Fachbeitrag

Sven Koesling

Automatisiertes Testen von Webapplikationen

Wie man Testaufwand reduziert und gleichzeitig die Softwarebeschaffung optimiert

DOI 10.1515/abitech-2017-0025

Zusammenfassung: Der aufwendige und immer wiederkehrende Prozess des Testens kann automatisiert und vom Computer übernommen werden. Dabei ist nicht nur die Arbeitsersparnis von Vorteil, auch der Beschaffungsprozess neuer Software profitiert von der für die Tests notwendigen klaren Formulierung der Anforderungen und der Kommunikation unter allen Beteiligten. Der Autor zeigt an einem vereinfachten Beispiel, wie verbreitete Probleme bei der Implementierung neuer Software durch die Rahmenbedingungen, die automatisiertes Testen setzt, vermieden werden können.

Schlüsselwörter: Automatisierung, Webapplikation, testen

Automated testing of web applications

How to reduce costs for testing and optimize software procurement at the same time

Abstract: The complex and recurrent process of testing can be automated and taken over by the computer. Not only does this result in the saving of resources, but also in an improvement of the procurement process. The latter profits from clear definition of the requirements and improved communication among all parties involved in the tests. The author uses a simplified example to explain how common problems with the implementation of new software can be avoided by the framework set up by automated testing.

Keywords: automation, web application, testing

Der Prozess des Testens von Software ist aufwendig. Immer wieder müssen Gruppen von Personen – idealerweise mit unterschiedlichen Rollen – für Tests zur Implementierung neuer Software bzw. Features koordiniert werden, Testvorlagen erstellt, Tests durchgeführt und Protokolle ausgefüllt werden. Und das Testen selber ist eine uninteressante Tätigkeit, besonders, wenn sie durch iterative Entwicklung häufig wiederholt werden muss.

Entwickler setzen darum schon lange auf Test-Tools, Frameworks, die es ihnen ermöglichen, Software schnell und unkompliziert zu testen. Je nach Anforderung kommen unterschiedliche Test-Tools zum Einsatz. Für das Testen des Verhaltens von Webapplikationen, die heute durch Cloud Computing und „Software as a Service“-Angebote immer weitere Verbreitung finden, bietet sich die Software *Cucumber* an, die auch im Folgenden in den Beispielen verwendet werden wird. *Cucumber* ist ein Werkzeug zur textuellen Spezifikation von Software-Anforderungen und zur automatisierten Überprüfung dieser Beschreibung auf ihre korrekte Implementierung. *Cucumber* zeichnet sich dadurch aus, dass die zu testenden Features in nahezu natürlicher Sprache in Szenarien beschrieben werden, so dass Anwender und Techniker zu einem gemeinsamen Verständnis der Anforderungen an die zu testende Software kommen. Der Computer übernimmt die stupide Arbeit des regelmäßigen Testens, und die notwendige standardisierte Formulierung der Spezifikationen trägt zur Qualitätssicherung bei, wie im Folgenden aufgezeigt wird.

Dazu wird ein typischer Beschaffungsprozess von Software skizziert, in dem zuerst die Anforderungen definiert, dann bei einem geeigneten Anbieter die Software bestellt und schließlich getestet wird, ob sie die geforderten Features aufweist.

Was sind die Anforderungen?

Bei der Einführung neuer Software oder auch nur neuer Features geht es darum, den Nutzenden einen Vorteil zu bieten. Dabei spielt es keine Rolle, ob die Neuerung Angestellten die Arbeit oder Kunden und Kundinnen den Zugang zu den Angeboten erleichtern soll.

Drei Fragen sind für die Bewertung von Neuerungen entscheidend:

1. Bietet die neue Software bzw. das neue Feature der Zielgruppe einen Mehrwert?

2. Leistet die neue Software bzw. das neue Feature das, was gefordert wurde?
3. Funktionieren die bereits bestehenden Features wie bisher?

Stellt man diese Fragen in einem Meeting, wird man feststellen, dass sie von allen unterschiedlich eingeschätzt und interpretiert werden. Jede und jeder hat ein anderes Verständnis von dem betreffenden Feature.

