Testen in agilen Projekten

Methoden und Techniken für Softwarequalität in der agilen Welt

1 Einleitung

Software ist allgegenwärtig. Nahezu jedes komplexere Produkt ist heute softwaregesteuert und auch viele Dienstleistungen stützen sich auf Softwaresysteme. Software und Softwarequalität sind daher ein entscheidender Wettbewerbsfaktor. Ein Unternehmen, das neue oder bessere Software in kürzerer Zeit in sein Produkt integrieren bzw. auf den Markt bringen kann (Time-to-Market), ist seinen Mitbewerbern überlegen.

Agile Entwicklungsmodelle versprechen eine schnellere »Time-to-Market« bei gleichzeitig besserer Ausrichtung an den Kundenanforderungen und nicht zuletzt bessere Softwarequalität. So ist es nicht verwunderlich, dass in immer mehr Unternehmen agile Methoden eingesetzt werden – auch in großen, internationalen Projekten und in Produktentwicklungseinheiten großer Konzerne, quer durch alle Branchen. In den meisten Fällen bedeutete oder bedeutet dies den Umstieg von einer bisher praktizierten Entwicklung nach V-Modell auf eine agile Entwicklung nach Scrum[1].

Weder die initiale Umstellung auf »agil« noch das dann notwendige nachhaltige agile Arbeiten sind jedoch einfach, insbesondere dann nicht, wenn mehr als nur ein Team davon betroffen ist. Jedes Teammitglied, das Projektmanagement, aber auch das Management in der Linienorganisation muss teils gravierende Änderungen gewohnter Abläufe und Arbeitsweisen vollziehen. Dabei sind Softwaretest und Softwarequalitätssicherung ganz entscheidend daran beteiligt, ob ein Team, eine Softwareabteilung oder ein ganzes Unternehmen die agile Entwicklung langfristig erfolgreich beherrscht und damit die erhofften Vorteile nachhaltig realisieren kann.

Zu den populären agilen Entwicklungsmethoden gibt es eine Fülle auch deutschsprachiger Literatur. Einige empfehlenswerte Einführungen, z. B. zu Scrum, finden sich im Literaturverzeichnis dieses Buches.

1. Umfragen [URL: Testpraxis], [URL: TrendsInTestingAgile] und Studie [URL: StatusQuoAgile].

In der Regel wird das Thema »agile Softwareentwicklung« in diesen Büchern aus der Sicht des Entwicklers und Programmierers betrachtet. Demgemäß stehen agile Programmiertechniken und agiles Projektmanagement im Vordergrund. Wenn das Thema Testen erwähnt wird, geht es meistens um Unit Test und zugehörige Unit-Test-Werkzeuge, also im Wesentlichen um den Entwicklertest. Tatsächlich kommt dem Testen in der agilen Entwicklung aber eine sehr große und erfolgskritische Bedeutung zu und Unit Tests alleine sind nicht ausreichend.

Dieses Buch möchte diese Lücke schließen, indem es agile Softwareentwicklung aus der Perspektive des Testens und des Softwarequalitätsmanagements betrachtet und aufzeigt, wie »agiles Testen« funktioniert, wo »traditionelle« Testtechniken auch im agilen Umfeld weiterhin benötigt werden und wie diese in das agile Vorgehen eingebettet werden.

1.1 Zielgruppen

Verstehen, wie Testen in agilen Projekten funktioniert

Das Buch richtet sich zum einen an Personen, die in das Thema agile Entwicklung erst einsteigen, weil sie künftig in einem agilen Projekt arbeiten werden oder weil sie Scrum oder agile Vorgehensweisen in ihrem Projekt oder Team einführen wollen oder gerade eingeführt haben:

- Entwicklungsleiter, Projektmanager, Testmanager und Qualitätsmanager erhalten Hinweise und Tipps, wie Qualitätssicherung und Testen ihren Beitrag dazu leisten können, das Potenzial agiler Vorgehensweisen voll zu entfalten.
- Professionelle (Certified) Tester und Experten für Softwarequalität erfahren, wie sie in agilen Teams erfolgreich mitarbeiten und ihre spezielle Expertise optimal einbringen können. Sie lernen auch, wo sie ihre aus klassischen Projekten gewohnte Arbeitsweise umstellen oder anpassen müssen.

Wissen über (automatisiertes) Testen und agiles Qualitätsmanagement erweitern

Ebenso angesprochen werden aber auch Personen, die bereits in agilen Teams arbeiten und eigene »agile« Erfahrungen sammeln konnten und die ihr Wissen über Testen und Qualitätssicherung erweitern wollen, um die Produktivität und Entwicklungsqualität in ihrem Team weiter zu erhöhen:

- Product Owner, Scrum Master, Qualitätsverantwortliche und Mitarbeiter mit Führungsverantwortung erfahren in kompakter Form, wie systematisches, hoch automatisiertes Testen funktioniert und welchen Beitrag Softwaretester in agilen Teams leisten können, um kontinuierlich, zuverlässig und umfassend Feedback über die Qualität der entwickelten Software zu liefern.

- Programmierer, Tester und andere Mitglieder eines agilen Teams erfahren, wie sie hoch automatisiertes Testen realisieren können, und zwar nicht nur im Unit Test, sondern auch im Integrations- und im Systemtest.

Das Buch enthält viele praxisorientierte Beispiele und Übungsfragen, sodass es auch als Lehrbuch und zum Selbststudium geeignet ist.

1.2 Zum Inhalt

Kapitel 2

Kapitel 2 gibt eine knappe Charakteristik des agilen Projektmanagement-Frameworks Scrum und der aus dem Lean Product Development stammenden und zu Scrum einige Ähnlichkeiten aufweisenden Projektmanagementmethode Kanban. Dabei werden auch die Bezüge zu Extreme Programming (XP), aus dem wichtige agile Entwicklungstechniken stammen, erläutert. Diesen agilen Vorgehensweisen wird das Vorgehen in Projekten, die sich an klassischen Vorgehensmodellen orientieren, gegenübergestellt. Personen, die ihr Projekt oder ihre Unternehmenseinheit auf eine agile Vorgehensweise umstellen oder agiler ausrichten wollen, erhalten hier einen Überblick und einen Eindruck von den organisatorischen Veränderungen, die mit der Einführung agiler Ansätze im Unternehmen, der betroffenen Abteilung und den betroffenen Teams einhergehen.

Kapitel 3

Kapitel 3 zeigt auf, welche leichtgewichtigen Techniken und Instrumente zur Planung und Steuerung der Entwicklungsarbeiten zum Einsatz kommen und wie Produkt- bzw. Kundenanforderungen »agil« ermittelt, überprüft und dokumentiert werden. Denn »agil« zu arbeiten bedeutet keineswegs »planlos« zu arbeiten. Das Kapitel richtet sich an Personen, die neu in das Thema »agile Entwicklung« einsteigen. Die Erläuterungen und Hinweise, welchen Beitrag die jeweiligen Instrumente zur Fehlervermeidung beitragen, sind jedoch auch für die Zielgruppe mit agiler Projekterfahrung wertvoll.

