Automatisierung von Akzeptanztests im Bereich der Web-Entwicklung

Maximilian Schuierer BSc.
OSRAM Opto Semiconductors GmbH Regensburg

OS AM S&G - AE
Leibnizsstraße 4, 93055 Regensburg

maximilian.schuierer@osram-os.com

Prof. Dr. Frank Herrmann
Ostbayerische Technische Hochschule Regensburg

Innovationszentrum für Produktionslogistik und Fabrikplanung
Galgenbergstraße 32,
93053 Regensburg

frank.herrmann@oth-regensburg.de

Philip Rueck MSc.
OSRAM Opto Semiconductors GmbH Regensburg

OS AM S&G - AE
Leibnizsstraße 4, 93055 Regensburg

philip.rueck@osram-os.com

Abstract

Jede Art von Software muss heutzutage höchsten Anforderungen genügen. Das Testen von Software bildet demnach einen wesentlichen und zunehmend wichtigeren Teil des Softwareentwicklungszyklus. In Zeiten der Digitalisierung ist in vielen Softwareprojekten mittlerweile eine Form der Testautomatisierung integriert. Testautomatisierung ermöglicht die Überprüfung von Anforderungen in hohen Tempo und reduziert zudem den manuellen Prüfungsaufwand auf ein Minimum. Diese Arbeit untersucht die Möglichkeit einer Automatisierung von Akzeptanztests im Rahmen der Webentwicklung in einem agilen Umfeld. Anhand eines Anwendungsbeispiels wird dargestellt, wie mithilfe der Werkzeuge Cucumber und Selenium ein erster Ansatz zur Testautomatisierung umgesetzt werden kann.

Schlüsselwörter

Testautomatisierung, Scrum, Akzeptanztests, Selenium, Cucumber, SpecFlow, Behavior Driven Development

Einleitung

Osram OS stellt auf der Firmenhomepage mehrere Tools für unterschiedliche Bereiche der LED-Nutzung zur Verfügung. Einerseits bieten diese Tools potenziellen Kunden einen Einstiegspunkt in die verschiedenen Serviceleistungen, die das Unternehmen anbietet. Mithilfe des Horticulture Tools beispielsweise können Pflanzenzüchter selbst erste Konfigurationen durchführen, um eine geeignete Kombination aus LEDs zu finden, die zu einer optimalen Belichtung von Zuchtpflanzen beitragen und deren Wachstum steigern. Basierend darauf können Kunden im Anschluss konkrete Anfragen an die jeweiligen Fachexperten starten. Andererseits nutzen auch interne Mitarbeiter diese Applikationen, um für ihre Kunden maßgeschneiderte Lösungen speziell für ihre Bedürfnisse erstellen zu können. Die Bereitstellung von Services via Web Tools soll in den kommenden Jahren noch weiter ausgebaut werden. Um zu gewährleisten, dass die entwickelten Tools allen Anforderungen entsprechen und wie erwartet funktionieren, sind vor Auslieferung hinreichende Tests durchzuführen. Durch ihre Automatisierung soll der erforderliche Aufwand signifikant reduziert werden. Der Artikel ist wie folgt strukturiert. Zunächst wird die Ausgangssituation und die Zielsetzung dargestellt. Dem schließt sich die Erläuterung der Grundlagen von Continuous Integration, des Scrum-Prozesses und der Akzeptanztests an. Für die Automatisierung werden zwei Werkzeuge herangezogen und vorgestellt. Mithilfe dieser Werkzeuge wird ein automatisierter Test für ein typisches Beispiel aufgesetzt und durchgeführt. Die Darlegung der erzielten Ergebnisse sowie eine Zusammenfassung schließen den Artikel ab.

Ausgangssituation

Bisher erfolgte die Entwicklung von Webanwendungen bei OS Regensburg nach dem Vorgehen des Wasserfallmodells. Der zuständige Fachbereich erstellte in Zusammenarbeit mit der IT und externen Consultants eine fachliche Spezifikation, die alle Anforderungen an die Webapplikation definiert. Anhand dieser Anforderungsspezifikation erfolgte die Implementierung der Webanwendung von externen Entwicklern. Während der Testphase stellten manuelle Funktions- und Abnahmetests seitens des Spezifikationsteams die korrekte Umsetzung der Anforderungen sicher. Der Entwicklungsprozess wird nun im Rahmen eines Projekts zu einem agilen Prozess umgebaut. In Zukunft sollen Webanwendungen innerhalb eines Scrum-Projektes realisiert werden. Im Zuge dieses Wechsels ändert sich auch das Vorgehen im Bereich des Testens. Viele der manuellen Akzeptanztests bestehen aus zahlreichen Klick- und Eingabeaktionen, die einen beträchtlichen zeitlichen Aufwand hervorrufen. Ein wesentlicher Anteil dieser Tests könnte durch

den Einsatz geeigneter Tools allerdings auch automatisiert durchgeführt und damit besser in einen agilen Prozess eingegliedert werden. Aus diesem Grund soll in Zukunft die korrekte Umsetzung von Anforderungen an Webanwendungen mithilfe automatisierter Funktions- und Akzeptanztests überprüft werden. Die Konzeption eines Test-Frameworks, der die Durchführung von automatisierten Akzeptanztests im Bereich der Web-Entwicklung ermöglicht, ist daher ein wesentlicher Bestandteil des Projekts und Gegenstand dieser Arbeit.

ZIELSETZUNG

Für die Durchführung zukünftiger Akzeptanztests soll ein Test-Framework erstellt werden. Anschließend gilt es einige funktionale Tests für das Horticulture Web Tool zu generieren, welche mithilfe des Frameworks für weitere Ergebnisse ausgeführt werden. Im Vorfeld wird dazu nach Tools recherchiert, die zur Ausführung von automatisierten Akzeptanztests verwendet werden können. Dabei soll beachtet werden, dass diese als Open-Source-Werkzeug zur Verfügung stehen. Im Rahmen des Wechsels zur Scrum-Methode wird außerdem ein kontinuierlicher Integrationsprozess (Continuous Integration) aufgebaut. Die erstellten Tests sollen abschließend in diesen Prozess eingebunden werden.
Im folgenden wird nun zunächst der Begriff Continuous Integration genauer erläutert sowie kurz auf die Grundlagen der Scrum-Methode eingegangen.