Es ist für die Einführung von neuer Software bzw. von neuen Features wichtig, dass alle das gleiche Verständnis der Anforderungen haben. „Software beginnt mit einer Idee. [...] Die Idee muss aus dem Kopf einer Person in die Köpfe anderer Personen reisen. Sie muss kommuniziert werden.“[1] Für die Automatisierung von Tests ist es sogar zwingend notwendig, die Anforderungen an die Software und die Zielgruppe eindeutig zu beschreiben. Nebenbei hilft das Formulieren der Testszenarien dabei, allen Beteiligten ein gemeinsames Verständnis der Anforderungen zu geben. Tatsächlich ist das eindeutige Formulieren von Anforderungen die Lösung vieler Probleme.

Nachfolgend wird ein Beispiel konstruiert, das die typischen Probleme bei der Einführung von Software illustriert und zeigt, welche Lösungen und Verbesserungen durch automatisiertes Testen entstehen können.

Ein Beispiel: Anforderungen an eine Kaffeekasse

Eine Abteilung möchte eine Software einführen, mit der die interne Kaffeekasse abgerechnet werden kann. Automatisiertes Testen ist hier noch unbekannt, die Software wird von den Anwendern getestet. Es wird ein Anforderungsdokument erstellt, mit dem der Softwareanbieter das Produkt entwickeln soll. Das Dokument wird auf dem internen Dokumentenmanagementsystem abgelegt, so dass jeder Mitarbeiter seine Anforderungen dort eintragen kann.

Der Mitarbeiter Reto Müller – er verwaltet die Kasse – schreibt in das Anforderungsdokument, dass zwingend eine Userliste benötigt würde. Ohne sie kann er die Kasse nicht verwalten. Die Mitarbeiterin Sabrina Meier – zuständig für Statistik und Auswertungen – findet eine Userliste praktisch. So bekommt sie leichter einen Überblick über die Mitarbeiter. Dann geht die Innovationsmanagerin Marisa Schmitt über das Anforderungsdokument – sie findet die Liste überflüssig und kontraproduktiv. Drei Personen, von denen jede den Mehrwert unterschiedlich einschätzt. Für Reto Müller ist die Liste zum Arbeiten zwingend notwendig, Sabrina Meier bietet sie eine willkommene Erleichterung ihrer Arbeit und Marisa Schmitt braucht sie gar nicht.

Der Softwareanbieter bekommt den Auftrag, eine Userliste zu programmieren. In den ersten Tests der neuen Software stellt der Mitarbeiter Reto Müller fest, dass die Userliste zwar da ist, aber vollkommen unnütz, da in ihr alle Mitarbeiter vorkommen. Er kann nicht jedes Mal alle 250 Mitarbeiter durchsehen, um für fünf von ihnen den Kaffeeumsatz auszurechnen. Sabrina Meier dagegen ist begeistert – sie hat endlich mal eine Übersicht über alle Mitarbeiter. Marisa Schmitt ist beruhigt. Sie hatte befürchtet, dass es bei der Liste um einen Papierausdruck ginge, das hätte nicht zu einem innovativen Unternehmen gepasst. Offensichtlich hatten alle drei unterschiedliche Vorstellungen von der Userliste.

Da Reto Müller mit der Software so nicht arbeiten kann, ergeht der Auftrag an den Software-Anbieter, eine Liste der Kaffeetrinkenden zu programmieren. Bei seinen folgenden Tests ist Reto Müller zufrieden. Seine Liste zeigt ihm, welche Personen er an die Einzahlungen in die Kaffeekasse erinnern muss. Die Software wird abgenommen und bezahlt. Als Frau Meier aus ihren Ferien zurückkommt, stellt sie fest, dass die Liste der Mitarbeiter nicht mehr funktioniert – es werden nicht mehr alle Mitarbeiter angezeigt. Das ist ein uns allen bekanntes Problem: Nach einem Update funktionieren die bereits bestehenden Features nicht mehr wie bisher.

Frau Meier ruft also die IT an und meldet, dass ihre Liste unvollständig ist. Die IT programmiert ja nicht selbst und schreibt deshalb einen Support-Case an den Softwareanbieter. Der First Level Support des Anbieters findet keinen Fehler, übergibt das Problem an den Second Level Support, der reicht es weiter an die Entwicklung und von dort kommt dann sechs Wochen später zurück: ,,Works as designed.“

Das Beispiel zeigt die typischen Probleme bei der Softwarebeschaffung bzw. -entwicklung. Zwischen ,,works as designed“, ,,works as requested“ und ,,works as it should“ gibt es Unterschiede. Es kommt immer zu Diskussionen, ob das, was geliefert wurde, das ist, was bestellt wurde. Und bei jedem Update besteht das Risiko, dass benötigte Funktionen nicht mehr da sind.