Kapitel 4

Kapitel 4 behandelt das Thema Unit Tests und »Test First«. Es erklärt, was Unit Tests leisten und wie Unit Tests automatisiert werden. Systemtester, Fachtester oder Projektbeteiligte ohne oder mit wenig Erfahrung im Unit Test finden hier Grundlagen zu Techniken und Werkzeugen im entwicklungsnahen Test, die ihnen helfen, enger mit Programmierern und Unit-Testern zusammenzuarbeiten. Programmierer und Tester mit Erfahrung im Unit Test erhalten hilfreiche Tipps, um ihre Unit Tests zu verbessern. Ausgehend von diesen Grundlagen wird Test First (testgetriebene Entwicklung) vorgestellt und die hohe Bedeutung dieser Praktik für agile Projekte erläutert.

Kapitel 5 Kapitel 5 erklärt Integrationstests und »Continuous Integration«. Auch Programmierer, die ihren Code intensiv mit Unit Tests prüfen, vernachlässigen dabei oft Testfälle, die Integrationsaspekte überprüfen. Daher werden in diesem Kapitel zunächst wichtige Grundlagen zur Softwareintegration und zu Integrationstests vermittelt. Anschließend wird die Continuous-Integration-Technik vorgestellt und erläutert, wie ein Continuous-Integration-Prozess im Projekt eingeführt und angewendet wird.

Kapitel 6 Kapitel 6 befasst sich mit Systemtests und »Continuous Testing«. Aufbauend auf den Grundlagen zu Systemtests werden wichtige Techniken für manuelle und automatisierte System- und Akzeptanztests im agilen Umfeld erläutert. Anschließend wird gezeigt, wie auch Systemtests effizient automatisiert und in den Continuous-Integration-Prozess des Teams eingebunden werden können. Kapitel 6 richtet sich dabei nicht nur an Systemtester und Fachtester, sondern auch an Programmierer, die besser verstehen wollen, welche Testaufgaben jenseits des entwicklungsnahen Tests im agilen Team gemeinsam zu bewältigen sind.

Kapitel 7 Kapitel 7 stellt klassisches und agiles Verständnis von Qualitätsmanagement und Qualitätssicherung gegenüber und erläutert die in Scrum »eingebauten« Praktiken zur vorbeugenden, konstruktiven Qualitätssicherung. Die Leserinnen und Leser erhalten Hinweise und Tipps, wie Qualitätsmanagement »agiler« realisiert werden kann und wie QS- und Testexperten ihr Know-how in agile Projekte einbringen und so einen wertvollen Beitrag für ein agiles Team leisten können. Auch wie sich Agile Scaling und DevOps auf das Qualitätsmanagement auswirken, wird diskutiert.

Kapitel 8 In Kapitel 8 werden mehrere Fallstudien aus Industrie, Onlinehandel und Unternehmen der Softwarebranche vorgestellt. Diese spiegeln die Erfahrungen und Lessons Learned wider, die die Interviewpartner bei der Einführung und Anwendung agiler Vorgehensweisen in ihren jeweiligen Unternehmen gesammelt haben.

Kapitelstruktur Die Kapitel 2, 3, 7 und 8 behandeln Prozess- und Managementthemen und richten sich daher an Personen, die an Managementaspekten interessiert sind. Die Kapitel 4, 5 und 6 erörtern das (automatisierte) »agile Testen« auf den verschiedenen Teststufen und sprechen (auch) technisch interessierte Personen an. Dabei wird ausführlich auf die Ziele und Unterschiede von Unit Tests, Integrations- und Systemtests eingegangen. Denn wie bereits erwähnt, wird Testen leider in vielen agilen Projekten oft mit Unit Tests gleichgesetzt. Abbildung 1–1 illustriert die Kapitelstruktur nochmals:

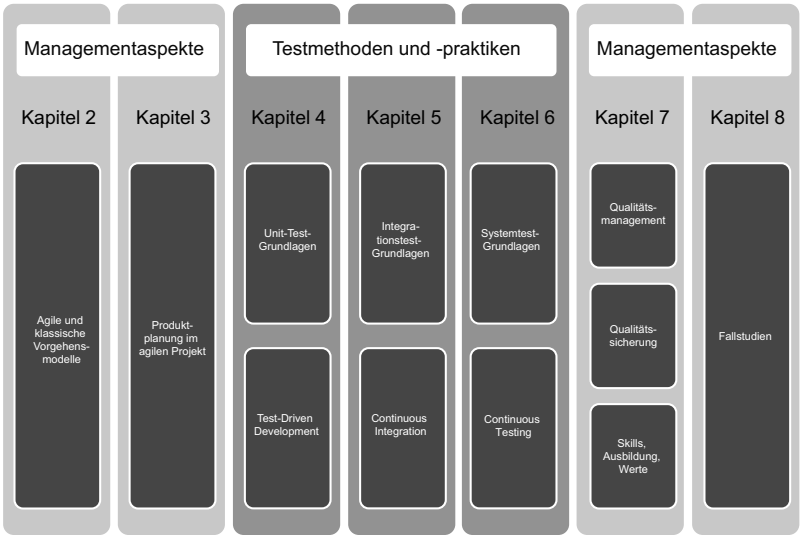


Abb. 1–1
Kapitelstruktur

Das Buch deckt den Stoff folgender ISTQB®-Lehrpläne ab:

Cross-Referenz zu den ISTQB®-Lehrplänen

- *ISTQB® Certified Tester – Foundation Level Extension Syllabus Agile Tester,* EN 2014, DE 2017,
- *ISTQB® Certified Tester – Advanced Level Syllabus – Agile Technical Tester (ATT),* EN 2019, DE 2019,
- *ISTQB® Certified Tester – Advanced Level Syllabus – Agile Test Leadership at Scale (CTAL-ATLaS),* EN 2023 sowie
- die agiles Testen betreffenden Inhalte im *ISTQB® Certified Tester – Foundation Level Syllabus, v4.0,* EN/DE 2023.

Die Gliederung des Buches folgt jedoch nicht der Gliederung der Lehrpläne. Darüber hinaus werden verschiedene Aspekte, die beim Testen in agilen Projekten eine Rolle spielen, im Buch über die Lehrplaninhalte hinausgehend oder zusätzlich behandelt.

In Anhang A sind daher Cross-Referenz-Tabellen enthalten, mittels derer diejenigen Buchabschnitte, die den Inhalt der in den betreffenden Lehrplänen jeweils genannten Lernziele erläutern, gezielt nachgeschlagen werden können.

