparation(){
18         System.out.println("forgot: " + calculator.memory);
19         calculator.forget();
20     }
21     @AfterMethod
22     public void finalization(){
23         calculator.memorize();
24         System.out.println("remembered: " + calculator.memory);
25     }

```

In diesem Code-Beispiel in Abbildung 3 ist ein Teil der Klasse CalculatorTest zu sehen, in der mit TestNG die Klasse Calculator getestet wird. Zu Beginn ist die Methode `start()` mit der Annotation `@BeforeClass` gekennzeichnet und die Methode `finish()` mit `@AfterClass`. Somit wird `start()` nur einmal vor dem ersten Testfall in dieser Klasse und `finish()` nur einmal nach dem letzten Testfall in dieser Klasse ausgeführt. Die Methoden `preparation()` und `finalization()` werden jeweils vor oder nach jeder Testmethode, die in der Klasse aufgerufen wird, ausgeführt.

■ **Abbildung 4** @Test-Annotation

```

70     @Test(enabled = false, priority = 10)
71     public void subTest(){
72         assertEquals( expected: 0.0, calculator.sub( a: 1, b: 1));
73     }

```

In Abbildung 4 ist ein konkreter Testfall der `sub(x, y)` Methode in der Klasse Calculator zu erkennen. Dieser wird mit der Annotation `@Test` gekennzeichnet. Mithilfe von `@Test`

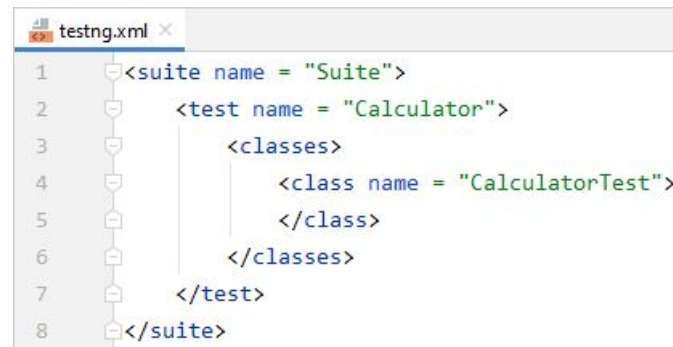
Testframeworks

kann man eine Testmethode noch genauer beschreiben und dieser neue Eigenschaften zuweisen, wie hier zum Beispiel die Priorität von 10. Mit *enabled = false* wird der Testfall bei der Ausführung der Testklasse ignoriert. Das sind nur zwei Beispiele von allen Möglichkeiten eine Testmethode innerhalb der *@Test*-Annotation genauer zu definieren.

3.3.2 XML-Datei

Eine sehr einfache XML-Datei ist in folgender Abbildung 5 zu sehen.

■ **Abbildung 5** XML-Datei



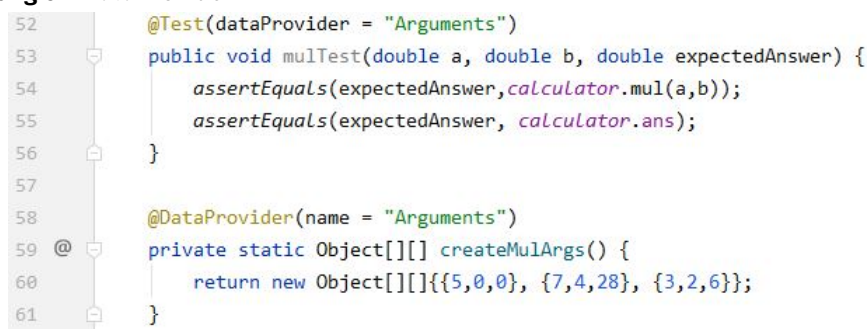
```
1 <suite name = "Suite">
2   <test name = "Calculator">
3     <classes>
4       <class name = "CalculatorTest">
5     </class>
6   </classes>
7 </test>
8 </suite>
```

Die Datei beginnt mit einer Suite-Umgebung und deren Namenszuweisung, woraufhin ein Testfall beschrieben wird, der in diesem Fall "Calculator" genannt wird. Es folgt eine *classes*-Umgebung in der die Testklasse mit *classname = "CalculatorTest"* angegeben ist. Da hier der Name der Klasse angegeben wird und kein konkreter Testfall genauer beschrieben ist, werden beim Ausführen der XML-Datei alle Testmethoden in der Klasse *CalculatorTest* ausgeführt.

Weitere Variationen von XML-Konfigurationen werden in den folgenden Unterabschnitten anhand von Beispielen erläutert.

3.3.3 DataProvider

■ **Abbildung 6** DataProvider



```
52 @Test(dataProvider = "Arguments")
53 public void mulTest(double a, double b, double expectedAnswer) {
54     assertEquals(expectedAnswer, calculator.mul(a,b));
55     assertEquals(expectedAnswer, calculator.ans);
56 }
57
58 @DataProvider(name = "Arguments")
59 @
60 private static Object[][] createMulArgs() {
61     return new Object[][]{{5,0,0}, {7,4,28}, {3,2,6}};
62 }
```

Auf Abbildung 6 wird ein Beispiel für einen Parametrisierten Test gezeigt, dessen Datenübermittlung mittels einem DataProvider abläuft. In der Zeile 58 wird die Methode *createMulArgs()* mit der Annotation *@DataProvider* als DataProvider gekennzeichnet und in der darauffolgenden Klammer wird ihm der Name "Arguments" zugewiesen. Die Testmethode *mulTest()* wird in Zeile 52 mit der *@Test* Annotation als Test gekennzeichnet und ihr wird daraufhin der zuvor genannte DataProvider "Arguments" zugewiesen. Der

DataProvider liefert hier bei der Ausführung der Testmethode ein *Object[][]*-Objekt mit drei verschiedenen Einträgen, die in *createMulArgs()* erzeugt werden. Jeder dieser Einträge führt zu einem neuen Testlauf der Testmethode.

3.3.4 Gruppierung von Tests

■ **Abbildung 7** Gruppierung von Tests

```

83     @Test (groups = "reasonable")
84     public void powTest(){
85         assertEquals( expected: 1448.15, calculator.pow( a: 2, b: 10.5), delta: 0.01);
86     }
87
88     @Test (groups = "reasonable")
89     public void log2Test(){
90         assertEquals( expected: 3.0, calculator.log2( a: 8));
91         assertEquals( expected: 3.0, calculator.ans);
92         assertEquals( expected: 10.0, calculator.log2( a: 1024));
93         assertEquals( expected: 10.0, calculator.ans);
94         assertThrows(ArithmeticException.class, () -> calculator.log2( a: 0));
95         assertThrows(ArithmeticException.class, () -> calculator.log2( a: -8));
96     }
97
98     @Test (groups = {"fast", "reasonable"})
99     public void fastFibTest(){
100         assertEquals( expected: 832040, calculator.fastFib( n: 30));
101         assertEquals( expected: 832040.0, calculator.ans);
102         assertEquals( expected: 9227465, calculator.fastFib( n: 35));
103         assertEquals( expected: 9227465.0, calculator.ans);
104     }

```

Hier werden die drei Testmethoden *powTest()*, *log2Test()* und *fastFibTest()* abgebildet, die alle innerhalb der *@Test*-Annotation der Gruppe "reasonable" zugewiesen werden. Die Methode *fastFibTest()* wird zusätzlich noch der Gruppe "fast" zugewiesen.

Mit der dazugehörigen XML-Datei wird deutlich wie man solche Gruppen in Testläufe mit einbinden oder ausschließen kann.

■ **Abbildung 8** Gruppierung mittels XML

