10 Verwenden Sie Beispiele

In den ersten beiden Teilen dieses Buchs begann die eigentliche Arbeit mit Beispielen, die sich von den Geschäftsvorgaben ableiteten. Für gute Beispiele braucht man Domänen-Kenntnisse – oder zumindest Zugang zu jemandem, der sie hat. Im ersten Beispiel hatte das Team der Flughafenbetreibergesellschaft Blaport AG noch nicht ausreichend Kenntnisse über die Domäne, was es durch das anfängliche Scheitern auch zu spüren bekommen hat. Das Team hatte sich daher zu einem Workshop zusammengefunden. Um ihre fehlenden Domänen-Kenntnisse zu kompensieren, hatten sie den Business-Experten Bernd zu sich geladen. Zusammen mit Bernds Hilfe konnten Petra und Thorsten die Beispiele der Geschäftsbedingungen hinsichtlich der Flughafenparkplatz-Gebühren ableiten.

Im zweiten Beispiel haben wir uns zusammen durch ein Problem gearbeitet, bei dem ich davon ausgegangen bin, ausreichende Domänen-Kenntnisse zu haben. Für die Tiefe, in der wir dieses Beispiel behandelt haben, hatten wir ausreichende Domänen-Kenntnisse, so dass wir nach kurzem Überlegen anfangen konnten.

Die Beispiele sind die erste Säule des Arbeitens mit Akzeptanztest-getriebener Entwicklung, Verhaltensgetriebener Entwicklung (engl.: Behavior-driven Development, BDD) oder wie Sie es auch immer nennen wollen. Sie können stets mit jedweder Art von Beispielen arbeiten, unabhängig davon, ob Sie sie später automatisieren oder nicht.

Ich habe diesen Ansatz einmal bei einer Firma verwendet, die Produktentwicklung anhand einer Abwandlung des Wasserfall-Modells betrieben hat. Dort habe ich in der Abteilung für System-Integration gearbeitet. Das zu integrierende System war eine Lösung für Mobilfunkanbieter zum Rating und Billing. Wir haben das System für den Endkunden konfiguriert, während die Abteilung für Produktentwicklung das Produkt für einen größeren Kundenstamm entwickelt hat. Das Produkt war in einem hohen Maße für unterschiedliche Tarife in der Mobiltelefon-Domäne konfigurierbar. Eines Tages präsentierte die Produktentwicklung eine Neugestaltung der GUIs zur Konfigurierung des Produkts. Leider wurde dabei ein Feature aus einem früheren Release, das sich mit unterschiedlichen Workflows und Freigaben neuer Tarife befasste, zu einem echten Hindernis. Da wir die Kunden

und damit die Domänen-Experten für die Produktkonfiguration waren, haben uns die Kollegen aus der Produktentwicklung um Hilfe gebeten.

Wir trafen uns zu drei Meetings, um das Problem zu klären und zu einer Neugestaltung zu kommen, die beiden Abteilungen zugute kam. Vorher hatte das System einen Workflow, der die neuen Tarife in das System gebracht hat. Für einen realen Kunden hatte das System nie wirklich funktioniert. Wenn das Produkt modulare Teile für die bis dahin einzige Konfigurations-Datei unterstützen sollte, hätte der alte Workflow alles nur schlimmer gemacht. Die Konfigurations-Komponente wäre zur Validierungsunterstützung mehrerer Arbeitsbereiche und zur Validierung von Änderungen und Abhängigkeiten über mehrere Arbeitsbereiche hinweg gebraucht worden. Das hätte unter Umständen zu einem heillosen Durcheinander in der Abteilung für das Testen des Produkts geführt.

Innerhalb eines dreistündigen Workshops hatten wir den ganzen Workflow besprochen. Mit der Vision der neuen Lösung im Kopf kamen wir zu Beispielen, die den Workflow mit einer aufgeteilten Konfiguration beschrieben. Als die aktualisierte Version sechs Monate später fertig war, waren alle zufrieden damit.

Die Beispiele, die wir im Workshop gefunden haben, waren durchweg auf sehr hohem Niveau. Die Testabteilung hatte zu diesem Zeitpunkt noch nicht das Vermögen, die Tests auf dem Niveau zu automatisieren, wie wir sie besprochen hatten. Allein schon das Gespräch über die Anforderungen hatte zu einer besseren Implementierung des Produkts und, von unserer Seite aus, zu einer besseren Kundenzufriedenheit geführt.

Egal, ob Sie Ihre Beispiele nun automatisieren oder nicht, einige Dinge sollten Sie beachten. In diesem Kapitel geht es um diese Dinge, auf die Sie beim Gebrauch von Beispielen achten sollten. Die Hauptfaktoren, die den Erfolg eines Ansatzes bestimmen, sind das Format, in dem die Beispiele verfasst sind, wie detailgenau die Beispiele sind und ob Sie an irgendwelche Lücken in Ihrem Gesamtansatz gedacht haben.

Verwenden Sie das richtige Format

Die Tests durch Beispiele zum Ausdruck zu bringen, wirft die Frage auf, wie das geschehen soll. Auch wenn Sie vielleicht denken, dass das Verfassen von Beispielen ziemlich einfach sei, haben sich doch im Verlauf der letzten zehn Jahre bei den Teams, die ATDD anwenden, gewisse Muster herausgebildet.

