04-2014 | Winter | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

Mit Lambda-Ausdrücken einfacher programmieren

Go for the Money
Währungen und Geldbeträge in Java

Oracle-Produkte
Die Vorteile von Forms und Java vereint

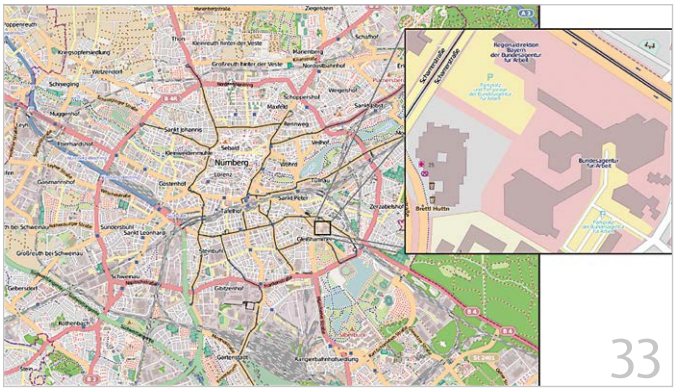
Pimp my Jenkins
Noch mehr praktische Plug-ins

Sonderdruck

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

4 191978 304903 04

iJUG Verbund



OpenStreetMap ist vielen kommerziellen Diensten überlegen, Seite 33

Alles über Währungen und Geldbeträge in Java, Seite 20

5 Das Java-Tagebuch
Andreas Badelt, Leiter der DOAG SIG Java

8 JDK 8 im Fokus der Entwickler
Wolfgang Weigend

15 Einmal Lambda und zurück – die Vereinfachung des TestRule-API
Günter Jantzen

20 Go for the Money – eine Einführung in JSR 354
Anatole Tresch

24 Scripting in Java 8
Lars Gregori

28 JGiven: Pragmatisches Behavioral-Driven-Development für Java
Dr. Jan Schäfer

33 Geodatensuche und Datenanreicherung mit Quelldaten von OpenStreetMap
Dr. Christian Winkler

38 Pimp my Jenkins – mit noch mehr Plug-ins
Sebastian Laag

41 Apache DeviceMap
Werner Kei

46 Schnell entwickeln – die Vorteile von Forms und Java vereint
René Jahn

50 Oracle-ADF-Business-Service-Abstraktion: Data Controls unter der Lupe
Hendrik Gossens

54 Neue Features in JDeveloper und ADF 12*c*
Jürgen Menge

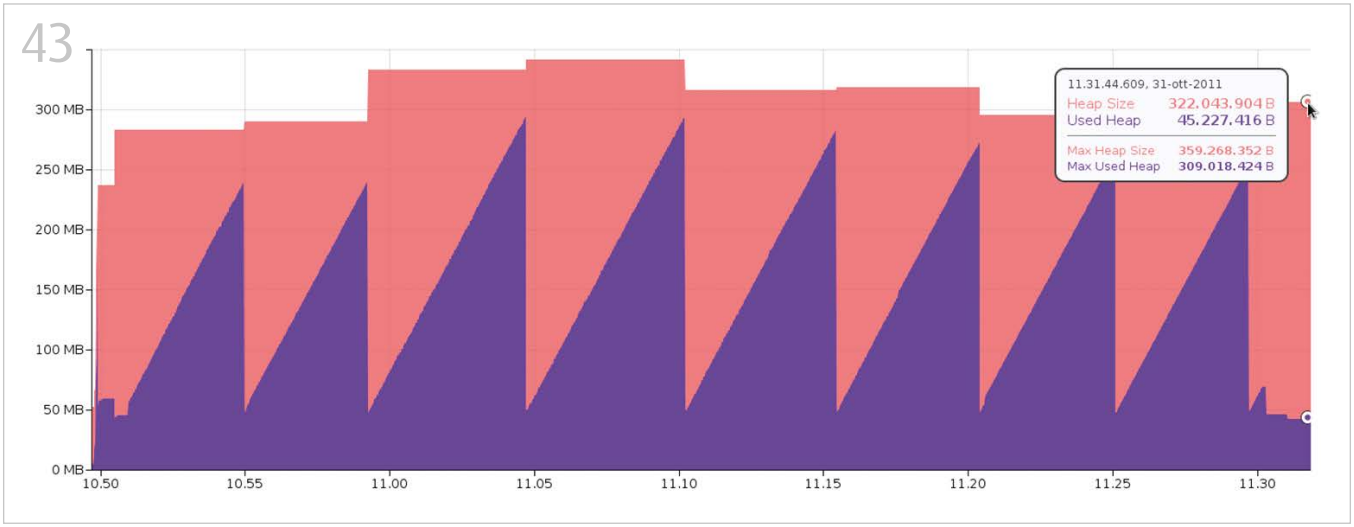
57 Der (fast) perfekte Comparator
Heiner Kücker