Solche Probleme vermeidet man, indem die verschiedenen beteiligten Personen miteinander über die Anforderungen detailliert sprechen. *Sprechen* – nicht mailen oder auf SharePoint Protokolle austauschen. Die Wichtigkeit des Miteinander-Sprechens wird von vielen Autoren von

1 Wynne, Matt, Aslak Hellesøy. *The Cucumber Book. Behaviour-Driven Development for Testers and Developers (Pragmatic Programmers)*. Dallas (Texas), Raleigh (North Carolina), 2012. (Übersetzung S. K.)

Werken zu agilen Methoden betont, z. B. von Jeff Patton in seinem Buch *User Story Mapping*.[2]

Es genügt nicht, wenn jede Person für sich Requirements aufschreibt – die Spezifikationen sollten gemeinsam formuliert werden. Dabei müssen die Anforderungen so formuliert werden, dass allen – auch Außenstehenden – aus den Texten klar wird, worum es bei einem Feature gehen soll. Wie oben gezeigt wurde, verstehen schon die internen Mitarbeiter nicht das Gleiche, wenn sie vom selben Feature reden. Wie soll ein externer Anbieter die Anforderungen dann umsetzen?

Gemeinsames Ausarbeiten von Anforderungen

Es wichtig, für das Ausarbeiten der Anforderungen genügend Zeit einzuplanen, so dass jedes Feature gut ausformuliert werden kann. Wenn die Anforderungen sauber formuliert sind, kann die Zeit durch Automatisierung der Tests wieder hereingeholt werden, wie im Folgenden gezeigt wird.

Eine sehr wirksame Technik ist es, aus einer Anforderung mit den Fragen „Wer?“, „Was?“ und „Warum?“ eine Story zu machen. Allgemein kann man sich gut an folgende Formulierung halten:

Als **ROLLE** möchte ich **FEATURE**, um **ZIEL** zu erreichen.

Vor dem Hintergrund des skizzierten Beispiels lautet eine Anforderung (die des Reto Müller) so:

Als **Kassenwart** möchte ich **eine Liste haben, in der alle an der Kaffeekasse Beteiligten aufgelistet sind**, so **dass ich sie an die Einzahlungen in die Kaffeekasse erinnern kann**.

Und eine weitere Anforderung (Sabrina Meier):

Als **Business Analyst** möchte ich **eine Liste aller Mitarbeitenden** haben, um eine **Übersicht über alle Abteilungen** zu bekommen.

Diese Formulierung macht die Anforderung eindeutig. Derartige Stories lassen sich auf Karten notieren, sortieren, gruppieren, priorisieren, mappen usw. Indem man Stories gruppiert, bilden sich Szenarien, die standardisiert aufgeschrieben und so dokumentiert werden können. Durch eine standardisierte Formulierung und eine standardisierte Notation wird es leichter, mit allen Personen zum gleichen Verständnis vom Umfang und Inhalt eines Features zu kommen.

2 Patton, Jeff, Peter Economy: *User Story Mapping. discover the whole story, build the right product*. O'Reilly: Sebastopol Cal., 2014.

Standardisierte Formulierungen von Anforderungsszenarien

Die Standardisierung ist zunächst etwas anstrengend für alle Beteiligten, aber standardisierte Formulierungen von Anforderungen haben zwei entscheidende Vorteile:

1. Alle Beteiligten wissen genau, wovon die Rede ist.
2. Standardisierte Formulierungen lassen sich maschinell auswerten.

Tatsächlich lassen sich standardisierte Formulierungen benutzen, um Software automatisch auf die gewünschten Features zu testen. Für die Standardisierung benutzt das hier verwendete Framework *Cucumber* die Sprache *Gherkin*. *Gherkin* ist keine eigene Sprache, sondern eher eine Regelsammlung zum Notieren von Anforderungsszenarien. Sie wurde entwickelt, um das Verhalten von Software zu beschreiben und dient dazu, Anforderungen zu dokumentieren und automatisiert zu testen.

