ÜbunGszettel TDD

# Aufgabe 1: Theorie

Arbeite dich in die Theorie zum Thema Testen bzw. Test-Driven-Development (TDD) ein. Nutze dazu die Materialien aus den verschiedenen „Input“ Abschnitten im Themenbereich „Test-Driven Development“ unseres Moodle-Kurses (https://moodle.tsn.at/course/view.php?id=24763#section-8). Nach dieser Aufgabe musst du folgende Punkte erklären können:

* Testdriven Development
* Red-Green-Refactor
* FIRST-Acronym
* Kent Beck (welche Rolle spielt er in Bezug auf TDD)
* Testarten
  + Unit-Tests (Sociable, Solitary, Mocks)
  + Integrationstests
  + UI-Tests / End-To-End Tests / Systemtests
  + Akzeptanztests
* Testpyramide
* JUNIT (Junit5)
* Mockito (Sinn und Funktionsweise von Mocking-Bibliotheken)

## Testdriven Development

wir häufig bei der agilen Entwicklung eingesetzt  
Softwaretests werden vor den zu testenden Komponenten erstellt

## Red-Green-Refactor

* Red: Was soll entwickelt werden?
* Green: Wie soll der Test bestanden werden?
* Refactor: Wie soll die existierende Implementierung verbessert werden?

**Red:**

Zweck dieser Phase: Definition was implementiert werden soll.

Der Test wird nur bestanden, wenn die Anforderungen erfüllt werden.

**Green:**

In dieser Phase wird der Code implementiert, um den Test zu bestehen. Ziel ist es, eine Lösung zu finden, ohne auf die Optimierung des Codes zu achten.

Nach dieser Phase soll der Test erfolgreich sein.

**Refactor:**

In der Refactor-Phase ist man immer noch im grünen Bereich.

Code soll verbessert werden. Der Output des Tests soll der gleiche bleiben, aber der Code soll schneller und übersichtlicher werden.

## FIRST-Acronym

Fast, Independent, Repeatable, Self-validating, Timely

**Fast:**

Tests sollten schnell sein, damit sie nicht dazu verleiten, sie nicht mehr durchzuführen.

**Independent:**

Test sollten sich nicht auf Tests davor stützen, weder auf Objekte noch auf Methoden.

**Repeatable:**

Tests sollten in jeder Umgebung, ohne unterschiedliche Ergebnisse, ausführbar sein.

**Self-validating:**

Jeder Test soll einen einzigen boolschen Wert zurückgeben.

**Timely:**

Unit-Tests sollten geschrieben werden, kurz vor der Implementierung des Codes, den der Test braucht.

## Kent Beck (welche Rolle spielt er in Bezug auf TDD)

gilt als einer der drei Begründer von Extreme Programming.

gilt als Erfinder des Smalltalk-Frameworks SUnit, das er auch auf Java portiert hat (JUnit)

## Testarten

* Komponententest: abgrenzbare Einzelteile wie Module, Unterprogramme, Units, Klassen -> überprüft technische Lauffähigkeit
* Integrationstest: Zusammenarbeit voneinander abhängiger Komponenten -> überprüft komplette Abläufe
* Systemtest: gesamtes System gegen gesamte Anforderungen -> Testumgebung soll Produktivumgebung des Kunden simulieren
* Abnahmetest: Test der Software durch Kunde

### Unit-Tests (Sociable, Solitary, Mocks)

ist eine frühe Teststufe, in der die inneren, detailliertesten Komponenten der Software getestet werden.

**Sociable:**

Ein solcher Test nimmt an, dass die anderen Dependencies funktionieren. Dieser Test testet nur eine Einheit.

Im Gegensatz dazu, testet ein Integrationstest, ob verschiedene Einheiten zusammenarbeiten können.

**Solitary:**

In diesem Test wird nur ein einziger Service getestet (alle anderen Dependencies, außer für data/value-Tyoen werden gemockt).

**Mocks:**

Der Zweck ist es den Code zu isolieren und den Fokus auf den zu testenden Code zu legen, nicht auf das Verhalten oder den Status von externen Dependencies.

### Integrationstest

Eine aufeinander abgestimmte Reihe von Einzeltests, die dazu dienen, verschiedene voneinander abhängige Komponenten eines komplexen Systems im Zusammenspiel miteinander zu testen.

Die Komponenten haben die jeweiligen Modultests erfolgreich bestanden und sind für sich isoliert fehlerfrei funktionsfähig.