Fallbeispiel, Checkfragen und Übungen

Viele, wenn nicht sogar die meisten agilen Ideen, Techniken und Praktiken sind, wenn man den Ausführungen in der entsprechenden Literatur folgt, einfach nachzuvollziehen. Auch die Ideen, Hinweise und Tipps in den folgenden Kapiteln werden vielleicht zunächst einfach erscheinen. Die Knackpunkte stellen sich erst in der Praxis bei der

Umsetzung heraus. Das Buch geht auf diese Hürden ein, und um praktisch nachvollziehbar und »erlebbar« zu machen, wo die Herausforderungen liegen, finden sich im Text die folgenden Elemente:

- Ein durchgängiges Fallbeispiel, anhand dessen die jeweils vorgestellten Methoden und Techniken veranschaulicht werden.
- Checkfragen und Übungen, anhand derer die Leserinnen und Leser am Ende eines Kapitels die besprochenen Inhalte rekapitulieren, aber auch ihre Situation und ihr Agieren im eigenen Projekt kritisch hinterfragen können.

1.3 Fallbeispiel

Dem Fallbeispiel des Buches liegt folgendes fiktives Szenario zugrunde: Die Firma »eHome-Tools« entwickelt Systeme zur Hausautomation. Das Funktionsprinzip solcher Systeme ist folgendes:

Fallbeispiel eHome-Controller

- **Aktoren:**
  Lampen und andere elektrische Verbraucher werden mit elektronischen Schaltern verbunden (sog. Aktoren). Jeder Aktor ist (per Kabel- oder Funkverbindung) an einen Kommunikationsbus angeschlossen und über diesen »fernsteuerbar«.
- **Sensoren:**
  An den Bus können zusätzlich elektronische Temperaturfühler, Windmesser, Feuchtigkeitssensoren usw. angekoppelt werden, aber auch einfache Kontaktsensoren, die z.B. geöffnete Fenster erkennen und melden.
- **Bus:**
  Schaltkommandos an die Aktoren, aber auch Statusmeldungen der Aktoren und Messwerte der Sensoren werden in Form von sogenannten Telegrammen über den Bus von und zum Controller übertragen.
- **Controller:**
  Der Controller sendet Schaltkommandos an die Aktoren (z.B. »Licht Küche ein«) und empfängt Statusmeldungen der Sensoren (z.B. »Temperatur Küche 20 Grad«) und Aktoren (z.B. »Licht Küche eingeschaltet«). Er ist in der Lage, ereignisgesteuert (also abhängig von eingehenden Meldungen) oder auch zeitgesteuert Folgeaktionen auszulösen (z.B. »20:00 Uhr → Rollo Küche schließen«).

- **Bedienoberfläche:**
  Der Controller bietet den Bewohnern des eHome auch eine geeignete Bedienoberfläche. Diese visualisiert den aktuellen Status des eHome und ermöglicht es den Bewohnern, Befehle (z.B. »Licht Küche aus«) per »Mausklick« an die Hauselektrik zu senden.

»eHome-Tools« steht mit seinen Produkten in einem harten Wettbewerb mit einer Vielzahl von Anbietern. Um in diesem Wettbewerb bestehen zu können, wird beschlossen, eine neue Controller-Software zu entwickeln. Allen Beteiligten ist klar, dass Schnelligkeit ein wesentlicher Erfolgsfaktor für das Vorhaben ist. Denn immer mehr Interessenten und Kunden fragen »eHome-Tools« nach einer Bediensoftware, die auf Smartphones und anderen mobilen Geräten läuft. Auch die Offenheit und Erweiterbarkeit des Systems für Geräte von Fremdherstellern ist enorm wichtig, um Marktanteile hinzuzugewinnen. Wenn das neue System Geräte konkurrierender Hersteller steuern kann, rechnet man sich Chancen aus, auch die Kunden dieser Hersteller z.B. beim Ausbau ihrer Systeme für die eigenen Geräte begeistern zu können. Dazu muss man nicht nur möglichst schnell eine möglichst breite Palette von Wettbewerbs-Hardware unterstützen, sondern auch künftig in der Lage sein, neu am Markt auftauchende Gerätemodelle kurzfristig einzubinden.

Daher wird entschieden, den Controller »agil« zu entwickeln und monatlich eine verbesserte, neue Version des Controllers herauszubringen, die mehr Geräte und weitere Protokolle unterstützt.

1.4 Webseite

Die im Buch enthaltenen Codebeispiele sind auf der Webseite zum Buch unter [URL: SWT-knowledge] veröffentlicht und herunterladbar. Die Leserinnen und Leser können diese Beispiele so auf ihrem eigenen Rechner nachvollziehen und mit eigenen Testfällen experimentieren. Auch die Übungsfragen sind dort zu finden.

Trotz der hervorragenden Arbeit des Verlags und der Reviewer sowie mehrerer Korrekturdurchgänge sind Fehler im Text nicht auszuschließen. Notwendige Korrekturhinweise zum Buchtext werden ebenfalls auf der Webseite veröffentlicht.

Did we build the system right?

Da in Scrum der Product Owner der Vertreter des Kunden ist, sind Abnahmetests in Scrum in erster Linie ein Qualitätssicherungsinstrument des Product Owners, während Unit Tests, Integrations- und Systemtests ein Qualitätssicherungsinstrument des Teams sind (»Did we build the system right?«). Kunde/Product Owner und Team verfolgen mit ihren Tests unterschiedliche Ziele. Inhaltlich müssen Abnahmetests aber nicht grundverschieden von den Tests des Teams sein. So kann das Team Schritte eines vom Product Owner beschriebenen Abnahmetests als Testablauf auf einer anderen geeigneten Teststufe wiederverwenden. Andererseits kann auch der Product Owner Teile seiner Abnahmetests durchaus aus der Menge bereits vorhandener Systemtests zusammensetzen.

Manuelle Abnahmetests ergänzen die automatisierten Tests.

In der Praxis wird der Abnahmetest innerhalb eines Sprints inhaltlich meistens auf die im jeweiligen Sprint geänderten oder neu entstandenen Produktaspekte eingeschränkt. Das Team führt diese Abnahmetestfälle dann in der Sprint-Demo dem Product Owner als manuellen explorativen Test vor. Wenn die übrigen Tests automatisiert erfolgen, ist das auch ohne großes Risiko machbar. Wenn aber außer diesen eingeschränkten Abnahmetests keine weiteren Systemtests im Sprint stattfanden, dann ist das Risiko groß, dass unerwünschte Seiteneffekte der Änderungen, die im Sprint gemacht wurden, unentdeckt bleiben.

6.4 Automatisierte Systemtests

Für Unit Tests und Integrationstests ist es in agilen Projekten Stand der Praxis, dass diese Tests nicht manuell durchgeführt werden, sondern in Form von xUnit-Testskripten automatisiert vorliegen. Wenn diese Testskripte in einer CI-Umgebung eingebunden sind, werden sie bei jeder Codeänderung automatisch aufgerufen und ausgeführt (vgl. Kap. 4 und 5). Unit Tests und Integrationstests werden in diesem Sinne kontinuierlich durchgeführt und liefern schnelles, kontinuierliches Feedback an das Team.