```

1  <suite name = "Suite">
2    <test name="Calculator" >
3      <classes>
4        <class name="CalculatorTest" />
5      </classes>
6    </test>
7    <test name="TestByGroup">
8      <groups>
9        <run>
10         <include name="reasonable" />
11         <exclude name="fast" />
12       </run>
13     </groups>
14     <classes>
15       <class name="CalculatorTest" />
16     </classes>
17   </test>
18 </suite>

```

Diese wurde um den Testfall "TestByGroup" erweitert in der die Gruppe "reasonable" ausgeführt werden soll, dabei sollen allerdings alle Testfälle, die der Gruppe "fast" angehören, ignoriert werden. Dies wird hier mit den Wörtern *include* und *exclude* beschrieben. Zudem wird innerhalb des Tests eine *groups*-Umgebung erschaffen in der mit *run* angegeben ist, welche Tests durchgeführt werden.

3.3.5 Multithread-Testing

■ Abbildung 9 Multithread-Testing

```

109      @Test (priority = 2, groups = "slow", threadPoolSize = 2)
110      public void fibTest(){
111          assertEquals( expected: 832040, calculator.fib( n: 30));
112          assertEquals( expected: 832040.0, calculator.ans);
113          // dauert sehr lange
114          assertEquals( expected: 9227465, calculator.fib( n: 35));
115          assertEquals( expected: 9227465.0, calculator.ans);
116      }

```

Um Testfälle innerhalb einer Methode auf einem oder mehreren Threads parallel ausführen zu lassen, kann man bei der `@Test`-Annotation die `threadPoolSize` festlegen, wie hier bei der Methode `fibTest()` in Zeile 109 gezeigt.

Um diese Funktionalität auf ganze Testmethoden, Testgruppen oder auch Testsuites auszuweiten, muss man die Anzahl der Threads in TestNG in der XML-Datei festlegen.

■ Abbildung 10 Multithread-Testing mittels XML

```

testng.xml
1  <suite name = "Suite" parallel = "methods" thread-count = "3" >
2      <test name="Calculator" >
3          <classes>
4              <class name="CalculatorTest" />
5          </classes>
6      </test>
7  </suite>

```

Wie in dieser Abbildung zu erkennen ist, kann man beim Erstellen der Suite-Umgebung mit `parallel = "methods"` angeben, dass die Testmethoden parallel ablaufen sollen und mit `thread-count = "3"` bestimmen, dass die Tests auf drei unterschiedlichen Threads durchgeführt werden. Für Testgruppen oder einzelne Testmethoden kann die XML-Datei analog angepasst werden.

3.4 Fazit

Insgesamt ist TestNG ein Open-Source Framework zum Testen von Java-Code, das auf bekannten Konzepten von JUnit aufbaut und um neue Funktionalitäten erweitert. Diese beinhalten insbesondere die Möglichkeit mit DataProvidern zu arbeiten, verschiedene Testfälle und -methoden flexibel miteinander zu gruppieren und Testabläufe parallel ausführen zu können auf mehreren Threads.

4 Selenium

4.1 Einleitung

Selenium ist ein open-source Testframework zur Automatisierung von Browserapplikationen und unterscheidet sich damit grundlegend von JUnit(Jupiter) und TestNG. Denn JUnit und TestNG sind manuelle Testingverfahren und spezialisieren sich eher auf die Funktionalität des Programms während Selenium oberflächenbasierend testet. Selenium beschränkt sich auf Browserapplikationen. [Sel21a]

4.1.1 Relevanz

Heutzutage werden immer mehr Softwaresysteme als Webanwendungen implementiert. Mit steigender Anzahl, steigt auch die Wichtigkeit eines effizienten Frameworks um jene Webanwendungen zu testen. Hier kommt Selenium ins Spiel. Selenium ist ein automatisiertes Testframework, welches die Funktionalität der Oberfläche testen kann.

4.1.2 Historie

Die Geschichte von Selenium beginnt 2004. Zu dieser Zeit arbeitet Jason Huggins bei der Firma "ThoughtWorks". Mit Unterstützung von Paul Gross und Jie Tina Wang programmiert er das Fundament von Selenium unter dem Namen "JavaScriptTestRunner", welches das automatische Testen von Webanwendungen ermöglicht und somit der agilen Arbeitsweise der Firma "ThoughtWorks" sehr zu Gute kommt.

Sein Kollege Paul Hamant verfolgte das Ziel, das Framework als ein Open-Source-Programm unter der Apache-2.0-Lizenz zu veröffentlichen und so entstand Selenium. Selenium wurde über die Jahre von vielen Programmierern weiterentwickelt, um zu dem bekannten, schnellen, flexiblen und verlässlichen Testframework von Heute zu werden. Die aktuellste Version ist Selenium 3. [Sel21a] [Sel21b]

4.1.3 Anwendungsweise

Selenium ist so bekannt und beliebt geworden, weil man unmittelbares und intuitives, visuelles Feedback erhält. Außerdem für sein Potenzial, als wiederverwendbares Testframework für andere Webanwendungen zu agieren. Im Wesentlichen ist Selenium eine Sammlung von Werkzeugen, um Webbrowser zu automatisieren. Es simuliert Benutzereingaben und testet, ob die Anwendung wie gewünscht reagiert. [Mol18]

4.1.4 Zielsetzung

Ziel dieses Abschnittes ist es, die Funktionalität von Selenium 3 darzustellen, zu testen und im Nachhinein mit den anderen Frameworks dieser Arbeit zu vergleichen. Die Funktionalität wird anhand von Codebeispielen, die in Java geschrieben wurden, veranschaulicht.

4.2 Überblick

4.2.1 Open-source unter Apache-2.0

Da Selenium ein Open-Source-Framework ist, das unter der Apache 2.0-Lizenz (veröffentlicht 2004) freigegeben wurde, darf jeder dieses Framework frei verwenden, modifizieren und veröffentlichen.

4.2.2 Testautomatisierung

Automatisierte Testsoftwares verfolgen eine bestimmte Strategie beim Testen, um eine stabile, sichere, nutzerfreundliche und effiziente Software zu sichern. Dabei wird auf schnelles Feedback und hohe Effizienz wert gelegt.

Für gewöhnlich wird die zu testende Software von der automatisierten Testsoftware gesteuert und nach jeder Aktion wird der aktuelle Stand ausgewertet, woraufhin sie auf Basis von festgelegten Regeln überprüft wird. Wird eine Abweichung festgestellt, wird diese für gewöhnlich notiert und der Programmierer kann anschließend manuell entscheiden, ob sie einen Fehler darstellt.

Je nach Software werden unterschiedliche Ebenen getestet. Es kann die Grafische Oberfläche (GUI oder Graphic User Interface) getestet werden, die API (Application-Programming-Interface) Schnittstellen oder man führt diese Tests direkt auf dem Code aus. Dies sind sogenannte Unit-Tests auch Modultests oder Komponententests.

Ziel der Testautomatisierung ist es nicht in erster Linie den Entwicklungsprozess zu beschleunigen, sondern die Qualität der Software zu wahren und zu fördern. [Che]

4.3 Testautomatisierung mit Selenium

Selenium ist das beliebteste Framework für Browseranwendungen, weil es unter anderem auch eine API bietet, um den Webbrowser zu automatisieren und die Tests mit mehreren Betriebssystemen und Browserkombinationen implementiert werden können.

Man schreibt Automatisierungsskripte, welche Anweisungen zur Durchführung der Tests enthalten. Einschließlich dem Lokalisieren und Manipulieren der Objekte auf der Webseite, der Verwendung von Informationen als Eingabe, der Überprüfung von Nachrichten aus der Webanwendung und der Interaktion mit Elementen. Der Selenium WebDriver bietet Schnittstellen, Klassen und Methoden für die Interaktion mit Browsern und Elementen in der Webanwendung. Die Webdriver-API wird normalerweise zum Implementieren von UI-Regressionstests und anderen Aufgaben verwendet, die über einen Webbrowser ausgeführt werden können. Browseranbieter und Drittanbieter haben Browsertreiber für die Arbeit mit der WebDriver-API entwickelt. Wodurch es Selenium möglich ist, Tests in allen gängigen Webbrowsern durchzuführen. [Sel21a][Wal]

Welche Sprachen unterstützt Selenium? Um möglichst alle Webapplikationsumgebungen zu bedienen, unterstützt die API viele verschiedene Programmiersprachen und Webbrowser. Dazu gehören Java, Ruby, Python, C# und JavaScript. Zudem sind alle bekannten Webbrowser verfügbar. [Sel21a]

Testframeworks

4.3.1 Selenium Tools [Sel21a]

Selenium bietet mehrere Tools zur Testautomatisierung.

Selenium Webdriver Eines davon ist der Selenium Webdriver. Für das Automatisieren von Webseiten werden die Webdriver APIs verwendet. Diese nutzen die von den Browserherstellern zur Verfügung gestellten APIs um den Browser zu steuern und die Tests auszuführen. Die Ausführung soll einen echten Benutzer simulieren um die Applikation zu testen, die später veröffentlicht werden soll.

Selenium IDE Die Selenium IDE (Integrated Development Environment) ist eine Entwicklungsumgebung, um die Tests zu erstellen. Es handelt sich um eine Erweiterung für Chrome und Firefox. Man kann mit dieser sehr lukrativ Testfälle erstellen indem man Benutzerinteraktionen mithilfe von Befehlen aufzeichnet. Auf diese werde ich in dieser Arbeit jedoch nicht genauer eingehen.

Selenium Grid Mithilfe von Selenium Grid können Testfälle auf verschiedenen Plattformen ausgeführt werden. Dies ist praktisch um die Webdrivertests nach der Entwicklung auf mehreren Browser- und Betriebssystemkombinationen ausführen zu können.

4.3.2 Selenium verwenden

Funktionale Endbenutzertests wie die, die mithilfe des Selenium-Frameworks geschrieben werden sind sehr teuer in der Umsetzung, da sie in der Regel eine umfangreiche Infrastruktur benötigen um effektiv betrieben werden zu können. Wenn die Webanwendung auch mithilfe von Komponententests (Unit-Tests) getestet werden kann, ist es ratsam die Tests auf dieser niedrigeren Ebene zu halten.

Sollte man sich dafür entschieden haben auf funktionaler Ebene zu testen, so sollte man nach folgender Reihenfolge vorgehen:

Workspace einrichten Wie bei allen Programmertätigkeiten bietet es sich auch beim Programmieren automatisierter Seleniumtests an eine IDE (Integrated Development Environment) zu verwenden. In dieser Arbeit wurde "IntelliJ IDEA Community Edition" verwendet. Es ist nun auch Zeit sich für eine der verfügbaren Programmiersprachen zu entscheiden. Die folgenden Code-Beispiele wurden, wie bereits erwähnt, in Java implementiert. Es wird empfohlen, sich das entsprechende Build-Management-Tool, in diesem Fall Maven, herunterzuladen. Als letztes fehlen noch die Installation der entsprechenden Browsertreiber (in meinem Fall der Chrome-treiber) und das legen der Dependencies für Selenium in *Maven* oder Ähnlichem. [Mol18]

4.3.3 7 Aktionen zum Selenium Skript [Mat19]

1. Das Instanziieren eines Webdriver-Objekts, um den Browser zu steuern.

