Dr. Ralph Guderlei

(ralph.guderlei@exxcellent.de)
ist Project Manager und Software Architect bei der eXXcellent solutions GmbH in Ulm. Dort ist er in unterschiedlichen Kundenprojekten mit der Planung und Realisierung von Enterpriseanwendungen beschäftigt. Neben Technologiethemen interessiert ihn besonders die effiziente Umsetzung von QS-Maßnahmen im Projektalltag.

„Gegurke“ für Fortgeschrittene

Testen während der Entwicklung durch Entwickler gehört spätestens seit dem Aufkommen agiler Softwareentwicklungsprozesse wie beispielsweise Extreme Programming (vgl. [Bec04]) zu den festen Bestandteilen zeitgemäßer Softwareentwicklung. Dabei werden bevorzugt automatisierte Tests verwendet. Je mehr sich der Testgegenstand von Implementierungsdetails (also klassischen Unit-Tests) hin zu Systemtests und dem Prüfen von komplexen Kundenanforderungen entwickelt, desto umständlicher wird die Umsetzung der Testautomatisierung. Dadurch ist es oft schwierig, den eigentlichen Testfall im Automatisierungscode zu erkennen. Personen, die mit der Automatisierungstechnologie nicht vertraut sind, sind dadurch typischerweise nicht in der Lage, den automatisierten Test nachzuvollziehen.

Die Beschreibung der Testfälle erfolgt bei Systemtests entweder rein textuell oder in automatisch verwertbarer Form. Als Werkzeuge für textuelle Beschreibungen dienen hier oft noch Office-Anwendungen oder spezielle Testmanagementwerkzeuge, die in gewissem Rahmen auch eine Testautomatisierung zulassen. Da die Automatisierung der Tests aber typischerweise von der Beschreibung getrennt ist, entsteht so die Situation, dass kaum nachvollzogen werden kann, ob Testfallbeschreibungen mit den automatisierten Tests übereinstimmen. Änderungen an Testfallbeschreibungen führen damit oft zu hohen Aufwänden bei der Anpassung automatisierter Tests, da einerseits unklar ist, welche automatisierten Testfälle überhaupt von der Änderung betroffen sind und andererseits die zu ändernden Stellen im komplexen Automatisierungscode nur schwer zu identifizieren sind.

Testfälle für Systemtests werden bei kleineren Projekten, wenn überhaupt, oft erst gegen Ende des Entwicklungszyklus entworfen. Bei größeren Projekten besteht dagegen oft das Problem, dass die entworfenen Systemtests oft nur einfache Fälle abbilden. Details von Prozessen und Geschäftsregeln werden dabei vernachlässigt. In den vorhandenen Anforderungen fehlen diese Details ebenfalls häufig. Damit sind die Anforderungen nicht präzise genug, um aus ihnen Tests ableiten zu können. Die Folge ist ein gewisser Interpretationsspielraum. Dieser Interpretationsspielraum sorgt sowohl bei der Implementierung als auch bei der Testfallentwicklung für unnötige Missverständnisse zwischen Fachseite und Entwicklung.

Der Artikel zeigt im Folgenden auf, wie die eingangs beschriebenen Probleme reduziert werden können und beschreibt die dazu notwendigen Vorgehensweisen und mögliche Technologien. Ziel ist es, automatisierte Systemtests zu erhalten, die von allen Beteiligten verstanden und gepflegt werden können. Dazu sind eine Reihe von Fragen zu klären:

- Wie können Testfälle verständlich beschrieben werden?
- Lassen sich die Beschreibungen so erstellen, dass diese 1:1 für eine Testausführung verwendet werden können?
- Wie kann die Lesbarkeit und Verständlichkeit der automatisierten Tests verbessert werden?
- Wie kann man erreichen, dass automatisch ausführbare Systemtests nicht erst am Ende einer langen Entwicklung erstellt werden?