60 Clientseitige Anwendungsintegration: Eine Frage der Herkunft
Sascha Zak

64 Unbekannte Kostbarkeiten des SDK Heute: Die Klasse „Objects“
Bernd Müller

66 Inserenten

66 Impressum



WURFL ist verglichen mit selbst dem nicht reduzierten OpenDDR-Vokabular deutlich speicherhungriger, Seite 43

JGiven: Pragmatisches Behavioral-Driven-Development für Java

Dr. Jan Schäfer, TNG Technology Consulting GmbH

JGiven ist ein Open-Source -Tool für Java, das Test-Szenarien in einer Java-DSL anstatt in Text-Dateien schreibt. JGiven erzeugt daraus eine Verhaltensdokumentation, die dann von der Fachabteilung beurteilt und mit den Anforderungen verglichen werden kann. Szenarien in JGiven sind modular aufgebaut, wodurch eine hohe Wiederverwendbarkeit von Test-Code entsteht.

Behavioral-Driven-Development (BDD) ist eine Testmethode, bei der automatisierte Tests in der Sprache des jeweiligen Fachgebiets der Anwendung geschrieben werden. Dadurch können BDD-Tests als Spezifikation und Dokumentation an die Fachabteilung gegeben und dort mit den Anforderungen abgeglichen werden. Klassische BDD-Tools für Java wie JBehave [1] oder Cucumber [2] formulieren Test-Szenarien in einfachen Text-Dateien. Dadurch soll es der Fachabteilung sogar möglich sein, automatisierte Tests selbst zu schreiben. Obwohl dies natürlich ideal wäre, sind es am Ende jedoch oft die Entwickler, die diese Text-Dateien schreiben und warten müssen.

Szenarien in Text-Dateien zu schreiben, hat jedoch diverse Nachteile. Erstens stehen einem nicht die bekannten Hilfsmittel einer modernen Entwicklungsumgebung (IDE) zur Verfügung, etwa automatische Vervollständigung, Typsicherheit und Refactoring. Plug-ins liefern zwar zumindest Autovervollständigung und Syntax-Highlighting nach, kommen aber ansonsten nicht an den Komfort heran, den man von einer Java-IDE gewöhnt ist. Zweitens ist man in der Text-Sprache gefangen und kann nicht auf die Mächtigkeit einer vollen Programmiersprache zurückgreifen. Dies zwingt oft dazu, Szenarien per „Copy & Paste“ zu erstellen, was die Wartbarkeit erheblich erschwert. Drittens müssen die Schritte der jeweiligen in Textform geschriebenen Szenarien durch reguläre Ausdrücke an ausführbaren Java-Code gebunden werden. Dies stellt in der Praxis einen nicht unerheblichen Aufwand dar.

