Gurkensalat, aber richtig

Agile Acceptance Testing mit BDD – Teil 2: Tests implementieren

Carl Anders Düvel, Roland Jülich

Wie stellt man sicher, dass man die richtige Software entwickelt? Eine schwierige Aufgabe, die seit jeher die Softwareentwicklung begleitet. Wie versteht man den Kunden und wie macht dieser sich am besten verständlich? Wie kommuniziert man die Anforderungen effizient mit allen an der Entwicklung und am Testen beteiligten Kollegen? Ach ja, die Software muss auch dokumentiert und automatisch auf Akzeptanzkriterien getestet werden! Viele Anforderungen, für deren Bewältigung wir Ihnen „Behavior Driven Development" mit Cucumber vorschlagen wollen.

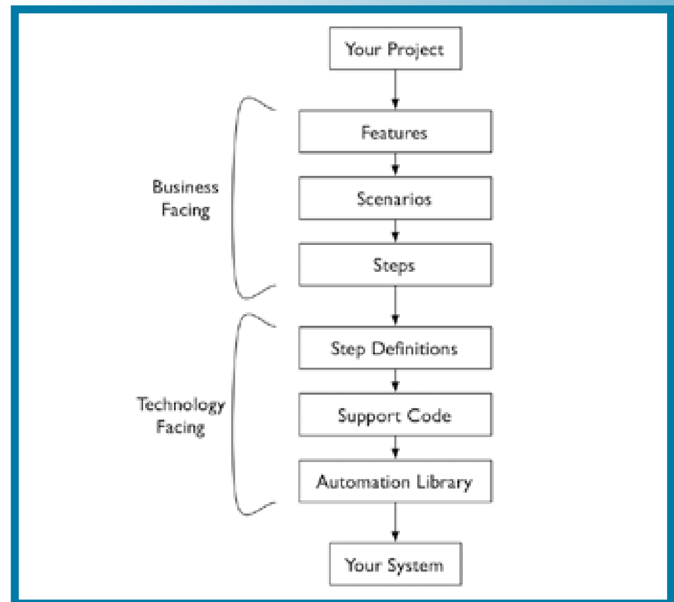

Abb. 1: Architekturübersicht Cucumber (vgl. [Wyn12], S. 8)

Einleitung

Im ersten Teil dieses Artikels [DüJü12] haben wir Ihnen anhand einer kleinen Geschichte *Behavior Driven Development (BDD)* als Methode eines akzeptanztestgetriebenen Vorgehensmodells zur Softwareentwicklung vorgestellt. Dabei haben wir aufgezeigt, wie zunächst anhand von „Specification-by-Example" geeignete Szenarios ermittelt und in natürlicher Sprache erfasst werden.

Im zweiten Teil zeigen wir nun, wie zu dieser Spezifikation Tests implementiert werden, um letztlich zu einer „ausführbaren Spezifikation" zu gelangen, die zu jeder Zeit die Funktionalität des zu entwickelnden Systems transparent macht. Dazu werden wir zeigen, wie Cucumber-JVM in Kombination mit Selenium für das Testen von Webanwendungen eingesetzt werden kann.

Cucumber aus der Vogelperspektive

Cucumber wurde zunächst in Ruby geschrieben – mit der Folge, dass auch die Tests in Ruby implementiert werden mussten. Mit Cucumber-JVM, um das es hier geht, werden viele unterschiedliche Sprachen auf der JVM unterstützt. Neben JRuby und Java auch Clojure, Groovy, Rhino, Ioke und Scala. Das neue Projekt kann dabei auf bereits fertige Komponenten wie Gherkin, den Parser für die Feature-Dateien, zurückgreifen. Wer Lust hat, sich an der aktiven Entwicklung zu beteiligen oder auch nur ein wenig im Quellcode zu stöbern, findet das Projekt auf Github [Gita].