Die Antworten auf diese Fragen sind sowohl in technischen als auch organisatorischen Maßnahmen zu finden. Die organisatorischen Maßnahmen werden an einem an Scrum (vgl. [Sch01]) angelehnten Prozess erläutert. Eine mögliche technische Umsetzung stellen wir in diesem Artikel anhand von *Cucumber* (vgl. [Cuc]) vor. Cucumber ist in der Ruby-on-Rails-Community ein etabliertes Werkzeug zum Erstellen automatisierter Testfallbeschreibungen. Von Cucumber existiert zudem eine Variante für die Java Virtual Machine (vgl. [Git]) und mit SpecFlow (vgl. [Spe]) eine Variante für die .Net-Plattform. Das Werkzeug wird von einer aktiven Community entwickelt und gepflegt. Der Artikel beschreibt im Folgenden eine Vorgehensweise, die sich an *Behavior Driven Development* (vgl. [Nor06]) bzw. *Acceptance Test Driven Development* (vgl. [Fre10]) orientiert.

Die Vorgehensweise ist in **Abbildung 1** schematisch dargestellt. Zunächst werden Testfallbeschreibungen erstellt. Die Testfallbeschreibungen dienen als Ausgangsbasis für die Ausführungslogik und werden durch diese interpretiert. Zur automatischen Durchführung des Testfalls interagiert die Ausführungslogik mit der zu testenden Anwendung und vergleicht deren Verhalten mit dem im Testfall spezifizierten erwarteten Verhalten. Die Ergebnisse des Tests werden zusammen mit der ursprünglichen

Abb. 1: Schematische Darstellung der Vorgehensweise.

Testfallbeschreibung für das Reporting verwendet. Beim Reporting können sowohl direkt lesbare HTML-Reports erstellt als auch Drittsysteme angebunden werden.

Um eine verständliche und gut lesbare Testfallbeschreibung zu erhalten, muss es möglich sein, Testfälle quasi allgemeinsprachlich zu formulieren. Die allgemeinsprachliche Formulierung erlaubt eine bessere Kommunikation von Fachseite und Entwicklung über Testfälle, erleichtert damit die Erstellung und Priorisierung der Testfälle und auch die Klärung von offenen Fragen. Das folgende Beispiel stammt aus einem realen Projekt:

```
Feature: create projects
In order to enable other users to collaborate, as an administrator,
I want to create project.

Scenario: create an empty project
        Given the project 'emptyproject' does not exist
        When I create a new project called 'emptyproject'
        Then the project 'emptyproject' should be listed in the project list
```

Die Systemtests beschreiben das gewünschte Verhalten des Systems in der Fachsprache. Um die Beschreibungen später besser zur Testautomatisierung heranziehen zu können, muss die Sprache zur Beschreibung formalisiert werden. Cucumber bietet hierfür eine einfache *Domain Specific Language (DSL)* (vgl. [Fow10]) an. Die Beschreibung der Testfälle erfolgt in einem festgelegten Schema. Ein sogenanntes *Feature* besteht aus einem oder mehreren *Szenarien*. Die Szenarien stellen die eigentlichen Testfälle dar. Features und Szenarien können zusätzlich noch mit einem Freitext beschrieben werden, der nicht Teil der Testausführung ist.

Jedes Szenario besteht aus Vorbedingungen, Aktionen und Nachbedingungen. Die einzige Einschränkung der DSL im Vergleich zu frei formulierten Texten ist nun, dass alle Zeilen mit einem Schlüsselwort beginnen müssen. Jeder dieser Sätze wird als *Step* bezeichnet. Die Schlüsselworte zeigen an, ob es sich um eine Vorbedingung (*Given*), eine Aktion (*When*) oder eine prüfbare Nachbedingung (*Then*) handelt. Die Schlüsselworte sind selbstverständlich internationalisierbar, sodass auch bei Testfallbeschreibungen in einer anderen Sprache als Englisch keine unschöne Vermischung unterschiedlicher Sprachen entsteht. Ein Szenario kann zudem jeweils mehrere Vorbedingungen, Aktionen und Nachbedingungen enthalten. Dadurch werden auch komplexe Szenarien wie z. B. komplette Prozessabläufe ermöglicht. Steps können parametrisiert und in unterschiedlichen Szenarien wiederverwendet werden.