Zur Veranschaulichung soll wieder das Beispiel „Kaffeekasse“ mit der Anforderung der Userliste dienen. Das Unternehmen hat Zeit und gemeinsame Sitzungen für die Anforderungsanalyse eingeplant, und in einer dieser Sitzungen hat Reto Müller die Forderung nach einer Userliste aufgeworfen. Im gemeinsamen Gespräch kommt heraus, dass auch Sabrina Meier eine Userliste benötigt, aber eine andere, als Herr Müller. Gemeinsam wird daraus nun mit *Gherkin* eine Funktionalität für die gesamte Applikation geschrieben.

Die Beschreibung beginnt mit einem Titel für die Funktionalität, zum Beispiel „Userlisten“. Darunter werden die Userstories notiert, Grundlagen definiert und schließlich die Szenarien und deren Testschritte aufgeschrieben.

```
Funktionalität: Userlisten
 Als Kassenwart möchte ich eine Liste haben, in der alle an der
    Kaffekasse Beteiligten aufgelistet sind, so dass ich sie an die
    Einzahlungen in die Kaffekasse erinnern kann.

 Als Business Analyst möchte ich eine Liste aller Mitarbeitenden haben,
    um eine Übersicht über alle Abteilungen zu bekommen.

 Grundlage:
  Gegeben sei , dass in der DB "50" Mitarbeiter angelegt sind
  Und dass davon "5" an der Kaffeekasse beteiligt sind.

 Szenario: Liste aller Kaffeetrinkenden
  Wenn ich als Kassenwart die Userliste öffne,
  Dann sollen in ihr alle Kaffeetrinkenden angezeigt werden.

 Szenario: Liste aller Mitarbeiter
  Wenn ich als Business Analyst die Userliste öffne,
  Dann sollen in ihr alle Mitarbeiter angezeigt werden.
```

Abb. 1: Beschreibung einer Funktionalität mit *Gherkin*

Damit ist die Anforderung klar beschrieben, und dem Tester ist sofort klar, was er oder sie testen muss und was das Ziel des jeweiligen Features ist. Und auch dem Softwareanbieter lässt sich damit genau mitteilen, was benö-

tigt wird. Aber zusätzlich lässt sich die Anforderung durch die Standardisierung nun maschinell überprüfen!

Automatisiertes Testen

Trotz allen Fortschritts können Computer immer noch keine Texte auf dem Niveau interpretieren, das für das Testen von Features notwendig wäre. Darum übersetzt der freundliche Kollege von der IT die Anforderung in einen Code[3]:

```
Wenn(/^ich als Kassenwart die Userliste öffne,$/) do
  visit login_path
  fill_in("login", with: @kassenwart.login)
  fill_in("password", with: @kassenwart.password)
  click_button("anmelden")
  visit users_path
end

Dann(/^sollen in ihr alle Kaffeetrinkenden angezeigt werden\.$/) do
  anzahl_in_liste = find_all("tbody tr").size
  anzahl_in_db = User.where(drinking_coffee: true).size
  expect(anzahl_in_liste).to equal anzahl_in_db
end

Wenn(/^ich als Business Analyst die Userliste öffne,$/) do
  visit login_path
  fill_in("login", with: @ba.login)
  fill_in("password", with: @ba.password)
  click_button("anmelden")
  visit users_path
end

Dann(/^sollen in ihr alle Mitarbeiter angezeigt werden\.$/) do
  anzahl_in_liste = find_all("tbody tr").size
  anzahl_in_db = User.all.size
  expect(anzahl_in_liste).to equal anzahl_in_db
end
```

Abb. 2: Vom Computer lesbarer Code

3 Hier wird ein weiteres Framework namens *RSpec* eingesetzt, um die vom Browser zurückgegebenen Werte mit den erwarteten Werten zu vergleichen.

Die wesentlichen Inhalte werden selbst Programmierlaien in diesem Code noch deutlich.

Wenn man diesen Test auf einem entsprechend ausgestatteten PC ausführt, startet dieser automatisch einen Browser, wie z. B. Firefox, öffnet die entsprechende Seite, führt das jeweilige Login durch, ruft dann die Userliste auf und vergleicht sie mit dem Inhalt der Datenbank.