Abbildung 1 bietet einen guten Überblick über die Funktion von Cucumber: Features bestehen aus einem oder mehreren Szenarios, die sich wiederum aus einzelnen Schritten zusammensetzen. Für solche Szenarios sind uns schon einige Beispiele in der einführenden Geschichte begegnet. Sie sind in Gherkin verfasst – einer Sprache mit minimalen Anforderungen: Allein die Signalwörter am Anfang der Sätze sind wichtig und entscheidend.

Jede Zeile in einem Szenario stellt einen Schritt dar. Für jeden von diesen findet Cucumber dann die entsprechende „Step Definition" und arbeitet den darin enthaltenen Code ab. Dieser greift dann möglicherweise auf ein Framework zur Testautomatisierung, wie zum Beispiel Selenium, zurück.

Dieser sehr geradlinige Aufbau macht den Einstieg sehr einfach, auch wenn im Detail noch der eine oder andere Fallstrick lauert. Das liegt vor allem daran, dass die Dokumentation des Projekts zu wünschen übrig lässt*. Unserer Erfahrung nach ist die Implementierung aber sehr stabil und reif für den Einsatz in Projekten.

Gherkin

Wir haben im ersten Teil schon einfache Beispiele in Gherkin gesehen. Die Sprache bietet aber noch einige andere Möglichkeiten, von denen wir Ihnen zwei besonders nützliche vorstellen möchten: Tabellen und Scenario Outlines.

Tabellen sind ein nützliches Werkzeug, wenn in einem Step eine größere Anzahl von Daten benötigt wird. Ein Beispiel, das zu unserer Geschichte aus Teil 1 über die Anzeige von Mitarbeiterbewertungen passt, zeigt Listing 1. Die in der Tabelle erfassten Werte sind Argumente für den Schritt darüber.

```
# language: de
Szenario: Angenommen, ich bin Abteilungsleiter Müller
  Wenn ich den Menüpunkt Mitarbeiterliste anklicke
  Dann sehe ich das folgende Ergebnis
  |Name          | Personalnummer | Lohngruppe |
  |Max Muster    |12345           |B           |
  |Berta Beispiel |58723          |B           |
```

Listing 1: Datentabelle in Gherkin

Scenario Outlines dagegen bieten die Möglichkeit, ein Szenario mit verschiedenen Werten mehrfach zu durchlaufen. So lässt sich damit die finale Version der Feature-Datei wesentlich kürzer und prägnanter formulieren: Das Ergebnis zeigt Listing 2.

* Mit der Ausnahme von Gherkin, das unter [Gitb] über eine Dokumentation verfügt.

```
Funktionalität: Anzeige von Mitarbeiterbewertungen
  Um sich über die Bewertungen von Mitarbeitern zu informieren,
  müssen Berichte für disziplinarische Vorgesetzte anzeigbar sein.

    Szenariogrundriss: Als Vorstandsvorsitzender kann ich
      alle Bewertungen sehen.
      Angenommen ich bin <Rolle>
      Wenn ich die Ansicht für Berichte öffne
      Dann kann ich <Bewertungen> sehen
      Beispiele:
      |Rolle                |Bewertungen                |
      |Vorstandsvorsitzender|Bewertungen aller Mitarbeiter |
      |Teamleiter           |meine eigenen Bewertungen    |
      |Sachbearbeiter       |meine eigenen Bewertungen    |
```

Listing 2: Ein „Szenariogrundriss“ in Gherkin

Wahrscheinlich ist Ihnen schon die deutsche Form der Signalwörter am Anfang der Sätze aufgefallen. Gherkin bzw. Cucumber lässt einem die freie Wahl der Sprache, und man ist gut beraten, diese Freiheit zu nutzen und die Sprache zu wählen, die alle Stakeholder am besten beherrschen. Wer eine Übersicht über die verschiedenen Übersetzungen gewinnen will, muss einfach nur einen Blick in eine YAML-Datei werfen, die im Gherkin-Projekt [Gitd] liegt. Für eine weitere Sprache müssen dort ein paar Zeilen hinzugefügt werden. Dass das notwendig ist, ist aber ziemlich unwahrscheinlich: Mittlerweile findet sich dort sogar eine geeignete Übersetzung für Piraten: Statt Funktionalität heißt es dann: „Ahoy matey!“. Um die Sprache einzustellen, genügt ein entsprechender Kommentar in der ersten Zeile (wie zum Beispiel in Listing 1).