CONTINUOUS INTEGRATION

Bei der Zusammenarbeit mehrerer Entwickler kommt es früher oder später zu Fehlern. Werden die einzelnen Code-Fragmente nicht in regelmäßigen Abständen zusammengeführt, können diese Fehler letztendlich nur schwer oder im schlechtesten Fall gar nicht mehr beseitigt werden. Mit Hilfe der kontinuierlichen Integration lässt sich dieses Problem jedoch lösen. Als Continuous Integration (CI) wird eine Praxis aus der Softwareentwicklung bezeichnet, bei der Softwareentwickler eines Teams ihre Änderungen am Sourcecode regelmäßig in einem zentralen Verzeichnis zusammenführen. Nach jeder Integration wird die gesamte Codebasis durch einen automatisierten Build erstellt und gleichzeitig getestet, um Integrations- und Implementierungsfehler so schnell wie möglich zu erkennen [vgl. Mar06].
Wesentliche Grundlage für eine kontinuierliche Integration ist die Nutzung eines Version Control Systems (z.B. Git), in welchem der Quellcode und sonstige Projektdateien gepflegt werden. Darüber hinaus ist die Automatisierung des Build-Vorgangs eine grundlegende Voraussetzung. Damit sich ein Continuous Integration-Prozess auf lange Sicht lohnt, sollte ein Entwickler-Team ihre Quellcodeänderungen häufig einchecken. Zusätzlich sollten in den Build-Prozess auch eine entsprechende Menge an automatisch durchgeführten Tests eingebunden werden. Auf diese Weise erhalten Entwickler nach dem Einchecken schon in kurzer Zeit Feedback über die Qualität ihrer Änderungen. Automatisch ausgeführte Akzeptanztests gewährleisten eine korrekte Implementierung der gewünschten Funktionalität.

Continuous Integration reduziert nicht nur das Risiko einer lang andauernden Zusammenfügung von Quellcode mehrerer Entwickler. Ein häufiges Einchecken hilft auch, Fehler frühzeitig zu finden und dadurch schnell wieder zu beheben. Darüber hinaus ermöglicht die kontinuierliche Integration ein regelmäßiges Deployment des aktuellen Projekts. Häufige Deployments haben einen wertvollen Nutzen, da sie den zukünftigen Anwendern ermöglichen, neue Funktionen schneller zu erhalten und diese selbst zu testen. Dies sorgt wiederum für ein zügiges Feedback für die Entwickler, die sich dadurch vergewissern können, ob Anforderungen auch den Vorstellungen der Anwender entsprechend umgesetzt wurden [vgl. Mar06; FS12].

Nun wird auf den Ablauf des aufgebauten CI-Prozesses eingegangen, den das Sequenz-Diagramm aus Abbildung 1 darstellt. Ausgangspunkt dieses Prozesses ist das Einchecken von Code in das Version-Control-System, in diesem Fall Git. Dieser Check-In löst auf dem CI-Server einen neuen Build-Vorgang aus. Der Build-Agent ruft daraufhin den Quellcode aus dem Git-Repository ab, kompiliert diesen und führt Unit Tests aus. Nach erfolgreicher Ausführung dieses Schritts startet der Build-Server die Durchführung automatisierter Akzeptanztests. Dies sind jene Tests, die im Rahmen dieses Projekts automatisiert werden sollen. Für das Horticulture Tool wäre dies z.B. die korrekte Berechnung einer Belichtungslösung. Verlaufen alle Tests erfolgreich, ist der Build-Vorgang beendet. Der CI-Server verpackt schließlich die die Dateien in unterschiedliche NuGet-Pakete. *NuGet* ist der von Microsoft unterstützte Mechanismus zur Freigabe von Code in .NET. NuGet-Pakete enthalten kompilierten Quellcode zusammen mit weiteren Inhalten anderer Entwickler. Auf diese Weise wird eine einfache Weitergabe (z.B. an einen Deployment-Server) des “gebauten“ Projektes ermöglicht. Im letzten Schritt benachrichtigt der Server das Entwicklungsteam über den erfolgreichen Build des Projekts. Treten im Laufe des Build-Prozesses Fehler auf, wie beispielsweise fehlerhaftes Kompilieren oder Scheitern von Tests, so bricht der Server den Vorgang ab und benachrichtigt das Team über die aufgetretenen Fehler.

SCRUM

Da von nun an neue Webanwendungen in Scrum-Projekten realisiert werden sollen, folgt in diesem Abschnitt ein grober Überblick über den Ablauf dieser agilen Projektmethode. Scrum ist ein Framework für das Management agiler Softwareprojekte und stellt heutzutage eine der bekanntesten agilen Methoden dar. Scrum ist weniger ein streng definierter Prozess, sondern viel mehr ein Rahmen, in den das Entwicklungsteam ihre bewährte Vorgehensweise einbetten kann. Es besteht aus wenigen Regeln, Prinzipien und Artefakten, die relativ schnell erlernt und produktiv eingesetzt werden können.

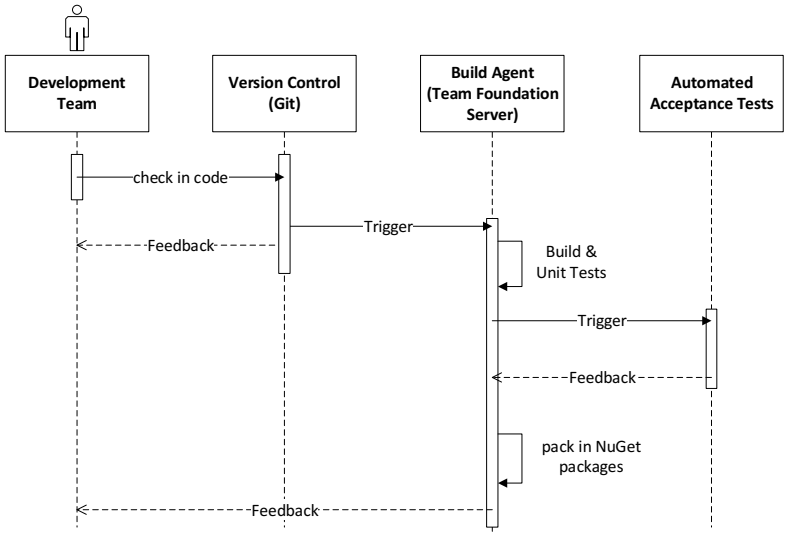

Abbildung 1: Sequenzdiagramm Continuous Integration Prozess (eigene Darstellung)

