Immer das Richtige

Agile Acceptance Testing mit BDD – Teil 1: Grundlagen

Carl Anders Düvel, Roland Jülich

Wie stellt man sicher, dass man die richtige Software entwickelt? Eine schwierige Aufgabe, die seit jeher die Softwareentwicklung begleitet. Wie versteht man den Kunden und wie macht dieser sich am Besten verständlich? Wie kommuniziert man die Anforderungen effizient mit allen an der Entwicklung und am Testen beteiligten Kollegen? Ach ja, dokumentiert werden muss die Software auch – sowie auf Akzeptanzkriterien getestet! Viele Anforderungen, bei deren Bewältigung „Behavior Driven Development" (BDD) mit Cucumber helfen kann.

Es war einmal ...

Wir möchten Ihnen in diesem zweiteiligen Artikel über Agile Acceptance Testing mit BDD - Teil 1: Grundlagenweise, nämlich Behavior Driven Development, und zum anderen ein Framework für automatisierte Akzeptanztests, nämlich Cucumber, vorstellen. Wir beginnen dafür mit einer Geschichte*:

Dora Developer steht am Taskboard und blickt auf die offenen User Stories (etwa Anwendererzählungen) für den aktuellen Sprint, das aktuelle Entwicklungsintervall. Erst vor wenigen Tagen war sie zusammen mit ihrem Team beim Kunden, einer mittelständischen Firma, die ein System zur Erfassung und Analyse von Mitarbeiterbewertungen entwickeln lässt. In diesem Sprint geht es vor allem um die Realisierung wichtiger Features für die Analyse der Bewertungen. Auf der nächsten Storycard steht: „Als Vorgesetzter will ich die Bewertung meiner Mitarbeiter ansehen können." Daneben hängen einige Zettel mit den für diese Story zu erledigenden Tasks: Erstellen eines entsprechenden Controllers, ein Entwurf der entsprechenden Seite und andere Aufgaben.

Dora setzt sich an ihren Rechner und beginnt damit, die Story umzusetzen. Zunächst einmal muss sie die für einen Benutzer einsehbaren Bewertungen ermitteln. Dabei kommt ihr entgegen, dass in dem System bereits eine Repräsentation des Organigramms der Firma als gerichteter Graph existiert. Dora schaut sich den entsprechenden Code kurz an und implementiert dann – testgetrieben – eine Methode in der Klasse `Mitarbeiter`, die alle direkt untergebenen Mitarbeiter zurückgibt. Die Funktionalität, alle Berichte für einen Mitarbeiter zu ermitteln, existiert schon. Also kann Dora beginnen, die Oberfläche zu erstellen, auf der die Mitarbeiter und die Bewertungen in einer Baumstruktur angezeigt werden sollen.

Wir springen einige Tage in die Zukunft und treffen Tom Tester. Tom steht müde an der Kaffeemaschine und macht sich gähnend einen doppelten Espresso. Der Grund für seinen Zustand ist einfach: Überstunden – genauer gesagt: zu viele davon. Die Entwickler haben sich mal wieder verspätet, aber der Termin für den Produktivgang kann nicht verschoben werden. Mit anderen Worten: Sein Chef hat ihm gestern klargemacht, dass er nur die Hälfte der geplanten Zeit zur Verfügung hat, um dem System auf den Zahn zu fühlen. Dabei kennt er das neue Bewertungssystem nicht und die Testfälle, durch die er sich arbeiten muss, sind ihm ebenso wenig geläufig.

„Wahrscheinlich hat jemand ganz oben den Livegang des Systems noch in diesem Jahr in den Zielvorgaben stehen ...", denkt er sich schlecht gelaunt und setzt sich vor seinen Rechner. Als Nächstes steht das von Dora entwickelte Feature zur Anzeige der Berichte an. Tom hetzt sich durch die Dialoge und testet zwei einfache Fälle: einen Mitarbeiter ohne Führungsverantwortung und einen Kollegen auf mittlerer Führungsebene. Tom ist zufrieden: Im ersten Fall werden nur die eigenen Bewertungen angezeigt und im zweiten Fall sind alle untergebenen Mitarbeiter und deren Berichte sichtbar. Eigentlich würde Tom gern noch den einen oder anderen Testfall durchgehen. Aber er hat keine Zeit, also setzt er das Feature im Projekt-Tracking-Tool auf „erfolgreich getestet" und wendet sich dem nächsten Testfall zu.