Es gibt noch einige andere interessante Ausdrucksmöglichkeiten in Gherkin. Für eine umfassendere Einführung können wir die genannte Dokumentation sowie das Buch der Entwickler empfehlen: [Wyn12].

Schritt für Schritt

Wie kommt man nun von dem natürlichsprachlichen Text zu ausführbarem Code? Während Cucumber für die Umsetzung der Step Defintions vor allem auf Monkey Patching und Closures zurückgreift, muss Cucumber-JVM andere Mechanismen verwenden: Es findet den auszuführenden Code durch die Markierung von Methoden mit entsprechenden Annotationen (s. Listing 3). Dabei ist es egal, welche genau gewählt wird – die Annotation führt dazu, dass ihr Wert als regulärer Ausdruck verwendet wird, um einen Step innerhalb eines Szenarios umzusetzen. Man kann sich also anstelle der Annotationen `@Wenn`, `@Aber`, `@Dann` ein `@*` vorstellen.

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.is;
import com.google.inject.Inject;
import cucumber.annotation.de.Angenommen;
import cucumber.annotation.de.Dann;

public class SomeSampleStep {
  @Inject
  SomePage somePage;

  @Inject
  LoginPage loginPage;

  @Angenommen(
"^ich logge mich mit den Credentials \"([^\"]*)\" und \"([^\"]*)\" ein$")
  public void pubic void login(String username, String password) {
    loginPage.open().fill(username,password).submit();
  }

  @Dann("^kann ich die Fehlermeldung \"([^\"]*)\" sehen$")
  pubic void checkTheErrorMessage(String expectedMessage) {
    assertThat(page.getErrorMessage, is(expectedMessage));
  }
}
```

Listing 3: Einige Step Definitions

Die Klasse aus Listing 3 wird für jedes Szenario, in dem sie benötigt wird, einmal via Dependency Injection (DI) instantiiert. Unterstützung für Guice, CDI und Picocontainer kann man direkt durch Einbindung der jeweiligen Module aus dem Cucumber-JVM-Projekt beziehen. Wenn man ein anderes DI-Tool bevorzugt, muss man – wenn auch in übersichtlichem Maße – selbst Hand anlegen.

Es ist möglich, Werte aus der Step Definition durch reguläre Ausdrücke „auszuscheiden“ und als Argument für die Methode zu verwenden, wie es in dem Beispiel für die Credentials des Logins in Listing 3 gemacht wird. Da es sich in diesem Fall um eine Webanwendung handelt, werden sogenannte „PageObjects“ (vgl. [Goog]) benutzt. Deren Prinzip ist einfach: Sie stellen die Möglichkeiten der Interaktion des Anwenders für eine Seite zusammen. Dieses Ordnungsmuster hilft dabei, den Testcode für Webapplikationen zu strukturieren.

Auch bei der Konstruktion der PageObjects können wir auf Dependency Injection und darüber hinaus auf die Unterstützung durch Selenium für das PageObject-Muster zurückgreifen (s. Listing 4). Die Klasse `PageFactory` ist Teil von Selenium und übernimmt die Instantiierung der verschiedenen Elemente der Seite. Die einzelnen Methoden geben dabei immer eine Referenz auf die eigene Instanz zurück, um es dem Client zu ermöglichen, die Methodenaufrufe aneinanderzureihen.