Im Fokus des Frameworks steht das sich selbst-organisierende Team mit klar definierten Rollen sowie die Konzentration auf kurze Interaktionszyklen, genannt Sprints. Ein Scrum-Team beinhaltet grundsätzlich drei Hauptrollen: den Product Owner, den Scrum-Master und das eigentliche Development Team. Da ein Scrum-Team ein cross-funktionales Team ist, sind nicht nur Entwickler, sondern auch Designer, Tester und Mitarbeiter aus dem Fachbereich Teil des Development Teams. Das Scrum-Team zählt in der Regel 5-9 Mitglieder.
Das Arbeiten in Sprints ist ein wesentliches Merkmal der Projektmethode Scrum. Die Dauer von Sprints sind zeitlich festgelegt und umfassen in der Regel einen Zeitraum von einer bis maximal vier Wochen. In dieser Zeit arbeitet das Team selbstorganisiert und ohne Störungen von außen. Tägliche Stand-up Meetings helfen dem Team, sich gegenseitig zu synchronisieren und über den aktuellen Stand des Projektes auszutauschen. Das Stand-up Meeting dauert exakt 15 Minuten. Jedes Teammitglied beantwortet dabei folgende Fragen:

- Was habe ich gestern gemacht?
- Was habe ich heute vor?
- Welche Hindernisse stören mich bei meiner Arbeit?

Mit Hilfe des täglichen Austauschs kann sehr leicht festgestellt werden, wo das Team im aktuellen Sprint steht und welche Probleme das Erreichen des Sprint-Ziels eventuelle gefährden können. Das Ergebnis eines Sprints ist ein potenziell auslieferbares Produktinkrement. Das Zusammenspiel von aufeinanderfolgenden Sprints und darin enthaltene Daily Scrums ist das Herz der agilen Methode *Scrum* und wiederholt sich bis zum Abschluss des Projekts [vgl. WM17, S.25-45].

USER STORIES

Mit dem Wechsel vom Wasserfallmodell zur Scrum-Methode ändert sich auch das Erfassen von Anforderungen. Während sämtliche Requirements in klassischen Entwicklungspraktiken noch vor dem eigentlichen Projektbeginn in umfangreichen Lasten- und Pflichtenheften detailliert beschrieben werden, verfolgt ein agiles Vorgehen einen wesentlich flexibleren Ansatz. In Scrum-Projekten werden dafür sehr häufig so genannte User Stories verwendet. Mithilfe von User Stories werden Anforderungen an das Projekt oder an die zu entwickelnde Software aus der Sicht des Nutzers bzw. aus der Sicht des Kunden beschrieben. Eine User Story repräsentiert eine kurze, simple Beschreibung eines Features in einem Satz, sie folgt in der Regel einem einfachen Schema:

> *As a <type of user>, I want <some goal> in order to <some reason>.*

Meist bestehen User Stories aus drei Bestandteilen. Den ersten Teil bildet die kurze, prägnante Beschreibung der Anforderung nach dem eben genannten Schema. Zum Beispiel: *As a User, I want a Button in order to be able to calculate a system solution.* Die vage und offene Formulierung von User Stories lässt viel Raum für spätere Anpassungen und fördert den Dialog zwischen den Projektbeteiligten. Dies ist ein großer Vorteil, da sich entsprechende Details in den Anforderungen erst im Laufe des Entwicklungsprozesses genauer bestimmen lassen und erst durch intensive Gespräche mit dem Kunden herausgefunden werden können. Sind die Einzelheiten bekannt werden im zweiten Teil die konkreten Merkmale und Eigenschaften der User Story stichpunktartig hinzugefügt, die sich aus der Kommunikation innerhalb des Development Teams bezüglich der Story ergeben. Vervollständigt wird eine User Story durch die Bestimmung der Akzeptanzkriterien, auch *Definition of Done* bezeichnet. Die Akzeptanzkriterien

definieren das Ziel einer User Story und legen fest, wann sie vollständig umgesetzt ist und damit als “fertig“ betrachtet werden kann. Sie beschreiben also die Bedingungen, welche zur Abnahme der User Story vom Product Owner erfüllt sein müssen. Darüber hinaus bilden die Definition of Done auch die Grundlage für spätere Akzeptanztests. Das Ausformulieren von Abnahmekriterien fällt streng genommen in den Aufgabenbereich des Product Owners, allerdings ist es viel mehr Aufgabe des gesamten Teams über die jeweilige Story zu sprechen, um geeignete Akzeptanzkriterien zu generieren. Betrachtet man das Horticulture Tool, könnte zum Beispiel darüber diskutiert werden, was im System passiert, wenn der Berechnen-Button gedrückt wird, aber nicht alle notwendigen Daten für eine Berechnung vorhanden sind. Wird die Berechnung trotzdem durchgeführt? Bleibt der Button bis zu vollständigen Dateneingabe deaktiviert? Welche Infos werden dem Nutzer zur Hilfestellung angeboten? Aus diesem Dialog ergibt sich ein Pool an Akzeptanzkriterien für die jeweilige User Story, auf deren Basis anschließend Akzeptanztests formuliert werden können[vgl. WM17, S.49-54] .