Diesen Komfort und diese Feedbackgeschwindigkeit möchte man gerne auch für die Systemtests erreichen. Leider gelingt das nicht so leicht. Zum einen behindert (wie oben erläutert) die komplexere Systemtestumgebung. Zum anderen fällt es schwerer, Systemtestfälle zu automatisieren, als xUnit Tests zu schreiben. Die Gründe dafür sind vielfältig:

- Als wichtigste Testschnittstelle dient die Bedienoberfläche (Graphical User Interface, GUI) des zu testenden Produkts. Hier werden spezielle GUI-Testwerkzeuge benötigt (s. [URL: Toolliste]). Die Ausführungsgeschwindigkeit solcher GUI-Testwerkzeuge ist wesentlich geringer als die Geschwindigkeit typischer xUnit Tests. Auch müs-

sen die Tests häufig auf die Reaktion und Rückmeldung anderer Komponenten (beispielsweise auf die Antwort einer unterlagerten Datenbank) oder externer Systeme warten. Die Laufzeit eines GUI-Tests ist daher typischerweise um eine Größenordnung langsamer als die eines typischen Unit Test.

- Die Systemkonfiguration muss in klar definierte Ausgangszustände zurücksetzbar sein, auf denen die Testskripte zuverlässig aufsetzen können.
- Oft sind zusätzlich zur GUI weitere Testschnittstellen anzusprechen. Dies erfordert dann den Einsatz zusätzlicher Testautomaten. Es müssen dann Tests für verschiedene Testautomaten geschrieben und deren Ausführung koordiniert gesteuert werden.
- Für viele Testfälle muss die Testauswertung durch einen Menschen vorgenommen werden, da ein Soll-Ist-Vergleich der Ergebnisse maschinell nicht oder nur sehr aufwendig möglich ist.
- Aus ähnlichen Gründen sind für manche Systemtestfälle manuelle Bedieneingriffe notwendig, die ebenfalls nicht automatisiert werden können.
- Oft fehlen im Scrum-Team auch Personen mit dem Wissen und der Erfahrung über Systemtestautomatisierung.

6.4.1 Capture and Replay

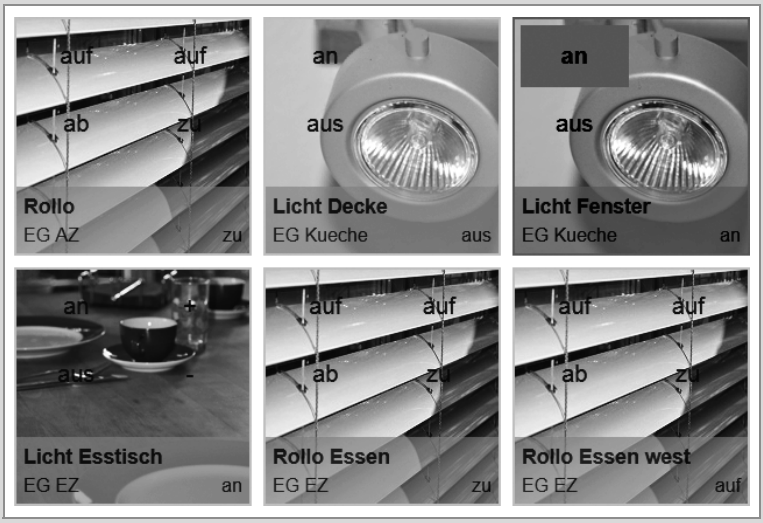
Im Unit Test und Integrationstest liegt die Testschnittstelle auf Ebene des Programmcodes. Die Testfälle werden mittels xUnit erstellt und ein Testfall trägt jeweils zur Prüfung genau einer Methode des API des Testobjekts bei. Der Inhalt der Testfälle ergibt sich direkt aus der Sollfunktion jeder einzelnen API-Methode. Das »Vokabular«, in dem diese Tests formuliert sind, ist durch die Menge der API-Methoden und deren Parameter gegeben. Auf Ebene der Systemtests ist das anders gelagert. Als Testschnittstelle muss dort in der Regel die Bedienoberfläche des Produkts angesprochen werden. Als geeignetes Vokabular für Systemtests bieten sich daher die »Bedienschritte« an, die in der Bedienoberfläche möglich sind.

Capture-and-Replay-Tools zeichnen Bediensequenzen auf.

Mittels sogenannter Capture-and-Replay-Tools können solche Bediensequenzen aufgezeichnet werden. »Das Capture-and-Replay-Tool zeichnet während einer Testsitzung alle manuell durchgeführten Bedienschritte (Tastatureingaben, Mausklicks) auf, die der Tester am Testobjekt ausführt. Diese Bedienschritte speichert das Werkzeug als Testskript. Durch ›Abspielen‹ des Testskripts kann der aufgezeichnete Test beliebig oft automatisch wiederholt werden« [Spillner & Linz 19]. Im eHome-Beispiel könnte ein Testskript wie das folgende entstehen:

Fallbeispiel eHome-Controller 6–4:
Bediensequenz im Systemtest

Der Tester öffnet die eHome-Controller-Applikation in seinem Browser und schaltet per Mausklick das »Licht am Küchenfenster« ein und aus. Nachfolgende Abbildung zeigt die eHome-Controller-GUI im Browser.



Seine Bedienschritte zeichnet er mit dem Capture-and-Replay-Tool »Selenium IDE« (s. [URL: Toolliste]) auf. Als Ergebnis erhält er beispielsweise folgendes Testskript:

```
open    http://ehome/eHomeController/index.php
clickAndWait xpath=(//a[contains(text(),'an')])[2]
clickAndWait xpath=(//a[contains(text(),'aus')])[2]
```

Das Testskript weist einige Schwächen auf. Zum einen testet es nicht wirklich, ob das »Licht am Küchenfenster« eingeschaltet wurde, sondern es löst lediglich Klicks auf den »an«- und »aus«-Button aus, ohne zu überprüfen, ob das System diese Kommandos tatsächlich ausführt. Zum anderen verknüpft es den Testfall eng mit der Bedienoberfläche und auch mit der Testumgebung. Denn der Testfall erwartet eine deutschsprachige Bedienoberfläche (»an«, »aus«), in der das »Licht am Küchenfenster« als zweites »Licht«-Icon erscheint. Und der Befehl `clickAndWait` verwendet eine im Testtool eingestellte Wartezeit, die sich aber je nach Testumgebung verändern kann.