Einige Tage später wird das System ausgeliefert. Aber nach wenigen Stunden ruft der Kunde völlig außer sich bei Doras Vorgesetztem an und verlangt die sofortige Abschaltung der neuen Software: „Die Anwendung kann auf keinen Fall so live bleiben: Teamleiter können die Bewertungen ihrer Mitarbeiter einsehen! Wenn der Betriebsrat davon Wind bekommt, gibt das einen Riesenkrach!"

Wer ein paar Projekte hinter sich hat, hat eine ähnliche Situation bestimmt schon einmal erlebt. Ein fachliches Missverständnis schafft es bis in die produktive Software und hinterlässt dort einen mehr oder minder großen Schaden. Mit Sicherheit jedoch einen Mehraufwand beim Aufarbeiten und ein unangenehmes Gefühl bei allen Beteiligten.

Und noch einmal von vorn!

Lassen Sie uns diese Episode noch einmal Revue passieren. Diesmal läuft es jedoch ein wenig anders: Kunde, Entwickler und Tester haben ein gemeinsames Meeting. Dora hat ihren Rechner dabei und die Stories für den aktuellen Sprint vor sich liegen. Sie begrüßt ihre Kollegen: „Schön, dass Ihr Euch Zeit genommen habt. Ich möchte mal etwas Neues ausprobieren

* Diese Geschichte ist allerdings nicht rein fiktiv, sondern basiert auf einer Projekterfahrung, die den meisten Lesern bekannt vorkommen dürfte.

und brauche dafür Eure Hilfe.“ Sie erklärt ihrem Kunden und Tom, dass sie die Stories mit Beispielen unterfüttern möchte, aus denen sie dann einen automatisierten Akzeptanztest bauen wird. Dafür hat sie einen einfachen Texteditor geöffnet und schon die Zeilen in Listing 1 geschrieben.

```
# language: de
Funktionalität: Anzeige von Mitarbeiterbewertungen
  Um sich über die Bewertungen von Mitarbeitern zu informieren,
  müssen die Berichte anzeigbar sein.
```

Listing 1: Die ersten Zeilen der Feature-Datei

„Hm“, meint der Kunde, „Das sieht aus wie eine der Stories für diesen Sprint.“ „Ja, richtig“, bestätigt Dora, „Aber dieses Dokument hier soll länger existieren und später als lebendige Dokumentation des Systems dienen.“ „Meinetwegen, aber was sollen wir denn jetzt tun?“ „Ich möchte mit Euch ein paar konkrete Beispiele formulieren. Das wird dann ein automatisierter Test, der prüft, ob das System sich für diese konkreten Beispiele auch genauso verhält, wie wir das erwarten. Mit anderen Worten: eine ausführbare Spezifikation!“ Tom und der Kunde scheinen skeptisch und Dora denkt sich, statt viele Worte zu verlieren, sollte sie am besten einmal konkret zeigen, was sie meint. Also schreibt sie einfach drauf los – mit dem Ergebnis von Listing 2.

```
Szenario: Als Vorstandsvorsitzender kann ich alle Bewertungen sehen.
  Angenommen ich bin Vorstandsvorsitzender
  Wenn ich die Ansicht für Berichte öffne
  Dann kann ich die Bewertungen aller Mitarbeiter einsehen
```

Listing 2: Das Szenario für den Vorstandsvorsitzenden

„Ah. Das haben Sie gemeint!“, meint der Kunde, „Na, das ist einfach. Da habe ich dann auch ein weiteres Beispiel: Als einfacher Mitarbeiter kann ich keine Bewertungen einsehen.“ Dora konkretisiert das Beispiel und schreibt das Listing 3.

```
Szenario: Als Sachbearbeiter kann ich nur meine eigenen Bewertungen sehen
  Angenommen ich bin Sachbearbeiter
  Wenn ich die Ansicht für Berichte öffne
  Dann kann ich nur meine eigenen Bewertungen einsehen
```