Bei der Formulierung der Testfälle hat es sich als vorteilhaft herausgestellt, Beschreibungen technischer Abläufe (z. B. Informationen, welche Buttons gedrückt werden müssen) zu vermeiden. Technische Abläufe stellen immer die Implementierung in den Vordergrund, statt sich auf die gewünschte fachliche Funktionalität zu konzentrieren. Im Idealfall ist die Art der technischen Realisierung des Systems nicht aus den Tests ableitbar.

Die nächste Frage ist, wie Testfallbeschreibung und Automatisierung direkt miteinander verbunden werden können. Basierend auf der Beschreibung muss nachvollziehbar sein, was bei der Automatisierung der Tests abläuft, um die Automatisierung verifizieren zu können. Diese Verbindung von Testfallbeschreibung und Testautomatisierung ist ein wesentlicher Bestandteil von Cucumber. Bei der Ausführung wird jede Zeile der Testfallbeschreibung auf eine Ausführungsmethode abgebildet. Diese enthält dann die eigentliche Automatisierungslogik. Cucumber unterstützt hierbei Ruby, eine Vielzahl an JVM-Sprachen (u. a. Java, Scala, Clojure) und .Net-Sprachen, um die Automatisierungslogik zu implementieren. Die Automatisierungslogik wiederum greift programmatisch auf die zu testende Anwendung zu. Die Voraussetzung hierfür ist, dass geeignete Werkzeuge für diesen Zugriff existieren. Für Webanwendungen hat sich beispielsweise Selenium bewährt. Aber auch für andere Oberflächentechnologien (z. B. Swing oder WPF) gibt es entsprechende Werkzeuge. Im Zweifelsfall kann auch mit Capture&Replay-Werkzeugen gearbeitet werden. Diese Möglichkeit sollte aber aufgrund des hohen Aufwands bei Erstellung und Pflege der Tests nur in absoluten Ausnahmefällen genutzt werden.

Bei allen JVM-Sprachen wird JUnit für die Testausführung verwendet. Dadurch integriert sich Cucumber in alle üblichen IDEs, Build-Tools und Continuous-Integration-Systeme. Darüber hinaus gibt es spezielle Cucumber-Plugins für IDEs, die nicht nur die Testausführung, sondern auch das Schreiben von Testfallbeschreibungen durch Syntax-Highlighting und Auto-Vervollständigung unterstützen.

Für den ersten Beschreibungssatz im Beispiel könnte die Umsetzung der Implementierung der Automatisierungslogik mit Hilfe von Cucumber wie folgt aussehen:

```
@Given ("^the project '([\w-.]+)' does
not exist$")
public void
checkIfProjectDoesNotExist(String
projectName){
assertFalse(projectList.containsProjec
tWithName(projectName));}
```

Der Zusammenhang zwischen Beschreibung und Automatisierungslogik wird über reguläre Ausdrücke hergestellt. Diese ermöglichen es, variable Inhalte in den

Testfallbeschreibungen unterzubringen. Die variablen Inhalte werden dann als Parameter der Automatisierungslogik übergeben. Die Automatisierungslogik enthält sowohl die Ansteuerung der zu testenden Anwendung als auch die Prüfung des erwarteten Verhaltens der Anwendung. Die Ansteuerung der zu testenden Anwendung verbirgt sich im Beispiel hinter `„projectList.containsProjectWithName()“`, die Prüfung des erwarteten Verhaltens erfolgt über JUnit-Assertions (`„assertFalse()“`).

Es empfiehlt sich, die Zugriffe auf die zu testende Anwendung weiter zu abstrahieren. Bei Webanwendungen kann man dazu beispielsweise auf das Page Object Pattern (vgl. [Pag]) zurückgreifen. Ein Page Object kapselt die Funktionalität für den Zugriff auf einzelne Seiten bzw. Komponenten einer Webanwendung und stellt der Automatisierungslogik Funktionen für die bequeme Ansteuerung der zu testenden Anwendung bereit. Das Page Object ist im Beispiel durch die Variable `„project List“` angedeutet. Die weitere Abstraktion erleichtert die Wiederverwendung der Zugriffslogik in unterschiedlichen Ausführungsmethoden und die Wartbarkeit der Zugriffslogik.

