hhu.

NKT
Tag 4: Datentypen, Bedingungen und Arrays

Sven Hoops

10.03.2025

Gliederung Tag4

- **Test-Doubles:** Stub, Dummy, Mock
- **Mocking mit Mockito:** Ein praktisches Beispiel (z.B. MailSender, Datenbank-Stub)
- **Häufige Probleme beim Testen:** Seiteneffekte und nicht-deterministisches Verhalten
- **Entkopplung im Arrange-Schritt:** Factory-Methoden und das Builder Pattern
- **Testdaten-Management:** Object Mother Pattern und Test Data Builder
- **Lesbare Tests:** Custom Assertions und das Design guter Assertions (z.B. mit AssertJ)
- **Test-Lifecycle:**Setup/Teardown-Methoden (@BeforeAll, @AfterEach)

Test-Doubles: Stub, Dummy, Mock

Heinrich Heine Universität Düsseldorf

- **Dummy:** Platzhalterobjekt, das übergeben, aber nie wirklich genutzt wird. Dient nur zur Erfüllung von Methodensignaturen.
- **Stub:** Liefert vordefinierte, "hartcodierteÄntworten auf bestimmte Aufrufe. Wird genutzt, um den Test unter kontrollierte Bedingungen zu setzen.
- **Mock:** Ein Objekt, das Erwartungen über die Art und Weise der Aufrufe definiert. Ein Test schlägt fehl, wenn diese Erwartungen nicht erfüllt werden. Mocks prüfen das Verhalten (Interaction Testing).

Aufgabe: Mocking

Heinrich Heine
Universität Düsseldorf

Wir benutzen eine Klasse `NotificationService`, die zur konstuktion eine Instanz von `MailSender` benötigt. Wir wollen testen, ob der Service diese Abhängigkeit korrekt aufruft, ohne eine echte Mail zu senden.

Probleme beim Testing und Seiteneffekte

- Änderungen am globalen Zustand, an Dateien, Datenbanken oder externen Systemen, die über den Test hinaus bestehen bleiben sind extrem schwer zu testen.
- **Probleme, die sie verursachen:**
  - **Flaky Tests:** Tests, die mal erfolgreich sind und mal fehlschlagen.
  - **Abhängige Tests:** Die Reihenfolge der Testausführung wird plötzlich relevant.
  - **Schlechte Performance:** Tests werden langsam (z.B. durch Datenbankzugriffe).
- **Lösungen:** Test-Doubles, In-Memory-Datenbanken[1].

[1] häufig nicht sinnvoll

Kopplung im Arrange-Schritt

Heinrich Heine Universität Düsseldorf

- **Problem:** Komplexe Objekterzeugung im `Arrange`-Teil des Tests macht ihn unleserlich und fragil.
- Lösung 1: Factory-Methoden
  - Private Hilfsmethoden, die gültige Testobjekte erstellen.
  - private Order createValidOrder()
- Lösung 2: Builder Pattern
  - Bietet eine flüssige API zur Erzeugung von Objekten mit spezifischen Eigenschaften.
  - Order order = new OrderBuilder().withStatus(Status.SHIPPED).build();

Object Mother / Testobjekte

- **Object Mother Pattern:** Eine Klasse, die eine Sammlung von statischen Factory-Methoden zur Erzeugung von typischen Testobjekten bereitstellt.
- **Vorteile:** Fördert die Wiederverwendung und sorgt für konsistente Testdaten.
- **Nachteile:** Kann zu unflexiblen und überladenen Klassen führen. Der **Test Data Builder** ist oft die flexiblere Alternative.

Builder Pattern (Gratis Punkte yay! :))

Heinrich Heine
Universität Düsseldorf

- Fluent-Api
- Besitzt build Methode, die das zu erstellende Objekt zurück gibt
- alle anderen Methoden geben sich selbst zurück

Custom Assertions und Assertions-Design

Heinrich Heine Universität Düsseldorf

- **Problem mit Standard-Assertions:**
  - `assertEquals(expected, actual)` sagt wenig über die Domäne aus.
  - Führt zu vielen einzelnen Assertions und unklaren Fehlermeldungen.
- **Lösung: Custom Assertions (z.B. mit AssertJ)**
  - Erstelle domänenspezifische Assertions, die die Lesbarkeit erhöhen.
  - Beispiel: `assertThat(user).isActive();` statt `assertTrue(user.isActive());`
  - Oder:
    `assertThat(order).hasStatus(OrderStatus.COMPLETED).containsProduct(product`
- **Vorteile:** Bessere Lesbarkeit, ausdrucksstärkere Tests und klarere Fehlermeldungen.

Setup/Teardown Methods

Heinrich Heine
Universität Düsseldorf

- **Setup und Teardown**
  - **`@BeforeAll / @AfterAll:`** Werden einmal pro Testklasse ausgeführt (z.B. für Datenbankverbindungen).
  - **`@BeforeEach / @AfterEach:`** Werden vor/nach jeder einzelnen Testmethode ausgeführt (z.B. um Objekte zurückzusetzen).

