T 03 Modul 450: Testdoubles

Softwaretests spielen eine zentrale Rolle bei der Entwicklung von zuverlässigen Anwendungen. Dabei stellt der Umgang mit Abhängigkeiten in Tests eine besondere Herausforderung dar.
Test Doubles sind Stellvertreter für echte Objekte oder Komponenten in Tests. Sie ermöglichen es, Abhängigkeiten zu isolieren und das Verhalten der zu testenden Einheiten gezielt zu überprüfen. In dieser Dokumentation werden die inhaltlichen Schwerpunkte der Präsentation beschrieben, inklusive Problemstellungen, Lösungsansätze und die unterschiedlichen Arten von Test Doubles.

Inhaltsverzeichnis

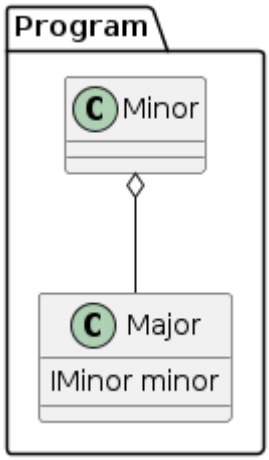
1. Probleme durch Abhängigkeiten

Tests von Komponenten, die von anderen Klassen oder externen Systemen abhängen, bergen typische Herausforderungen:

- **Langsamkeit**: Abhängigkeiten greifen auf externe Dienste oder Datenbanken zu, was die Tests verlangsamt.
- **Aufwand**: Die Vorbereitung von Abhängigkeiten in einem bestimmten Zustand ist oft kompliziert.
- **Schwierigkeit**: Abhängigkeiten in ein spezifisches Verhalten zu zwingen, kann komplex sein.

1.1. Beispielproblem

Ein typisches Szenario besteht darin, dass eine Klasse (z. B. `Major`) eine andere Klasse (z. B. `Minor`) verwendet. Tests für `Major` beinhalten somit indirekt auch Tests für `Minor`. Dies führt dazu, dass Fehler oder unerwartetes Verhalten von `Minor` die Tests von `Major` beeinflussen.



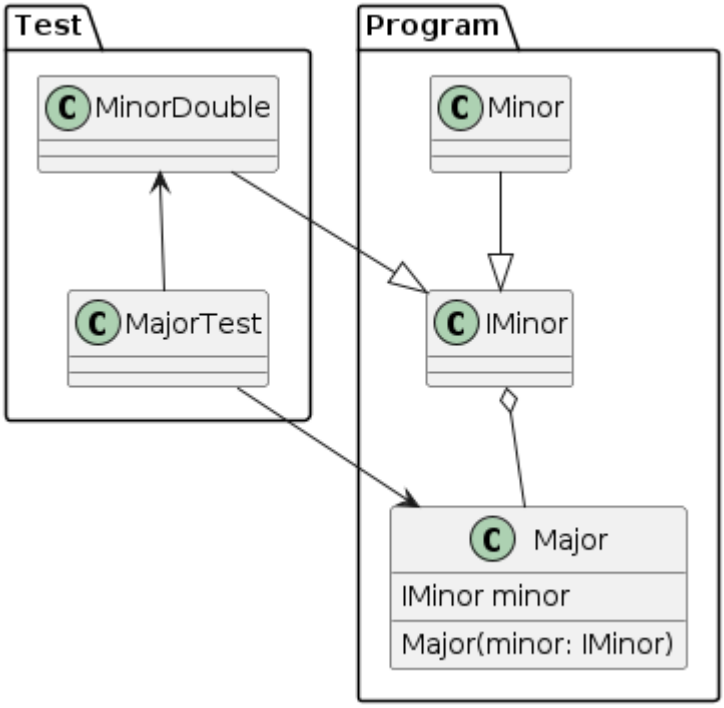
2. Dependency Injection

Ein zentraler Lösungsansatz zur Trennung von Abhängigkeiten ist **Dependency Injection**. Anstelle einer direkten Instanziierung von Abhängigkeiten werden diese durch Schnittstellen abstrahiert und von aussen bereitgestellt. Dies erlaubt es, im Test eine spezialisierte Implementierung (z. B. ein Test Double) zu verwenden.

2.1. Beispiel

Das folgende Diagramm zeigt die Anwendung von Dependency Injection:

- Die Klasse `Major` erwartet eine Implementierung von `IMinor` im Konstruktor.
- Im Test kann ein spezielles Test Double (`MinorDouble`) als Abhängigkeit bereitgestellt werden.



3. Mocking

Mocking bezeichnet die Verwendung von Frameworks, um das Verhalten von Abhängigkeiten zu simulieren. Beispiele für Mocking-Frameworks:

- **Java**: Mockito
- **C#**: Moq
- **Python**: unittest.mock
- **JavaScript**: mocha

Mit diesen Frameworks lassen sich:

- Verhalten von Objekten simulieren (z. B. Rückgabewerte festlegen).
- Aufrufe von Methoden prüfen (z. B. Häufigkeit und Parameter).

4. Arten von Test Doubles

Test Doubles können je nach Anwendungsfall in verschiedene Kategorien eingeteilt werden:

4.1. Dummy

- **Definition**: Platzhalterobjekt, das für den jeweiligen Testfall irrelevant ist.
- **Beispiel**: Eine Klasse hat zwei Abhängigkeiten (z. B. Cache und Berechnung). Der Cache wird benötigt, die Berechnung jedoch nicht. Für die Berechnung genügt ein Dummy.

4.2. Fake

- **Definition**: Vereinfachte Implementierung einer Abhängigkeit.
- **Beispiele**:
  - Datenbank wird durch eine `ArrayList` simuliert.
  - DNS-Lookup wird durch eine statische `Map` nachgebildet.

4.3. Stub

- **Definition**: Pseudo-Implementierung mit hart kodierten Antworten.
- **Eigenschaften**:
  - Keine Programmlogik.
  - Ignoriert Methodenparameter.
  - Liefert stets dieselben Antworten.

4.4. Mock

- **Definition**: Erweiterter Stub mit "Erinnerungsvermögen".
- **Eigenschaften**:
  - Merkt sich, wie oft und mit welchen Parametern Methoden aufgerufen wurden.
  - Kann verwendet werden, um Aufrufbedingungen zu überprüfen.

4.5. Spy

- **Definition**: Produktive Implementierung, die von einem Wrapper umschlossen ist.
- **Eigenschaften**:
  - Aufrufe und Parameter werden aufgezeichnet.
  - Test bleibt ein Integrationstest mit zusätzlichen Prüfungen.