AKZEPTANZTESTS

Als Akzeptanztests oder Abnahmetests werden funktionale Tests bezeichnet, mit deren Hilfe beschrieben wird, wie Akzeptanzkriterien aus den User Stories getestet werden können. Akzeptanztests vertreten die Interessen des Kunden. Die Tests geben ihm die Sicherheit, dass die Anwendung die gewünschten Eigenschaften aufweist und sich wie erwartet verhält. Infolgedessen haben Akzeptanztests eine große Bedeutung. Abnahmetests konzentrieren sich auf die reine Funktionalität der Anwendung. Die integrierte Logik und der Programmcode werden dabei ignoriert. Aus diesen Grund bezeichnet man Akzeptanztests auch als Black-Box-Tests. Abnahmetests können sowohl manuell als auch automatisiert durchgeführt werden. Durch das zyklische Vorgehen in einem agilen Umfeld bietet sich allerdings vor allem die Automatisierung dieser Tests an. So können beispielsweise die Tests, welche die User Stories im ersten Sprint betreffen, auch in nachfolgenden Sprints erneut durchlaufen werden. Auf diese Weise wird sichergestellt, dass bisher umgesetzte Anforderungen auch nach wie vor fehlerfrei funktionieren [vgl. WM17; Roy11].

Das Automatisieren von Akzeptanztests wird auch als Akzeptanztest-getriebene Entwicklung (ATDD = Acceptance-Test-Driven-Development) bezeichnet, das auf die testgetriebene Entwicklung aufbaut (TDD = Test-Driven-Development). Nach diesem Vorgehen schreibt ein Entwickler vor der eigentlichen Implementierung einen Test der fehlschlägt, da die zugrundeliegende Funktion noch nicht programmiert wurde. Anschließend wird die Funktion so umgesetzt, dass der Test erfolgreich ausgeführt wird. Auf Basis dieser Methode können vor der Umsetzung der funktionalen Anforderungen einer User Story aus den Akzeptanzkriterien automatisierte Testbeispiele generiert werden [vgl. WM17, S.198-199].

WERKZEUGE ZUR TESTAUTOMATISIERUNG

Der Markt bietet zahlreiche verschiedene Tools und Anwendungen, die eine Automatisierung von Akzeptanztests ermöglichen. Hauptkriterium bei der Auswahl passender Tools war deren Verfügbarkeit als Open-Source-Werkzeug. Weiterhin wichtig war, dass eine Automatisierung mithilfe der Programmiersprache .NET erfolgen konnte, da diese den Standard bei Osram darstellt. Im Bereich der Webentwicklung ist Selenium eines der beliebtesten Werkzeuge für eine Testautomatisierung und für alle gängigen Sprachen verfügbar. Als weiteres Tool wurde *Cucumber* in Betracht gezogen. Da sich nach ersten Untersuchungen das Verfassen von Tests mit Cucumber sehr gut in die Erstellung von User Stories integrieren lässt, wurde beschlossen, sich genauer mit diesem Werkzeug auseinander zu setzen. Im folgenden werden nun die beiden Open-Source-Tools vorgestellt.

SELENIUM WEBDRIVER

Der Selenium WebDriver ist eine Komponente aus dem Selenium-Framework. Selenium ist eine Open-Source-Anwendung zur automatisierten Steuerung von Webbrowsern und eines der meist genutzen Tools, um Webapplikationen automatisiert zu testen. Mithilfe von Selenium lassen sich Nutzeraktionen in einem Browser simulieren. Weitere Komponenten des Werkzeugs sind die Selenium IDE sowie das Selenium Grid. Die wohl wichtigste Komponente bildet aber der Selenium WebDriver. Dieser repräsentiert eine objektorientierte API (Application Programming Interface) zum automatisierten Testen von Webanwendungen. Das Testwerkzeug steht unter anderem für die Programmiersprachen Java, JavaScript, C#, Ruby, Perl, Phyton und PHP zur Verfügung. Für die Simulation der Tests können dabei zahlreiche verschiedene Browser verwendet werden. Alle gängigen InternetBrowser, wie zum Beispiel Chrome, Firefox, Internet Explorer, Safari und Opera werden von Selenium unterstützt. Darüber hinaus ist sogar die Automatisierung von Webanwendungen auf mobilen Endgeräten möglich. Einzige Voraussetzung für die Verwendung eines Browsers ist dessen vorhandene Installation auf dem Testsystem. Für jeden von Selenium unterstützten Webbrowser existiert eine Driver-Klasse. Mit einer Objektinstanz des Typs WebDriver kann anschließend ein Browser gestartet und eine Webseite geöffnet werden. Um nun gezielt Aktionen, wie beispielsweise den Klick auf einen Button oder die Eingabe von Text durchführen zu können, sind vorher die Elemente der Webseite zu identifizieren. Für die Identifizierung bzw. Lokalisierung können so genannte *Element Locators* benutzt werden. Ein Beispiel dafür stellt die ID oder der Name eines Webelements dar. Können die Elemente vom Driver-Objekt eindeutig zugeordnet werden, ist von nun an die Ausführung von Aktionen im Browser möglich [vgl. Sel; And15]. Der folgende C#-Code zeigt exemplarisch eine Websuche nach dem Begriff “Selenium“ im Chrome-Browser.

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using OpenQA.Selenium;
using OpenQA.Selenium.Chrome;

namespace UnitTestProject
{
  [TestClass]
  public class UnitTest
  {
    [TestMethod]
    public void TestMethod()
    {
      //WebDriver Instanz
      IWebDriver driver = new ChromeDriver();

      //Aufruf der Webseite google.com
      driver.Navigate().GoToUrl("https://www.google.
    com");

      //Input Feld anhand des Namens identifizieren
      IWebElement GoogleSearchBox = driver.
    FindElement(By.Name("q"));

      //Das Wort 'Selenium' mithilfe des WebDrivers
    in die Suche eingeben
      GoogleSearchBox.SendKeys("Selenium");

      //Suche ausfuehren
      GoogleSearchBox.Submit();

