

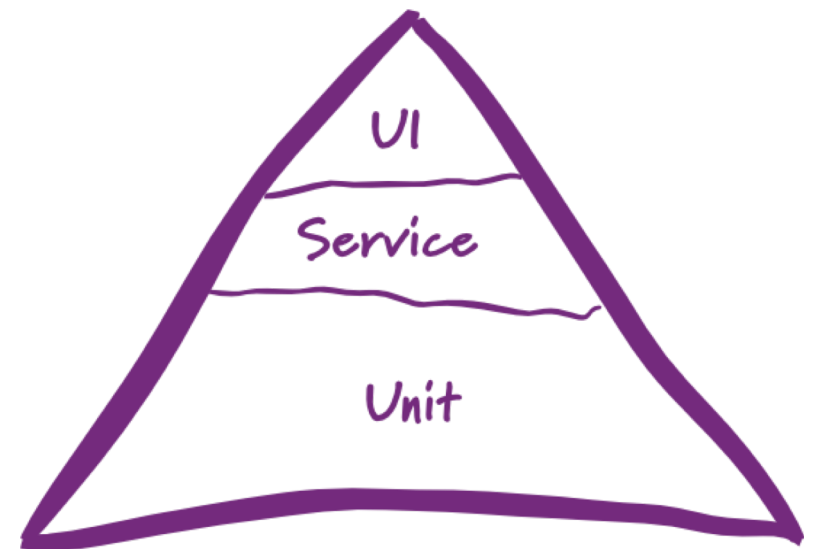
# **SOFTWARETECHNIK II (INF 4)**

Prof. Dr. Milena Zachow

## **2.3 Teststufen (Wiederholung)**

# Testpyramide

- Ideale Verteilung der verschiedenen **automatisierten** Teststufen in einem Projekt. Typischerweise gibt es noch **manuelle Checklistentests** und/oder *Smoketests*
- **Unit Tests**: testen atomare Einheiten im Quellcode (z.B. Methoden, Klassen, Komponenten). Gehören typischerweise in die Entwicklung
- **Service-Level Tests/Integrationtests**: testen das Zusammenspiel verschiedener Komponenten. Gehören typischerweise in die Entwicklung
- **UI Tests/Akzeptanztests/Systemtests**: testen das Gesamtsystem aus Usersicht gegen die Anforderungen. Gehören teilweise in Produkt-Management oder QS



# Unit Tests

- **Unit Tests** sollten die **Basis** (die größte Menge) an Tests sein, da sie sich schnell und automatisiert ausführen lassen
- **Unit Tests** lassen gute Fehlerbestimmung zu
- **Unit Tests** sind Aufgabe der Software-Entwickler (also von Ihnen)
- **Unit Test** sind sehr detailliert
- Was auf dieser Ebene getestet wird ist verschieden. Ich versuche z.B. alle Methoden zu testen, in denen es Verzweigungen gibt. Einfache *Accessor* und *Mutator* Methoden lasse ich aus
- **Unit Tests** machen nur Sinn, wenn sie regelmäßig ausgeführt werden und sofort auf Testfehler reagiert wird (später mehr bei *Continuous Integration*)

# Unterschiede von Unit und Integration Tests

Unit Tests	Integration Tests
Ergebnis hängt vom reinen (Java) Code unter Test ab	Ergebnis kann von externen System abhängen
Leicht aufzusetzen und zu schreiben	Eventuell kompliziertes Setup möglich
Klasse oder Unit wird isoliert getestet	Zusammenspiel von Komponenten wird getestet
Abhängigkeiten werden über Mocks oder Stubs bereit gestellt	Es wird nicht gemockt (außer es gibt Abhängigkeiten, die nicht getestet werden sollen)
Alle Unit Tests einer (Enterprise) Anwendung auszuführen, sollte höchstens einige Minuten dauern	Alle Integration Tests einer (Enterprise) Anwendung auszuführen, kann auch automatisiert Stunden dauern
Von Interesse nur für die Entwicklung	Kann auch für QA oder andere Abteilungen interessant sein