Da nicht selten der Entwicklungs- und Wartungsaufwand von automatisierten Tests den des eigentlichen Produktiv-Codes übersteigt, bedeutet jeder zusätzliche Aufwand, der für Tests erforderlich ist, erhebliche Zusatzkosten für die gesamte Entwicklung. Schlussendlich führt dies oft dazu, dass BDD ganz aufgegeben wird. Aus diesen Gründen hat der Autor JGiven entwickelt. Es legt den Fokus auf den Entwickler und macht es so einfach wie möglich, BDD-Tests zu schreiben und zu warten.

Prinzipien

JGiven verabschiedet sich von der Idee, Szenarien in Text-Dateien zu formulieren. Stattdessen werden diese in einer Java-DSL geschrieben. Sämtliche der oben genannten Nachteile entfallen dadurch, da bei der Szenario-Erstellung auf die volle Mächtigkeit der Java-IDE zurückgegriffen werden kann. Die DSL selbst wird von den jeweiligen Anwendungsentwicklern definiert, JGiven liefert nur die nötigen Hilfsmittel dazu mit.

JGiven vermeidet – soweit wie möglich – unnötige Zusatz-Annotationen. Dies wird dadurch erreicht, dass Methoden-Aufrufe zur Laufzeit von JGiven abgefangen und eingelesen werden. Technisch funktioniert dies ähnlich wie die aus dem Mockito-Framework [3] bekannten „Spys“. Ein weiteres wichtiges Prinzip ist der modulare Aufbau von Szenarien aus wiederverwendbaren Einheiten. Dadurch wird das Erstellen neuer Szenarien deutlich vereinfacht und gleichzeitig die Duplikation von Test-Code vermieden.

JGiven selbst ist ein kleines, leichtgewichtiges Framework. Es überwacht lediglich die Ausführung von Szenarien und erstellt daraus entsprechende Berichte. Die Tests selbst werden entweder durch „JUnit“ oder „TestNG“ ausgeführt. JGiven ist daher sehr leicht in existierende Test-Infrastrukturen zu integrieren und ermöglicht es so, bestehende Tests Stück für Stück in Szenario-Tests umzuwandeln.

In den folgenden Beispielen wird ein imaginärer Web-Shop mit JGiven getestet. Alle Beispiele in diesem Artikel sind auf Deutsch geschrieben, es ist aber selbstverständlich genauso gut möglich, Test-Szenarien auf Englisch oder in jeder anderen Sprache zu schreiben. Bei Verwendung von Nicht-ASCII-Zeichen muss das Datei-Encoding allerdings „UTF-8“ sein. Das gesamte Beispiel-Projekt ist auf „GitHub“ verfügbar [4]. In der ersten Story „ABC-1“ soll sich ein Kunde registrieren können. Dazu formulieren wir ein erstes Szenario (siehe Listing 1).

Das Beispiel zeigt eine „JUnit 4“-Testmethode („@Test“), in der das JGiven-Szenario definiert ist. Der Methodenname selbst ist die Beschreibung des Szenarios in „Snake_Case“. Innerhalb der Testmethode werden dann die einzelnen Schritte des Szenarios in der „Gegeben, Wenn, Dann“-Notation als Methodenaufrufe geschrieben, jeweils wieder in „Snake_Case“. Abgesehen von ein paar von Java benötigten Klammern, Punkten und Unterstrichen enthält der Code keinerlei unnötige Boilerplates und ist für sich allein schon fast so gut lesbar wie ein in reinem Text geschriebenes Szenario. Wenn der Test nun von JUnit ausgeführt wird, generiert JGiven auf der Konsole eine Text-Ausgabe (siehe Listing 2).

```
@Test @Story("ABC-1")
public void Kunden_können_sich_registrieren() throws Exception {
   gegeben().die_Registrierungsseite_ist_geöffnet()
      .und().eine_valide_Email_ist_angegeben()
      .und().ein_valides_Passwort_ist_angegeben();
   wenn().der_Kunde_auf_den_Registrieren_Knopf_drückt();
   dann().ist_der_Kunde_registriert()
     .und().der_Kunde_erhält_eine_Bestätigungsemail();
}
```