      //WebDriver Instanz und Browser beenden
      driver.Quit();
    }
  }
}
```

Listing 1: Google-Suche nach dem Wort “Selenium“

CUCUMBER

Cucumber ist ein Werkzeug aus der verhaltensgetriebenen Softwareentwicklung, besser bekannt als Behavior-Driven-Development (kurz: BDD). Mit Cucumber können Softwareanforderungen spezifiziert und anschließend als automatisierte Akzeptanztests auf korrekte Umsetzung überprüft werden. Behavior-Driven-Development ist ein Ansatz in der agilen Softwareentwicklung, welcher auf dem Test-Driven-Development (TDD) aufbaut. Wie bei der test-getriebenen Entwicklung wird vor der Programmierung ein Test erstellt. Dieser Test beschreibt das erwünschte Softwareverhalten aus Sicht des Anwenders, ähnlich wie die User Stories. Zentrales Merkmal des Behavior-Driven-Development ist dabei die Verwendung einer einfachen, ubiquitären Sprache, die von jedem Mitglied im Team - auch ohne technischen Hintergrundverstanden werden kann. Im Gegensatz zur testgetriebenen Entwicklung, in welcher die erstellten Tests nur schwer für Nicht-Programmierer zu verstehen sind, dient die ubiquitäre Sprache als Bindemitglied zwischen allen Beteiligten im Team. Cucumber-Tests können infolgedessen von jedem Teammitglied leicht gelesen oder erstellt werden. Dies ist auch der wesentliche Aspekt, den Cucumber von anderen Testwerkzeugen unterscheidet. Die Akzeptanztests fungieren somit nicht nur als reine Tests, sondern als Kommunikations- und Kollaborationswerkzeug. Cucumber fördert nicht nur die Zusammenarbeit zwischen Fachbereich und Entwicklung, sondern stärkt zugleich die Einbeziehung der Stakeholder und hilft ihre Anforderungen zu verstehen.

Cucumber ist standardmäßig ein Kommandozeilen-Werkzeug aus dem Ruby on Rails-Umfeld. Mittlerweile wurde es allerdings auch auf viele weitere Programmiersprachen portiert. Bei der Ausführung von Cucumber wird der beschriebene Test, bezeichnet als *Feature*, eingelesen und auf zu testende *Szenarios* durchsucht. Jedes Szenario besteht aus einer Liste von Schritten, die Cucumber durchläuft. Damit Cucumber diese Features “versteht“, müssen sie eine grundsätzliche Syntax befolgen. Der Name dieser Syntax ist *Gherkin*. Das folgende Listing zeigt ein Beispiel eines Cucumber-Akzeptanztests.

```
Feature: Sign up

  Sign up should be quick and friendly.

Scenario: Successfull sign up
  New users should get a confirmation email and be
    greeted personally
  by the site once signed in.

  Given I have chosen to sign up
  When I sign up with valid details
  Then I should receive a confirmation email
  And I should see a personalized greeting message
```

Listing 2: Cucumber Beispiel

Um nun die erstellten Szenarien auch testen zu können, muss zuvor noch definiert werden, wie die einzelnen Given/When/Then-Anweisungen ausgeführt werden. Dazu gilt es so genannte *Step Definitons* zu implementieren, welche die Schritte aus dem Feature-File auf den Sourcecode abbilden, der anschließend die Aktion ausführt, die in dem jeweiligen Schritt beschrieben ist. Die Erstellung dieser Step Definitions ist Plattform-spezifisch. Ein ausführliches Beispiel zeigt im weiteren Verlauf die Erstellung eines lauffähigen Tests. Das Listing 2 zeigt, wie aus diesem Vorgehen ausführbare Beispiele entstehen, die exakt beschreiben, wie sich das System in bestimmten Situationen verhalten soll. Nach diesem Prinzip verfasste Akzeptanztests können infolgedessen auch als ausführbare Spezifikationen bezeichnet werden. Dieses Vorgehen fördert nicht nur die Kommunikation innerhalb des Projektteams. Es hat darüber hinaus einen starken Effekt, wenn es darum geht, das System zu visualisieren, bevor es entwickelt wurde. Jedes Teammitglied kann einen derartigen Test lesen und darüber entscheiden, ob das beschriebene Verhalten dem Verständnis entspricht, was das System ihrer Meinung nach tun sollte [vgl. WHT17, S.3-9].

In der Praxis wird Cucumber wie die meisten Behavior-Driven-Development-Werkzeuge selten alleine benutzt. Für das vernünftige Testen von Webapplikationen sind dazu bei-

spielsweise noch andere Bibliotheken notwendig, die das Simulieren von Aktionen im Browser ermöglichen. Für diesen Zweck kann Cucumber mit dem Selenium WebDriver kombiniert werden [vgl. Bas13]. Aus diesem Grund wird beschlossen, für die Erstellung automatisierter Akzeptanztests beide Werkzeuge in Kombination zu verwenden.

Aufsetzen eines Frameworks

Im Vorfeld der Erstellung eines Anwendungsbeispiels wird nun ein Test-Framework aufgebaut, der die beiden Tools Selenium und Cucumber integriert. Die Standard-Programmiersprache im Web Tool Bereich bei Osram ist .NET, daher wird auch der Framework in dieser umgesetzt. Im ersten Schritt erfolgt das Anlegen eines Komponententestprojekts in Visual Studio, welches grundsätzliche Basisfunktionen für spätere Tests bereitstellt. Die Tests selbst werden in einem separaten Projekten implementiert, die anschließend auf die grundlegenden Funktionen zurückgreifen. Dazu werden die Bibliotheken für Cucumber und Selenium eingebunden. Um Cucumber in .NET verwenden zu können, muss im Vorfeld die Erweiterung *SpecFlow* installiert werden. SpecFlow ist ein Teil der Cucumber-Familie und repräsentiert die Umsetzung von Behavior-Driven-Development in .NET [Tecb]. Darüber hinaus wird zur Ausführung von Features in .NET ein TestRunner benötigt, z.B. der NUnit 3 Test Adapter. NUnit ist ein Software-Framework zur Ausführung von Unit-Tests für die .NET-Plattform [NUn18].

Zu den Basisfunktionen gehören beispielsweise einige Selenium-Befehle, wie die Initialisierung und das Beenden des Web-Drivers oder das Navigieren zu einer Webseite. Das Nutzen dieser Selenium-Befehle kann in einem Testdurchlauf auch zu technischen Fehlern führen. Insbesondere bei der Ausführung von Browser-Aktionen ist es beispielsweise erforderlich, auf eine Aktualisierung des Testobjekts zu warten. Denn in den meisten Fällen dauert das Laden der Webapplikation länger als der Durchlauf der codierten Anweisungen. Infolgedessen kann zum Beispiel ein Klick auf einen Button nicht ausgeführt werden, da dieser noch nicht erzeugt wurde. In diesem Fall würde der TestRunner die Ausführung mit der Fehlermeldung “NoSuchElementException“ abbrechen. Daher ist es wichtig, auf gewisse Zustände, wie das vollständige Laden der Webseite oder die Sichtbarkeit eines Buttons zu warten. Der Selenium WebDriver kann durch die Konfiguration von expliziten und impliziten Warte-Befehlen präziser gesteuert werden. Um dies zu gewährleisten, werden in das Basis-Projekt zusätzlich einige Warte-Methoden integriert.

Durch die Implementierung dieser Funktionen in das Basisprojekt sind die WebDriver-Funktionen, die für jeden Test erforderlich sind, vom eigentlichen Test gekapselt. Die Übersicht und Wartbarkeit wird dadurch erheblich gesteigert. Der Framework wird im folgenden um ein Anwendungsbeispiel erweitert.

Anwendung auf ein Test-Beispiel

Der folgende Beispiel-Test soll nun zeigen, wie ein automatisierter Akzeptanztest unter Verwendung von Cucumber und Selenium erstellt und ausgeführt wird. Als Beispiel-Webapplikation dient dazu das eingangs erwähnte Horticulture Web Tool. Für den Test wird ein neues Komponententestprojekt erzeugt und das Basisprojekt als Referenz darin integriert. So kann nun auf die Selenium-Methoden des Frameworks zurückgegriffen werden. Des Weiteren erfolgt die Einbindung der Tool-Bibliotheken, wie zuvor bei der Erstellung des Basisprojekts.

Feature-Datei anlegen

Nach dem Integrieren der Referenzen wird ein Feature-File erstellt. Mithilfe des Horticulture Tools kann unter anderem errechnet werden, wie viele LEDs einer bestimmten Art nötig sind, um einen bestimmten Helligkeitswert zu erreichen. Diese Rechnung wird im folgenden als Anwendungsbeispiel verwendet. Da das Beispiel zugleich eine Reihe von Testdaten umfasst, repräsentiert es einen Daten-getriebenen Test.