```
import javax.annotation.PostConstruct;

import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.CacheLookup;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.How;
import org.openqa.selenium.support.PageFactory;
import com.google.inject.Inject;

public class public class LoginPage {
  @Inject
  SharedDriver driver;

  @FindBy(how = How.ID, using = "login")
  @CacheLookup
  private WebElement loginField;

  @FindBy(how = How.ID, using = "password")
  @CacheLookup
  private WebElement passwordField;

  @FindBy(how = How.ID, using = "submit")
  @CacheLookup
  private WebElement submitButton;

  @PostConstruct
  pubic void init() {
    PageFactory.initElements(driver, this);
  }

  public LoginPage open() {
    driver.get("http://thePathToTheLoginPage");
    return this;
  }

  public LoginPage fill(String username, String password) {
```

```
      loginField.sendKeys(username);
      passwordField.sendKeys(password);
      return this;
    }

    public LoginPage submit() {
      submitButton.click();
      return this;
    }
  }
```

Listing 4: Beispiel eines PageObject

Wenn es nach der Ausführung eines Szenarios Aufräumarbeiten zu verrichten gibt, dann greift Cucumber dem Entwickler mit einer `@After`-Annotation unter die Arme. In unserem Beispiel einer Webanwendung gibt es dafür in der Klasse `SharedDriver` Bedarf, deren Instanzen ein Browserfenster darstellen, das nach dem Ende jedes Szenarios geschlossen werden soll. Ferner erstellen wir am Ende jedes Szenarios einen Screenshot (s. Listing 5), der dann im erzeugten Testbericht erscheint, was im Fehlerfall die Ursachenfindung erleichtert.

```
import java.io.ByteArrayInputStream;

import org.openqa.selenium.OutputType;
import org.openqa.selenium.WebDriverException;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.events.EventFiringWebDriver;

import cucumber.annotation.After;
import cucumber.runtime.ScenarioResult;

/**
 * Dieser Driver wird von Cucumber für ein Szenario benutzt und danach
 * geschlossen. Es erbt von {@link EventFiringWebDriver}, da kein
 * Adapter in Selenium zu Verfügung steht.
 */
public class SharedDriver extends EventFiringWebDriver {
  public SharedDriver() {
    super(new FirefoxDriver());
  }

  @After
  public void close(ScenarioResult result) {
    mkScreenshot(result);
    super.close();
  }

  private void mkScreenshot(ScenarioResult result) {
    try {
      byte[] screenshot = this.getScreenshotAs(OutputType.BYTES);
      result.embed(new ByteArrayInputStream(screenshot), "image/png");
    } catch (WebDriverException somePlatformsDontSupportScreenshots) {
      System.err.println(
        somePlatformsDontSupportScreenshots.getMessage());
    }
  }
}
```

Listing 5: SharedDriver

Einbindung via JUnit

Cucumber-JVM stellt einen Testrunner für JUnit zur Verfügung, was dank der guten Unterstützung von JUnit durch Entwicklungsumgebungen und die gängigen Buildtools in der Java-Welt eine sehr einfache Integration ermöglicht. Listing 6 zeigt einen solchen Test. Die Klasse darf keine sonstigen Methoden deklarieren und verweist auf den Ordner mit den Feature-Dateien. Zusätzlich geben wir mit dem Attribut `glue` an, in welchem Package die nötigen Step Definitions und Hilfs-

```
import org.junit.runner.RunWith;
import cucumber.junit.Cucumber;