Listing 1: Definition des Registrierungs-Szenarios in JGiven

```
Scenario: Kunden können sich registrieren

  Gegeben die Registrierungsseite ist geöffnet
      Und eine valide Email ist angegeben
      Und ein valides Passwort ist angegeben
     Wenn der Kunde auf den Registrieren Knopf drückt
     Dann ist der Kunde registriert
      Und der Kunde erhält eine Bestätigungsemail
```

Listing 2: Text-Ausgabe während der Test-Ausführung

```
public class RegistrierungsTest extends
SzenarioTest<GegebenRegistrierungsSeite<?>,
WennRegistrierungsSeite<?>, DannRegistrierungsSeite<?>> {
      // Szenarien ...
}
```

Listing 3: Definition der „RegistrierungsTest"-Klasse

Die Text-Ausgabe von JGiven dient im Wesentlichen dem Entwickler während der Entwicklung des Szenarios. Zusätzlich generiert JGiven auch JSON-Dateien während der Ausführung, die dann in einem späteren Schritt in HTML-Berichte konvertiert werden. Abbildung 1 zeigt die HTML-Darstellung des Registrierungsszenarios.

Snake_Case

Für Java-Entwickler etwas ungewöhnlich und auch gegen die Java-Code-Konventionen ist die Verwendung von „Snake_Case" in Methodennamen. „Snake_Case" ist essenziell für JGiven, da dadurch die korrekte Groß- und Kleinschreibung verwendet werden kann, was entscheidend zur Lesbarkeit der generierten Berichte beiträgt. In der Praxis hat sich die parallele Verwendung von „CamelCase" für normalen Java-Code und „Snake_Case" für Test-Szenarien als problemlos herausgestellt.

Falls die Verwendung von „Snake_Case" aufgrund von Projekt-Vorgaben unmöglich sein sollte, lässt sich die korrekte Schreibweise mit der „@Description"-Annotation angeben. Dies ist auch nützlich, wenn Sonderzeichen im Bericht erscheinen sollen, die nicht in Java-Methoden-Namen erlaubt sind.

Stage-Klassen

Schrittmethoden sind in JGiven in sogenannten „Stage-Klassen" definiert. Entsprechend der „Gegeben, Wenn, Dann"-Notation besteht ein Szenario in der Regel aus drei Stage-Klassen: eine Klasse für die „Gegeben"-, eine für die „Wenn"- und eine für die „Dann"-Schritte. Diese Modularisierung der Szenarien hat insbesondere für die Wiederverwendung von Test-Code große Vorteile. Szenarien können so nach dem Baukastenprinzip aus den Stage-Klassen zusammengesetzt werden.

Bewährt hat sich auch die Verwendung von Vererbung innerhalb der Stage-Klassen, bei denen speziellere, seltener gebrauchte Stages von allgemeineren, öfter gebrauchten Stages erben. Ein Szenario ist

Abbildung 1: HTML-Darstellung des Beispiel-Szenarios

nicht auf drei Stage-Klassen beschränkt, sondern kann aus beliebig vielen Stages bestehen. Für unser Beispiel definieren wir drei Stage-Klassen: „GegebenRegistrierungsSeite", „WennRegistrierungsSeite" und „DannRegistrierungsSeite". Damit die Stage-Klassen in unserem Test verwendet werden können, muss die Test-Klasse von der von JGiven bereitgestellten „SzenarioTest"-Klasse erben und deren Typ-Parameter entsprechend setzen (siehe Listing 3). Für englische Szenarien erbt man von „ScenarioTest".

Die „Gegeben"-Stage

Kommen wir nun zu den Stage-Klassen. Listing 4 zeigt die „GegebenRegistrierungsSeite"-Klasse, die wir für obiges Szenario benötigen. Der Einfachheit halber simulieren wir den Web-Shop mithilfe von statischen HTML-Seiten. Beim Testen kommt der Selenium WebDriver [5] zum Einsatz.