```
Feature: Calculation

Scenario Outline: Calculate Solution using different
      LEDs
  Given I have opened the horticulture webtool
  And I have given a <value> for Target Photon Flux
  | value |
  | 1000  |
  When I select a <LED>
  And I click the calculate button
  Then I should see the results as <LED Quantity>

Examples:
| LED             | LED Quantity |
| GA PSLR31.13    | 786          |
| GD CS8PM1.14    | 440          |
| GD PSLR31.13    | 604          |
| GF CSSPM1.24    | 640          |
| GW CS8PM1.CM    | 569          |
| GW DASPA1.EC    | 2106         |
| LA CN5M         | 3273         |
| LA G6SP         | 3218         |
```

Listing 3: LED Auswahl

Listing 3 zeigt das gewünschte Verhalten in der Gherkin-Syntax. Im ersten Schritt des Anwendungsbeispiels wird das Horticulture Web Tool im Browser geöffnet. Anschließend wird der Zielwert für die Photon Flux auf den Wert 1000 gesetzt. In der nächsten Anweisung erfolgt das Auswählen einer LED. Infolgedessen wird berechnet, wie viele dieser LED benötigt werden, um den Wert von 1000 Photon Flux zu erreichen. Im letzten Schritt erfolgt die Überprüfung auf die richtige Anzahl. Der Test aus diesem Feature wird für jede Zeile aus der Examples-Tabelle einmal ausgeführt und die Parameter in den Anweisungen durch die Werte in der Tabelle ersetzt. Im ersten Durchlauf wird also <LED> mit *GA PSLR31.13* und <LED Quantity> mit *786* ersetzt. Auf diese

Weise kann der gleiche Test mehrmals mit unterschiedlichen Daten durchgeführt werden, muss zur Ausführung allerdings nur einmal gestartet werden.

Nach der Erstellung der Feature-Datei fällt auf, dass SpecFlow im Hintergrund eine Datei mit automatisch generierten Sourcecode erstellt. Dieser Code dient als Setup zur Ausführung des erstellten Features, zum Beispiel wird darin der zur Durchführung benötigte TestRunner assoziiert. Außerdem dient diese Datei als so genanntes "Mapping" zwischen den Feature-Files und den Step Definitions, welche zur Ausführung dieses Szenarios zusätzlich noch erstellt werden müssen.

Step-Definitions generieren

In Visual Studio lassen sich diese sehr einfach generieren. Durch den Klick der rechten Maustaste in der Feature-Datei und der Auswahl "Generate Step Definitions" aus dem Kontextmenü können die Schritte deklariert werden. Nun wird dem Projekt eine weitere .cs-Datei hinzugefügt, welche die Schritte zum obigen Feature enthält. Listing 4 zeigt dazu den Inhalt dieser Datei. Die einzelnen Given/When/Then-Anweisungen aus dem Feature-File werden durch das [Binding]-Attribut und den regulären Ausdrücken mit den Methoden in dieser Datei verbunden. Ein regulärer Ausdruck ist eine Methode, die in der Programmierung für die Mustererkennung verwendet wird [vgl. Teca]. SpecFlow generiert die Methodennamen automatisch aus den Wörtern der Step-Anweisung im Feature-File. Diese können allerdings auch umbenannt werden, da die Bezeichnung der Methoden zur weiteren Nutzung irrelevant ist. Durch die regulären Ausdrücke sind die Anweisungen bereits eindeutig referenziert.