Zwei Strategien:

* testzielorientiert: benötigt nur die zum Testen notwendigen System-Komponenten
* vorgehensorientiert: ist abhängig von der Integrationsreihenfolge aus der Systemarchitektur

Arten:

* inkrementell: fügt Module schrittweise zusammen
* nicht-inkrementell: fügt alle Komponenten gleichzeitig zusammen

### UI-Tests / End-To-End Tests / Systemtests

**UI-Tests (User-Interface-Test):**

Überprüft:

* Funktionalität
* Design
* Performance
* Verwendbarkeit
* Compliance (Fügbarkeit)

Testfälle:

* Datentyp-Errors: nur valide Daten werden akzeptiert
* Feldgröße: Input-Feld darf keine zu langen oder kurzen Werte haben
* Navigations-Elemente: sollen richtig funktionieren
* Fortschritts-Balken: wenn Daten geladen werden, soll ein Fortschritts-Balken gezeigt werden
* Type-ahead: für Drop-Down-Listen
* Table scrolling: User soll durch große Tabellen scrollen können
* Error Logging: System soll Errors korrekt mitloggen
* Shortcuts: testet ob alle Shortcuts richtig funktioniert

**End-To-End-Tests:**

Testet komplette Software vom einen zum anderen Ende, um den ganzen Fluss zu testen, ob alles wie gewünscht funktioniert.

Hauptzweck ist, der Test vom Benutzer (Benutzer-Simulation), über Daten-Validierung bis zu den Komponenten für Integration und Datenintegrität.

**Systemtests:**

Hier wird das gesamte System gegen die Anforderungen (funktional und nicht-funktional) getestet. Findet gewöhnlich in einer Testumgebung statt und wird mit Testdaten durchgeführt. Testumgebung soll Produktivumgebung des Kunden simulieren.

### Akzeptanztests

UAT (User Acceptance Test)

Testet, ob die Software aus Sicht des Benutzers wie beabsichtigt funktioniert und dieser die Software akzeptiert.

Sollten von einem Fachexperten (Kunde|Eigentümer) durchgeführt werden. Danach eventuell Software überarbeiten.

Fungiert als abschließende Überprüfung der erforderlichen Geschäftsfunktionen und das ordnungsgemäße Funktionieren des Systems unter realen Einsatzbedingungen im Namen des zahlenden Kunden.

## Testpyramide

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Klärt die Frage: Wie viele Tests jeder Kategorie sollen durchgeführt werden?

1. Basis bilden die Unit Tests: prüfen Großteil der Business Logik
2. Einbeziehung der Persistenzschicht: kann in zwei Teilebenen unterteilt werden
   1. Test gegen eine In-Memory-DB: beschränkt Tests auf lokalen Entwicklerrechner
   2. Arbeiten mit Datenbestand: stellt Eindeutigkeit von Testergebnis sicher
3. Integrationstest mit Serverumgebung: Korrektheit von Konfiguration, Firewall, Netzwerkperformance, Zusammenspiel aller Anwendungen

## JUNIT (JUNIT5)

Framework zum Testen von Java-Programmen. Ist besonders geeignet für automatisierte Unit-Tests.

JUnit-Test kenn nur zwei Ergebnisse: Entweder grün oder rot. Das Misslingen kann als Ursache einen Fehler (Error) oder ein falsches Ergebnis (Failure) haben, die beide per Exception signalisiert werden.  
Failures (AssertionFailedError) werden erwartet, Errors sind unerwartet.

Vorgehensweise:

Programmierer schreibt zuerst einen automatisch wiederholbaren Test und dann den zu testenden Code. Der Test ist selbst ein Stück Software und wird ebenso wie der zu testenden Code programmiert. Tests werden nach jeder Änderung durchgeführt. Wenn nach einer Änderung ein Fehler auftritt, wars scheiße.

## Mockito (Sinn und Funktionsweise von Mocking-Bibliotheken)

Ein Mock ist ein Dummy-Element, das ein System/einen Baustein simuliert.

Um einen vollständig isolierten Test zu erreichen, müssen die Schnittstellen, über die das zu testende Objekt auf seine Umgebung zugreift, durch Mock-Objekte ersetzt werden. Mock-Objekte sind dabei Platzhalter für die echten Objekte. Mockito hilft diese Mock-Objekte samt ihrem Verhalten zu generieren und gegebenenfalls auch zu prüfen, wie diese vom zu testenden Code aufgerufen wurden. Mock-Objekte werden zur Laufzeit dynamisch generiert. Mockito kann Klassen und Interfaces mocken.

Verhalten des Mocks kann genau definiert werden (z.B. bei Eingabe von x soll immer y zurückgegeben werden). -> Test-Verhalten ist so immer voraussagbar.

* Mock-Objekt (Klasse|Schnittstelle) erzeugen:  
  
* Mock-Objekt im Unit-Test benutzen:  
  Ein Bild, das Text enthält.