Das Beispiel zeigt mehrere wichtige Konzepte. Das erste ist, dass Stage-Klassen in der Regel von einer von JGiven vordefinierten Klasse erben, in unserem Fall der „Stufe"-Klasse (die deutsche Variante der „Stage"-Klasse). Sie stellt eine Reihe von vordefinierten Hilfsmethoden zur Verfügung, wie „und()", „aber()" etc., die in der Regel für Szenarien gebraucht werden. Stage-Klassen folgen dem „Fluent Interface"-Pattern, jede Methode gibt also als Rückgabewert das aufgerufene Objekt wieder zurück.

Damit das „Fluent Interface"-Pattern auch unter Vererbung korrekt funktioniert, wird außerdem ein Typ-Parameter an die Super-Klasse durchgereicht, der dem eigenen Typ der Klasse entspricht. Die „self()"-Methode liefert nun diesen Typ wieder zurück. Die Annotationen „@ProvidedScenarioState" und „@AfterScenario" sind nachfolgend erläutert.

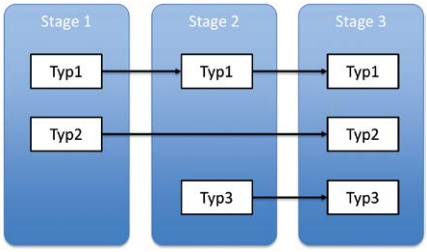


Abbildung 2: Zustand von Stage-Feldern wird von Stage zu Stage übertragen

Zustands-Transfer

Ein typisches Szenario funktioniert in der Regel so: In der „Gegeben"-Stage wird ein bestimmter Zustand hergestellt. Darauf wird dann in der „Wenn"-Stage zugegriffen, eine Aktion ausgeführt und ein Ergebniszustand produziert. Schließlich wird in der „Dann"-Stage der Ergebniszustand ausgewertet.

Um einen Zustand zwischen Stages zu transportieren, besitzt JGiven einen Mechanismus, der automatisch Felder von Stages ausliest und in nachfolgende Stages schreibt. Felder, die mit „@ScenarioState", „@ProvidedScenarioState" oder „@ExpectedScenarioState" annotiert sind, werden dabei berücksichtigt. Man kann den Mechanismus mit Dependency Injection vergleichen, mit dem Unterschied, dass der Zustand nicht nur injiziert, sondern auch extrahiert wird (siehe Abbildung 2).

Im Beispiel wird die WebDriver-Instanz für die nachfolgenden Stages verfügbar gemacht und deswegen das entsprechende Feld mit „@ProvidedScenarioState" annotiert. Nachfolgende Stages können sich nun die WebDriver-Instanz injizieren lassen, indem sie ein entsprechendes Feld deklarieren und es mit „@ExpectedScenarioState" annotieren. Der Zustandstransfer-Mechanismus von JGiven ist entscheidend für die Modularität. Dadurch, dass eine Stage nur deklariert, welchen Zustand sie benötigt, und nicht, von welcher Stage der Zustand kommen muss, lassen sich beliebige Stages miteinander kombinieren, solange die jeweils benötigten Zustände von den vorherigen Stages bereitgestellt sind.