```

1 @Test
2 public void chromeTest(){
3     //set location of chromedriver
4     System.setProperty("webdriver.chromedriver","ressources/chromedriver.exe");
5     //start session (opens browser)
6     Webdriver = new ChromeDriver();

```

2. Das Navigieren zu der Webseite, die getestet werden soll.

```

7     //Navigate to a website
8     driver.get("https://example.cypress.io/");

```

3. Das Lokalisieren des zu testenden Elements auf der Webseite. Es gibt verschiedene Möglichkeiten ein Element zu lokalisieren. Google Chrome bietet hierzu eine IDE und Selenium hat auch eine eigene IDE entwickelt.

```

9     //Find element
10    driver.findElement(By.id("sign-in"));

```

4. Sicherstellen, dass der Browser in der Lage ist mit dem Element zu interagieren.

```

11    //find element umwandeln in:
12    WebdriverWait wait = new WebdriverWait(driver,10);
13    WebElement signIn = wait.until(
14        ExpectedConditions.presenceOfElementLocated(By.id("sign-in")));

```

5. Tests auf dem Element durchführen.

```

15    // Aktion auf dem Element durchfuehren
16    signIn.click();

```

6. Die Testergebnisse aufnehmen.

```

1 //Testwatcher comes with JUNIT
2 @Rule
3 public TestWatcher watcher = new TestWatcher(){
4     @Override
5     protected void failed(Throwable e, Description description){
6         System.out.println(description.getMethodName() + ": Failed");
7     }
8     @Override
9     protected void succeeded(Description description){
10        System.out.println(description.getMethodName() + ":Succeeded");
11    }
12 };

```

7. Den Driver und damit den Browser schließen.

```

17    driver.quit();
18 }

```

Testframeworks

Die Bibliothek WebDriverManager WebDriverManager ist eine Bibliothek, die es uns ermöglicht, die Verwaltung der von Selenium WebDriver benötigten Treiber zu automatisieren. Wie man diese in ein Projekt einfügt und benutzt steht unter <https://github.com/bonigarcia/webdrivermanager/>. Der Aufbau eines SeleniumTests ist also unter Verwendung von JUnit- Annotations und der Bibliothek WebDriverManager folgender:

```
1 public class SeleniumTest{
2     private WebDriver driver;
3     @BeforeClass
4     public static void setupDriver() {
5         WebDriverManager.chromedriver.setup();
6     }
7     @Before
8     public void setupTest() {
9         driver = new ChromeDriver();
10    }
11    @After
12    public void teardown(){
13        driver.quit();
14    }
15    @Test
16    public void test(){...}
17 }
```

Durch die Annotations wird der Code schöner, verliert an Duplikationen und wird automatisierter. [bon]

4.3.4 Lokalisieren von Elementen mithilfe von Chrome:

Im Chromebrowser können mit F12 die Entwicklertools geöffnet werden. Mit einem Klick auf den Pfeil oben links im Entwicklerfenster oder mit der Tastenkombination ctrl + shift + c wechselt man in den Modus, in dem man Elemente auf den Bildschirm auswählen kann. Anschließend werden entsprechender html und css-code markiert und dort kann dann bspw. die *id* eines Elements abgelesen werden. Mithilfe dieser kann daraufhin das Element im Selenium-Skript lokalisiert werden, um dann Tests darauf durchzuführen. Außerdem kann mit ctrl + f eine Suchleiste geöffnet werden, in der man die html-Datei nach strings, selektoren oder XPath's durchsuchen kann.

Kurze Einführung in html: Jedes Element in einem html Dokument besteht aus einem Starttag

< > und einem Endtag </>. Die Tags haben Bezeichnungen wie: "html", "head", "title", "base", "meta", "link", "style", usw. Alle html-tags findet man unter 4.5.1.

Außerdem stehen innerhalb des Starttags noch mögliche Attribute, die den tag beschreiben. Diese Attribute bestehen jeweils aus einem Attributnamen und einem Wert. Häufige Beispiele hierfür sind: "id", "class", "style", "title", usw. Alle html-Attribute findet man unter 4.5.1.

Syntax: <tag attribut1="Wert1" attribut2="Wert2" ...> Inhalt </>.

[HTM21] [HTM20]

Selenium Lokatoren: In Selenium können Elemente mithilfe von acht verschiedenen Locators identifiziert werden. Diese sind: “id”, “name”, “cssSelector”, “className”, “tagName”, “linkText”, “partialLinkText” und “xpath”. Zur Umsetzung wird in diesem Codebeispiel “id” durch den entsprechenden Locator ersetzt. Es ist zu beachten, dass id’s die besten Locators sind, da diese immer einzigartig sein sollten. Die nächstbesten Locatoren wären xpaths und cssSelectors. Zu beachten ist, dass “id”, “name”, “className”, “tagName”, “linkText” und “partialLinkText” alles Attribute sind, die ein html-element beschreiben. “cssSelector” und “xpath” hingegen sind Kombinationen aus einem oder keinem “tag-name” und keinem, einem oder mehreren Attributen eines Elements. Diese werden verwendet um eindeutige *Locators* zu definieren, wenn es zum Beispiel keine id gibt oder diese doch nicht eindeutig sein sollte. Sie haben ihre eigene Syntax! [Sel21a][Ang]

■ **Tabelle 4** XPath - Mehr Informationen unter 4.5.1

XPath Beispiele	Beschreibung
<code>//img[@id='img1']</code>	sucht images mit der id ‘img1’.
<code>//*[contains(text(), "Subscribe")]</code>	sucht alle elemente mit inneren Text, der “Subscribe” enthält.
<code>//*[starts-with(@id, 'Id')]</code>	sucht alle Elemente, deren ID mit ‘Id’ startet.
<code>//*[ends-with(@id, 'Id')]</code>	sucht alle Elemente, deren ID auf ‘Id’ endet.
<code>//*[matches(@id, 'r*')]</code>	sucht alle Elemente, deren ID unter die <i>Regular Expression</i> ‘r*’ fallen.
<code>//div/child::parent::div</code>	sucht das erste child/parent von div, dass ein div ist.

■ **Tabelle 5** CssSelector - Mehr Informationen unter 4.5.1

CssSelector Beispiel	Beschreibung
<code>#button1</code>	sucht Element mit der id ‘button1’.
<code>input.form_input</code>	sucht ein Element mit dem input-tag, das den Klassennamen form_input hat.
<code>form > input</code>	sucht ein Inputelement, das ein direktes child von einem Formelement ist.
<code>input[id='button1']</code>	sucht ein input element mit der id ‘button1’ .
<code>ul[@id]</code>	ul-elemente mit id als Attribut
<code>*[name='button1'][value='cambria']</code>	sucht alle elemente mit name=‘button1’ und value=‘cambria’.

4.3.5 Mögliche Aktionen auf lokalisierten Elementen: [Sel21a][Met]

Im folgenden sind die wichtigsten und am häufigsten verwendeten Methoden aufgelistet, die Selenium anbietet.

■ **Tabelle 6** Methoden auf Webelementen:

Elementinteraktionen	Beschreibung
void click()	Simuliert das Klicken eines Buttons, eines Links oder eines Radiobuttons.
void clear()	Löscht alles aus einem veränderbaren Element.
void submit()	Reicht ein Element ein. Bsp: Suchfeld in Google nach dem reinschreiben: submit()
Informationenlieferer	Beschreibung
String getText()	Ermittelt den Text eines Elements.
String getTagName()	Ermittelt den TagName eines Elements.
String getAttribute(String name)	Ermittelt den Wert des Attributes "name" eines Elements.
boolean isDisplayed()	Prüft ob ein Element auf dem Display angezeigt wird.
boolean isEnabled()	Prüft, ob das Element aktiviert ist.
boolean isSelected()	Prüft, ob das Element ausgewählt ist.

Erweiterte Elementinteraktionen Selenium hat außerdem eine Actions Klasse mit der Maus- und Tastaturaktionen simuliert werden können. Die folgenden Methoden werden auf Actions-Elemente ausgeführt.

■ **Tabelle 7** Erweiterte Elementinteraktionen

<code>void build()</code>	Generiert eine zusammengesetzte Aktion, die alle bisher ausgeführten Aktionen enthält, die ausgeführt werden können.
<code>void perform()</code>	Führt eine Aktion aus.
Mausaktionen	Beschreibung
<code>void click(WebElement target)</code>	Simuliert das Klicken in der Mitte des Elements.
<code>void clickAndHold(WebElement target)</code>	Simuliert das Klicken (ohne Loszulassen) in der Mitte des Elements.
<code>void contextClick(WebElement target)</code>	Simuliert einen Rechtsklick auf ein Element.
<code>void doubleClick(WebElement target)</code>	Simuliert einen Doppelklick auf ein Element
<code>void dragAndDrop(WebElement source, WebElement target)</code>	Führt <code>clickAndHold(source)</code> aus, zieht das Element anschließend zu den der Lage von <code>target</code> und simuliert dort das Loslassen der Maustaste.
<code>void dragAndDropBy(WebElement source, int xOffset, int yOffset)</code>	Führt <code>clickAndHold(source)</code> aus, zieht das Element anschließend zu den dem gegebenen Offset und simuliert dort das Loslassen der Maustaste.
<code>void moveToElement(WebElement target)</code>	Simuliert das Bewegen der Maus zu der Mitte eines Elements.
<code>void moveToElement(WebElement target, int xOffset, int yOffset)</code>	Simuliert das Bewegen der Maus zu einem Offset (von oben links) eines Elements.
<code>void release()</code>	Simuliert das Loslassen der linken Maustaste.
Tastaturaktionen	Beschreibung
<code>void keydown(CharSequence key)</code>	Simuliert das Drücken einer modifizierten Taste
<code>void keyup(CharSequence key)</code>	Simuliert das Loslassen einer modifizierten Taste
<code>void sendKeys(CharSequence... keysToSend)</code>	Schreibt eine gegebene <code>charSequence</code> in ein veränderbares Element.

Beispiel für erweiterte Elementinteraktionen mit der Actions Klasse von Selenium:

```

1 WebElement element = driver.findElement(By.id("log-in"));
2 Actions action = new Actions(driver);
3 action.click(element).build().perform();

```

4.3.6 JavaScriptExecutor

JavaScriptExecutor ist ein Interface von Selenium, welches es uns ermöglicht JavaScript-Code mithilfe der Objektmethode *executeScript* auf dem aktiven Browserwindow auszuführen. Um dieses zu benutzen, deklarieren wir ein Objekt der Klasse *JavaScriptExecutor* und casten unseren *WebDriver* darauf. Das Interface implementiert zwei Methoden. *executeScript()* und *executeAsyncScript()*, wobei

```
1 JavascriptExecutor js = (JavascriptExecutor) driver;  
2 js.executeScript("javascriptcode;");
```

Dieses ermöglicht es uns viele Dinge zu tun. Unter anderem können der Style von Elementen mithilfe von JavaScript den beeinflusst werden. So kann zum Beispiel in ein Dokument rein- und rausgezoomt werden. Außerdem kann nun auch das Element direkt geklickt werden, anstelle von der Position des Elements, wie bei *WebElement.click()*. Jene Funktion stellt sich vor allem bei der *ElementClickInterceptedException*, als nutzvoll dar, denn diese wird geworfen, wenn das Element, welches geklickt werden soll durch ein anderes blockiert wird. Mithilfe von *javascript* wird diese umgangen, denn es wird das Element selber geklickt. Im folgenden eine kleine Vorschau der erwähnten Möglichkeiten. Natürlich gibt es noch viel mehr wertvolle Beispiele, um javascript auf Webseiten anzuwenden.