Listing 3: Das Szenario für den Sachbearbeiter

Während der Kunde mittlerweile überzeugt ist, scheint Tom noch skeptischer geworden zu sein: „Das ist ja schön und gut, aber was habe ich damit zu tun?“ „Du kannst uns helfen, weitere Testfälle zu finden. Schließlich haben wir jetzt nur zwei sehr eindeutige Beispiele. Als Tester bist Du schließlich sehr gut darin, Konstellationen zu finden, an die während der Entwicklung nicht gedacht wurde und die deshalb zu Fehlern führen.“ „Ach so, na ja, dann versuche ich das einmal.“ Tom schnappt sich die Tastatur und tippt das Listing 4.

```
Szenario: Als Teamleiter kann ich die Bewertungen aller
    Teammitglieder einsehen.
  Angenommen ich bin Teamleiter
  Wenn ich die Ansicht für Berichte öffne
  Dann kann ich die Bewertungen aller Teammitglieder sehen
```

Listing 4: Das Szenario für die Rolle Teamleiter

„Nein, auf keinen Fall!“, platzt der Kunde heraus. Dora und Tom sehen ihn fragend an. „Teamleiter sind ja keine disziplinarischen Vorgesetzten – daher dürfen sie auf keinen Fall die Bewertungen ihrer Teamkollegen sehen!“ Nach kurzer Verwunderung verstehen die beiden, was der Kunde meint und Dora sagt: „Das ist ein sehr wichtiger Punkt. Leider wird er durch die Beschreibung des Features so nicht ganz klar. Super, dass wir das jetzt klären können.“ Die drei einigen sich auf eine neue Version, die sich in Listing 5 wiederfindet.

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

  Szenario: Als Vorstandsvorsitzender kann ich alle Bewertungen sehen.
    Angenommen ich bin Vorstandsvorsitzender
    Wenn ich die Ansicht für Berichte öffne
    Dann kann ich die Bewertungen aller Mitarbeiter einsehen

  Szenario: Als Sachbearbeiter kann ich nur meine
      eigenen Bewertungen sehen.
    Angenommen ich bin Sachbearbeiter
    Wenn ich die Ansicht für Berichte öffne
    Dann kann ich nur meine eigenen Bewertungen einsehen