Life-Cycle-Annotationen

Ähnlich wie bei JUnit bietet JGiven die Möglichkeit, bestimmte Aktionen zu bestimmten Zeitpunkten des Szenarios auszuführen. Dazu gibt es eine Reihe von Methoden-Annotationen: „@BeforeStage" und „@AfterStage", um Aktionen vor beziehungsweise direkt nach einer Stage auszuführen, „@BeforeScenario" und „@After Scenario", um Aktionen vor beziehungsweise nach dem ganzen Szenario auszuführen. In unserem Beispiel muss nach dem Szenario der Browser wieder geschlossen werden. Entsprechend wird die „closeBrowser"-Methode definiert und mit „@AfterScenario" annotiert.

```
public class GegebenRegistrierungsSeite<SELF extends GegebenRegis-
trierungsSeite<?>> extends Stufe<SELF> {
   @ProvidedScenarioState
   WebDriver webDriver = new HtmlUnitDriver(true);
   @ProvidedScenarioState
   String email = "testuser@test.com";
   @ProvidedScenarioState
   String passwort = "Passwort1234!$";
   public SELF die_Registrierungsseite_ist_geöffnet() throws Ex-
ception {
      File file = new File( "src/main/resources/webshop/regis-
trierung.html" );

      webDriver.get( file.toURI().toURL().toString() );
      return self();
   }

   public SELF eine_valide_Email_ist_angegeben() {
      webDriver.findElement( By.id( "emailInput" ) )
         .sendKeys( email );
      return self();
   }
   public SELF ein_valides_Passwort_ist_angegeben() {
      webDriver.findElement( By.id( "passwortInput" ) )
         .sendKeys( passwort );
      return self();
   }
   @AfterScenario
   public void closeBrowser() {
      webDriver.close();
   }
}
```

Listing 4: Definition der „GegebenRegistrierungsSeite"-Klasse

Die „Wenn"-Stage

Die Klasse „WennRegistrierungsSeite" ist sehr übersichtlich (siehe Listing 5). Wie oben schon kurz erwähnt, erwartet die Stage eine „WebDriver"-Instanz und besitzt ein entsprechend mit „@ExpectedScenarioState" annotiertes Feld. Typischerweise würde die „Wenn"-Stage selbst auch den Zustand für folgende Stages bereitstellen, in unserem Beispiel ist allerdings der Ergebnis-Zustand wiederum im WebDriver gekapselt. Ansonsten implementiert die Klasse die Registrierungsaktion, indem sie einen Knopfdruck durchführt.

Die „Dann"-Stage

Da wir in unserem Beispiel nur gegen statische HTML-Seiten testen, ist die „Dann"-Stage nicht vollständig implementiert. Ob der Kunde registriert ist oder eine E-Mail versendet wurde, würde man in der Praxis normalerweise gegen die Datenbank testen. Listing 6 zeigt die implementierten Teile. In „Dann"-Klassen sind typischerweise die Assertions eines Tests implementiert, wie auch in diesem Beispiel.

Parametrisierte Schrittmethoden

Nachdem die erfolgreiche Registrierung getestet wurde, kommen nun die Fehlerfälle an die Reihe. Unter anderem soll die E-Mail-Adresse bei der Registrierung validiert werden. Da es mühselig wäre, für jeden Wert eine eigene Schrittmethode zu schreiben, kön-

```
public class WennRegistrierungsSeite<SELF extends WennRegis-
trierungsSeite<?>> extends Schritte<SELF> {
   @ExpectedScenarioState
   WebDriver webDriver;

   public SELF der_Kunde_auf_den_Registrieren_Knopf_drückt() {
      webDriver.findElement( By.id( "registrierenButton" ) )
         .click();
      return self();
   }
}
```

Listing 5: Die „WennRegistrierungsSeite"-Klasse

```
public class DannRegistrierungsSeite<SELF extends DannRegis-
trierungsSeite<?>> extends Schritte<SELF> {
   @ExpectedScenarioState
   WebDriver webDriver;

   public SELF wird_die_Willkommensseite_geöffnet() {
      assertThat( webDriver.getTitle() )
         .isEqualTo( "Willkommen beim WebShop!" );
      return self();
   }
   // ...
}
```

Listing 6: Ausschnitt aus der „DannRegistrierungsSeite"-Klasse

```
public SELF als_Email_ist_$_angegeben( String email ) {
   this.email = email;
   webDriver.findElement( By.id( "emailInput" ) )
      .sendKeys( email );
   return self();
}
```

