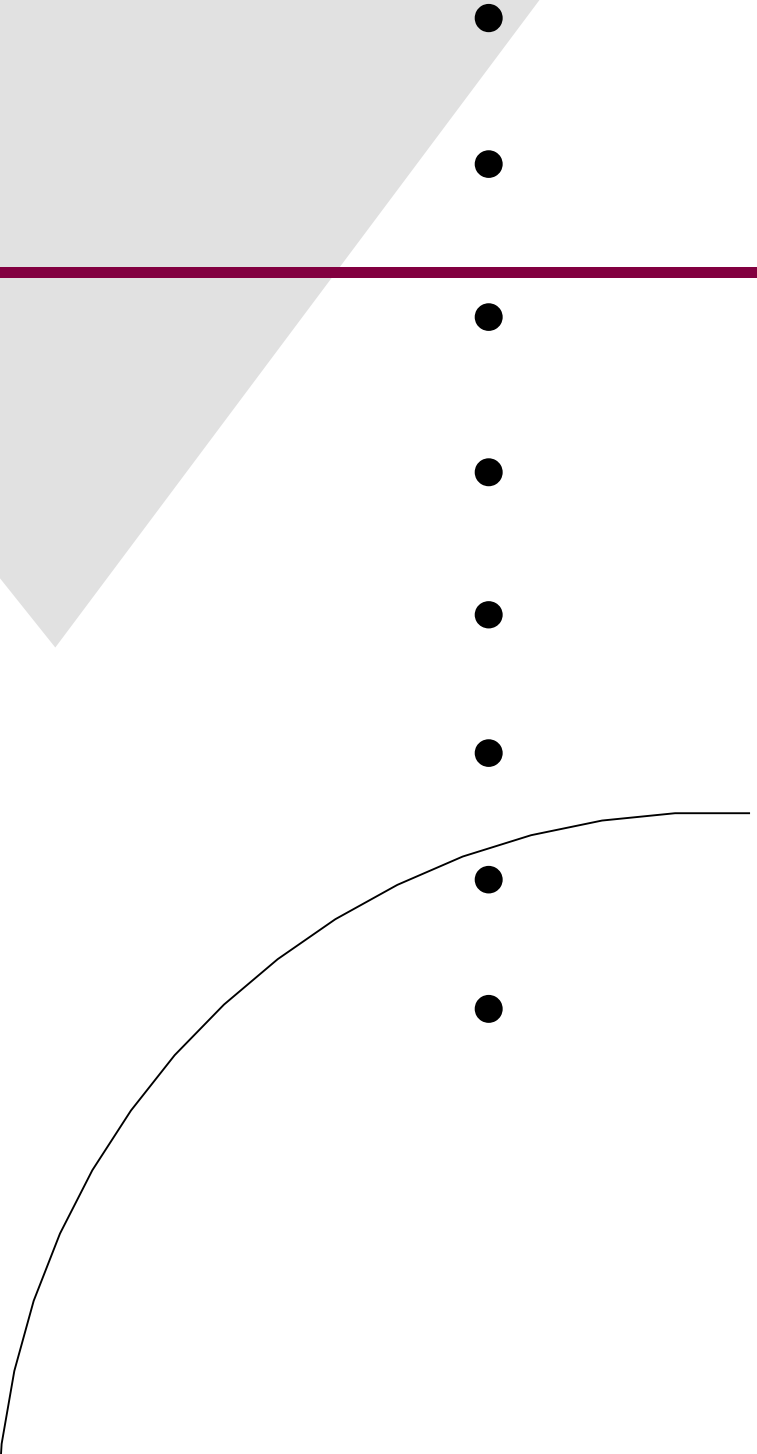




Unit Tests

Übersicht über **das**
Testframework xUnit

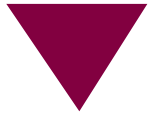




Übersicht

- Etwas Philosophie zum Unit Testing.
- Grundlagen zu den xUnit Test Frameworks.
- Einfache Beispiele zum Einstieg.
- Arbeitsweise und praktische Nutzung des JUnit Frameworks.





Steckbrief xUnit Framework

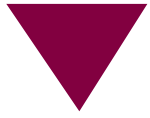
- Erste Versionen Ende der 1990er Jahre
 - von Kent Beck im Rahmen von eXtreme Programming entwickelt.
 - Erste Version für Smalltalk gefolgt von Java.
- Mauserte sich schnell zum weltweiten Standard.
 - Hohe Akzeptanz bei Entwicklerteams.
 - Einfach, robust und ressourcenschonend.
 - Für nahezu alle wichtigen OO-Sprachen erhältlich.
- Tests entstehen ohne "Medienbruch" in derselben Sprache und IDE wie der eigentliche Quellcode.
- Paradigmenwechsel: Tests werden nicht mehr nach dem Programmieren in einer eigenen Testphase erstellt, sondern zeitgleich mit dem Programmcode.