Endlich wieder Frühling

Heinrich Heine
Universität Düsseldorf

- **Spring Dependency Injection (DI):**
  - Kernkonzept: Inversion of Control (IoC)
  - Annotationen: `@Component`, `@Configuration`, `@Bean`
- **Injection-Arten:** Konstruktor, Setter, Field-Injection
- **Scopes:** Singleton (Standard), Prototype, Request, Session
- **Konfiguration:** Properties, YAML, Umgebungsvariablen
- **Logging-Frameworks:** SLF4J als Abstraktion, Logback/Log4j2 als Implementierung

Spring Dependency Injection (DI)

Heinrich Heine
Universität Düsseldorf

- **Inversion of Control (IoC):** Nicht der Entwickler, sondern das Framework ist für die Erzeugung und Verknüpfung von Objekten (Dependencies) zuständig.
- **Der Spring IoC-Container:** Verwaltet den Lebenszyklus von Objekten (genannt "Beans").
- **Zentrale Annotationen:**
  - **`@Component`:** Markiert eine Klasse als Spring-Bean.
  - **`@Configuration`:** Markiert eine Klasse, die Bean-Definitionen enthält.
  - **`@Bean`:** Markiert eine Methode in einer `@Configuration`-Klasse, die eine Bean erzeugt.

Injection-Arten

- **Constructor Injection (bevorzugt):**
  - Abhängigkeiten werden über den Konstruktor übergeben.
  - Macht Abhängigkeiten explizit und ermöglicht finale Felder (Immutability).
  - `@Autowired` am Konstruktor ist optional, wenn es nur einen gibt.
- **Setter Injection:**
  - Abhängigkeiten werden über Setter-Methoden injiziert.
  - Nützlich für optionale Abhängigkeiten.
- **Field Injection (abzuraten):**
  - Abhängigkeiten werden direkt in die Felder injiziert.
  - **Nachteile:** Erschwert das Testen, verletzt das Prinzip der Kapselung und versteckt Abhängigkeiten.

Scopes (Singleton, Prototype)

Heinrich Heine
Universität Düsseldorf

- **Scope:** Definiert den Lebenszyklus und die Sichtbarkeit einer Bean.
- **`Singleton` (Standard):**
  - Es existiert nur **eine einzige Instanz** dieser Bean im gesamten Spring-Container.
  - Jede Anfrage nach dieser Bean liefert dieselbe Instanz zurück.
- **`Prototype`:**
  - Jedes Mal, wenn die Bean angefordert wird, wird eine **neue Instanz** erzeugt.
  - Nützlich für zustandsbehaftete Beans.
  - Spring verwaltet nicht den kompletten Lebenszyklus von Prototype-Beans.
- **Weitere Scopes (Web-Kontext):** `request`, `session`, `application`.

Konfiguration mit Properties, YAML

Heinrich Heine Universität Düsseldorf

- **Ziel:** Applikations-Konfiguration von der Logik trennen.
- **`application.properties`:**
  - Standard-Format in Spring Boot.
  - Einfaches Key-Value-Format, z.B. `server.port=8080`.
- **`application.yml` (YAML):**
  - Strukturiertes, hierarchisches Format.
  - Oft besser lesbar für komplexe Konfigurationen.
- Werte können via `@Value` oder `@ConfigurationProperties` in Beans injiziert werden.
- **Priorität:** Umgebungsvariablen überschreiben YAML/Properties.

Race Conditions und Deadlocks

- **Race Condition:**[2]
  - Entsteht, bei ungünstigem Timing von verschiedenen Threads.
  - Populärstes Beispiel: Dirty-COW
- **Deadlock:**
  - Eine Situation, in der zwei (oder mehr) Threads blockiert sind und permanent aufeinander warten.
  - **Beispiel:** Thread A sperrt Ressource 1 und wartet auf Ressource 2. Thread B sperrt Ressource 2 und wartet auf Ressource 1. Keiner kann fortfahren.

[2] Nicht Klausurrelevant

Entitäten und Value Objekte

- **Entity (Entität):**
  - Ein Objekt, das nicht durch seine Attribute, sondern durch seine **eindeutige Identität** definiert wird.
  - Die Identität bleibt über den gesamten Lebenszyklus konstant, auch wenn sich die Attribute ändern.
  - Beispiel: Ein `Kunde` mit der Kundennummer 123. Der Name oder die Adresse kann sich ändern, es bleibt aber derselbe Kunde.

- **Value Object (Werteobjekt):**
  - Ein Objekt, das durch seine **Attribute** definiert wird. Es hat keine eigene Identität.
  - Sollte **immutable** (unveränderlich) sein.
  - Beispiel: Eine `Adresse` (Straße, PLZ, Stadt). Wenn sich die Straße ändert, ist es eine neue Adresse, keine geänderte.

Aggregate

Heinrich Heine Universität Düsseldorf

- Eine Gruppe von zusammengehörigen Entities und Value Objects, die als eine Einheit behandelt wird.
- Das **Aggregate Root** ist die primäre Entity, über die alle Zugriffe auf das Aggregat erfolgen.
- Schützt die Konsistenz und stellt sicher, dass Geschäftsregeln (Invarianten) eingehalten werden.
- Beispiel: Eine `Bestellung` (Aggregate Root) mit ihren `Bestellpositionen` (Entities).