Listing 7: Parametrisierte Schrittmethode

```
public SELF wird_die_Registrierung_mit_der_Fehlermeldung_$_abge-
lehnt( String fehlerMeldung ) {
     assertThat(
        webDriver.findElement( By.id( "fehlerMeldung" ) )
           .getText() ).isEqualTo( fehlerMeldung );
   return self();
}
```

Listing 8: Parametrisierte Schrittmethode in der „Dann"-Stage

nen Schrittmethoden Parameter haben. Die „GegebenRegistrierungsSeite"-Klasse erhält nun eine neue Methode (siehe Listing 7).

Das Dollarzeichen im Methodennamen ist ein Platzhalter, um den Parameter an die richtige Stelle des Satzes setzen zu können. Im generierten Bericht wird es durch den Wert des Parameters ersetzt. Falls kein „$" vorkommt, werden Argumente einfach am Ende angefügt. Zusätzlich ist noch eine neue Schrittmethode in der „DannRegistrierungsSeite"-Klasse notwendig, die ebenfalls parametrisiert ist (siehe Listing 8).

Parametrisierte Szenarien

Parametrisierte Schrittmethoden sind für sich schon nützlich, um in verschiedenen Szenarien verschiedene Werte übergeben zu können. Oft möchte man allerdings das Szenario selbst parametrisieren. Im Beispiel werden verschiedene ungültige E-Mail-Adressen mit dem JUnit-DataProvider [6] getestet. Dazu wird eine neue Testmethode erstellt, die einen Data-Provider verwendet und eine E-Mail-Adresse als Parameter erwartet (siehe Listing 9). JGiven erzeugt daraus einen Bericht, der die verschiedenen Fälle in einer Tabelle übersichtlich darstellt (siehe Abbildung 3).

Tags

Es ist sicher aufgefallen, dass die bisherigen Tests alle mit „@Story(„ABC-1")" annotiert sind. Dies ist keine spezielle JGiven-Annotation, sondern eine für das Beispiel definierte Annotation. „@Story" ist selbst wiederum mit „@IsTag" annotiert, wodurch JGiven es als Tag erkennt. Tags erscheinen im generierten HTML-Bericht und für jedes Tag wird eine extra Seite generiert. Dadurch erhält man nach Themen gruppierte Szenarien, unabhängig von der Test-Klasse, in der sie definiert sind.

Anders als in anderen BDD-Tools hat JGiven keinen Mechanismus, um Features zu beschreiben. In der Regel werden Features beziehungsweise Stories (oder auch Bugs) schon in anderen Tools verwaltet. Anstatt die Beschreibung der Features in JGiven zu wiederholen, können Tags verwendet werden, um eine Zuordnung zwischen Szenarien und Features zu ermöglichen. Idealerweise enthalten die Story-Tickets auch die entsprechenden Akzeptanzkriterien, die dann direkt mit den geschriebenen Szenarien verglichen werden können.

Einbindung ins Projekt

JGiven wird entweder zusammen mit JUnit oder TestNG verwendet. Für JUnit benötigt man folgende Maven-Abhängigkeit (siehe Listing 10). Für „TestNG" hat die „artifactId" den Namen „jgiven-testng". Um nach der Test-Ausführung den HTML-Bericht zu generieren, bindet man noch das JGiven-Maven-Plug-in ein (siehe Listing 11). Ein „mvn verify" führt nun die Tests aus und generiert anschließend die JGiven-HTML-Berichte.