Bedeutung der Unit Tests

Auflistung ist
nicht vollständig

- Für den Entwickler
 - Tests für eigenen Code können schnell und einfach erstellt werden.
 - Sicherheit beim Arbeiten & Refactoring.
 - Tests geben Feedback, wann die Entwicklung fertig ist (Test First Strategie).
 - Die Testsuites als Ganzes ergeben:
 - ➔ eine technische Dokumentation des Quellcodes.
 - ➔ eine Zusammenfassung der Anforderungen an den Code.
- Für den Projektleiter / Testmanager
 - Messung des Projektfortschritts.
 - ➔ Nur wenn neben dem Quellcode auch alle Tests existieren und grün sind, sind die Anforderungen erfüllt.
 - Qualitätskontrolle über den aktuellen Zustand des Projekts.
 - Sicherstellung der geforderten Testabdeckung.
- Für das Projekt
 - Refactoring und Wartung bestehenden Codes sind abgesichert.
 - Jederzeit kann eine Auskunft gegeben werden über den Qualitätszustand des produktiven Codes.
 - Versehentliches Einschleichen von Seiteneffekten wird erkannt.
- Für den Endkunden
 - Höheres Vertrauen in die Entwicklung des Produktes.
 - Testprotokolle und Unit-Tests beweisen den qualitativen Zustand des Produktes.
 - Durch existierende Testsuiten wird eine spätere Wartung und Weiterentwicklung erleichtert.



Der Minimaltest

- Vollständige Testsuites können sehr komplex werden.
- Für einen einfachen Test reichen hingegen wenige Zeilen:

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
import mysource.Droid;

class HelloWorldTest {
    Droid myFirstDroid = new Droid();

    @Test
    void speak2me() {
        assertEquals("Hello World", myFirstDroid.speak());
    }
}
```



Übersicht wichtiger Annotationen

- Die folgende Liste zeigt die wichtigsten Annotationen für JUnit-Tests:

Annotation	Description
@Test	Denotes that a method is a test method.
@BeforeEach	Denotes that the annotated method should be executed before each @Test method in the current class.
@AfterEach	Denotes that the annotated method should be executed after each @Test method in the current class.
@BeforeAll	Denotes that the annotated method should be executed before all @Test methods in the current class.
@AfterAll	Denotes that the annotated method should be executed after all @Test methods in the current class.
@Disabled	Used to disable a test class or test method.
@Timeout	Used to fail a test method if its execution exceeds a given duration.
@DisplayName	Declares a custom display name for the test class or test method.

Die vollständige Liste findet sich unter: junit.org/junit5/docs/current/user-guide

Einsatz von Annotationen

- **@BeforeEach**
 - Methode wird vor **jedem einzelnen** Test durchlaufen.
 - Hiermit wird ein bestimmter Zustand hergestellt (=Initialisierung), der für jeden einzelnen Tests benötigt wird **und** der sich durch die Testausführung ändern kann.
- **@AfterEach**
 - Methode wird nach **jedem einzelnen** Test durchlaufen.
 - Hiermit können z.B. mögliche Artefakte, die während der einzelnen Testdurchläufe erzeugt werden, aufgeräumt werden
 - ➔ z.B.: Löschen einer Datei, die für die folgenden Tests nicht existieren soll.
- **@BeforeAll**
 - Einmalige Initialisierung bevor die Tests der Testklasse ausgeführt werden.
 - Wird z.B. verwendet um benötigte Ressourcen (z.B. Datenbank) zu initialisieren.
- **@AfterAll**
 - Einmaliger "Tear Down" nach dem Durchlauf aller Tests.
 - Wird zum abschließenden Aufräumen der Testumgebung und dem Freigeben genutzter Ressourcen verwendet.
 - ➔ z.B. Löschen eines von den Tests genutzten Temp-Verzeichnisses und Freigabe der genutzten Datenbank.

Einsatz von Annotationen

- **@Test**
 - Markiert die Methode als Testmethode.
 - Diese Methoden enthalten den eigentlichen Testcode.
 - Die @Test-Methoden bilden in ihrer Gesamtheit den vollständigen Unit-Test ab.
- **@Timeout**
 - Hiermit kann eine maximale Zeit für den Testdurchlauf angegeben werden.
 - Nützlich für zeitkritische Prozesse bzw. Prüfung von Performance-Anforderungen.
- **@DisplayName**
 - Zuweisung eines Anzeigenamens für einen Test oder eine Testklasse.
 - Hiermit können auch Leerzeichen, Umlaute und Piktogramme definiert werden, die anstatt des Methodennamens bei der Auswertung angezeigt werden.
- **@Disabled**
 - Die wahrscheinlich wichtigste Annotation überhaupt: Deaktiviert und ignoriert die folgende Testmethode bzw. eine vollständige Testklasse!
 - Wird hauptsächlich verwendet um einzelne Testmethoden, die nicht funktionieren "vorübergehend" auszuschalten und einen positiv-grünen Testdurchlauf sicher zu stellen.
 - Beliebt bei unerwarteten Projektleiter- oder Kundenbesuchen.
 - Kann bei reichlicher Verwendung auch zu lustigen Wutausbrüchen überzeugter Agile-Entwickler, Projektleiter und Testmanager führen...

Annotationen im Detail

- **@Test**

- Markiert die eigentlichen Tests.
- Es existieren mit `@ParameterizedTest` und `@RepeatedTest` weitere Alternativen für spezielle Testszenarien (-> wird an dieser Stelle nicht weiter vertieft).