# Ansätze für Integrationstest

- **Big Bang:** (fast) alle Module oder Komponenten des System werden zusammen getestet
- **Top-Down:** man beginnt die Komponenten auf höchster Ebene zu testen und bewegt sich dann immer tiefer in die Hierarchie
- **Bottom-Up:** zuerst testet man *low-level* Komponenten und arbeitet sich dann weiter hoch
- **Ad-hoc:** es wird die Komponenten getestet, die fertig sind

# Beispiele

- Tests gegen Datenbanken können Integration Test sein
- UI-Tests können Integration Tests sein
- Test gegen eine Service-Schnittstelle (SOA, oder z.B. eine Rest-Schnittstelle oder eine Service Klasse) können Integration Tests sein
- Die oben genannten Beispiele können auch Systemtests sein
- *Wir gehen hier aus Zeitgründen detaillierter auf UI Tests als Beispiele ein*

# UI-Tests

- Sind auch Integration-Tests, es wird aber auf jeden Fall die UI getestet, UI-Test können auch End-to-end bzw. Systemtests sein
- Tests laufen langsam und sind fehleranfällig (besonders im Browser)
- Man kann auf zwei Arten testen
  - Funktional
  - Gegen einen Style-Guide (Layout)
- Eine gängige Variante ist **Capture and Replay**, um UI Tests zu automatisieren
  - **Capture**: Anwenderinteraktionen werden mit einem Werkzeug aufgezeichnet
  - **Checkpoints/Asserts** werden erstellt
  - **Replay**: die Anwenderinteraktion wird automatisch wieder abgespielt, *Asserts* werden geprüft



# Identifikation von Testfällen

- Äquivalenzklassen und Grenzwertanalyse
  - In der Praxis können nicht alle Eingaben getestet werden. Daher ist es notwendig geeignete Testfälle zu identifizieren.
  - Dazu eignen sich Äquivalenzklassen, die ähnliche Daten zusammenfassen.
  - Insbesondere Grenzwerte sollten dabei besonders berücksichtigt werden.

## 2.3 TESTARTEN

# Identifikation von Testfällen Beispiel aus Beneken et al. (2022)

**Tab. 18.1** Das Datum, zu dem wir unsere Tests anschauen, ist der 18.03.2020, alle Angaben sind relativ zu diesem Wert. Wir gehen davon aus, dass wir bereits ein plausibles Datum erhalten haben, also keinen 30.02.1919, und dass auch Fehleingaben wie 18.02. oder ‚ABCD‘ oder ’’ (Leerstring) bereits abgefangen wurden. So haben wir für die verschiedenen denkbaren Tarifmodelle der Kfz-Versicherung sechs Äquivalenzklassen gefunden, davon sind zwei Klassen für zu große oder zu kleine Eingaben vorgesehen

Äquivalenzklasse	Unterer Grenzwert	Oberer Grenzwert	Reaktion des Produkts
Kinder (0–17)	19.03.2002	18.03.2020	Antrag ablehnen aus Altersgründen
Fahranfänger (18–24)	19.03.1995	18.03.2002	Antrag annehmen, erhöhter Tarif
Normaltarif (25–60)	19.03.1959	18.03.1995	Antrag annehmen, normaler Tarif
Seniorentarif (61–100)	19.03.1919	18.03.1959	Antrag annehmen, Seniorentarif
Zu große Werte	19.03.2020	Maximal-Datum	Fehler melden
Zu kleine Werte	Minimal-Datum	18.03.1919	Fehler melden

# Identifikation von Testfällen: Heuristiken

- Wenn Sie schon häufiger Softwareprojekte durchgeführt haben, werden Sie mögliche Fehlerquellen ggf. schnell identifizieren. Dies nennt man häufig: Heuristiken Erfahrene Entwickler:innen / Softwaretester:innen suchen häufig noch Mustern, die zu Fehlern führen können, z. B.:
  - Leere Listen erstellen und Elemente löschen lassen
  - Liste mit Elementen löschen, ohne das Element vorher zu löschen
  - Integeroverflows, Negative Zahlen, ...