Fazit

Hinter Behavioral-Driven-Development steht die Idee einer gemeinsam verwendeten Sprache zwischen Product-Ownern,

```
@Test @Story("ABC-1")
@DataProvider( { "abc.com", "ungültig", "1234" } )
public void Kunden_können_sich_nur_mit_valider_
Emailadresse_registrieren( String email ) {
  gegeben().die_Registrierungsseite_ist_geöffnet()
    .und().als_Email_ist_$_angegeben( email )
    .und().ein_valides_Passwort_ist_angegeben();
  wenn().der_Kunde_auf_den_Registrieren_Knopf_
drückt();

dann().wird_die_Registrierung_mit_der_Fehlermel-
dung_$_abgelehnt( "Ungültige Emailadresse" )
   .und().der_Kunde_ist_nicht_registriert();
}
```

Listing 9: Parametrisiertes Szenario

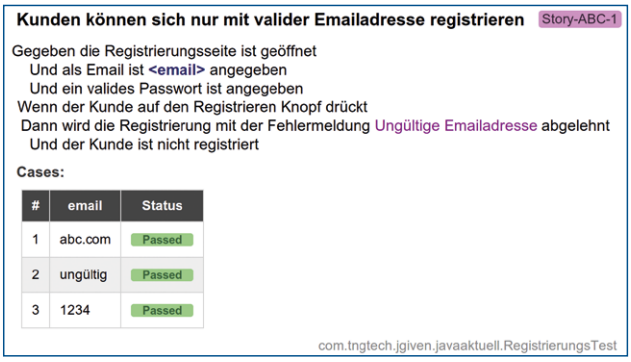


Abbildung 3: HTML-Darstellung von parametrisierten Szenarien

```
<dependency>
  <groupId>com.tngtech.jgiven</groupId>
  <artifactId>jgiven-junit</artifactId>
  <version>0.3.0</version>
  <scope>test</scope>
</dependency>
```

Listing 10

```
<build>
  <plugins>
    <plugin>
      <groupId>com.tngtech.jgiven</groupId>
      <artifactId>jgiven-maven-plugin</artifactId>
      <version>0.3.0</version>
      <executions>
        <execution>
          <goals>
            <goal>report</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Listing 11: Maven-Plug-in zur Erzeugung des HTML-Berichts

Business-Analysten, Entwicklern und Testern. Dadurch werden Kommunikationsfehler vermieden und sichergestellt, dass Anforderungen so wie gewünscht umgesetzt werden. Bestehende BDD-Tools für Java zwingen Entwickler, mit einfachen Text-Dateien zu arbeiten, was den Aufwand der automatisierten Testerstellung deutlich erhöht. JGiven geht einen anderen Weg, indem es, der TDD-Philosophie folgend, den Entwickler in den Mittelpunkt stellt. Dies steht nicht im Gegensatz zur BDD-Philosophie. Business-Analysten schreiben die Anforderungen weiterhin in der „Gegeben, Wenn, Dann"-Notation. Anstatt deren Eingabe allerdings direkt auszuführen, werden die generierten Szenario-Berichte vom Business-Analysten beziehungsweise Product-Owner abgenommen und gegen die Akzeptanz-Kriterien abgeglichen, idealerweise zusammen mit den Entwicklern. Die von JGiven generierten Berichte dienen zusätzlich als automatisch validierte Verhaltens-Dokumentation der Anwendung. Aktuelle Informationen, weitere Beispiele und natürlich der Source-Code von JGiven sind über die offizielle Webseite [7] verfügbar.

Quellen

[1] jbehave.org
[2] cukes.info
[3] code.google.com/p/mockito
[4] github.com/janschaefer/jgiven-java-aktuell
[5] docs.seleniumhq.org/projects/webdriver
[6] github.com/TNG/junit-dataprovider
[7] jgiven.org

Dr. Jan Schäfer
jan.schäfer@tngtech.com

Dr. Jan Schäfer ist Senior Consultant und seit drei Jahren bei der TNG Technology Consulting GmbH in München tätig. Er ist leidenschaftlicher Software-Entwickler und programmiert seit mehr als fünfzehn Jahren in Java. Während seiner Promotion befasste er sich mit der formalen Definition von Objekt-orientierten Programmiersprachen und Aktor-basierten Nebenläufigkeitsmodellen.

http://ja.ijug.eu/14/4/7