```
3 js.executeScript("arguments[0].click();", element); //klicken eines Elements  
4 js.executeScript("window.scrollTo(0,50);"); //scrollt die Webseite um 50 Pixel nach unten  
5 js.executeScript("document.body.style.zoom='40%';"); //zoomt raus auf 40 %
```

[Jav]

4.3.7 Windows, Frames und Alerts [Mul][IFr][Fra]

Windows Sobald auf einen Link gedrückt wird, ein neues Tab oder ein neues Fenster geöffnet wird, arbeitet unser *WebDriver* mit mehreren *Windows*. Um zwischen diesen zu switchen gibt es für *WebDriver* folgende Objektmethoden:

■ **Tabelle 8** Windows

WebDriver-Objektmethoden - Windows	Beschreibung
<code>.getWindowHandle();</code>	gibt einen String, der das aktuelle <i>Window</i> beschreibt zurück
<code>.getWindowHandles();</code>	gibt ein Set von Strings, die jeweils ein <i>Window</i> beschreiben zurück.
<code>.getTitle();</code>	gibt den Titel des aktuellen <i>Windows</i> als String zurück.
<code>.getCurrentUrl();</code>	gibt die url des aktuellen <i>Windows</i> als String zurück.
<code>.close();</code>	schließt das aktuelle <i>Window</i> .
<code>.quit();</code>	Schließt den Webtreiber.

Frames Frames kennzeichnen Teilbereiche in einer html-Datei. Hierbei gehören mehrere Frames zu einem Frameset. Dies kann zu Komplikationen führen, wenn nicht klar ist, wie damit umzugehen ist. Denn das Wechseln in den richtigen Frame ist essentiell, um dessen Elemente anzusprechen. Oftmals wird eine *NoSuchElementException* geworfen, weil die Elemente, die angesprochen werden sollen, sich nicht im aktuellen Frame befinden und Selenium das Element somit nicht finden kann. Um zwischen Frames zu switchen und diese zu behandeln gibt es folgende Methoden für den WebDriver:

■ **Tabelle 9** Frames

WebDriver-Objektmethoden - Frames	Beschreibung
<code>.switchTo().frame(o);</code>	wechselt den Fokus zum <code>i</code> .Frame (nach Index)
<code>.switchTo().frame("frame-name");</code>	wechselt den Fokus zu dem Frame mit dem angegebenen Namen
<code>.switchTo().parentFrame();</code>	wechselt den Fokus zu dem parent-frameobjekt des aktuellen frames
<code>.switchTo().defaultContent();</code>	wechselt den Fokus zurück auf das ursprüngliche Fenster.

Alerts Alerts sind Benachrichtigungen an den Nutzer, die in einer Webseite integriert sind. Um mit diesen umzugehen gibt es folgende Methoden für den WebDriver:

■ **Tabelle 10**

WebDriver-Objektmethoden - Alerts	Beschreibung
<code>.switchTo().alert().accept();</code>	wechselt den Fokus zu dem aktuellen Alert und akzeptiert diesen
<code>.switchTo().alert().dismiss();</code>	wechselt den Fokus zu dem aktuellen Alert und lehnt diesen ab
<code>.switchTo().alert().getText();</code>	wechselt den Fokus zu dem aktuellen Alert und gibt die Alert-Message zurück
<code>.switchTo().alert().sendKeys();</code>	wechselt den Fokus zu dem aktuellen Input-Alert und schreibt eine CharSequenz in dessen Inputfeld.

Siehe dazu auch das Beispiel zu Alerts, Frames und Windows.

4.3.8 Downloads und Uploads [Upl]

Download Das Uploaden von Dateien lässt sich durch deren Dateipfad realisieren. Dieser wird mithilfe von *sendKeys()* an das entsprechende Input-Feld übergeben.

Upload Das Downloaden von Dateien kann Selenium alleine nicht automatisieren. Jedoch bietet *Wget* die Möglichkeit die Dialogfenster zum Hernuterladen der Dateien zu umgehen. *Wget* wird dann das Herunterladen durchführen. Wie genau das funktioniert findet ihr hier.

4.3.9 Auswerten von Tests

Um Tests auszuwerten werden die Assert-Methoden von JUnit genutzt. Diese werden verwendet, um bspw. zu schauen ob ein Element, dass vorher nicht angezeigt wurde nun angezeigt wird. Oder ob ein bestimmter Text angezeigt wird. Aber letzten Endes kann der Entwickler selbst und flexibel entscheiden, wie er die Tests auswertet. Sehen Sie dazu, die Assert-Methoden hier.

```
6 WebElement element = driver.findElement(By.id("id"));
7 Assert.assertTrue("Element is not displayed", element.isDisplayed());
8 Assert.assertTrue("Element does not display the expected Test",
9     "Text".equals(element.getText()));
```

4.4 Tests

Eine Webseite, auf der grundsätzlich alle Browserinteraktionen getestet werden können, ist demoqa. Auf dem ersten Blick fällt direkt auf, dass diese Seite für das Lernen von Selenium programmiert wurde. Sie bietet eine Auswahl an Überkategorien, die gemeinsam alle grundlegenden Webelemente beinhalten, die mit Selenium getestet werden sollten. Zum Anfang wird wieder die Grundstruktur des Tests gebildet. Dazu werden wichtige Variablen definiert und es werden JUnit-Annotations verwendet, um den Treiberstart und das Ende zu definieren. Außerdem werden die, zu Beginn definierten, Variablen nun initialisiert.

```

1 public class demoqaTests {
2     public WebDriver driver;
3     JavascriptExecutor js;
4     Actions action;
5
6     WebElement widgets;
7     WebElement forms;
8     WebElement alertsFrameWindows;
9     WebElement elements;
10    WebElement interactions;
11
12    @BeforeClass
13    public static void setupDriver() {
14        WebDriverManager.chromedriver().setup();
15    }
16
17    @Before
18    public void setupTest() throws InterruptedException {
19        driver = new ChromeDriver();
20        driver.get("https://demoqa.com");
21
22        widgets = driver.findElement(By.xpath("//div[@class='card mt-4 top-card'][.='Widgets']"));
23        forms = driver.findElement(By.xpath("//div[@class='card mt-4 top-card'][.='Forms']"));
24        alertsFrameWindows = driver.findElement(By.xpath("//div[@class='card mt-4 top-card'][.='
        ↳ Alerts, Frame & Windows']"));
25        elements = driver.findElement(By.xpath("//div[@class='card mt-4 top-card'][.='Elements']"));
26        interactions = driver.findElement(By.xpath("//div[@class='card mt-4 top-card'][.='
        ↳ Interactions']"));
27    }
28
29    @After
30    public void teardown() {
31        driver.quit();
32    }

```

Testframeworks

Die erste Kategorie beinhaltet die elements-Tests. In dieser Kategorie, kann man an allen möglichen Elementen Interaktionen ausführen und Testen. Im folgenden zeige ich Interaktionen an den *Buttons*, den *Links* und den *Uploads* und *Download*.

```
33 @Test
34 public void elementsTest() {
35     elements.click();
36     //WebTables
37     driver.findElement(By.xpath("//li[@class='btn btn-light '][.='Web Tables']")).click();
38     //add max mustermann
39     driver.findElement(By.id("addNewRecordButton")).click();
40     driver.findElement(By.xpath("//input[@id='firstName']")).sendKeys("Max");
41     driver.findElement(By.xpath("//input[@id='lastName']")).sendKeys("Mustermann");
42     driver.findElement(By.xpath("//input[@id='userEmail']")).sendKeys("
        ↳ mmustermann@examplemail.com");
43     driver.findElement(By.xpath("//input[@id='age']")).sendKeys("29");
44     driver.findElement(By.xpath("//input[@id='salary']")).sendKeys("20000");
45     driver.findElement(By.xpath("//input[@id='department']")).sendKeys("Compliance");
46     driver.findElement(By.xpath("//button[@id='submit']")).click();
47     List<String> valuesOrigin = getwebtablevalues();
48     List<String> values = getwebtablevalues();
49     Assert.assertTrue("The added Person was not found in the table", values.contains("Max") &&
        ↳ values.contains("Mustermann"));
50     //search max mustermann
51     driver.findElement(By.xpath("//input[@id='searchBox']")).sendKeys("
        ↳ mmustermann@examplemail.com");
52     values = getwebtablevalues();
53     Assert.assertTrue("The searched Person could not be found in the table while it should have
        ↳ been", values.contains("mmustermann@examplemail.com") );
54     //delete max mustermann
55     String delete = "delete-record-";
56     delete = delete + (Integer) ((valuesOrigin.indexOf("mmustermann@examplemail.com") + 1) / 7
        ↳ + 1);
57     driver.findElement(By.id(delete)).click();
58     //search max mustermann
59     driver.findElement(By.xpath("//input[@id='searchBox']")).sendKeys("
        ↳ mmustermann@examplemail.com");
60     values = getwebtablevalues();
61     Assert.assertTrue("The searched Person was found in the table while it should not have been
        ↳ ", values.contains("mmustermann@examplemail.com")==false );
62
63     //Buttons
64     driver.findElement(By.xpath("//li[@class='btn btn-light '][.='Buttons']")).click();
65     //double click button
66     action.doubleClick(driver.findElement(By.id("doubleClickBtn"))).build().perform();
67     Assert.assertTrue("The desired action was not performed on the 'Double Click Me' - Button",
        ↳ "You have done a double click".equals(driver.findElement(By.id("
        ↳ doubleClickMessage")).getText()));
68     //right click
69     action.contextClick(driver.findElement(By.id("rightClickBtn"))).build().perform();
70     Assert.assertTrue("The desired action was not performed on the 'Right Click Me' - Button", "
        ↳ You have done a right click".equals(driver.findElement(By.id("rightClickMessage")).
        ↳ getText()));
71     //dynamic click
72     driver.findElement(By.xpath("//button[@class='btn btn-primary'][.='Click Me']")).click();
```