Anschließend wird ein Protokoll des Tests ausgegeben. Im Beispiel ,,Kaffeekasse“ hätte der Test bei der ersten Softwareversion, in der immer alle Mitarbeiter in der Liste angezeigt werden, Folgendes gezeigt (Abb. 3).

Das Protokoll zeigt genau, um welche Funktionalität es bei dem Test ging, gibt dann die Grundlagen an und geht die Szenarien Schritt für Schritt durch. Darunter gibt es eine Zusammenfassung, nämlich dass Szenarien (in diesem Fall eines) fehlgeschlagen sind und welche. Abschließend wird in einer Statistik angezeigt, wie viele Szenarien getestet wurden, wie viele Testschritte durchgeführt wurden und wie lange es gedauert hat: 0,5 Sekunden. Ein Mensch hätte allein zum Öffnen des Browsers länger gebraucht.

Im Szenario ,,Liste aller Kaffeetrinkenden“ konnte der Kassenwart noch die Userliste öffnen (in grün gesetzt), der Inhalt entsprach aber nicht den Erwartungen. In rot wird neben detaillierten Fehlerangaben gezeigt, dass in der Liste fünf Benutzer erwartet, aber 50 angezeigt wurden: expected => 5, got => 50

Das zweite Szenario, die ,,Liste aller Mitarbeiter“ funktioniert wie gewünscht.

Wenn beide Features korrekt umgesetzt sind, wird der Test fehlerfrei durchlaufen, und das Protokoll sieht dann aus wie in Abb. 4.

```
Funktionalität: Userlisten
   Als Kassenwart möchte ich eine Liste haben, in der alle an der Kaffekasse Beteiligten aufgelistet sind, so dass ich
 sie an die Einzahlungen in die Kaffekasse erinnern kann.
   Als Business Analyst möchte ich eine Liste aller Mitarbeitenden haben, um eine Übersicht über alle Abteilungen zu b
ekommen.

  Grundlage:                                          # features/userlisten.feature:7
    Gegeben sei , dass in der DB "50" Mitarbeiter angelegt sind # features/step_definitions/userlisten_step.rb:3
    Und dass davon "5" an der Kaffeekasse beteiligt sind.      # features/step_definitions/userlisten_step.rb:9

  Szenario: Liste aller Kaffeetrinkenden              # features/userlisten.feature:11
    Wenn ich als Kassenwart die Userliste öffne,      # features/step_definitions/userlisten_step.rb:23
    Dann sollen in ihr alle Kaffeetrinkenden angezeigt werden. # features/step_definitions/userlisten_step.rb:32

      expected #<Fixnum:11> => 5
           got #<Fixnum:101> => 50

      Compared using equal?, which compares object identity,
      but expected and actual are not the same object. Use
      `expect(actual).to eq(expected)` if you don't care about
      object identity in this example.

       (RSpec::Expectations::ExpectationNotMetError)
      ./features/step_definitions/userlisten_step.rb:35:in `/^sollen in ihr alle Kaffeetrinkenden angezeigt werden\.$/'
      features/userlisten.feature:13:in `Dann sollen in ihr alle Kaffeetrinkenden angezeigt werden.'

  Szenario: Liste aller Mitarbeiter                   # features/userlisten.feature:15
    Wenn ich als Business Analyst die Userliste öffne,    # features/step_definitions/userlisten_step.rb:27
    Dann sollen in ihr alle Mitarbeiter angezeigt werden. # features/step_definitions/userlisten_step.rb:38

Failing Scenarios:
cucumber features/userlisten.feature:11 # Szenario: Liste aller Kaffeetrinkenden

2 scenarios (1 failed, 1 passed)
8 steps (1 failed, 7 passed)
0m0.459s
```

