SEMINARARBEIT

im Studiengang Informatik

Lehrveranstaltung Softwaretest

BDD mit Cucumber in Gilded Rose Kata

Ausgeführt von: Lena Krammer, Maria Sattler, Jacob Suchorabski

07.05.2020, Wien

FH University of Applied Sciences
TECHNIKUM WIEN

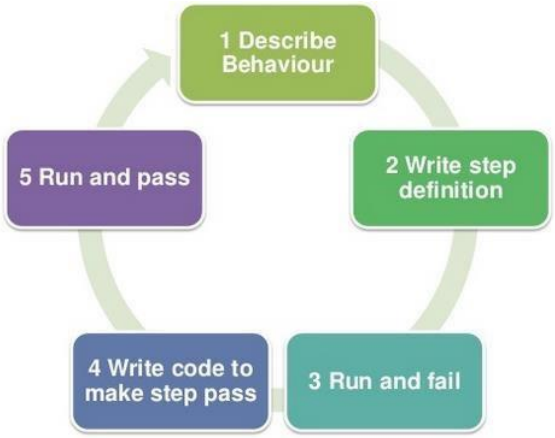
Inhaltsverzeichnis

BDD – Behavior Driven Development

Beim Behavior Driven Development wird das Format "Given When Then" verwendet. Dabei wird eine Ausgangssituation im "Given"-Teil beschrieben. Die Aktion, welche aus dieser Ausgangssituation geschehen soll, wird im "When"-Teil definiert. Anschließend wird im "Then"-Teil angegeben, welches Ergebnis das System liefern soll. [1]
Der große Vorteil bei BDD und dem "Given When Then"-Format ist, dass die Tests nicht in Code-Form von EntwicklerInnen geschrieben werden müssen, sondern in klaren, verständlichen Sätzen von jedem (z. B. Business AnalystInnen ohne technischem Wissen) geschrieben werden können.

Ablauf

Der geschriebene Code wird anhand der Testskripte, die aus den definierten Szenarien entstanden sind, getestet. Jener Code-Anteil, der die Tests nicht besteht, wird einem Refactoring unterzogen. Der Code-Anteil, der die Tests bestanden hat, wird eingefroren.



Um dieses Konzept bestmöglich umzusetzen, verwendet man ein Behavior Driven Development Framework. Eines davon ist Cucumber. [2]

Cucumber

Cucumber ist ein Open Source Testing Tool, welches Behavior Driven Development unterstützt. Mit Cucumber können Tests geschrieben werden, die jeder versteht und formulieren kann – nicht nur geschulte EntwicklerInnen. Cucumber ist nämlich ein Testing Framework, das englischen Klartext in Testskripte verarbeiten kann.

AnwenderInnen schreiben BDD Szenarien oder acceptance tests, die das Verhalten des Systems aus Sicht des Kunden und der Kundin beschreiben sollen. Anschließend werden diese Szenarien über Code für das System getestet.
Der große Vorteil ist das bei Cucumber die Dokumentation, das Tool zur Testautomatisierung und zur Entwicklungsunterstützung in einem zu finden ist. [1] [2]

Gherkin

Zu Cucumber gehört auch Gherkin, jene Sprache, in der die "Given When Then"-Szenarien geschrieben werden. Gherkin ist hauptsächlich englischer Klartext. Gherkin besteht aus einem Wortschatz von üblichen englischen Wörtern, die von den SzenarienschreiberInnen verwendet werden können, und ein paar Grammatikregeln. Der Grund für das Einführen von Gherkin war, dass nun SzenarienschreiberInnen verschiedene Formulierungen eines Satzes angeben können, aber trotzdem auf dasselbe Ergebnis kommen in Form eines Testskriptes. Außerdem definiert Gherkin ein strukturiertes Format, welches von Cucumber verstanden werden kann. [2][3]

Funktionsweise

Die Tests bestehen aus zwei Files – dem Feature-File und dem Code-File.
Im Feature-File werden die Szenarien in Klartext definiert. Im Code-File befinden sich die ausformuliert Tests in Codeform.
Cucumber liest den Klartext (geschrieben in Gherkin) vom Feature-File ein und sucht sich die exakte Übereinstimmung jedes Schrittes in der sogenannten "Step definition", welche im Code-File steht. [2]

Feature-File

Ein Feature ist meist eine unabhängige Funktion des zu testenden Systems. Mehrere Szenarien für das Feature, die getestet werden sollen, werden in einem Feature-File zusammengefasst. Zum Best Practice zählt, dass im Feature-File eine kleine Beschreibung des Features eingefügt wird, um als Dokumentation zu dienen.
Somit sieht der Aufbau des Feature-Files wie folgt aus:

- Feature – Name des zu testenden Features
- Beschreibung (optional) – kurze Beschreibung des Features
- Szenario – was ist das Testszenario
- Given – Voraussetzungen bevor der Test ausgeführt wird
- When – Bedingung, die erfüllt sein muss, damit der nächste Schritt ausgeführt wird
- Then – was soll passieren, wenn die Bedingung von "When" erfüllt ist [2]

Code-File mit Step Definition

Im Code-File werden dann die sogenannten "Step Definitions" definiert. In diesen Step Definitions wird beschrieben, welcher Satzbau in den Feature-Files zu welcher Testmethode gehört. Wenn nun also Cucumber einen Schritt des Szenarios im Feature-File ausführt, wird nach der passenden Step Definition gesucht und diese ausgeführt. Für jeden Teil des "Given When Then"-Szenarios kann eine eigene Code-Funktion implementiert werden. [2]

Installation

Cucumber ist für mehrere Plattformen verfügbar. In unserem Beispiel haben wir uns auf Java fokussiert. In Java kann Cucumber ganz einfach als Dependency über Maven oder Gradle importiert werden und ist auf der „Getting Started“ Seite von Cucumber beschrieben. [4] Bei der Installation von Cucumber ist zu Entscheiden ob mit neuen Lambda-Expression oder mit klassischen Methoden gearbeitet wird, da hierfür jeweils unterschiedliche Dependencies zu installieren sind.

```
1. //Benötigte Dependencies in build.gradle
2. dependencies {
3.     testImplementation 'org.junit.jupiter:junit-jupiter:5.5.2'
4.     testImplementation 'io.cucumber:cucumber-java:5.6.0'
5. }
6.
7. configurations {
8.     cucumberRuntime {
9.         extendsFrom testImplementation
10.     }
11. }
```

Konfiguration

Für die Erstellung des Feature-Files und der eigentlichen Implementation der Tests muss man lediglich die Standard Java Struktur respektieren. Die Feature-Files müssen als Ressource gespeichert werden und die Implementation im Test Ordner anlegt werden. Die Cucumber-Tests können danach schon ausgeführt werden.
Cucumber ermöglicht zusätzlich die Konfiguration von mehreren Profilen, um beispielsweise gewisse Tests nur in der DEV Umgebung auszuführen.
Typregistrierungen können auch konfiguriert werden, um sicherzustellen das benötigte Klassen richtig erkannt werden.

Szenarien für Gilded Rose Kata mit Cucumber

Wir haben uns in diesem Projekt zur Aufgabe gemacht, die Gilded Rose Kata mit Cucumber zu testen.
Als Testumgebung dient IntelliJ und als Programmiersprache wurde Java gewählt.
Somit haben wir begonnen, die Anforderungen der Gilded Rose Kata mit "Given When Then"-Szenarien abzudecken.
Vorab wurde überlegt, wie die Tests aussehen sollen. Uns ist aufgefallen, dass alle Szenarien mit einem Satzformat abgedeckt werden können: Ein gegebenes Item mit gegebener Quality und SellIn soll nach beliebig vielen Tagen eine gewisse Veränderung bezüglich Quality und SellIn haben.
Wichtig ist, immer die gleiche Satzstellung zu verwenden, damit die passende Funktion für das Szenario aufgerufen wird. Somit benötigen wir nur wenig Code für das Testskript und können aber viele Szenarien damit abdecken. Der große Vorteil darin ist, dass es für die EntwicklerInnen weniger Aufwand bedeutet, wenn sich die SzenarienschreiberInnen an das vorgegebene Format halten.
Die Szenarien wurden in einzelne Feature-Files aufgeteilt – je ein Feature-File wurde für normale Items, conjured Items, Brie, Backstage passes und Sulfuras verwendet. In den jeweiligen Feature-Files wurden die Szenarien für das jeweilige Item definiert.

```
#Feature File Beispiel
Feature: Verify Aged Brie behaviour

  Scenario: Aged brie increases in quality by 1 before sellIn
    Given item named "Aged Brie" with quality 10 and sellIn 2
    When 1 days pass
    Then quality should be 11 and sellIn should be 1

  Scenario: Aged brie increases in quality by 2 after sellIn
      Given item named "Aged Brie" with quality 10 and sellIn -1
      When 1 days pass
      Then quality should be 12 and sellIn should be -2

   Scenario: Aged brie quality does not rise over 50
      Given item named "Aged Brie" with quality 50 and sellIn 5
      When 1 days pass
      Then quality should be 50 and sellIn should be 4
```