```

73     Assert.assertTrue("The desired action was not performed on the 'Click Me' - Button", "You
        ↳ have done a dynamic click".equals(driver.findElement(By.id("dynamicClickMessage"))
        ↳ .getText()));
74
75     //Links
76     driver.findElement(By.xpath("//li[@class='btn btn-light '][.='Broken Links - Images']")).click();
77     //valid link
78     driver.findElement(By.xpath("//a[@href= 'http://demoqa.com']")).click();
79     Assert.assertTrue("Chrome did not switch to the correct Tab","https://demoqa.com/".equals(
        ↳ driver.getCurrentUrl()));
80     driver.get("https://demoqa.com/broken");
81     //broken link
82     driver.findElement(By.xpath("//a[@href= 'http://the-internet.herokuapp.com/status_codes
        ↳ /500']")).click();
83     String stringtocheck = "This page returned a 500 status code.\n" + "\n" + "For a definition
        ↳ and common list of HTTP status codes, go here";
84     Assert.assertTrue("the broken link did not show the right status message",stringtocheck.
        ↳ equals(driver.findElement(By.xpath("//div[@class = 'example']/p")).getText()));
85     driver.get("https://demoqa.com/broken");
86 }
87 private List<String> getwebtablevalues(){
88     List<WebElement> webtable = driver.findElements(By.xpath("//div[@class='rt-tr-group']/*/*"
        ↳ ));
89     List<String> values = new ArrayList<>();
90     for (WebElement element:webtable) {
91         values.add(element.getText());
92     }
93     return values;
94 }

```

Aus der Kategorie *Widgets* zeige ich das Testen von Tool Tips beim Hovering über ein Element.

```

95 @Test
96 public void widgetsTest(){
97     widgets.click();
98     driver.findElement(By.xpath("//li[@class='btn btn-light '][.='Tool Tips']")).click();
99     //Buttonhover
100    action.moveToElement(driver.findElement(By.id("toolTipButton"))).build().perform();
101    Assert.assertTrue("The Tooltip is not what it's supposed to be!" + "\n" +
102        "It is: " + driver.findElement(By.xpath("//div[@class ='tooltip-inner'][.='You hovered
        ↳ over the Button']")).getText() + "when it should be: You hovered over the
        ↳ Button",
103        "You hovered over the Button".equals(driver.findElement(By.xpath("//div[@class ='
        ↳ tooltip-inner'][.='You hovered over the Button']")).getText()));
104    //Textfieldhover
105    action.moveToElement(driver.findElement(By.id("toolTipTextField"))).build().perform();
106    Assert.assertTrue("The Tooltip is not what it's supposed to be!" + "\n" +
107        "It is: " + driver.findElement(By.xpath("//div[@class ='tooltip-inner'][.='You hovered
        ↳ over the text field']")).getText() + "when it should be: You hovered over the
        ↳ text field",
108        "You hovered over the text field".equals(driver.findElement(By.xpath("//div[@class
        ↳ ='tooltip-inner'][.='You hovered over the text field']")).getText()));
109 }

```

Testframeworks

Die nächste Oberkategorie ist *Forms*. Im folgenden werde ich ein Practice-Form ausfüllen, welches außerdem ein *Image – Upload* und ein *OptionMenu* beinhaltet.

```
110 @Test
111 public void formsTest(){
112     forms.click();
113     driver.findElement(By.xpath("//li[@class='btn btn-light '][.='Practice Form']")).click();
114     //Names
115     driver.findElement(By.xpath("//input[@placeholder='First Name']")).sendKeys("Max");
116     driver.findElement(By.xpath("//input[@placeholder='Last Name']")).sendKeys("Mustermann")
117         ↪ ;
118     //E-mail
119     driver.findElement(By.xpath("//input[@placeholder='name@example.com']")).sendKeys("
120         ↪ max@mustermann.de");
121     //Geschlecht
122     driver.findElement(By.xpath("//label[@class='custom-control-label'][.='Male']")).click();
123     //Nummer
124     driver.findElement(By.xpath("//input[@placeholder='Mobile Number']")).sendKeys("
125         ↪ 0123456789");
126     //Date of Birth
127     //TODO: element lokalisieren, dss man erst aufklappen muss
128     //Hobbies
129     driver.findElement(By.xpath("//label[@class = 'custom-control-label'][.='Sports']")).click();
130     driver.findElement(By.xpath("//label[@class = 'custom-control-label'][.='Reading']")).click();
131     driver.findElement(By.xpath("//label[@class = 'custom-control-label'][.='Music']")).click();
132     //Picture
133     //TODO: Bild hochladen
134     //current Address
135     driver.findElement(By.xpath("//textarea[@placeholder='Current Address']")).sendKeys("
136         ↪ Musterstrasse 1a");
137     //State and City
138     driver.findElement(By.xpath("//div[@class=' css-tlfecz-indicatorContainer']")).click();
139     //TODO: element lokalisieren, dss man erst aufklappen muss
140     //submit
141     driver.findElement(By.id("submit")).click();
142     Assert.assertTrue("The form has not been submitted", "Thanks for submitting the form".
143         ↪ equals(driver.findElement(By.xpath("//div[@class='modal-title h4']")).getText());
144 }
```

Anhand der Kategorie Interactions wird im Folgenden gezeigt, wie man die Funktionalität von Drag and Drop testet.

```
140 @Test
141 public void interactionsTest(){
142     interactions.click();
143     driver.findElement(By.xpath("//li[@class='btn btn-light '][.='Droppable']")).click();
144     WebElement droppable = driver.findElement(By.xpath("//div[@id = 'droppable'][@class = '
145         ↪ drop-box ui-droppable']"));
146     action.dragAndDrop(driver.findElement(By.id("draggable")),droppable).build().perform();
147     Assert.assertTrue("The draggable was not dropped into the Box. It saies: " + droppable.
148         ↪ getText(),"Dropped!".equals(droppable.getText()));
149 }
```

Eine weitere Oberkategorie bilden *Alerts, Frame and Windows*. Im Folgenden zeige ich einige Tests an unterschiedlichen Browser Windows, Alerts und Frames.

