Testkonzept (V1.1)

Dokumenteninformation

Erstellt von: Johannes Hüttinger

Bearbeitet von: Johannes Hüttinger

Dateiname: Testkonzept_v1.1.pdf

Versionierung

Version	Datum	Bearbeiter	Änderung	Stand
1.0	17.04.2021	Johannes Hüttinger	Initiale Erstellung	Abgenommen
1.1	22.04.2021	Johannes Hüttinger, Daniel Scherer	Grobe Überarbeitung, Fehlerkorrekturen	Abgenommen

Inhalt

1 Allgemeines

Das Ziel dieses Projektes ist, ein Softwareprodukt zu entwickeln, das nach Übergabe an den Kunden sowohl ausgeführt als auch weiterentwickelt werden kann. Um dies zu erleichtern und gleichzeitig auch die Softwarequalität zu gewährleisten, sollen umfangreiche Tests geschrieben werden. Dieses Dokument soll dazu Orientierung geben.

Alle Tests können vom Entwickler jederzeit in der Entwicklungsumgebung (IntelliJ, optional mit Plugin “kotest”) oder mit dem jeweiligen Buildtool ausgeführt werden. Außerdem werden sie bei einer Pull Request und einem Push auf den Branch master in der CI-Umgebung (Github Actions) ausgeführt. Die Tests des Servers befinden sich in einem eigenen Test-Ordner (server/src/**test**/), wobei das Package das gleiche wie das des Testlings ist. Auf dem Client dagegen sind diese im gleichen Ordner wie die jeweilige zu testende Klasse, wobei die Testklasse den gleichen Namen mit der Dateiendung `.spec.ts` statt nur `.ts` hat. UI- und Systemtests befinden sich im Ordner client/e2e/src/.

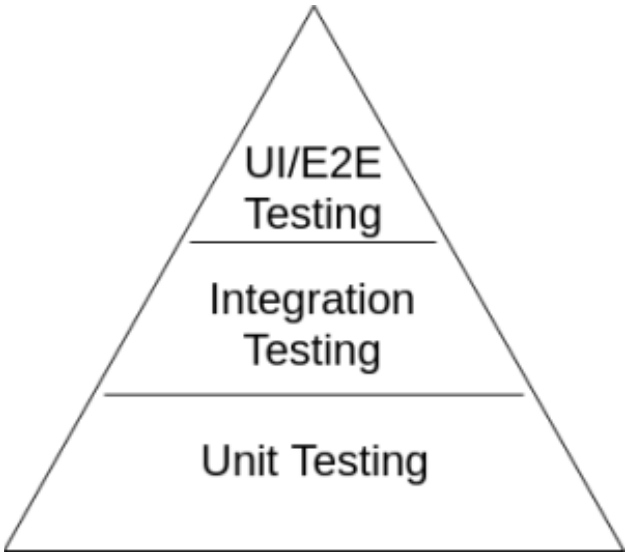
1.1 Testaufbau

Tests werden automatisch dokumentiert, weswegen aussagekräftige Testnamen äußerst wichtig sind. Die Ergebnisse werden automatisch auf tests.bm-games.net veröffentlicht. Der Name sollte auf Englisch formuliert werden und sowohl die Ausgangsbedingung nennen (“Without a network connection”), als auch das erwartete Ergebnis (“my login should fail”).

Im Test an sich wird mit Assert-Statements das Ergebnis des getesteten Codes gegen ein erwartetes Ergebnis geprüft. Ein Beispiel ist im Abschnitt Unit Tests aufgeführt. Die zu testende Funktionalität sollte sowohl nach Erfolg (“Login klappt”), als auch nach Misserfolg (“Falsches Passwort”, “User existiert nicht”) getestet werden. Dies ist gerade für UI- und System-Tests zu beachten. Tests an sich sollten immer nur eine Funktionalität überprüfen, das heißt mehrfache Assert-Statements *sollten* vermieden werden. Stattdessen können in einer Test-Klasse mehrere Tests hinzugefügt werden, die die unterschiedlichen Funktionalitäten prüfen.

2 Testarten

Die aus der Vorlesung bekannte Testhierarchie dient zur groben Orientierung bezüglich Form, Reihenfolge und Menge der Tests. Im Folgenden werden die einzelnen Stufen und ihre konkrete Anwendung näher beschrieben.



Quelle:
https://convincingbits.files.wordpress.com/2019/11/test_pyramid.png?w=300&h=265

2.1 Unit Tests

Unit Tests sind sogenannte White Box Tests, das heißt das “Innere” des getesteten Codes ist bekannt und wird gezielt getestet. Deshalb ist auch jeder Entwickler für Unit Tests seines eigenen Codes verantwortlich. Es sollen möglichst alle nichttriviale Methoden und Funktionalitäten jeder Klasse getestet werden. Eine Orientierung für die Menge der Tests bietet die Code Coverage (siehe Abnahme).

2.1.1 Kotlin

In Kotlin werden die Tests mit dem Framework Kotest geschrieben. Dabei können auch die aus JUnit bekannten Listener (z.B. beforeEach, beforeTest) verwendet werden. Ein Beispiel für einen Test mit Listener könnte so aussehen:

```
class MyTests : FunSpec({
    beforeTest {
      println("Starting a test $it")
    }
    afterTest { (test, result) ->
      println("Finished spec with result $result")
    }
    test("String length should return the length of the string") {
```

```
        "sammy".length shouldBe 5
        "".length shouldBe 0
    }
})
```

In einer Testklasse können auch mehrere Tests geschrieben werden.

