LU06a - Testing Doubles

Test Doubles sind Hilfsobjekte, Abhängigkeiten zu ersetzen und so Bereiche testen zu können, ohne, dass produktive, externe (bzw. ausserhalb des eigenen Testbereichs liegende) Systeme dafür benutzt werden müssen. In der folgenden Tabelle ist eine kleine Übersicht über verschiedene Test Doubles in aufsteigender Reihenfolge. Es ist grundsätzlich empfohlen, die einfachste (oberste) Variante zu wählen, die funktioniert.

Double-Typ	Beschreibung	Entscheidungsfrage
Dummy	Platzhalter	Brauche ich nur ein Objekt?
Stub	Vordefinierte Antworten	Welchen Wert soll zurückkommen?
Fake	Vereinfachte echte Implementierung	Kann ich eine einfache echte Version bauen?
Spy	Aufrufe beobachten	Brauche ich Zusatzcode, um etwas zu beobachten?
Mock	Überprüfung direkt bei Aufruf	Ist es sinnvoll, den Test bereits abzubrechen?

Dummy

Ein `Dummy` wird verwendet, um eine Funktion aufzurufen, die ein bestimmtes Objekt erfordert, welches aber nicht getestet werden soll. Anstelle eines realistischen Objekts wird lediglich ein Dummy-Objekt mitgegeben. Dieses Dummy-Objekt enthält keine Logik und dient nur dazu, dass der Code lauffähig ist.

```
class DummyLogger {
  log() {
    // Empty
  }
}

const logger = new DummyLogger()
createUserService(logger)
```

Im Beispiel ist ersichtlich, dass `createUserService` ein Objekt benötigt, auf welchem die Methode `log()` aufgerufen werden kann. Da die Implementation für den Test jedoch irrelevant ist, reicht es aus einen leeren Body zu verwenden.

Stub

Ein `Stub` gibt zu einem Input explizit vordefinierte Werte zurück. Anstatt, dass zum Beispiel ein Objekt aufwändig geladen oder erstellt wird, wird ein Testobjekt zurückgegeben.

Dadurch ist ein `Stub` im Prinzip wie eine Tabelle, in welcher zu einem bestimmten erwarteten Input der entsprechenden Output nachgeschlagen wird.

```
const userRepositoryStub = {
  findUserById(id) {
    return { id: 1, name: "Alice" }
```

```
  }
}

const service = new UserService(userRepositoryStub)

const user = service.getUser(1)
```

In diesem Beispiel wird für die Id 1 immer das Alice-Testobjekt zurückgegeben, auch wenn die Datenbank andere Werte hätte.

Fake

Ein Fake ist eine vereinfachte Implementierung. Zum Beispiel wird eine Heuristik oder sonst eine schnelle Abkürzung verwendet.

```
class FakeUserRepository {
  constructor() {
    this.users = new Map()
  }

  save(user) {
    this.users.set(user.id, user)
  }

  findUserById(id) {
    return this.users.get(id)
  }
}

const repo = new FakeUserRepository()
repo.save({ id: 1, name: "Alice" })

const service = new UserService(repo)

expect(service.getUser(1).name).toBe("Alice")
```

Oft ersetzen Fakes in Tests Services wie Datenbanken mit einer einfacheren In-Memory-DB wie auch in diesem Beispiel. Die Objekte sind dadurch nur zur Laufzeit vorhanden, was aber für die Ausführung des Tests vollkommen ausreicht.

Spy

Ein Spy enthält Logik, um den Testverlauf zu überwachen. Zum Beispiel merkt er sich, wie oft oder mit welchen Argumenten eine Methode aufgerufen wurde.

Es ist auch möglich, dass der Spy die echte Logik zusätzlich aufruft.

```
class EmailServiceSpy {
  constructor() {
    this.callCount = 0
    this.messages = []
  }

  sendEmail(message) {
    this.callCount++
    this.messages.push(message)
  }
}

class UserService {
  constructor(emailService) {
    this.emailService = emailService
  }

  registerUser(email) {
    this.emailService.sendEmail("Welcome " + email)
  }
}

const emailSpy = new EmailServiceSpy()
const service = new UserService(emailSpy)

service.registerUser("alice@test.com")

if (emailSpy.callCount !== 1) {
  throw new Error("Email should be sent exactly once")
}

if (emailSpy.messages[0] !== "Welcome alice@test.com") {
  throw new Error("Wrong email content")
}
```

Mock

Ein `Mock` beinhaltet Logik, um die Eingabe oder den Zustand zum Zeitpunkt des Aufrufs zu überprüfen. Im Fehlerfall wird der Test abgebrochen, anstatt erst am Schluss die Überprüfungen zu machen.

```
class OrderService {
  constructor(paymentGateway, repository) {
    this.paymentGateway = paymentGateway
    this.repository = repository
  }

  placeOrder(order) {
    this.paymentGateway.authorize(order.amount)
```

```
    this.repository.save(order)

    this.paymentGateway.capture(order.amount)
  }
}

class PaymentGatewayMock {
  constructor() {
    this.authorized = false
  }

  authorize(amount) {
    if (this.authorized) {
      throw new Error("authorize called twice")
    }
    this.authorized = true
  }

  capture(amount) {
    if (!this.authorized) {
      throw new Error("capture called before authorize")
    }
  }
}

const paymentMock = new PaymentGatewayMock()

const repositoryStub = {
  save() {}
}

const service = new OrderService(paymentMock, repositoryStub)

service.placeOrder({ amount: 100 })
```