  Automatisch generierte Beschreibung
* Verifizierung, ob das Mock-Objekt von den Methoden addCustomer() und dleteAllCustomers() richtig benutzt wurde:  
  

# Aufgabe 2: Ausgangsprojekt

Laden Sie sich das gegebene Maven-Ausgangsprojekt („TDD Kino Demo“, siehe Moodle) herunter. Laden Sie es als Maven-Projekt in ihre IDE und schauen Sie sich an, wie das Projekt aufgebaut ist:

* pom.xml (Dependencies, Java-Version etc.)
  + junit-jupter version 5.8.2
  + Mockito-junit-jupter version 4.3.1
* Gegebene Domänen-Klassen (Kinosaal, Ticket etc.)
  + Ticket:
    - String saal
    - Zeitfenster zeitfenster
    - LocalDate datum
    - char reihe
    - int platz
    - Konstruktor
    - Getter
  + KinoSaal:
    - String name
    - Map<Character, Integer> reihen
    - Konstruktor
    - boolean pruefePlatz(char reihe, int platz)
      * prüft ob es Platz gibt
    - public boolean equals(Object obj)
      * prüft ob obj instanceof KinoSaal ist
* Gegebene Start-Junit5-Tests in test /java/at.itkolleg/AppTest
  + @Mock:
    - KinoSaal kinosaalMock
    - Kinosaal kinosaalOriginial
  + @BeforeEach
    - setup
    - map mit Reihen und Plätzen füllen und einem Kinosaal zuweisen
  + @Test
    - shouldAnswerWithTrue()
    - testKinosaalMockName()
    - testKinosaalPlätze()
    - testKinoSaalName

Starten Sie den Test AppTest über den grünen Pfeil und versichern Sie sich, dass alles korrekt läuft. Starten Sie auch die App (main-Methode).

# Aufgabe 3: Einarbeitung in den gegebenen Code

Arbeiten Sie sich in den gegebenen Code zur Kinoverwaltung ein. Verwenden Sie die gegebenen Klassen KinoSaal, Ticket, Vorstellung, Kinoverwaltung in der App-Klasse (main-Methode), um ein Gefühl für die Funktionsweise des Programms zu bekommen. Führen Sie folgende Punkte durch:

* Kinosäle anlegen
* Vorstellungen anlegen
* Vorstellungen über die Kinoverwaltung einplanen
* Tickets für Vorstellungen ausgeben
* etc.

# Aufgabe 4: JUNIT-Tests für KinoSaal

Testen Sie alle Methoden der Klasse KinoSaal (Testklasse TestKinoSaal).

# Aufgabe 5: JUNIT-Tests für Vorstellung

Testen Sie alle Methoden der Klasse Vorstellung (Testklasse TestVorstellung).

# Aufgabe 6: JUNIT-Tests für KinoVerwaltung

Testen Sie alle Methoden der Klasse KinoVerwaltung (Testklasse TestKinoverwaltung).

# Aufgabe 7: JUNIT-Tests Advanced

Falls nicht schon in den vorhergehenden Aufgaben passiert, testen Sie folgende Punkte unter Verwendung der fortgeschrittenen Features von JUNIT 5:

1. Schreiben Sie einen Test, der validiert, dass das Anlegen einer Vorstellung korrekt funktioniert. Der Test sollte eine fachliche Bezeichnung haben und die Assertions sollten bei Validierungsfehler eine Hinweistext liefern.
2. Schreiben Sie einen Test, der validiert, dass das Einplanen mehrerer Vorstellungen korrekt funktioniert. Stellen Sie zudem sicher, dass beim möglichen Auftreten eines Fehlers trotzdem alle Validierungen ausgeführt werden.
3. Schreiben Sie einen Test, der sicherstellt, dass ein Fehler geworfen wird, wenn eine Veranstaltung doppelt eingeplant wird.
4. Schreiben Sie einen parametrisierten Test, der mehrere Ticketkäufe mit unterschiedlichen Parametern überprüft.
5. Schreiben Sie eine dynamische TestFactory die den Ticketkauf mit zufälligen Werten bombardiert. Der Test soll sicherstellen, dass der Ticketkauf entweder funktioniert oder nur einen der definierten Fehlermeldungen (z.B. new IllegalArgumentException("Nicht ausreichend Geld.")) ausgibt. Die Tests müssen reproduzierbar sein.

# Aufgabe 8: Mockito Einführung

Lesen Sie sich in das Mocking-Framework Mockito ein (Links siehe Moodle im Abschitt „Input zu Mockito“).

Verwenden Sie die wesentlichen Mockito-Möglichkeiten praktisch in kleinen Programmen.