```

149 @Test
150 public void alertsFrameWindowsTest(){
151     alertsFrameWindows.click();
152
153     //BrowserWindows
154     driver.findElement(By.xpath("//li[@class='btn btn-light '][.='Browser Windows']")).click();
155     //newTab
156     driver.findElement(By.id("tabButton")).click();
157     ArrayList<String> tabs = new ArrayList<>(driver.getWindowHandles());
158     driver.switchTo().window(tabs.get(1));
159     Assert.assertTrue("Chrome did not switch to the correct Tab","https://demoqa.com/sample".
        ↪ equals(driver.getCurrentUrl()));
160     driver.close();
161     driver.switchTo().window(tabs.get(0));
162     //new Window
163     driver.findElement(By.id("windowButton")).click();
164     tabs = new ArrayList<>(driver.getWindowHandles());
165     driver.switchTo().window(tabs.get(1));
166     Assert.assertTrue("Chrome did not switch to the correct Window","https://demoqa.com/
        ↪ sample".equals(driver.getCurrentUrl()));
167     driver.close();
168     driver.switchTo().window(tabs.get(0));
169     //new WindowMessage
170     driver.findElement(By.id("messageWindowButton")).click();
171     tabs= new ArrayList<>(driver.getWindowHandles());
172     driver.switchTo().window(tabs.get(1));
173     Assert.assertTrue("Chrome did not switch to the correct Window","Knowledge increases by
        ↪ sharing but not by saving. Please share this website with your friends and in your
        ↪ organization.".equals(driver.findElement(By.tagName("body")).getText()));
174     driver.close();
175     driver.switchTo().window(tabs.get(0));
176
177     //Alerts
178     driver.findElement(By.xpath("//li[@class='btn btn-light '][.='Alerts']")).click();
179     //alert after 5 seconds
180     driver.findElement(By.id("timerAlertButton")).click();
181     Assert.assertTrue("Alertbox does not display expected message", "This alert appeared after 5
        ↪ seconds".equals(driver.switchTo().alert().getText()));
182     driver.switchTo().alert().accept();
183     //confirm box
184     driver.findElement(By.id("confirmButton")).click();
185     Assert.assertTrue("Alertbox does not display expected message", "Do you confirm action?".
        ↪ equals(driver.switchTo().alert().getText()));
186     driver.switchTo().alert().accept();
187     Assert.assertTrue("Doesnt display expected Text after accepting", "You selected Ok".equals(
        ↪ driver.findElement(By.id("confirmResult")).getText() );
188     driver.findElement(By.id("confirmButton")).click();
189     driver.switchTo().alert().dismiss();
190     Assert.assertTrue("Doesnt display expected message after dismissing", "You selected Cancel"
        ↪ .equals(driver.findElement(By.id("confirmResult")).getText() );
191     //prompt box
192     driver.findElement(By.id("promtButton")).click();
193     System.out.println(driver.switchTo().alert().getText());

```

Testframeworks

```
194     Assert.assertTrue("Alertbox does not display expected message" , "Please enter your name".  
        ↪ equals(driver.switchTo().alert().getText()));  
195     driver.switchTo().alert().sendKeys("selenium");  
196     driver.switchTo().alert().accept();  
197     Assert.assertTrue("Doesnt display expected Text after accepting", "You entered selenium".  
        ↪ equals(driver.findElement(By.id("promptResult")).getText()));  
198 }
```

4.5 Fazit

4.5.1 Die Möglichkeiten und Grenzen von Selenium

Manuelles Testen kann oft sehr mühselig und ermüdend sein. Doch trotz der vielen offensichtlichen Vorteile des automatisierten Testens, ist es nicht immer sinnvoll sich für diese Methodik zu entscheiden. Insbesondere die Möglichkeit mit einem Testfall mehrere Browser zu testen macht das automatisierte Testen von Webseiten verlockend, doch bei kleineren Projekten sprengt der Aufwand und die Kosten oft das Budget und überwiegt den Nutzen automatisierter Tests.

Entscheidet man sich trotzdem für die automatisierten Testfälle trifft man mit Selenium die perfekte Wahl. Selenium hebt sich durch das unmittelbare und intuitive visuelle Feedback sowie von seinem Potenzial, als wiederverwendbares Testframework für andere Webanwendungen von anderen Frameworks ab und ist damit das beliebteste und meist verwendete Framework im Webdesign. Zudem ist anzumerken, dass Selenium eine tolle Community besitzt, die tagtäglich daran arbeitet Selenium programmierfreundlicher und effizienter zu machen.

Schlussendlich ist zu sagen, dass Selenium für alle Nutzereingaben im Webbrowser Methoden bietet, um diese zu simulieren. Schwierigkeiten entstehen, beim ausfindig machen von kaputten Downloads und Dateien, da dies schwer durch eine Nutzerinteraktion am Webbrowser getestet werden kann.

Tipps:

Im Folgenden sind ein paar Tipps aufgelistet, die beim Lernen von Selenium hilfreich sein könnten!

Es gibt bereits Dummy-Webseiten, die für das Testen mit Selenium erstellt wurden. Bsp:

- <https://the-internet.herokuapp.com/>
- <https://example.cypress.io/>
- <https://react-shopping-cart-67954.firebaseio.com/>
- <https://www.saucedemo.com/>
- <https://www.demoqa.com/>
- ...

Eine Webseite, die alle html-Tags abbildet:

<https://www.mediaevent.de/html/html5-tags.html>

Eine Webseite, die alle html-Attribute abbildet:

<https://www.mediaevent.de/html/kernattribute.html>

Eine Webseite, die beim Lokalisieren von Elementen hilft.

<https://automatetheplanet.com/selenium-webdriver-locators-cheat-sheet/>

Eine webseite, die erklärt, wie man anhand von Wget Dateien herunterlädt.

<https://www.guru99.com/upload-download-file-selenium-webdriver.html> Thread.sleep um zu sehen was der Test macht

5 Gemeinsame Tests

Anhand der Dummy Webseite *saucedemo* zeigen wir im folgenden Beispiel, wie man mithilfe von *click()*, *isDisplayed()* und *sendKeys(...)* Nutzereingaben zum Einloggen und anschließendem Checkout simulieren kann. Zur Verwaltung und Auswertung der Tests nutzen wir Annotations und Assert-Methoden von JUnit und TestNG:

```

1 public class SeleniumTest{
2     private WebDriver driver;
3     @BeforeClass
4     public static void setupDriver() {
5         WebDriverManager.chromedriver().setup();
6     }
7     @Before
8     public void setupTest() {
9         driver = new ChromeDriver();
10        driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
11    }
12    @After
13    public void teardown() {
14        driver.quit();
15    }
16    /**
17     * This method tests the checkout-implementation of "https://www.saucedemo.com/"
18     * It starts by logging in. Then adding a Backpack to the cart and checking out.
19     * Lastly it is simulating a user typing in their delivery information to finish
20     * the checkout and ending up at the checkout complete site.
21     */
22    @Test
23    public void test() {
24        driver.get("https://www.saucedemo.com/");
25
26        //logging in - finding elements using CSS Selectors
27        driver.findElement(By.cssSelector("#user-name")).sendKeys("standard_user");
28        driver.findElement(By.cssSelector("#password")).sendKeys("secret_sauce");
29        driver.findElement(By.cssSelector("#login-button")).click();
30
31        //checking out - finding elements using XPath
32        driver.findElement(By.xpath("//*[@id='add-to-cart-sauce-labs-backpack']")).click();
33        driver.findElement(By.xpath("//a[@class='shopping_cart_link']")).click();
34        driver.findElement(By.xpath("//button[@id='checkout']")).click();
35
36        //typing in delivery information - finding elements using id
37        driver.findElement(By.id("first-name")).sendKeys("max");
38        driver.findElement(By.id("last-name")).sendKeys("mustermann");
39        driver.findElement(By.id("postal-code")).sendKeys("01234");
40        driver.findElement(By.id("continue")).click();
41        driver.findElement(By.id("finish")).click();
42
43        //checking whether checkout was completed: Test evaluation !!!
44        Assert.assertTrue(driver.findElement
45        (By.cssSelector("#checkout_complete_container")).isDisplayed());
46    }
47 }

```

6 Vergleich und Fazit

Abschließend vergleichen wir JUnit, TestNG und Selenium miteinander.

6.1 JUnit verglichen mit TestNG

TestNG erweitert JUnit, wodurch beide Testframeworks viele gemeinsame Funktionalitäten haben. Eine Gemeinsamkeit bilden die Annotations, welche die Möglichkeit bieten Tests zu steuern. Dieses erweitert TestNG durch neue Annotations und die Alternative die Tests mittels XML-Konfiguration zu beeinflussen. Weiterhin bieten beide die gleichen Assert-Methoden um Tests durchzuführen. Sowohl JUnit als auch TestNG implementieren ein Grundgerüst für parametrisierte Tests, welche in TestNG durch die Annotation *DataProvider* und in JUnit durch die Annotation *MethodSource* ermöglicht werden. Darüber hinaus gibt es in TestNG die Möglichkeiten Testmethoden flexibel zu Gruppieren und verschiedene Tests durch *Multithreadtesting* parallel durchzuführen.

6.2 JUnit und TestNG verglichen mit Selenium

JUnit, TestNG und Selenium sind Frameworks, die man auf Java-Code anwenden kann, um automatisierte Tests zu erstellen. Wobei sich aber JUnit und TestNG in erster Linie für funktionale Tests eignen, während Selenium für nicht-funktionale Tests entwickelt wurde. Dementsprechend verwendet Selenium Black-Box Testing, wohingegen JUnit und TestNG überwiegend White-Box Testing einsetzen.

Anzumerken ist, dass man zum Testen von Webseiten mithilfe von Selenium in Java TestNG oder JUnit nutzt um die Tests auszuwerten.

6.3 Fazit

Beim Vergleich fällt auf, dass JUnit und TestNG größtenteils die gleichen Funktionalitäten aufweisen. Jedoch Selenium ein komplett anderes Anwendungsgebiet hat. Diese starke Diskrepanz verkompliziert es uns einen ausführlichen Vergleich von Testfällen und Funktionalitäten zu ziehen. Dies macht sich auch weiterhin bemerkbar, denn allgemein lässt sich sagen, dass es immer am besten ist sowohl funktionale, als auch nichtfunktionale Tests für eine Software zu schreiben um alle Aspekte abzudecken [Sys21], welches uns in dieser Arbeit jedoch schwer fällt, denn unser nicht-funktionales Testframework Selenium bezieht sich auf Webseiten, auf denen man mit TestNg und JUnit keine funktionalen Tests schreiben kann.

7 Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Marburg, 23.08.2021

Ort, Datum

Marie Osting Rn. A. Böcker

Unterschrift

8 Literatur

- [Ang] Anton Angelov. Selenium-locators. <https://www.automatetheplanet.com/selenium-webdriver-locators-cheat-sheet/>.
- [Api] Junit 5.7.2 api documentation. URL: <https://junit.org/junit5/docs/current/api>.
- [Baer19] Baeldung. A quick junit vs testng comparison. <https://www.baeldung.com/junit-vs-testng>, 2019.
- [Beuo4] Cédric Beust. Testng. <https://testng.org/doc/index.html>, 2004.
- [BN16] Andrew Butterfield and Gerard Ekembe Ngondi. *A Dictionary of Computer Science*. Oxford University Press, 2016.
- [bon] bonigarcia. Webdrivermanager doc and repo. <https://github.com/bonigarcia/webdrivermanager/>.
- [Che] Sebastian Chece. Expertenbericht-testautomatisierung definition, tutorial und artikel. <https://www.testing-board.com/testautomatisierung/>.
- [dRS21] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rancourt and Christian Stein. Junit 5 user guide. URL: <https://junit.org/junit5/docs/current/user-guide>, 2021.
- [Fra] Framesets und frames definieren. <https://www2.informatik.hu-berlin.de/Themen/www/selfhtml/html/frames/definieren.htm>.
- [Gup20] Lokesh Gupta. Testng tutorial. <https://howtodoinjava.com/java-testng-tutorials/>, 2020.
- [HTM20] Html-attribute class, id, style. <https://www.mediaevent.de/html/kernattribute.html>, jun 2020.
- [HTM21] Alle html-tags im Überblick. <https://www.mediaevent.de/html/html5-tags.html>, jul 2021.
- [iee90] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, page 79, 1990.
- [IFr] Top 25 selenium webdriver commands that you should know. <https://de.myservername.com/top-25-selenium-webdriver-commands-that-you-should-know5Handlingiframes>.
- [Jai21] Sonoo Jaiswal. Testng tutorial. <https://www.javatpoint.com/testng-tutorial>, 2021.
- [Jav] Javascriptexecutor in selenium webdriver with example. <https://www.guru99.com/execute-javascript-selenium-webdriver.html>.
- [Mat19] Mateus. 7 actions to a selenium script. <https://ultimateqa.com/selenium-dotnet-core-1/7-actions-of-selenium-script/>, 2019.
- [Met] Selenium-methoden. <https://www2.informatik.hu-berlin.de/Themen/www/selfhtml/html/frames/definieren>.
- [MH20] Vincent Massol and Ted Husted. *JUnit in Action*. Manning Publications, third edition, 2020.
- [Mol18] Diego Molina. *Selenium Fundamentals*. Packt Publishing, 2018.
- [Mul] How to handle multiple windows in selenium? <https://www.browserstack.com/guide/handle-multiple-windows-in-selenium>.
- [Nov20] Ekaterina Novoseltseva. 8 benefits of unit testing - dzone devops. URL: <https://dzone.com/articles/top-8-benefits-of-unit-testing>, 2020.
- [Par18a] Nicolai Parlog. Junit 5 - dynamic tests. <https://nipafx.dev/junit-5-dynamic-tests/>, 2018.

Testframeworks

- [Par18b] Nicolai Parlog. Junit 5 - parameterized tests. <https://nipafx.dev/junit-5-parameterized-tests/>, 2018.
- [Raj20] Rajkumar. Parallel test execution in testng. <https://www.softwaretestingmaterial.com/parallel-test-execution-testng/>, 2020.
- [Raj21] Harish Rajora. Testng vs junit. <https://www.toolsqa.com/testng/testng-vs-junit/>, 2021.
- [Sel21a] Selenium dokumentation. <https://www.selenium.dev>, 2021.
- [Sel21b] Selenium geschichte. <https://de.wikipedia.org/wiki/Selenium>, 2021.
- [Sys21] Softwaretest. <https://de.wikipedia.org/wiki/Softwaretest>, 2021.
- [Tal18] Tallence-warum software tests genauso wichtig sind wie software-entwicklung. <https://tallence.com/content/software-tests>, 2018.
- [TP221] Tutorialspoint - software testing dictionary. URL: https://www.tutorialspoint.com/software_testing_dictionary/unit_testing.htm, 2021.
- [Tut21] Tutorialspoint. Testng tutorial. <https://www.tutorialspoint.com/testng/index.htm>, 2021.
- [TY08] Dor Nir, Shmuel Tyszberowicz and Amiram Yehudai. Locating regression bugs. In *Hardware and Software: Verification and Testing*, pages 218–234. Springer Berlin Heidelberg, 2008.
- [Ull16] Christian Ullenboom. *Java ist auch eine Insel*. Galileo Press, twelfth edition, 2016.
- [Upl] Uploaddownload. <https://www.guru99.com/upload-download-file-selenium-webdriver.html>.
- [Vog21a] Lars Vogel. Junit 5 tutorial - learn how to write unit tests. URL: <https://www.vogella.com/tutorials/JUnit/article.html>, 2021.
- [Vog21b] Lars Vogel. What is software testing with unit and integration tests. URL: <https://www.vogella.com/tutorials/SoftwareTesting/article.html>, 2021.
- [Wal] Jakub Walczak. Expertenbericht-selenium webdriver tutorial 1. <https://www.testing-board.com/selenium-webdriver-tutorial-1-grundlagen-testautomatisierung-wordpress-und-basis-testframework>.
- [Wik21a] Wikipedia - system integration. URL: https://en.wikipedia.org/wiki/System_integration, 2021.
- [Wik21b] Wikipedia. Testng. <https://en.wikipedia.org/wiki/TestNG>, 2021.
- [Wol] Jan Wolter.
- [ZM17] Ahmed Zerouali and Tom Mens. Analyzing the evolution of testing library usage in open source java projects. pages 417–421, 02 2017.