  Szenario: Als Teamleiter kann ich nur meine eigenen Bewertungen sehen.
    Angenommen ich bin Teamleiter
    Wenn ich die Ansicht für Berichte öffne
    Dann kann ich nur meine eigenen Bewertungen einsehen
```

Listing 5: Die neue Feature-Datei

Im Verlauf des Meetings haben die drei noch einige andere Szenarien gefunden und auch weitere Features für den Sprint beschrieben. Diese Dateien sind für Dora der Ausgangspunkt für die Implementierung von Akzeptanztests. Vor der Umsetzung einer Story wird zunächst der entsprechende Test geschrieben. Im Laufe des Sprints zeigen diese Tests den Fortschritt der Implementierung an – immer mehr Schritte in den Szenarios können erfolgreich getestet werden. Hin und wieder merken die Entwickler, dass sie bestehende Funktionalität durch eine Änderung beeinträchtigt haben, aber dank des schnellen Feedbacks durch die Tests sind diese Fehler schnell erkannt.

Auch in der anschließenden Testphase ist das Vorgehen von Vorteil: Tom muss nun nicht die Testfälle durchklicken – das erledigt der automatisierte Akzeptanztest schneller und besser. Stattdessen hat er Zeit, explorativ zu testen, die Ergonomie zu prüfen und auch auf Dinge wie mögliche Verstöße gegen den Style Guide zu achten – also qualitätssichernde Tätigkeiten, die sich gar nicht oder nur mit sehr hohem Aufwand automatisieren lassen. Es kommt auch in dieser Phase zu wesentlich weniger „false positives“ bei den Fehlermeldungen: Grund dafür ist, dass Tom, Dora und der Kunde am Anfang des Sprints ein gemeinsames Verständnis entwickelt haben: Fehlermeldungen aufgrund verschiedener Meinungen über das richtige Verhalten der Software sind dadurch seltener geworden.

Behavior Driven Development (BDD)

Wir sind nun Zeuge zweier verschiedener Szenarios geworden. Im ersten Fall kommt es zu einem fachlichen Missverständnis, das sich am Ende in einer Software manifestiert, die vielleicht korrekt funktioniert, aber nicht die Erwartungen des Kunden erfüllt. Im zweiten Fall besteht dieses fachliche Missverständnis ebenfalls, wird aber wesentlich früher aufgedeckt, was für alle Beteiligten einen Vorteil darstellt.

Ein Grund dafür ist, dass sich alle am Entwicklungsprozess beteiligten Personen am Anfang an einen Tisch setzen. Sie be-

dienen sich dabei nicht der sonst üblichen abstrakten und damit oft unscharfen Begriffe, sondern arbeiten mit Beispielen. Diese werden in einer Form erfasst, die allen Beteiligten verständlich ist. Trotzdem ermöglichen die Beispiele automatische Akzeptanztests, die vor der Implementierung der Funktionalität erstellt werden. Diese Methode ist mittlerweile unter dem Begriff „Specification by Example“ bekannt.

Fachliche Missverständnisse aufzudecken, ist dabei nur ein Vorteil. Die Vorgehensweise ist auch dafür geeignet, vage Ideen des Auftraggebers für neue Funktionalitäten zu konkretisieren. Die Grundlage dieser Beispiele sind bei einem agilen Vorgehen die Akzeptanzkriterien einer User Story. Da diese Beispiele gleichzeitig Testfälle darstellen, ist es nur naheliegend, ihre Prüfung zu automatisieren. Dabei kommt Behavior Driven Development (BDD) ins Spiel, indem es die textuellen Beispiele direkt mit ausführbaren Tests verknüpft. BDD als agile Entwicklungsmethode baut auf dem Konzept von Test Driven Development (TDD) auf. Es ergänzt den TDD-Cycle um eine äußere Schleife. Dabei handelt es sich auch um eine moderne Form von Acceptance-TDD, denn die äußere Schleife ist genau dann durchlaufen, wenn die Akzeptanzkriterien erfüllt sind.

Abb. 1: (A)TDD-Cycle

Abbildung 1 verdeutlicht dies. Zunächst wird der Akzeptanztest vollendet. Dann wird so lange testgetrieben entwickelt, bis der Test erfolgreich verläuft. Beiden Schleifen gleich ist die strikte „Test First“-Regel. Allerdings gibt es auch Unterschiede: Das zweite TDD-Gesetz – kein Test wird weiter entwickelt, als nötig ist, um einen Fehler zu erhalten – gilt für BDD nicht. Es ist durchaus sinnvoll, den gesamten Test für ein Feature fertigzustellen, bevor man sich an die Umsetzung macht.

BDD-Frameworks

Ein BDD-Framework muss vor allem zwei Kriterien erfüllen:

- Die Beispiele müssen in einer Sprache verfasst sein, die auf der einen Seite von allen Beteiligten verstanden wird, aber auf der anderen Seite auch formalisiert genug ist, dass sie als Grundlage für einen automatischen Akzeptanztest dienen kann.