## 2.3 TESTARTEN

# Identifikation von Testfällen: weitere Verfahren

- Überdeckung
  - Kanten- bzw. Knotenüberdeckung
  - Code Coverage
  - Anforderungsüberdeckung
  - Zustandsüberdeckung

## 2.4 Testautomatisierung

# JUnit

- De Facto Standard ist die Bibliothek JUnit
- JUnit5 (5.x) ist aktuell und unterscheidet sich stark von den älteren Versionen (dessen Konzepte finden Sie aber noch in vielen anderen Bibliotheken)
- Sie können mit JUnit auch Integrationstests schreiben

```
package com.example.foo;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
/**
 * Tests for {@link Foo}.
 *
 * @author user@example.com (John Doe)
 */
class FooTest {

    @Test
    public void thisAlwaysPasses() {
        assertEquals(2, 1+1);
    }

    @Test
    @Disabled
    public void thisIsIgnored() {
    }
}
```



# Miniübung

- Implementieren Sie einen Unit Test im Projekt von letzter Woche

```
package com.example.foo;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
/**
 * Tests for {@link Foo}.
 *
 * @author user@example.com (John Doe)
 */
class FooTest {

    @Test
    public void thisAlwaysPasses() {
        assertEquals(2, 1+1);
    }

    @Test
    @Disabled
    public void thisIsIgnored() {
    }

}
```



## Zusicherungen (Asserts)

- **Asserts:** Annahmen über den Zustand eines Computerprogramms. Wir testen immer gegen bestimmte Annahmen über den Zustand eines Programmes. Einige Beispiele in JUnit:
  - `assertEquals`
  - `assertTrue`
  - `assertNotNull`
  - `assertAll`
- Die Parameter sind oft *message*, *expected*, *actual*
- Einige *Annotations*
  - `@AfterEach` `tearDown` pro Testmethode
  - `@BeforeEach` `setUp` pro Testmethode
  - `@BeforeAll`, `@AfterAll` wird vor/nach dem Testfall/Testklasse ausgeführt

Viele Testframeworks richten sich nach dem ursprünglichen Konzept von JUnit (3.x): Test haben ein *test* im Methodennamen, es gibt eine *setUp()* und eine *tearDown()* Methode..



## Testcases und Testsuits

- Die Java Konvention ist zu einer **Klasse** einen Testfall zu erstellen, der dann *KlassenNameTests* genannt wird
- **Testmethoden**: eine Methode, die einen einzelnen Unit Test enthält
- **Testklassen**: eine Klasse mit einer oder mehr Testmethoden (typischerweise nach der oben genannten Konvention)
- **Test Fixtures**: Setup für mehrere Tests (DRY)
- **Test Suite**: Gruppe von zusammengehörigen Tests

```
@SelectClasses({  
    TestFeatureLogin.class,  
    TestFeatureLogout.class,  
    TestFeatureNavigate.class,  
    TestFeatureUpdate.class  
})  
  
class FeatureTestSuite {  
    // the class remains empty,  
    // used only as a holder for the above annotations  
}
```


# Testdaten

- **Idee:** Testdaten sollen von Tests getrennt werden
- **JUnit 4:** *Theories* oder *ParameterizedTests*
- **JUnit 5:** *ParameterizedTests* oder *DynamicTests*

```
@ParameterizedTest
@ValueSource(strings = { "hallo!", "test", "de" })
void testEven(String candidate) {
    assertTrue(candidate.length()%2==0);
}
```

# Testen von Exceptions

- Schreibt man Code der *Exceptions* wirft, möchte man auch Testen, dass diese *Exceptions* an den richtigen Stellen geworfen werden
- **JUnit** bietet 2 Möglichkeiten:
  - *assertThrows* nutzen
  - In der Testmethode arbeitet man mit `try ... catch`