Testen der Szenarien

Zum Testen der Szenarien fehlte nach der Erstellung der Feature-Files noch das Code-File mit den Step Definitions. Hierbei gibt es einen kleinen Trick. Wenn man jetzt die Applikation startet, erscheint in der Konsole eine Fehlermeldung, dass zu den Szenarien keine passende Implementierung vorhanden ist und zusätzlich wird eine mögliche Implementierung der Step Definition vorgeschlagen. Somit kann man nun das Code-File erstellen und die vorgeschlagene Implementierung hineinkopieren. Dabei machten wir uns auch den vorab definierten Satzbau zunutze. Cucumber analysiert den Satzbau in den Feature-Files und benötigt je nach unterschiedlichem Satzbau Step Definitions. Dadurch, dass wir vorab eine Art von Satzbau definiert und die Szenarien anhand diesem geschrieben haben, benötigen wir hier nur drei Step Definitions – eine für den "Given"-Teil, eine für den "When"-Teil und eine für den "Then"-Teil der Szenarien.

```
//Unsere Test Implementation
public class CucumberTests {
    Item[] items;

    @Given("item named {string} with quality {int} and sellIn {int}")
    public void Given_Item_With_Name_Quality_And_SellIn(String name,
  Integer quality, Integer sellIn){
        items = new Item[]{
                new Item(name,sellIn,quality)
        };
    }

    @When("{int} days pass")
    public void When_Day_Passes(Integer days){
        GildedRose app = new GildedRose(items);

        for(int i = 0; i < days; i++)
            app.updateQuality();
    }

    @Then("quality should be {int} and sellIn should be {int}")
    public void Then_Quality_And_SellIn_Should_Equal(Integer
  quality, Integer sellIn){
        assertEquals(quality, items[0].quality);
        assertEquals(sellIn, items[0].sellIn);
    }
```

Fazit

Anwendung von Cucumber

Cucumber ist sehr einfach anzuwenden und man kann gut damit arbeiten. Mit der vorhandenen Dokumentation in diversen Internetquellen findet man sich schnell zurecht und kann auch ohne Vorkenntnisse in kurzer Zeit zu Cucumber-TesterIn werden.

Vor- und Nachteile

Ein großer Vorteil von Cucumber ist – wie bereits erwähnt – dass nicht nur die EntwicklerInnen Tests schreiben können, sondern auch andere MitgliederInnen in einem agilen Team die Tests bzw. Szenarien schreiben können. Hier entsteht also eine Arbeits- und Aufwandsteilung.

Ein weiterer Vorteil ist die Wiederverwendbarkeit der Step Definitions. Wie wir hier gezeigt haben, kann man mit guter Vorarbeit beim Satzbau für die Szenarien sich viel Aufwand als EntwicklerIn in der Zukunft sparen. Man kommt mit wenigen Step Definitions aus, die man auch nicht erweitern muss, wenn Szenarien im selben Format hinzukommen. Weniger Code heißt auch weniger Wartungsaufwand.

Als kleiner Nachteil kommt in diesem Zug aber mit, dass man nicht so intuitiv in das Testing gehen kann, wie zum Beispiel mit Unit Tests. Bei Unit Tests reicht es, wenn man die kleine Einheit betrachtet, die man testen möchte, jedoch bei BDD und den Szenarien ist es wichtig, auf das ganze Feature oder auch darüber hinaus zu achten. Als wir mit dem Schreiben der Szenarien in einer Unit-Test-Denkweise begonnen hatten, kamen wir schnell zu der Erkenntnis, dass wir so pro Szenario drei Step Definitions erhalten und keine Vorteile gegenüber Unit Tests gewinnen würden. Wichtig ist also, sich vorab Gedanken zu machen, WIE man die Tests schreibt.

Auch ein Vorteil vom Klartext ist, dass es als Dokumentation dienen kann. Durch das leserliche Gherkin-Format verstehen nicht nur EntwicklerInnen, was genau getestet wurde.

Literaturverzeichnis

[1] “What is Cucumber Testing Tool? Framework Introduction”, unter: https://www.guru99.com/introduction-to-cucumber.html (abgerufen am 06.05.2020)

[2] “Cucumber Overview”, unter: https://www.tutorialspoint.com/cucumber/cucumber_overview.htm (abgerufen am 06.05.2020)

[3] „Introduction“, unter: https://cucumber.io/docs/guides/overview/ (abgerufen am 06.05.2020)

[4] “Cucumber-JVM”, unter: https://cucumber.io/docs/installation/java/ (abgerufen am 06.05.2020)