Abb. 3: Der Test schlägt fehl

```
Funktionalität: Userlisten
   Als Kassenwart möchte ich eine Liste haben, in der alle an der Kaffekasse Beteiligten aufgelistet sind, so dass ich
 sie an die Einzahlungen in die Kaffekasse erinnern kann.
   Als Business Analyst möchte ich eine Liste aller Mitarbeitenden haben, um eine Übersicht über alle Abteilungen zu b
ekommen.

  Grundlage:                                          # features/userlisten.feature:7
    Gegeben sei , dass in der DB "50" Mitarbeiter angelegt sind # features/step_definitions/userlisten_step.rb:3
    Und dass davon "5" an der Kaffeekasse beteiligt sind.       # features/step_definitions/userlisten_step.rb:9

  Szenario: Liste aller Kaffeetrinkenden                      # features/userlisten.feature:11
    Wenn ich als Kassenwart die Userliste öffne,              # features/step_definitions/userlisten_step.rb:23
    Dann sollen in ihr alle Kaffeetrinkenden angezeigt werden. # features/step_definitions/userlisten_step.rb:32

  Szenario: Liste aller Mitarbeiter                           # features/userlisten.feature:15
    Wenn ich als Business Analyst die Userliste öffne,        # features/step_definitions/userlisten_step.rb:27
    Dann sollen in ihr alle Mitarbeiter angezeigt werden.     # features/step_definitions/userlisten_step.rb:38

2 scenarios (2 passed)
8 steps (8 passed)
0m0.433s
```

Abb. 4: Der Test läuft fehlerfrei durch

Ein Test lässt sich natürlich jederzeit durch weitere Szenarien ergänzen. So könnte z. B. im Laufe der Zeit der Wunsch auftauchen, dass die Liste alphabetisch sortiert sein soll. Das Szenario dazu würde so lauten:

```
Szenario: alphabetisch nach Nachname sortiert
  Wenn ich als Kassenwart die Liste der Kaffeetrinkenden öffne,
  Dann sollen die Kaffeetrinkenden alphabetisch nach Nachnamen sortiert
      angezeigt werden.
```

Abb. 5: Ein weiteres Szenario

Genauso wäre denkbar, dass der Kassenwart die Liste nach Vornamen sortiert haben möchte, die Business Analystin aber nach Nachnamen. Das alles lässt sich auf diese Weise klar formulieren und anschließend maschinell überprüfen. Und weil die Anforderungen eindeutig formuliert sind, können sie bei der Programmierung gleich richtig umgesetzt werden. Das spart Zeit und erhöht die Zufriedenheit aller Beteiligten.

In unserem Beispiel wurde erst **nach** der Programmierung deutlich, dass das Feature „Userliste“ eigentlich aus zwei Stories mit unterschiedlichen Anforderungen besteht. Mit standardisierten Formulierungen wäre das schon bei der Bestellung aufgefallen.

Fazit

Der Mehraufwand für standardisierte Formulierungen, der zur Vorbereitung von automatisiertem Testen notwendig ist, bietet langfristig Vorteile:

- Die gemeinsame Formulierung von Anforderungen führt zur Entwicklung einer gemeinsamen Sprache und zu einem gemeinsamen Verständnis der jeweiligen Features.
- Missverständnisse werden schon bei der Formulierung der Anforderungen erkannt.
- Die so formulierten Features dienen gleichzeitig als Dokumentation der Anforderungen und als Grundlage für automatisierte Tests.
- Ein automatisierter Test verhält sich immer gleich. So wird sich beispielsweise der PC nie versehentlich als Kassenwart einloggen, wenn er die Funktionen des Business Analysten testen möchte.
- Für einen Test muss kein Zeitfenster mehr für eine Gruppe von Testern koordiniert werden. Der PC macht keinen Urlaub.
- Bei Bedarf kann der Test beliebig oft wiederholt werden, ohne die Nerven von Mitarbeitern zu strapazieren.
- Und schließlich läuft der Test auf dem PC im Vergleich zur manuellen Ausführung auch noch etwas schneller ab. Ab einer bestimmten Menge von Wiederholungen der Tests hat man die Zeit, die für die Formulierungen benötigt wurde, wieder hereingeholt.

Bei nur einem Feature ist der Gewinn durch die Automatisierung von Tests natürlich gering. Sobald man sich das Ganze aber mit hunderten von Features, entsprechend vielen Szenarien und Testschritten und über die Laufzeit der Software durchrechnet, wird der Wert der Testautomatisierung erkennbar.

Autoreninformationen

Sven Koesling
Leitung Bibliotheks-IT-Services
ETH Zürich
ETH-Bibliothek Rämistrasse 101
8092 Zürich
Schweiz
sven.koesling@library.ethz.ch
orcid.org/0000-0002-0973-5340