Wenn man Systemtestfälle mittels Capture-and-Replay-Tool aufzeichnet und somit im Vokabular der »Bedienschritte« formuliert, läuft man sehr leicht in solche Fallen. Und man legt sich auf eine be-

stimmte Ausprägung oder Technologie der Bedienoberfläche (z.B. HTML) fest. Gerade die Bedienoberfläche ist aber ein Teil des Produkts, der sich über die Sprints hinweg erfahrungsgemäß stark verändert. Die Tests müssen dann in jedem Sprint immer wieder aufwendig an die veränderte Bedienlogik und/oder GUI-Technologie angepasst werden. Statt vorhandene Tests ständig zu pflegen, möchte man die Zeit im Sprint aber nutzen, um neue Tests für die neuen Funktionen zu erstellen. Ein weiteres Manko ist: Viele Produkte bieten dem Anwender alternative Oberflächen bzw. Layouts zur Auswahl an. Beispielsweise lässt sich der eHome-Controller nicht nur über die grafisch aufwendiger gestaltete PC-Browser-Webseite steuern, sondern auch über eine für Smartphones oder Tablets optimierte Webseite. Es ist weder wirtschaftlich noch zeitlich machbar, für jede dieser Oberflächen verschiedene Systemtests zu erstellen.

6.4.2 Daten- und schlüsselwortgetriebener Test

Das Team kann solche Schwierigkeiten vermeiden, wenn es als Systemtestvokabular statt der Bedienschritte ein abstrakteres Vokabular wählt, nämlich das Vokabular der Use Cases oder der Businesslogik der Anwendung.

Keyword-Driven Testing

Diese Methode, Systemtests zu formulieren, wird auch als »schlüsselwortgetriebener Test« oder »Keyword-Driven Testing«[10] bezeichnet und ist eine etablierte Technik zur Systemtestautomatisierung. Die Businesslogik des eHome-Controllers dreht sich um das »Schalten elektrischer Geräte« – und das eHome-Team beschließt, für seine Systemtests eine passende »domänenspezifische Testsprache«[11] zu definieren:

Fallbeispiel eHome-Controller 6–5a:
Entwicklung einer DSL für den Systemtest

Ein Hausbewohner und Anwender der eHome-Software steuert elektrische Geräte (z.B. Lampen) oder lässt sich Sensordaten anzeigen (z.B. Temperaturen). Lampen können geschaltet oder gedimmt werden, Sensoren können ausgelesen und mit Sollwerten belegt werden.

→

10. Eine ausführliche Darstellung dieser Methode findet sich in [Daigl & Rohner 22].
11. Die hier vorgestellte Kommando- bzw. Schlüsselwortnotation kann als eine einfache »Domain Specific Language« (DSL) aufgefasst werden. Als Werkzeug zur Testautomatisierung in agilen Projekten sind DSLs ein interessanter Ansatz. Komplexere DSLs besitzen Variablen, Kontrollstrukturen, Prozedurdefinitionen und andere Sprachkonstrukte. Einführungen in das Thema DSLs bieten z.B. [Ghosh 11], [Fowler & Parsons 10] und [Rahien 10]. Die Grundlagen zum Compilerbau finden sich z.B. in [Aho et al. 99].

»Sätze« einer sehr einfachen Sprache zum Umgang mit den Objekten dieser Domäne könnten daher nach dem Schema <Objektname> <Kommando> gebildet werden und wie folgt aussehen:

```
Küchenlampe einschalten;
Wohnzimmer-Temperatur anzeigen;
```

Eine verbessertes Schema erlaubt es, den Ort durch ein Zimmer und eventuell durch ein Geschoss zu identifizieren, und könnte so aussehen:

```
<Geschoss><Zimmer><Objekttyp><Objektname><Kommando><Parameter>
```

Dieses einfache Schema reicht aus, um schon relativ umfangreiche Steuerungssequenzen zu formulieren. So könnte die Hausherrin auf dem Nachhauseweg über ihr Smartphone folgende Befehlssequenz senden:

```
Garagentor auf
Wohnzimmer Heizung 20 Grad
Erdgeschoss Licht an[a]
Wohnzimmer Licht Sitzecke dimmen 60%[b]
Fernseher an[c]
```

a. Wenn der Objektname weggelassen wird, werden alle Geräte am angegebenen Ort gleichzeitig angesprochen.
b. Welche Parameter verfügbar sind, hängt vom jeweiligen Kommando ab.
c. Wenn der Objektname eindeutig ist, kann die Ortsangabe entfallen.

Wie man sieht, eignet sich diese DSL bereits gut, um in standardisierter Form praxisrelevante Use Cases zu notieren. Und als »Abfallprodukt« wurde gleichzeitig eine kommandoorientierte Bedienschnittstelle für den eHome-Controller spezifiziert. Um eine domänenspezifische Testsprache daraus zu machen, fehlt noch ein Schritt: Die Reaktionen des Systems müssen erkannt und mit Sollwerten verglichen werden können. Dazu wird außer den oben gezeigten »Aktionen« auch eine »check«-Funktion benötigt. Ein Systemtestfall könnte dann so spezifiziert werden:

Fallbeispiel eHome-Controller 6–5b: Systemtestfall

Kommandosyntax	Funktionsaufrufe der Testablaufsteuerung
`Kueche Licht Fenster an`	`switch('Kueche','Licht','Fenster','an');`
`Kueche Licht Fenster status? an`	`assert('Kueche','Licht','Fenster','an');`

Das »check«-Kommando `assert` liest den Status des Küchenlichts und prüft, ob dieser Status `'an'` ist. Je nachdem, wie viel »Intelligenz« man in den Parser für die Sprache und in die Testablaufsteuerung (die die Sprachkonstrukte letztlich ausführen können muss) einbaut, kann die Bedienschnittstelle, aber auch die Testsprache beliebig komfortabel werden.

Data-Driven Testing

Wenn die Syntax der Testschritte bzw. die Syntax der DSL es erlaubt, Parameter nicht nur als feste Werte (z.B. `'an'`, `'aus'`), sondern durch Variablen anzugeben, dann können die gewünschten Testdaten in eine Testdatentabelle ausgelagert werden. Die Testdaten werden dadurch von der Testlogik (d.h. der Sequenz der Testschritte) separiert. Das folgende Fallbeispiel zeigt die Idee:

Fallbeispiel eHome-Controller 6–5c: datengetriebener Test

```
switch  (FLOOR, ROOM, DEVICE, NAME, 'aus');[a]
assert  (FLOOR, ROOM, DEVICE, NAME, 'aus');
switch  (FLOOR, ROOM, DEVICE, NAME, 'an');
assert  (FLOOR, ROOM, DEVICE, NAME, 'an');
switch  (FLOOR, ROOM, DEVICE, NAME, 'aus');
```

FLOOR	ROOM	DEVICE	NAME
EG	Wohnzimmer	Licht	Sitzecke
EG	Kueche	Licht	Fenster
OG	Kind	Steckdose	Fernseher
...			

a. Man kann auch das Kommando `'an'` oder `'aus'` in der Testdatentabelle hinterlegen. Der Zweck der Testsequenz ist es aber, zu prüfen, ob ein Gerät korrekt an- und wieder ausgeschaltet werden kann, und das für eine Reihe von Geräten. Dieser Testzweck wird deutlicher, wenn an/aus im Testfall codiert und nicht in der Testdatenliste ist.