@RunWith(Cucumber.class)
@Cucumber.Options(glue = "de.example.cucumber",
  features = "src/test/resources/cucumber/features",
  format = "html:target/cucumber")
public class RunAllCucumberFeatures {
}
```

Listing 6: JUnit-Runner für Cucumber

klassen zu finden sind. Der Testrunner beginnt dann damit, die entsprechenden Feature-Dateien zu parsen und die Schritte sequenziell auszuführen. Dabei wird auch ein html-Report erstellt, der für die Fehlersuche sehr hilfreich ist.

Ein Cucumber-Test hat genau vier mögliche Ergebniszustände:

- *Undefined:* Es wurde keine passende Step Definition gefunden.
- *Pending:* Es trat eine PendingException auf.
- *Failed:* Bei der Ausführung ist eine andere Exception aufgetreten.
- *Passed:* Alle Steps wurden gefunden und ausgeführt.

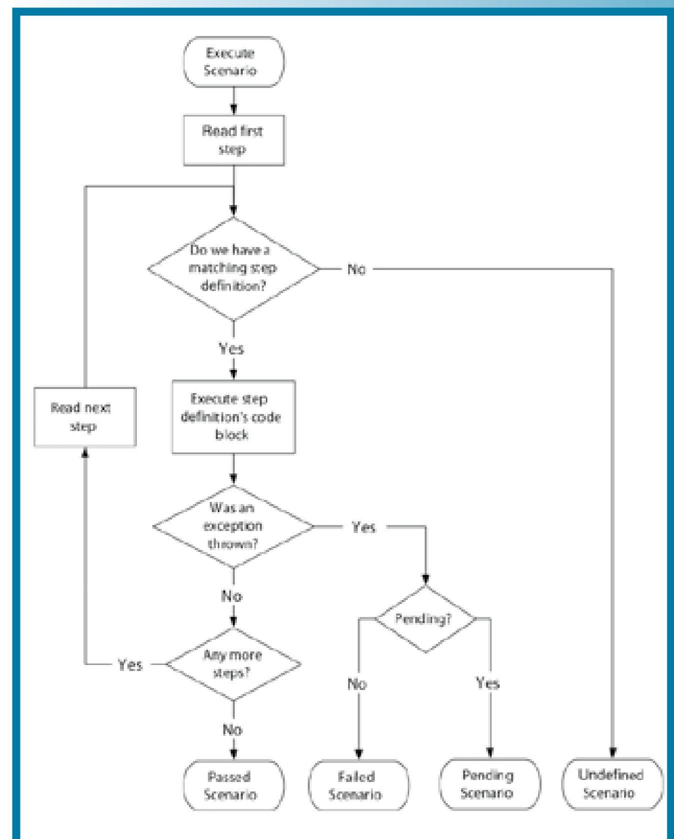

Abb. 2: Ablauf eines Szenarios (vgl. [Wyn12], S. 53)

Abbildung 2 stellt die Ausführung eines Szenarios dar und zeigt, wie diese Endzustände erreicht werden. JUnit kennt natürlich diese Zustände nicht, sie werden daher den bekannten JUnit-Ergebnissen zugeordnet:

- Undefined – Ignored
- Pending – Ignored
- Failed – Failed
- Passed – Passed

Beispiele gefällig?

Wer nun Lust bekommen hat, ein paar Tests mit Cucumber-JVM zu bauen, dem sei neben dem Wiki des Cucumber-Projekts und dem hervorragenden Cucumber-Buch [Wyn12] auch das Studium des einen oder anderen Beispiels ans Herz gelegt. Diese gibt es zum einen im Cucumber-JVM-Projekt im Unterordner „examples". Während dort Picocontainer verwendet wird, finden sich in einer Beispielapplikation, die wir mit Kollegen geschrieben haben, Tests, die auf Guice und Selenium zurückgreifen [Gitc].

Literatur und Links

[DüJü12] C. A. Düvel, R. Jülich,
Agile Acceptance Testing mit BDD – Teil 1: Grundlagen, in: JavaSPEKTRUM, 4/2012
[Gita] Cucumber-JVM, `https://github.com/cucumber/cucumber-jvm`
[Gitb] Gherkin, `https://github.com/Cucumber/Cucumber/wiki/Gherkin`
[Gitc] Ticket2Rock, `https://github.com/holisticon/ticket2rock`
[Gitd] Gherkin,
`https://github.com/cucumber/gherkin/blob/master/lib/gherkin/i18n.yml`
[Goog] PageObjects,
`http://code.google.com/p/selenium/wiki/PageObjects`
[Wyn12] M. Wynne, A. Hellesoy, The Cucumber Book: Behaviour-Driven Development for Testers and Developers, Pragmatic Programmers, 2012

Carl Anders Düvel arbeitet als Berater und Entwickler für die Holisticon AG. Er ist Mitglied der Softwerkskammer Hamburg.
E-Mail: carl.duevel@holisticon.de

Roland Jülich arbeitet als Berater, Entwickler und Coach für die Holisticon AG. Er ist Mitbegründer der Softwerkskammer Hamburg.
E-Mail: roland.juelich@holisticon.de