```
using System;
using TechTalk.SpecFlow;

[Binding]
public class CalculationSteps
{
  [Given(@"I have opened the horticulture webtool")]
  public void GivenIHaveOpenedTheHorticultureWebtool
    ()
  {
    ScenarioContext.Current.Pending();
  }

  [Given(@"I have given a (.*) for Target Photon
    Flux")]
  public void GivenIHaveGivenAForTargetPhotonFlux(
    string p0, Table table)
  {
    ScenarioContext.Current.Pending();
  }

  [When(@"I select a (.*)")]
  public void WhenISelecta(string LED)
  {
    ScenarioContext.Current.Pending();
  }

  [When(@"I click the calculate button")]
  public void WhenIClickTheCalculateButton()
  {
    ScenarioContext.Current.Pending();
  }

  [Then(@"I should see the results as (.*)")]
  public void ThenIShouldSeeTheResultsAs(int p0)
  {
    ScenarioContext.Current.Pending();
  }
}
```

Listing 4: Step Definitions

Nach der Erstellung der Step Definitions kann das Szenario ein erstes mal ausgeführt werden. Allerdings bleibt dieser Test vorerst ergebnislos. Alle Methoden enthalten bisher nur die Anweisung *Scenario.Current.Pending()*. Nun gilt es diese Anweisungen zu ersetzen und die Methoden dahingehend auszuarbeiten, dass sie die gewünschten Aktionen aus dem Feature-File an der Applikation ausführen, in diesem Fall im Horticulture Web Tool.

Anweisungen implementieren

Die erste Anweisung aus dem Szenario ist das Aufrufen des Horticulture Tools. Dafür muss der Selenium WebDriver den Browser (in diesem Beispiel Chrome) öffnen und anschließend zum Horticulture Tool navigieren. Die Methoden dafür sind bereits im Framework implementiert und müssen demnach nur noch aufgerufen werden. Im nächsten Schritt erfolgt die Eingabe eines Wertes für die Höhe der Photon Flux. Für diese Anweisung muss vorher das Eingabe-Element vom WebDriver-Objekt identifiziert und lokalisiert werden. Diese Lokalisierung könnte nun für jeden Button, jedes Eingabefeld etc. innerhalb der Step-Methode integriert werden. Dies ist allerdings eine eher unsaubere und unübersichtliche Herangehensweise. Für eine bessere Wartung des Testprojekts ist es daher klüger, die einzelnen Komponenten einer Webseite in separaten Modulen zu pflegen. Dafür bietet sich die Verwendung des Page Object Design Patterns an. Dieses Entwurfsmuster legt fest, dass Elemente und Funktionen einer zu testenden Applikation an anderer Stelle als die eigentlichen Tests implementiert werden. Der WebDriver greift anschließend über diese Funktionen auf die jeweiligen Objekte zu und führt die Aktionen aus. Auf diese Weise kann das Testprojekt leicht auf Änderungen in der echten Applikation reagieren [vgl. Gun15, S.181 ff.]. Folglich werden die für dieses Szenario benötigten Elemente und Funktionen des Horticulture Tools in einer separaten Klasse deklariert.

Im nächsten Schritt wird eine LED im Produkt-Selektor ausgewählt und die Berechnung durch den Klick auf den Calculate-Button gestartet. Schließlich überprüft die Then-Anweisung, ob die errechnete Anzahl der LEDs mit der erwarteten übereinstimmt. Diese Prüfung erfolgt unter der Verwendung des Assert-Befehls. Die Assert-Klasse ist in der Unit-Testing-Bibliothek von Microsoft enthalten und ermöglicht beispielsweise das Prüfen auf Gleichheit. Stimmt die Anzahl

der benötigten LEDs in der Berechnung mit der erwarteten Anzahl überein, gibt die Assert-Anweisung den Wert “true“ zurück und der Test wurde damit erfolgreich ausgeführt. Das nachfolgende Listing 5 zeigt nun einen Ausschnitt der implementierten Methoden.

Nach der Implementierung der Step-Methoden kann das Szenario in Visual Studio innerhalb des integrierten Test Explorers ausgeführt werden. Die Ausführung des Tests ist auch im Debug-Modus möglich. Aus dem erstellten Testprojekt lässt sich im Anschluss auch eine DLL (Dynamic Link Library) generieren. Diese kann schließlich in den Build-Prozess des Continuous Integration Systems eingebunden werden. Sobald das Entwicklungsteam nun Änderungen im Sourcecode in das Code-Repository einchecken (z.b. Git) und das Projekt erstellt wird, werden automatisch die erstellten Akzeptanztests durchlaufen.