Über die Variablen einer Testdatentabelle wird obige Testsequenz mit passenden Testdaten »versorgt«. So entsteht ein datengetriebener Test, der die in der Tabelle angegebenen Geräte aus- und anschaltet, prüft, ob die Schaltkommandos ausgeführt werden, und jedes Gerät zum Abschluss in den `'aus'`-Status zurückschaltet.

Die Vorgehensweise des datengetriebenen Testens kann in Testskripten jeder Teststufe (d.h. auch im Unit Test und im Integrationstest) angewendet werden. Auch in Testskripten, die ohne Schlüsselworte bzw. ohne DSL arbeiten, kann das Lesen der Testdaten aus externen Testdatentabellen realisiert werden.

Datengetriebenes Testen – Vorteile für agile Teams[a]

- Agile Teams können sich von Iteration zu Iteration schnell an die sich ändernden/erweiternden Funktionalitäten eines Produkts anpassen, da das Ändern/Hinzufügen neuer Datenkombinationen einfach ist und wenig oder keine Auswirkungen auf die bestehende Testautomatisierung hat.
- Agile Teams können die erforderliche Testüberdeckung einfach nach oben oder unten anpassen, indem sie Einträge in der Testdatentabelle hinzufügen, ändern oder entfernen. Auf diese Weise können agile Teams auch die Testdurchführungszeiten steuern, um die Bedingungen für ein Continuous Deployment (kontinuierliche Bereitstellung/Inbetriebnahme [URL: Continuous Deployment]) zu erfüllen.
- Datengetriebenes Testen unterstützt die Philosophie des multidisziplinären Arbeitens (working cross-functionally), da Testdatentabellen einfacher zu verstehen sind als Testskripte und daher eine effektivere Teilnahme von Teammitgliedern mit weniger technischem Know-how ermöglichen.
- Da Testdatentabellen leichter verständlich sind, unterstützt datengetriebenes Testen ein frühes Feedback zu Testfällen und Abnahmekriterien von Teammitgliedern mit weniger/keinem technischen Know-how und Kunden/Anwendern.
- Datengetriebenes Testen hilft agilen Teams, Testautomatisierungsaufgaben effizienter zu erledigen, da es nicht nur den Aufwand für die Entwicklung neuer Tests verringert, sondern auch die Wartung vorhandener datengetriebener Tests reduziert.

a. Da der Text der Syllabus-Kapitel »Data-Driven Testing« und »Keyword-Driven Testing« vom Autor dieses Buches (als Mitglied der für die erste Ausgabe zuständigen ISTQB®-Arbeitsgruppe) verfasst wurde, erlaubt er es sich hier, diese Textabschnitte wörtlich aus [URL: ISTQB CTAL-ATT] zu zitieren.

Da eine Schlüsselwortnotation oder eine DSL-basierte Notation in der Regel immer Variablen als Kommandoparameter kennt und zulässt, kann datengetriebenes Testen auch als »Vorstufe« von schlüsselwortgetriebenem Testen betrachtet werden. Den maximalen Nutzen entfalten die beiden Methoden jedoch in Kombination.

Vorteile einer domänenspezifischen Testsprache

Der Einsatz einer domänenspezifischen DSL als Testsprache (die schlüsselwortbasiert sein kann oder auch eine komplexere Syntax aufweisen kann) bietet folgende Vorteile:

- Die DSL verwendet einen Ausschnitt des Vokabulars der Anwendungsdomäne. Im Zuge der Aufnahme in die DSL oder spätestens durch die Art der Verwendung in den Testfällen werden die Begriffe in ihrer Bedeutung definiert oder zumindest gut eingegrenzt. Dies fördert eine klare und präzise Kommunikation im Entwicklungsteam und mit den Stakeholdern.
- Das DSL-Vokabular kann bereits zur Formulierung von Akzeptanzkriterien genutzt werden. Dies kann Missverständnisse reduzieren und unterstützt den Shift-Left-Gedanken (s. Abschnitt 3.8.4). Es kann nicht unerheblich zu schnellerem und präziserem Feedback an das Entwicklungsteam beitragen.
- Wenn die Testfälle im Vokabular der Anwendungsdomäne formuliert sind, können Personen, die mit der Anwendungsdomäne vertraut sind, diese verstehen. Mitarbeiter aus den Fachabteilungen und andere Stakeholder können daher leichter in die Erstellung der Tests (beispielsweise als Reviewer) einbezogen werden.
- Auch Mitarbeiter mit weniger Programmiererfahrung können automatisierte Testabläufe erstellen, indem sie die gewünschten Testabläufe aus dem DSL-Vokabular bausteinartig zusammensetzen.
- Ändert sich die Implementierung des Produkts, bleiben die Tests weiterhin gültig und ablauffähig. Die Tests im Beispiel können unverändert zum Test einer PC-Browser-Oberfläche verwendet werden, aber auch für den Test über eine unter Umständen ganz anders aussehende Oberfläche einer Smartphone-App.
- Die Testfälle eignen sich auch, um Tests auf niedrigeren Systemschnittstellen anzusteuern, ohne die Testfälle zu verändern! Im Beispiel könnte etwa ein entsprechender Testtreiber die Testfälle in Controller-Kommandos übersetzen und direkt den eHome-Controller ansprechen. Der Testtreiber kann dann als Stellvertreter (bzw. Doublette) der GUI dienen, sodass die Testfälle auch ohne GUI ablaufen können. Das reduziert die Testlaufzeit und es erlaubt den Test des Controllers schon im Rahmen der Unit Tests und Integrationstests – auch dann, wenn die GUI noch nicht fertiggestellt ist.

Natürlich ist dieser Komfort nicht umsonst zu haben: Im Team muss Einigkeit über die zu verwendenden Schlüsselworte herrschen und dieses Vokabular muss stabil sein. Es wird ein Interpreter benötigt, der die Kommandos in passende Funktionsaufrufe der Testablaufsteuerung übersetzt, eine passende Testablaufsteuerung, die die Kommandos ausführt, und ein Adapter, der die Testablaufsteuerung letztlich mit dem Testobjekt verbindet. Damit der Adapter im Testobjekt die jeweils anzusteuernden Objekte (im Falle einer Bedienoberfläche also Buttons, Checkboxen, Textfelder usw.) sicher erkennt, müssen diese Objekte

feste Identifiers besitzen, die über die Sprints hinweg unverändert bleiben müssen. Andernfalls muss der Testcode von Sprint zu Sprint an die wechselnden Identifiers angepasst werden, was unnötigen Wartungsaufwand erzeugt. Abbildung 6–1 veranschaulicht, wie die verschiedenen Schichten einer solchen Testautomatisierung zusammenwirken.

