Lösungen zum Praktikum Mock-Testing

1. Herz

Funktionserläuterung zum Herzschlag finden Sie in der Aufgabenstellung

2. Aufgaben

2.1. Einführung in Mockito

a. Studieren Sie die Testmethoden `HeartTest::testValveStatus()` und `HeartTest::testExecuteHartBeatErrorBehaviour()`. Dort wird **ein Teil** des Verhaltens des Herzens getestet.

b. Implementieren Sie die Methoden `Heart::executeDiastole()`, `Heart::executeSystole()` und `Heart::executeHeartBeat()` sodass die bestehenden Tests durchlaufen und das oben beschriebene Verhalten des Herzens modelliert wird.

c. (optional) Ein echtes Herz hat eine Schlagfrequenz. Implementieren Sie, dass das Herz nach jeder Systole pausiert.

Siehe Musterlösung: `Heart.executeDiastole()`, `Heart.executeSystole()` und `Heart.executeHeartBeat()`

2.2. Fragen zu Testing [TU]

a. Testing kann in zwei unterschiedliche Strategien aufgeteilt werden. Zum einen gibt es White-Box-Testing und zum zweiten Black-Box-Testing. Was für Java Libraries gibt es, um diese zwei Strategien zu testen? Wann wenden Sie welche Strategie an?

- JUnit ist ein Framework für Unit-Tests (auch Modul-Tests genannt). Es wird sowohl für White-Box-Testing als auch für Black-Box-Testing angewendet.
- Für White-Box-Testing eignet sich Mockito, weil damit die Abläufe innerhalb einer Klasse getestet werden können (behavior testing).
- Black-Box-Testing trifft keine Annahmen darüber, **wie** eine Klasse oder Methode eine Aufgabe erledigt. Es wird lediglich überprüft, ob auf eine bestimmte Eingabe die erwartete Ausgabe erfolgt.
- Nichttriviale Methoden sollten immer mindestens mit Black-Box-Testing getestet werden. Diese Tests werden am besten von einer anderen Person geschrieben als von der Programmierer:in.
- Für White-Box-Testing werden Informationen benötigt, **wie** eine Klasse oder Methode eine Aufgabe erledigt.

b. Das Erstellen von guten automatisierten Unit-Tests kann manchmal schwierig umzusetzen sein. Was ist der Hauptgrund dafür? Wie können Sie dieses Problem entschärfen?

- Das Setup eines Unit-Tests kann kompliziert werden, wenn mehrere Klassen dafür instanziiert und konfiguriert werden müssen.
- Das Problem kann entschärft werden, indem
  - die verschiedenen Klassen besser entkoppelt werden und
  - die Klassen, von denen die zu testende Klasse abhängig ist, zu mocken.

c. Der Testfokus war bisher auf der Klasse `Heart`. Testen Sie jetzt die Klasse `Half`. Wo verwenden Sie Stubbing, wo Mocking?

- Die Funktionalität der Klasse Half nicht besonders umfangreich. Alle Methoden der Klasse bestehen aus nur genau einer Zeile und die Klasse hat bis auf das Datenfeld `Half.side` keinen State. Was schieflaufen könnte ist, dass jemand beispielweise auf einem Ventil anstatt der `close()` Methode die `open()` Methode aufruft. Weil wir in der Klasse `Half` nicht wirklich einen Status prüfen können, macht es mehr Sinn das Verhalten der Klasse zu prüfen (ruft die Klasse die richtigen Methoden auf) → Das können wir mit Mocking (in Mockito mit `verify`) erreichen. Entsprechend würde sich hier ein Mock besser eignen als ein Stub. Über die Frage, wie sinnvoll es ist solch einfache Klassen zu simulieren, lässt sich Diskutieren und ist abhängig vom jeweiligen Testkonzept.
- Beim Auftrag die Klasse `Half` zu testen, stossen Sie auf das Problem, dass Sie weder Stubs noch Mocks anwenden können, weil die Klassen von welchen `Half` abhängt direkt im Konstruktor von `Half` initialisiert werden. Damit können Sie kein Test-Double verwenden (injecten). Das Problem könnten Sie lösen, wenn es einen entsprechenden Konstruktor in der Klasse `Half` gäbe, welche die Instanzen der Abhängigen Typen entgegennimmt (analog zu `Heart`).