Cucumber bietet keine Unterstützung bei den typischen Problemen bei Systemtests, es bietet ausschließlich einen Rahmen für die Organisation und Ausführung der Testfälle. Die technischen Herausforderungen bei Systemtests müssen weiterhin gelöst werden. Zuerst muss die notwendige Infrastruktur automatisiert aufgesetzt werden bzw. ein automatisches Deployment in eine Testinfrastruktur erfolgen. Hier hat sich eine Mischung aus Deploymentfunktionalitäten bestehender Build-Systeme und virtualisierten Infrastrukturen als praktikabel herausgestellt. Für die Durchführung der Tests ist zudem eine Initialisierung des Datenbestands (z. B. Datenbank) notwendig. Hier hängt die beste Lösung oft von den Gegebenheiten des jeweiligen Systems ab. Im einfachsten Fall können Systemtests so gestaltet werden, dass diese einen initialen Datenbestand in einer leeren Datenbank erzeugen. In komplexeren Fällen können Importfunktionalitäten oder Datenbank-Snapshots für die initiale Befüllung der Datenbanken genutzt werden.

Eine weitere Herausforderung ist die zur Durchführung der Tests notwendige Zeit. Systemtests haben gegenüber Unit-Tests typischerweise eine deutliche längere Laufzeit, dadurch sind Systemtests leider kaum bei jedem Einchecken in das Versionsmanagementsystem durchführbar. Vermeidet man Abhängigkeiten von Tests untereinander, kann auch die Ausführung von Systemtests gut parallelisiert werden. Eine Initialisierung des Datenbankbestands durch Tests ist so allerdings nur schwer möglich.

Um die vorgeschlagene Vorgehensweise optimal zu unterstützen, ist es notwendig, die Entwicklung featurebasiert durchzuführen, d. h. Entwickler arbeiten an kleinen, abgegrenzten Arbeitspaketen, die sich auf einzelne Funktionalitäten beschränken. Diese Features können in Form von User Stories oder Use Cases (vgl. [Coc00]) erfasst werden. Wenn man die Szenarien in den Use Cases bereits in der eingeführten Cucumber-DSL beschreibt, können die Szenarien der Use Cases bzw. die Beschreibungen der User Stories direkt als automatisierte Testfallbeschreibung herangezogen werden. Wenn Anforderungen nicht bereits in dieser Form vorliegen, empfiehlt es sich, die Anforderungen in die entsprechende Form zu überführen. Bei dieser Überführung zeigen sich erfahrungsgemäß schnell die Defizite der bestehenden Anforderungen. Um die Defizite auszugleichen, muss eine intensive Kommunikation der