```
[Binding]
public class CalculationSteps
{
[Given(@"I have opened the horticulture webtool")]
public void GivenIHaveOpenedTheHorticultureWebtool()
{
  WebDriver.NavigateToUrl(HorticultureToolPage.
    DevUrl);
  PropertiesCollection.currentPage = new
    HorticultureToolPage();
}
[Given(@"I have given a (.*) for Target Photon Flux"
    )]
public void GivenIHaveGivenAForTargetPhotonFlux(
    string p0, Table table)
{
  dynamic value = table.CreateDynamicInstance();
  PropertiesCollection.currentPage.As<
    HorticultureToolPage>().SetTargetPhotonFlux(
    value.value);
}
[When(@"I select a (.*)")]
public void WhenISelecta(string LED)
{
  PropertiesCollection.currentPage =
    PropertiesCollection.currentPage.As<
    HorticultureToolPage>().AddLEDButtonClick();
  PropertiesCollection.currentPage =
    PropertiesCollection.currentPage.As<
    LEDSelectionPage>().SelectLED(LED);
}
[When(@"I click the calculate button")]
public void WhenIClickTheCalculateButton()
{
  PropertiesCollection.currentPage.As<
    HorticultureToolPage>().Calculate();
}
[Then(@"I should see the results as (.*)")]
public void ThenIShouldSeeTheResultsAs(int
    LEDQuantity)
{
  string expectedQuantity = LEDQuantity.ToString();
  Assert.AreEqual(expectedQuantity,
    PropertiesCollection.currentPage.As<
    HorticultureToolPage>().GetLEDQuantityFirstLED()
    .ToString());
}
}
```

Listing 5: Step Definitions implementiert

Erzieltes Ergebnis

Der Einsatz von automatisierten Akzeptanztests in der Web-entwicklung erweist sich durchaus als sinnvoll. Denn dadurch können Anforderungen in hoher Geschwindigkeit auf korrekte Umsetzung geprüft werden. Manuelle Funktionstests, die im wesentlichen aus Klick-Operationen bestehen, müssen somit nicht mehr von Mitarbeitern durchgeführt werden. Eine wiederkehrende Ausführung von Tests während der fortschreitenden Entwicklung sorgen darüber hinaus für eine hohe Qualität und vermeidet das Entstehen komplexer Fehler. Durch die Integration der Tests in einen Continuous Integration Prozess erhalten Entwickler zudem ein schnelles Feedback, ob Ihre Änderungen auch fehlerfrei implementiert wurden.

Das Anwendungsbeispiel schafft einen Einblick, Akzeptanztests automatisiert mithilfe der Werkzeuge Cucumber (SpecFlow für .NET) und Selenium durchzuführen. Insbesondere das Vorgehen nach dem Behavior-Driven-Development kann gut in ein agiles Umfeld integriert werden. Jedes Teammitglied kann Feature-Dateien lesen und auch selbst erstellen. Das Schreiben von User Stories und ein gleichzeitiges generieren zugehöriger Features bietet sich an. Darüber hinaus dienen die Feature-Dateien auch als Dokumentation. Zudem sind Selenium und SpecFlow beides Open-Source-Werkzeuge und bieten eine umfassende Dokumentation. Die Arbeit mit beiden Tools ist leicht erlernbar und führt schnell zu brauchbaren und umfangreichen Ergebnissen. Darüber hinaus kann der Funktionsumfang der Werkzeuge durch zahlreich verfügbare Plug-Ins erweitert werden.

Allerdings muss bei allen Vorteilen, die aus der Automatisierung resultieren, auch bedacht werden, dass dieses Vorgehen auch mit nicht unerheblichen Aufwand verbunden ist. Die Feature-Files müssen erstellt und gepflegt werden, darüber hinaus sind für die Ausführung von aussagekräftigen Tests auch fundierte Testdaten notwendig. Zudem müssen die Anweisungen aus den Features in den Step Definitions implementiert werden. Den wohl größten Aufwand bildet dabei das “Nachbauen“ des zu testenden Objekts im Page Object Model, damit Selenium die Aktionen am Testobjekt auch durchführen kann. Darüber hinaus ist das Abfangen etwaiger Fehler zu berücksichtigen. Ohne umfassende Fehlerbehandlung ist eine Automatisierung von Tests wenig sinnvoll. All diese Aufwände gilt es bei der Projekt-Planung bzw. auch bei der Planung jedes Sprints zu bedenken. Mithilfe eines gut ausgebauten Frameworks können Änderungen zwar relativ einfach vorgenommen werden, jedoch erfordert dies ein konsequentes Vorgehen. Ein möglicher Ansatz für ein agiles Team wäre, zu Beginn des Projekts bzw. eines Sprints zu prüfen, in welchen Bereichen sich eine Testautomatisierung anbietet und welchen Bereich man doch besser manuell testet.

Zusammenfassung

Diese Arbeit bietet einen Überblick über die Automatisierung von Akzeptanztests im Bereich der Web-Entwicklung. Anhand

eines Anwendungsbeispiels wurde gezeigt, wie automatische Akzeptanztests unter Verwendung der Werkzeuge Selenium und Cucumber erzeugt werden können. Das vorgestellte Verfahren erlaubt Anforderungen an Webanwendungen in hohem Tempo und ohne großen manuellen Überprüfungs-Aufwand auf korrekte Umsetzung zu prüfen.

LITERATUR

[And15] Andreas Monschau. *Automatisiertes Testen von Weboberflächen - JAXenter*. UI-Test mit Selenium WebDriver. Hrsg. von JAXenter. 2015. URL: https://jaxenter.de/automatisiertes-testen-von-weboberflaechen-30703 (besucht am 21.08.2018).

[Bas13] Bastian Krol. *Cucumber: Setup und Grundlagen*. 2013. URL: https://blog.codecentric.de/2013/08/cucumber-setup-grundlagen/ (besucht am 21.08.2018).

[FS12] Björn Feustel und Steffen Schluff. *Continuous Integration in Zeiten agiler Programmierung*. Heise Medien. 2012. URL: https://www.heise.de/developer/artikel/Continuous-Integration-in-Zeiten-agiler-Programmierung-1427092.html?seite=all (besucht am 22.08.2018).

[Gun15] Unmesh Gundecha. *Selenium testing tools cookbook*. eng. Second edition. Quick answers to common problems. Gundecha, Unmesh (VerfasserIn). Birmingham, UK: Packt Publishing, 2015. 11 S.

[Mar06] Martin Fowler. *Continuous Integration*. 2006. URL: https://martinfowler.com/articles/continuousIntegration.html (besucht am 22.08.2018).

[NUn18] NUnit.org. *NUnit.org*. 3.07.2018. URL: http://nunit.org/ (besucht am 07.08.2018).

[Roy11] Roy W. Miller, Christopher T. Collins. „acceptance testing“. In: *SpringerReference*. Berlin/Heidelberg: Springer-Verlag, 2011. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.18.5040&rep=rep1&type=pdf (besucht am 29.08.2018).

[Sel] Selenium. *Selenium - Web Browser Automation*. URL: https://www.seleniumhq.org/ (besucht am 07.08.2018).

[Teca] Technopedia. *What is a Regular Expression? - Definition from Techopedia*. URL: https://www.techopedia.com/definition/25843/regular-expression (besucht am 13.08.2018).

[Tecb] TechTalk Software Support. *SpecFlow - Binding Business Requirements to .NET Code*. URL: https://specflow.org/ (besucht am 21.08.2018).

[WHT17] Matt Wynne, Aslak Hellesøy und Steve Tooke. *The cucumber book. Behaviour-driven development for testers and developers*. eng. Second edition. The pragmatic programmers. Wynne, Matt (VerfasserIn) Hellesøy, Aslak (VerfasserIn) Tooke, Steve (VerfasserIn). Dallas, Texas: Pragmatic Bookshelf, 2017. 11 S.

[WM17] Ralf Wirdemann und Johannes Mainusch. *Scrum mit User Stories*. ger. 3., erweiterte Auflage. Wirdemann, Ralf (VerfasserIn) Mainusch, Johannes (VerfasserIn). München: Hanser, 2017. 271 S.