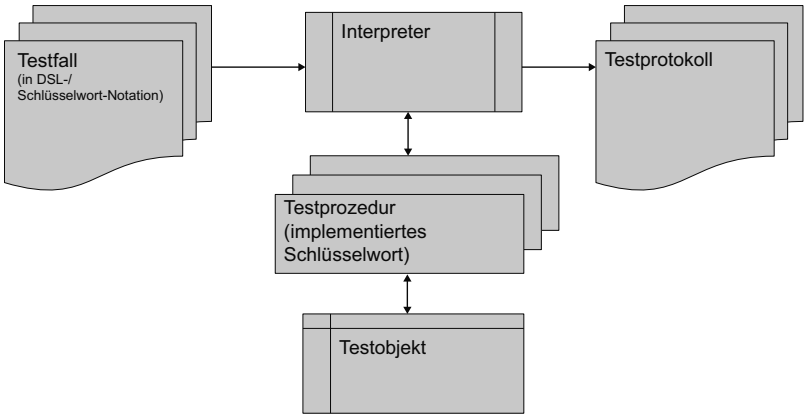


Abb. 6–1
Dreischichtige Testarchitektur

Dreischichtige Testarchitektur

Eine dreischichtige Testarchitektur sorgt für eine klare Trennung der Zuständigkeiten. Dinge, die in der Testlogik nichts zu suchen haben, sind in die Ablaufsteuerung oder den Adapter verschoben:

- **Entkopplung der Testumgebung:**
  Ein Parameter, wie z.B. eine einzustellende Timeout-Wartezeit, ist keine Eigenschaft des Testfalls, sondern eine Eigenschaft des Laufzeitverhaltens der Testumgebung. Daher sollten solche Parameter in die Ablaufsteuerung »ausgelagert« und dort z.B. als Methode einer Klasse »Testumgebung« (`TestEnvironment`) implementiert werden. Indem die Ablaufsteuerung diese Methode (z.B. `TestEnvironment->wait_for_MsgAck()`) aufruft, ist sichergestellt, dass automatisch immer genau so lange auf ein Quittungssignal gewartet wird, wie es die jeweilige Testumgebung erfordert. Wie lange gewartet werden muss, »entscheidet« jetzt die Testumgebung. Die Wartezeit ist nicht mehr Eigenschaft der Testlogik. Auf ähnliche Art können auch andere Eigenschaften der Testumgebung aus der Testlogik herausgehalten werden.
- **Entkopplung der Testschnittstelle:**
  Wenn die Testlogik zu sehr mit einer bestimmten Testschnittstelle verwoben ist, dann kann der Testfall nur über diese bestimmte Testschnittstelle ausgeführt werden. Im Gegensatz zu Unit Tests und Integrationstests gibt es im Systemtest häufig Situationen, wo die

gleichen Testfälle über verschiedene Testschnittstellen laufen sollen. Der eHome-Controller etwa kann über unterschiedliche Oberflächen bedient werden (verschiedene PC-Browser, verschiedene Smartphone-Browser und Apps, Kommandoschnittstellen). Indem die Testablaufsteuerung nicht direkt, sondern über einen Adapter auf die gewünschte Testschnittstelle zugreift, kann sie durch Austausch der Adapter mit verschiedenen Testschnittstellen kommunizieren, ohne dass die Testlogik verändert werden muss. Der Adapter leistet dann die »Übersetzung« der Testschritte in das Format der jeweiligen Testschnittstelle. Um Tests über die GUI des Testobjekts auszuführen, wird dann beispielsweise ein Adapter benötigt, der einen geeigneten GUI-Testautomaten (z. B. Selenium) ansteuert und in dessen Befehle übersetzt.

Schlüsselwortgetriebene Systemtestautomatisierung erfordert initiales Investment.

Wenn das Team seine Systemtests entwirft, muss es untersuchen, ob eine so hohe Modularität und Flexibilität im Projekt benötigt wird. In kleinen oder kurzlaufenden Projekten wird sich das vermutlich nicht auszahlen. Denn der Aufbau einer solchen dreischichtigen, schlüsselwortbasierten Systemtestautomatisierung erfordert ein nennenswertes initiales Investment. Allerdings gibt es am Markt heute Werkzeuge, die den Aufbau einer solchen Testarchitektur vereinfachen (s. [URL: Toolliste]).

Zum anderen kann die Aufbauarbeit inhaltlich stark entkoppelt werden von den Tasks, die zur Produktentwicklung nötig sind. In der Produktentwicklung werden die Produktfeatures implementiert und die zugehörigen Systemtests in Schlüsselwortnotation erfasst. Wenn die Testablaufsteuerung schon vorhanden ist, laufen die schlüsselwortgetriebenen Tests sofort! Ist die Testablaufsteuerung noch unvollständig, dann wird erst ein Teil der Systemtests automatisiert laufen. Diejenigen, die noch nicht ausführbar sind, müssen vorübergehend manuell ausgeführt werden. Aber die Testvorschrift, was der Tester manuell zu prüfen hat, die liegt in Schlüsselwortnotation bereits vor.

Zur Programmierung der Testablaufsteuerung ist nur geringes oder gar kein Wissen über die fachliche Logik der Anwendung nötig. Gefragt ist vielmehr gutes Testautomatisierungs-Know-how. Das Schreiben der Anwendung, das Schreiben der Systemtests (in Schlüsselwortnotation) und das Programmieren der Testablaufsteuerung sind deshalb Tasks, die an verschiedene Teammitglieder verteilt werden können. Die Sprint-Planung kann daher relativ gut »ausbalancieren«, dass die Testablaufsteuerung mit den Systemtests und dem Produkt in etwa Schritt hält. Ein größerer Aufwand ist freilich nötig, um eine geeignete Testablaufsteuerung erstmals im Team aufzubauen und in die CI-Umgebung zu integrieren. Dieser Aufgabenblock sollte im Rahmen des Aufbaus der CI-Umgebung vom ersten Sprint an angegangen und in den frühen Sprints stark gewichtet werden.

6.4.3 Behavior-Driven Testing

Ein weiterer Ansatz, um Tests in einer domänenspezifischen Sprache (DSL) zu formulieren und zu automatisieren, ist »Behavior-Driven Testing« (BDT). BDT ist ein Testautomatisierungsansatz aus dem sogenannten »Behavior-Driven Development« (verhaltensgetriebene Softwareentwicklung, s. [URL: BDD]). Im Unterschied zum oben beschriebenen schlüsselwortgetriebenen Test, wo mit einer tabellenorientierten Darstellung der Testfälle gearbeitet wird, bietet der BDT-Ansatz eine noch stärker an natürlicher Sprache angelehnte Notation.

Die Übergänge zwischen den Ansätzen sind allerdings fließend und lassen sich ineinander überführen. So ist es denkbar, auch für BDT-Testskripte die verwendeten Schlüsselworte in einem Repository zu verwalten und Möglichkeiten zur hierarchischen Strukturierung vorzusehen.

Für BDT steht eine Reihe von Frameworks zur Verfügung, z.B. Fit, FitNesse, Cucumber, JBehave, Specs2 oder Behat (vgl. [URL: Toolliste]), die die Automatisierung von BDT-Testfällen unterstützen. Der Automatisierungsansatz ist dabei vergleichbar mit dem oben beschriebenen Schlüsselwortansatz: Ein Produktfeature wird durch ein oder mehrere sogenannte »Scenarios« getestet. Ein Scenario entspricht dabei einem Systemtestfall und gliedert sich in `Given-When-Then`-Abschnitte, die in etwa den `setup-procedure-check`-Abschnitten eines Unit-Testfalls entsprechen (vgl. Abschnitt 4.1.2).