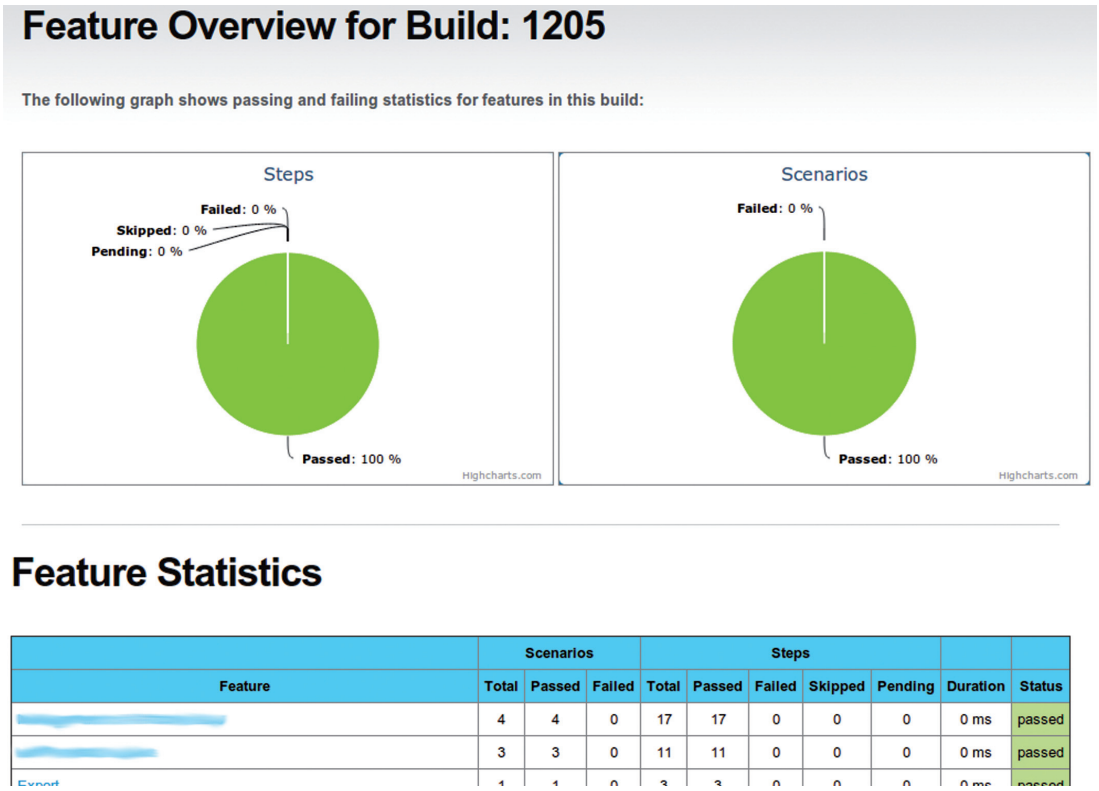

Abb. 2: Reportübersicht.

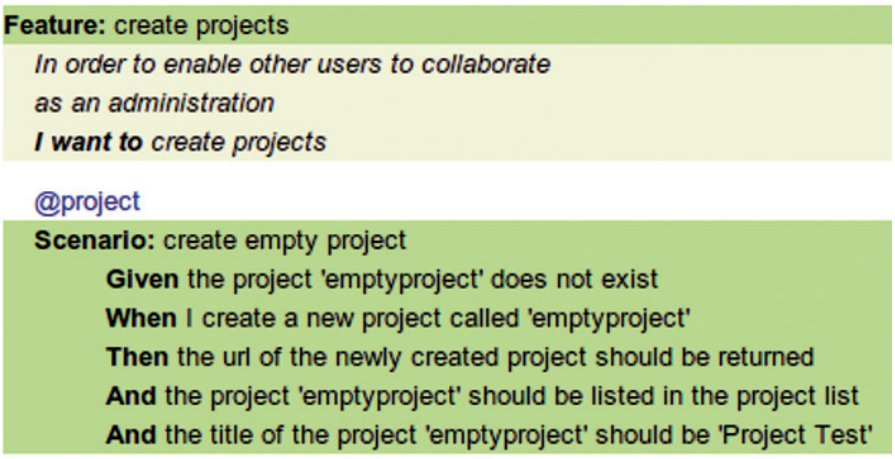

Abb. 3: Reportszenario – Status.

Beteiligten stattfinden. Aus dieser entwickelt sich dann ein gemeinsames Verständnis für die Anforderungen.

Findet dieser Abgleich der Anforderungen nicht statt, werden abweichende Interpretationen von unscharfen Anforderungen erst spät entdeckt und führen zu teuren Nacharbeiten. Die Aufarbeitung der Anforderungen muss am Anfang der Entwicklung, spätestens bei der Planung der nächsten Iteration (Sprint) erfolgen. Die Aufarbeitung erfolgt somit möglichst frühzeitig. Die gewünschte Testbarkeit der Anforderungen ergibt sich damit direkt bei der gezielten Aufarbeitung der bestehenden Anforderungen.

Der Entwicklungsprozess selbst orientiert sich am klassischen Test-Driven Development. Für jedes Feature werden zunächst ein oder mehrere (System-)Testfälle spezifiziert. Eine erste Durchführung des Tests sollte scheitern, da die zu testende Funktionalität noch nicht umgesetzt wurde. Die Entwicklung eines Features ist abgeschlossen, sobald die Tests des Features erfolgreich durchgeführt wurden. Bei einem an Scrum angelehnten Entwicklungsprozess sollte die erfolgreiche Durchführung der Systemtests in die Definition von *Done* für ein Arbeitspaket aufgenommen werden. Die Tests sichern dabei auch automatisch eine gewisse Qualität der Umsetzung ab, sodass Folgeaufwände für Integration oder Nacharbeiten gering ausfallen sollten.