Demo siehe  
*live coding*

# Mocks und Stubs

- *Problem*: Eine **Klasse** kann **Abhängigkeiten** haben. Sind diese nicht gesetzt, lassen sich manche **Methoden** nicht testen
- Lösungsansätze
  - **Stubs**: Implementierungen, die für einen Test genau die benötigte Antwort liefern. Kann zu viel „leeren“ Methoden führen, um Interfaces zu implementieren
  - **Mocks**: Objekte, die auf bestimmte *Calls* eine vorprogrammierte Antwort geben. Man benutzt Libraries wie *Mockito*(später mehr)

```
public class ServiceImpl implements IService {  
    private IDependency dependency;  
  
    public void setDependency(IDependency dependency){  
        this.dependency = dependency;  
    }  
  
    public boolean isServiceWorking() {  
        //does something here we want to test  
        return dependency.someMethod();  
    }  
}
```



Beispiel siehe  
Tafel

# Mockito

- <http://mockito.org> Bekannte *Mock Library* im Java Umfeld
- **Idee:** vor dem Test zeichnet man das Verhalten des Mock-Objekts auf, dann wird das Verhalten wiedergegeben
- Mit `verify(Mock)` kann man prüfen, ob die spezifizierten Methoden wirklich aufgerufen wurden
- Behavior-Driven Syntax möglich
- Andere bekannte Alternativen: **Easymock**, **JMock** oder **PowerMock**

# Integrationstests

- Bei Integrationstests werden Komponenten im Zusammenhang getestet, dazu muss also eine entsprechende Umgebung hergestellt werden
- Z.B. SpringBoot unterstützt das mit der *@SpringBootTest* Annotation
- So können Sie z.B. einen HTTP Request stellen und testen, ob die Antwort den Erwartungen entspricht

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class HttpRequestTest {
    @LocalServerPort
    private int port;
    @Autowired
    private TestRestTemplate restTemplate;
    @Test
    void greetingShouldReturnDefaultMessage() throws Exception {
        assertThat(this.restTemplate.getForObject("http://localhost:" + port + "/",
            String.class)).contains("Hello, World");
    }
}
```

# Testdurchführung

- *Manuell*
  - Kosten- und Zeitintensiv
  - Gängige Praxis: Checklisten Test
- *Automatisiert*
  - Liegt oft bei den Entwicklern
  - Auf Unit-, Integration- und UI-Testebene möglich
  - Aufwendig in der Erstellung, aber günstig im Betrieb
  - Laufzeit kann durchaus zum Problem werden

# Testdurchführung

- Möglichst oft
- Möglichst automatisiert
- So, dass man eine direkte Rückmeldung bekommt
- Mehr dazu im Kapitel *Continuous Integration* (SWT II)



## Fazit

- Wir haben uns mit funktionalen Tests auf verschiedenen **Teststufen** beschäftigt
- Nicht-funktionale Tests wurde nicht thematisiert
  - Performanz-Tests
  - Sicherheitstests
  - ...

# Testkonzept nach IEEE 829

- Ein Testkonzept ist ein Dokument, da die Aktivitäten im Bereich Test spezifiziert. Es ist nach IEEE 829 standardisiert und enthält folgende Punkte:

- **Einführung**

- Identifikation des Testkonzepts
- Geltungsbereich und Umfang
- Referenzen

- Zu testendes System und Testobjekt

- Überblick über die Testaufgaben

- Organisation
- Projekttestplan
- Integrationsstufen
- Ressourcenübersicht
- Zuständigkeiten
- Werkzeuge, Techniken, Methoden, Metriken

- **Details**

- Testprozess und Teststufen
- Dokumente
- Abweichungs- und Änderungsmanagement
- Berichtswesen

- **Allgemeines**

- Glossar
- Änderungsdienst und Historie

Für Ihr Praktikum brauchen Sie kein vollständiges Testkonzept nach IEEE 829. Es sollte aber deutlich werden, welche Tools und Teststufen Sie testen und für welche Fälle Tests geschrieben werden!