2.1.2 Angular

Angular bringt den Testrunner Karma mit, der die mit dem Framework Jasmine geschriebenen Tests ohne echten Browser ausführt. Dadurch wird die Performance verbessert. Die Syntax sieht wie folgt aus:

```
describe('AppComponent', () => {
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [
        RouterTestingModule
      ],
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  });

  it(`should have as title 'client'`, () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.componentInstance;
    expect(app.title).toEqual('client');
  });

});
```

Auch hier sind sowohl Listener (z. B. beforeEach), als auch mehrfache Tests pro Klasse möglich.

2.2 Integrationstests

Bei der Integration von einzelnen Komponenten (Units) wird die erfolgreiche Zusammenarbeit derselben getestet. Dabei ist das Innere der Testlinge unbekannt (Black Box Test). Es wird nur die äußere Schnittstelle derselben getestet. Verantwortlich dafür sind die integrierenden Entwickler, die aber auch von den Entwicklern der einzelnen Untermodule unterstützt werden sollten. Für Tests des Servers “von außen” steht die Ktor-Integration von Kotest zur Verfügung, die das Simulieren von Client-Anfragen erlaubt und so Komponenten des Servers “von außen” getestet werden:

```
class ApplicationTest : FunSpec({
    test("Root url should display 'Hello World'" ){
        withTestApplication({ configureRouting() }) {
            handleRequest(HttpMethod.Get, "/").apply {
                response shouldHaveStatus HttpStatusCode.OK
                response shouldHaveContent "Hello World!"
            }
        }
    }
})
```

2.2.1 Mocks

Oft müssen bei Integrationstests bestimmte Komponenten gemockt werden (z.B. Datenbankanbindung). Dazu steht auf der Serverseite das Framework mockk zur Verfügung. Oft ist es notwendig, für jede Methode den Mock zurückzusetzen, wofür die bereits beschriebenen Listener genutzt werden.

```
class MyTest : FunSpec({

    val repository = mockk<MyRepository>()
    val target = MyService(repository)

    test("Saves to repository") {
        every { repository.save(any()) } just Runs
        target.save(MyDataClass("a"))
        verify(exactly = 1) { repository.save(MyDataClass("a")) }
    }

    test("Saves to repository as well") {
        every { repository.save(any()) } just Runs
        target.save(MyDataClass("a"))
        verify(exactly = 1) { repository.save(MyDataClass("a")) }
    }

    afterTest {
        clearMocks(repository)  // <---- Reset Mock after each test
    }
})
```

2.3 UI-/ Systemtests

Hier wird aus der Perspektive des Endnutzers getestet, d. h. das Testframework simuliert diesen, indem es sich durch die Applikation klickt. Deshalb wird hier nur auf Clientseite getestet. Angular bietet hierzu den Testrunner Protractor, der die Tests in einem echten Browser ausführt. Die Tests an sich werden auch mit dem Framework Jasmine geschrieben, weshalb sie die gleiche Syntax wie die Angular-Integrationstests haben.

Es sind, wo möglich, die einzelnen vereinbarten Anforderungen aus dem Pflichtenheft zu testen, wobei diese im Test aus User-Sicht durchgegangen werden und jeweils der positive als auch der negative Ausgang getestet werden soll, falls dies sich anbietet. Die Testnamen sollten hier auch den jeweiligen Geschäftsprozess im Namen erwähnen. Beispiele für Tests von /F30/ (Login / Logout des Benutzers): 1. “/F30/ Login klappt mit richtigen Daten” 2. “/F30/ Login schlägt fehl mit falschen Daten” 3. “/F30/ Logout klappt”

2.3.1 Leistungstests

Diese Tests prüfen, ob bestimmte Funktionen der Anwendung auch unter Stress funktionstüchtig bleiben. Die Ausführung und Dokumentation erfolgt vorerst manuell, hierfür ist der Testbeauftragte verantwortlich. Diese Features sollen hier getestet werden: * Chatfunktion

3 Abnahme

Die Implementierung wird nach einzelnen Features, die in sogenannten User Stories festgehalten werden, aufgeteilt. Jedes Feature sollte auf einem eigenen Branch implementiert werden, welcher dann per Pull Request in die Abnahme gegeben werden kann. Hier laufen dann automatisch alle Tests durch und es wird die sogenannte Code Coverage mit dem Tool Codecov erzeugt, die angibt, welcher Anteil des Codes von Tests abgedeckt ist. Falls sowohl alle Tests durchgelaufen sind, als auch die Code Coverage mindestens 75% beträgt, kann der neue branch nach einem Codereview vom Team oder einem unbeteiligten Entwickler auf master gemergt werden. Falls die gewählte Coverage nicht umsetzbar oder zu niedrig ist, kann dies in den nächsten Wochen noch angepasst werden, jedoch bietet dieser Wert erstmal eine gute Balance zwischen Qualität und Machbarkeit.

4 Ausführen der Tests

```
//Client
cd client
npm test
npm e2e //UI-/ Systemtests ausführen

//Server
cd server
gradlew :test
```

5 Technologien und Weiterführende Links

- **Kotlin**
  - Kotest: https://kotest.io/docs/framework/framework.html
  - Assertions: https://kotest.io/docs/assertions/core-matchers.html
  - Ktor-Assertions: https://kotest.io/docs/extensions/ktor.html
  - mockk: https://mockk.io/
- **Angular:** https://angular.io/guide/testing
- **Codecov:** Analysiert Code Coverage