Es empfiehlt sich, die Systemtests so oft wie möglich durchzuführen. Bei realen Projekten bedeutet das typischerweise, dass alle Systemtests ein- bis zweimal pro Tag durchgeführt werden. Das reicht normalerweise aus, um Regressionen schnell genug zu entdecken, sodass diese noch während der Entwicklung eines Features behoben werden können.

Cucumber bietet die Möglichkeit, die Testergebnisse in unterschiedlichen Formaten zu dokumentieren. Eine Möglichkeit besteht darin, die Ergebnisse in einem HTML-Dokument aufzubereiten. Dieses bietet sowohl eine Übersicht über die Ergebnisse als auch detaillierte Informationen, welche die Testfallbeschreibungen und deren Ergebnisse bis auf die einzelnen Sätze der Testfallbeschreibung enthalten. Die HTML-Reports können allen Projektbeteiligten einfach zugänglich gemacht werden und ermöglichen so eine schnelle Übersicht über den Zustand des Projekts. Da es beim vorgeschlagenen Vorgehen eine direkte Verbindung von Features und Tests gibt, zeigt der Report an, welche Features momentan funktionieren und welche nicht. Bei klassischen Unit-Tests ist diese Interpretation der Testergebnisse nicht so einfach möglich.

Des Weiteren können die Testergebnisse auch als Json- oder XML-Datei protokolliert werden. Diese können gut automatisiert weiterverarbeitet werden. Auf diesem Weg können Drittsysteme wie beispielsweise Testmanagementlösungen oder Issue Tracker angebunden werden.

Wenn alle Systemtests zumindest rudimentär implementiert sind, ermöglicht das Reporting zudem eine einfache und aussagekräftige Bestimmung des Fertigstellungsgrades des Gesamtsystems bzw. der aktuellen Iteration (Sprint).

Bei einer Entwicklung mit Feature-Branches können Feature-Branches bei erfolgreichen (System-) Tests mit geringem Risiko automatisiert in einen stabilen Branch übernommen werden. Dadurch ist es mit vertretbarem Aufwand möglich, Kunden oder Testteams eine aktuelle, hinreichend stabile Version eines Systems zur Verfügung zu stellen, um damit neue Features frühzeitig ausprobieren oder weitere manuelle Tests durchführen zu können. Noch weiter gedacht sind durchgehend automatisierte Systemtests eine der Voraussetzungen für automatisierte Produktiv-Deployments (Continous Delivery (vgl. [Hum10])).

Automatisierte Systemtests sind ein wichtiger, aber nicht der einzige Bestandteil einer umfassenden Strategie zur Qualitätssicherung. Der beschriebene Ansatz für verständliche, automatisierte Systemtests hat sich in mehreren Projekten als überaus praktikabel erwiesen. Dabei ist die Testautomatisierung nur ein Teil der Vorteile des Ansatzes. Mindestens genauso wertvoll ist die gemeinsame Testfallentwicklung und das damit verbundene gemeinsame Verständnis der Anforderungen bei Fachseite und Entwicklung. ■

Referenzen

[BecO4] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 2004.
[CocO0] A. Cockburn, Writing Effective Use Cases, Addison-Wesley, 2000.
[Cuc] Cucumber, siehe http://cukes.info.
[Fow10] M. Fowler, R. Parsons, Domain Specific Languages,Addison-Wesley, 2010.
[Fre10] S. Freeman,N. Pryce, Growing Object-Oriented Software, Guided By Tests, Addison-Wesley, 2010.
[Git] Cucumber-JVM, siehe https://github.com/cucumber/cucumber-jvm.
[Hum10] J.Humble, D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, 2010.
[Nor06] D. North, Introducing BDD, Better Software Magazin, 2006.
[Pag] Page Object Pattern, siehe https://code.google.com/p/selenium/wiki/PageObjects.
[Sch01] K. Schwaber, M. Beedle, Agile Software Development with Scrum, Prentice Hall, 2001.
[Spe] SpecFlow, siehe http://www.specflow.org/specflownew/.