```
class MeinTest{  
    @Test  
    void testeIrgendwas() { ... }  
}
```

Annotationen im Detail

- @BeforeEach, @AfterEach, @BeforeAll, @AfterAll
 - Markieren jeweils eine Methode die innerhalb des Test-Lifecycles zu den definierten Zeitpunkten ausgeführt wird.

```
class MeinTest{

    @BeforeAll
    static void initialisiereTestumgebung () { ... }

    @BeforeEach
    void setzeTestVariablen() { ... }

    @Test
    void testeIrgendwas() { ... }

    @Test
    void testeNochWasAnderes() { ... }

    @AfterEach
    void testartefakteAufräumen() { ... }

    @AfterAll
    static void testumgebungAufräumen() { ... }

}
```

Annotationen im Detail

▪ @DisplayName

- Weist einer Testmethode bzw. Testklasse einen alternativen Namen für die Anzeige zu.
- Wird von vielen Testtools unterstützt.

```
@DisplayName("Mein erster Test")
class DisplayNameDemo {

    @Test
    @DisplayName("Leerzeichen sind erlaubt")
    void testWithDisplayNameContainingSpaces() { ... }

    @Test
    @DisplayName("Ümlaute und Sonderzeichen □ ° sind möglich")
    void testWithDisplayNameContainingSpecialCharacters() { ... }

    @Test
    @DisplayName("Emojis gehen auch "👁️")
    void testWithDisplayNameContainingEmoji() { ... }

}
```

Annotationen im Detail

▪ @Timeout

- Zusätzlich zu dem eigentlichen Testfall, schlägt ein Test mit @Timeout auch fehl, wenn die angegebene maximale Laufzeit überschritten wird.
- Die Zeiteinheit ist konfigurierbar (Default=Sekunden).

```
class TimeoutTest {  
    @Test  
    @Timeout(5)  
    void testeSekunden() {  
        // Falls Testausführung > 5 Sekunden → Fehler  
    }  
  
    @Test  
    @Timeout(value = 100, unit = TimeUnit.MILLISECONDS)  
    void testeMillisekunden() {  
        // Falls Testausführung > 100 Millisekunden → Fehler  
    }  
}
```

Annotationen im Detail

▪ @Disabled

- Deaktiviert die Ausführung eines einzelnen Tests oder einer Testklasse.
- Ist nur in seltenen Fällen eine gute Idee und dann auch nur wenn der Grund angegeben ist.
- Disabled Tests in Release Code haben einen "very, very bad code smell"!

```
@Disabled("Testklasse deaktiviert bis Datenbank repariert ist")
class DisabledClassDemo {

    @Test
    void testWillBeSkipped() {
    }
}
```

```
class DisabledTestsDemo {

    @Disabled("Deaktiviert bis Bug #1123 gefixed ist")
    @Test
    void testWillBeSkipped1() { // Akzeptabel, weil kommentiert }

    @Disabled()
    @Test
    void testWillBeSkipped2() { // Kein Kommentar => Das gibt Ärger!!! }

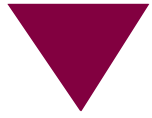
    @Test
    void testWillBeExecuted() { ... }
}
```



An die Tasten...

- Nach der Theorie nun die Praxis!
 - Schreiben sie eine Testsuite für das Labor-Projekt.
 - Entwicklergruppe 3-4 Personen.
 - Bitte gemeinsam abstimmen, analysieren, designen und dann entwickeln.
 - Kunde/Abnehmer ist ihr Dozent.
 - Abnahme/Präsentation der Testsuite gegenüber dem "Kunden"
 - Deadline ist am Ende des Labors.
 - Code wird vorab bereitgestellt.
 - Besprechung (inklusive Fragen des Kunden), maximal 45 Minuten.
 - Bezahlung und Motivation ;-)
 - Es gibt eine Note, welche in die Gesamtnote einfließt..





Zusammenfassung

- xUnit wurde für das automatisierte, regressive Testen (in agilen Projekten) entwickelt.
- xUnit eignet sich prinzipiell für Whitebox, Graybox & Blackbox Testing.
- Annotationen markieren Klassen und Methoden und legen dadurch ihre Benutzung fest.