- Das Implementieren der Testlogik sollte so einfach wie möglich sein, um die Aufwände für die Erstellung der Akzeptanztests gering zu halten.

Es existieren mittlerweile eine ganze Reihe von BDD-Frameworks. Zu den prominentesten Vertretern im Java-Umfeld zählen Cucumber, JBehave und Concordion. Aus unserer Sicht erfüllt Cucumber die oben genannten Kriterien besonders gut und besticht durch seine leichte Integrierbarkeit. Daher werden wir im zweiten Teil darlegen, wie unser Beispiel mit Cucumber im Umfeld von Webanwendungen realisiert werden kann.

Vorteile von BDD

Der Vorteil beim Einsatz von Behavior Driven Development liegt ganz klar in der Fokussierung auf Akzeptanzkriterien und damit den funktionalen Anforderungen einer Software. Dies legt schließlich bereits der Name nahe: Bei BDD dreht sich alles um das Verhalten von Software.

Durch die Anwendung des Angenommen-Wenn-Dann-Musters zur Beschreibung der Feature-relevanten Szenarios wird das antizipierte Verhalten einer Anwendung konkret und klar definiert. Je mehr relevante, sinnvoll voneinander abgrenzbare Beispiele sich finden lassen, desto genauer wird die Spezifikation.

Bezogen auf den Entwicklungsprozess forciert der Einsatz von BDD neben der Beteiligung von Domänenspezialisten auch die frühzeitige Einbindung von Testern. Der große Vorteil dabei liegt im frühen Aufdecken fachlicher Inkonsistenzen oder Missverständnisse, die sonst auf klassischem Weg von der Spezifikation über die Implementierung zum Test gelangen würden. Im günstigsten Fall werden sie dort noch erkannt. Nicht selten wird die Fehlerbehebung dann sehr aufwendig und damit auch kostspielig. Abgesehen davon werden Entwickler oft auf die Jagd nach vermeintlichen Bugs geschickt, die tatsächlich gar keine sind, sondern nur von verschiedenen Erwartungen an das Systemverhalten zeugen.

Tester werden aber auch bei Einsatz von BDD keineswegs überflüssig. Gerade sie ersinnen meist wichtige Szenarios abseits des „Happy Paths“, die von Domänenexperten und Entwicklern oft übersehen werden. Letztlich ist leider auch nicht jedes Feature mit sinnvollem Aufwand automatisiert testbar, von nichtfunktionalen Anforderungen wie Ergonomie ganz zu schweigen.

Dieses akzeptanztestgetriebene Vorgehen scheint im Vergleich zu „konventionellen“ Entwicklungspraktiken zunächst einen erhöhten Aufwand zu bedeuten. Spätestens seit Kent Becks „extreme Programming explained“ [Bec00] sollten jedoch automatisierte Akzeptanztests zur Routine in der Softwareentwicklung gehören. BDD forciert dies und schafft mit der Formulierung der Tests in natürlichsprachlicher Form die Basis einer „Ubiquitous Language“ [Eva04]. Weil diese Sprache sowohl in der fachlichen als auch in der technischen Domäne eindeutig ist, trägt sie erheblich zu einer effizienteren Kommunikation bei. Unserer Erfahrung nach lohnt es sich besonders, diese Verfahren einzusetzen, wenn:

- das zu entwickelnde System eine mittel- bis langfristige Lebenserwartung hat,
- sich von fachlicher Seite eine formale, abgrenzbare Beschreibung von Features als schwierig erweist,
- die Entwicklung von Features stets eine hohe Defektrate aufweist, die in der Regel nicht auf technische Fehler zurückzuführen ist,
- es wichtige Features gibt, die häufigen Änderungen unterliegen und an deren Umsetzung viele verschiedene Personen (Spezifikation, Implementierung, Test ...) beteiligt sind.

Für den Einsatz von BDD spricht die hohe Qualität der entstehenden Software, die sich in hoher Akzeptanz seitens der Anwender und deutlich geringeren Wartungskosten manifestiert. Darüber hinaus fördert BDD die Transparenz über alle Phasen und für die Beteiligten des Entwicklungsprozesses. Dies ge-

schieht vor allem durch den kontinuierlichen Auf- und Ausbau einer jederzeit „ausführbaren Spezifikation“. Da deren Ergebnisse aufgrund ihrer einheitlichen, natürlichsprachlichen Form intuitiv verständlich sind, steigt das Vertrauen in die Korrektheit des Produkts. Die Qualitätssicherung steht somit nicht mehr am Ende, sondern wird zum elementaren Bestandteil des gesamten Entwicklungsprozesses.

Literatur und Links

[Bec00] K. Beck, Extreme Programming explained, Addison-Wesley, 2000
[Eva04] E. Evans, Domain-Driven Design, Tackling Complexity in the Heart of Software, 2004

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

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