Im eHome-Beispiel könnte das Team das für PHP geeignete Framework »Behat« einsetzen und das folgende Testskript erstellen:

Fallbeispiel eHome-Controller 6–6:
Systemtestfall formuliert als Behavior-Driven Test mit Behat

```
Feature: Geraete_steuern
Als eHome-Bewohner kann ich Geraete unterschiedlicher Klassen
steuern, wobei der eHome-Controller nur die fuer eine Geraeteklasse
gueltigen Steuerbefehle zulaesst.

Scenario: Licht ein und ausschalten
    Given Geraet "Licht" in "Kueche" am "Fenster"
    When schalten "an" "Licht" in "Kueche" am "Fenster"
    Then status "Licht" in "Kueche" am "Fenster" ist "an"
```

Die im Scenario vorkommenden Schlüsselworte und Parameter (z.B. `schalten` und `"Licht"`) muss das eHome-Team durch unterlagerte Testskripte implementieren. Für das Kommando `schalten` ist beispielsweise eine PHP-Funktion nach folgendem Muster zu programmieren:

→

```
/**
* @When /^schalten "([^"]*)" "([^"]*)" in "([^"]*)" am "([^"]*)"$/
*/
public function schaltenInAm($arg1, $arg2, $arg3, $arg4) {
    ...
}
```

Bei der Ausführung des Tests ruft Behat dann diese PHP-Funktion mit folgenden Parametern auf:

```
schaltenInAm("an", "Licht", "Kueche", "Fenster");
```

Der Einsatz eines BDT-Frameworks führt also ebenfalls zu einer mehrschichtigen Testarchitektur (vgl. Abb. 6–1). Als Schnittstelle der Testautomatisierung zum Testobjekt nutzen BDT-Frameworks wie Unit-Test-Frameworks die APIs des betreffenden Testobjekts. Wenn diese APIs auf einer hohen Architekturebene angesiedelt sind, dann kann BDT Systemtestfälle, die ansonsten über die Bedienoberfläche des Produkts zugreifen müssten, teilweise ersetzen.

BDT und API-Tests

Auch Unit-Testfälle (s. Kap. 4) können prinzipiell in Form von BDT formuliert werden. Der zusätzliche Aufwand für die natürlichsprachliche Formulierung ist hier allerdings selten gerechtfertigt. Denn das API einer einzelnen Klasse implementiert in Relation zu einer Anforderung auf Systemebene meistens nur eine Teil- oder Basisfunktionalität. Und diese Funktionalität lässt sich im Vokabular des API (API-Methoden und deren Parameter) einfacher ausdrücken als im Vokabular der Systemanforderungen. Wenn man allerdings erreichen möchte, dass die Unit Tests auch von Personen verstanden werden, die keinen Programmcode bzw. keinen herkömmlichen Unit-Test-Code lesen können, dann kann BDT eine Lösung sein. Analoges gilt für Integrationstestfälle.

BDT und User Stories

Wenn im Team BDT eingesetzt wird und schon ein gewisser »Vorrat« an BDT-Tests entstanden ist, dann ist damit ein Vokabular und ein Schema vorhanden, das das Team auch im Requirements-Prozess einsetzen kann, um die User Stories und/oder deren Akzeptanzkriterien von vornherein in BDT-Notation als »ausführbare Szenarien« zu formulieren.

Dabei ist zu beachten, dass dieses Vokabular dazu dienen soll, die fachliche Sicht auf das System bzw. dessen fachlich gewünschtes Verhalten zu beschreiben. Technische Details, die für den Ablauf in Form von automatisierten Testfällen notwendig sind, aber für die Beschreibung des Systemverhaltens nicht relevant sind, müssen daher in unterlagerten Schichten der Testarchitektur angesiedelt werden.

BDT vs. ATDD vs. SBE

BDT und ATDD sind Methoden, um Tests auf Systemebene zu definieren. In BDD wird dazu das Gerkhin-Satzschema (»Given-When-Then«) verwendet und ein so formulierter BDT-Test ist per BDT-Tool direkt ausführbar. BDT legt den Schwerpunkt somit auf das Ziel »ausführbare Spezifikation«.

Das Ziel von ATDD ist primär, dass die Abnahmetests vor der Implementierung der Features definiert und geschrieben werden. ATDD legt den Schwerpunkt also auf den »Test First«-Gedanken. Wie der Test notiert wird, ist weniger streng festgelegt. Oft wird auch hier Gerkhin verwendet.

Eine weitere Variante ist unter dem Namen »Specification by Example« (SBE) populär. Der Gedanke hier ist, dass die Testfälle Beispiele dafür darstellen, wie das System durch seine Anwender eingesetzt werden kann. SBE zielt somit eher auf den inhaltlichen Stil der Testfälle ab. Der Nutzen ist, dass es Stakeholdern in der Regel leicht fällt, ihre Wünsche in Form von Beispielen zu beschreiben. Dies soll durch SBE unterstützt werden.

In der Praxis sind die Grenzen zwischen den Methoden fließend. Welche Variante dieser Methoden ein Team einsetzt, hängt letztlich davon ab, welches Testautomatisierungstool das Team verwendet, und die Bezeichnung der Methode richtet sich dann danach, ob der jeweilige Toolhersteller sein Tool als BDT-, ATDD- oder SBE-Tool vermarktet.

6.5 Test First im Systemtest

Test First bedeutet, dass der Entwickler, bevor er ein Stück Code ändert oder ein neues Stück Code schreibt, zuerst einen oder mehrere Testfälle entwirft und automatisiert, die den geänderten oder neuen Code hinreichend testen[12]. Der fehlerfreie Durchlauf dieser Testfälle wird als »Done«-Kriterium des Programmiertasks aufgenommen. Im Unit Test und Integrationstest funktioniert das sehr gut (vgl. Kap. 4 und 5). Aber lässt sich Test First auch für Systemtests praktizieren?

Wenn die Tests in einem GUI-Testwerkzeug aufgezeichnet oder codiert werden, dann muss das Testobjekt mit Bedienoberfläche vorhanden sein. Denn das GUI-Testwerkzeug greift unmittelbar auf die Bedienoberfläche des Testobjekts zu. Test First ist so nicht machbar.

Systemtestfälle per Schlüsselwortmethode oder BDT

Wenn die Systemtestfälle per Schlüsselwortmethode oder BDT im Vokabular der Businesslogik formuliert werden, dann entstehen Testfälle, die unabhängig sind von der technischen Realisierung des Produkts und insbesondere unabhängig von dessen Bedienoberfläche.

12. »Write a failing automated test before changing any code« [Beck & Andres 04, Kap. 7].