Das richtige Format für Sie hängt von Faktoren in Ihrem Team, Ihrem Projekt und Ihrer Firmenorganisation ab. Zunächst müssen Sie berücksichtigen, wer die Tests liest, nachdem Sie sie erstellt haben. Das sind vermutlich Sie, der Programmierer, der das durch die Beispiele beschriebene Feature entwickelt, und Ihr Product-Owner, respektive der Kunde. Desgleichen müssen Sie an einen möglichen zukünftigen Test-Pfleger denken, für den Fall, dass Sie Ihre Tests einem anderen Team, einer anderen Abteilung oder gar einer anderen Firma übergeben. Alle diese zukünftigen Testleser möchten vielleicht einen schnellen Überblick über das

Feature, das Sie heute beschreiben. An dieser Stelle wird es wichtig, dass die Beispiele in einer einheitlichen Art und Weise formuliert werden.

Mit all diesen verschiedenen Zielgruppen im Hinterkopf sollten Sie Beispiele anstreben, die von allen verstanden werden können. Das bedeutet, dass Sie keine technischen Beschreibungen des Workflows verfassen sollten, bevor alle Beteiligten in der Lage sind, sich den Sinn daraus zu erschließen. Die Beispiele unseres Ampel-Controllers hätten wir folgendermaßen schreiben können:

Falls der Schaltkreis für das grüne Signal für Richtung A und in Richtung B eingeschaltet ist, fahre den Ampel-Controller in einen ausfallsicheren Zustand herunter. Im ausfallsicheren Zustand blinken die gelben Signale ständig. Nur ein Techniker darf die Ampel aus diesem ausfallsicherem Zustand nach Prüfung zurückversetzen. Der Controller muss in solchen Fällen alle Zustandsänderungen zur späteren technischen Analyse protokollieren.

Dieser Stil ist mehr erzählerisch, als man es in den üblichen Spezifikationen vorfindet. Jetzt überlegen Sie noch einmal, wie wir unsere Beispiele niedergeschrieben haben. Das erste, was Ihnen vielleicht auffällt, ist, dass wir uns überhaupt nicht um die Logging-Anforderung gekümmert haben – das wird vielleicht ein Bestandteil späterer Entwicklung. Wir haben auch alle Umstände aufgezählt, in denen der ausfallsichere Modus in Gang gesetzt werden soll. Dieser erklärerische Stil beim Verfassen der Beispiele hält für später viel mehr Informationen bereit, wenn wir nachlesen und die Beispiele erweitern müssen. Diese Informationen werden uns auch als Kommunikationsmittel für reale Kunden dienen – und sogar Benutzern des Systems wie Ihnen und mir.

ATDD-freundliche Testautomatisierungs-Frameworks teilen Testdaten und Testautomatisierungs-Code, der für die Ausführung der Tests nötig ist, auf. Da der Testautomatisierungs-Code die Entwicklung Ihrer Tests und die Entwicklung Ihrer Anwendung »zusammenklebt«, nenne ich ihn auch Glue-Code. Der Glue-Code kann innerhalb der verfügbaren Tools auch in verschiedenen Sprachen implementiert sein. Die meisten Tools gründen sich auf eigene Konventionen zum Aufrufen der Funktionen im Glue-Code. Durch die Trennung der Testdaten vom Glue-Code können Sie die Beispiele unabhängig von jeder bestimmten Implementierung der Anwendung definieren.

Es gibt noch viele andere Wege beim Verfassen der Beispiele für Ihre Anwendungs-Anforderungen. Der beliebteste Ansatz war in den letzten paar Jahren der der Verhaltensgetriebenen Entwicklung (BDD). Das BDD-Format entwickelt sich aus den Begriffen »Gegeben sei«, »Wenn« und »Dann« (»Given«, »When«, »Then«), die dabei helfen, die Erwartungen an ein bestimmtes Feature zu strukturieren.

Ein anderer beliebter Weg ist das Formulieren der Beispiele in *Tabellenform*. Es gibt zwar einige Variationen, aber sie alle haben drei verschiedene Formate gemeinsam: eine Form des Mapping von Ein- und Ausgabe, eine Form der Abfragen und eine Form der Aktionen wie in Workflows.

Eine dritte Möglichkeit Beispiele zu verfassen, sind *Keywords* oder *Datengetriebene Tests*. Durch die Keywords können Sie unterschiedliche Abstraktionsniveaus in Ihrer Test-Sprache kombinieren. Sie können sich dazu entschließen, Lower-Level-Funktionen für Tests zu nutzen, die sich auf niedrigere Ebenen Ihrer Anwendung konzentrieren, oder aber auch Lower-Level-Funktionen so kombinieren, dass Sie ein höheres Abstraktionsniveau erreichen. Durch Keywords werden diese verschiedenen Abstraktionsniveaus in Ihrer Testbeschreibung flüssiger und ergeben so eine Sprache für Ihre Domäne in Ihren Tests.

Verhaltensgetriebene Entwicklung

Die Verhaltensgetriebene Entwicklung (BDD) wurde als Erstes von Dan North beschrieben [Nor06]. In diesem Artikel hat North das *Gegeben sei-Wenn-Dann*-Paradigma das erste Mal erwähnt. Obwohl BDD aus mehr als nur *Gegeben sei-Wenn-Dann* besteht, um als Methode das Verhalten der Features auszudrücken, ist diese spezielle Syntax in den letzten Jahren doch fast zum Synonym für BDD geworden. Das *Gegeben sei-Wenn-Dann*-Format haben wir im Flughafen-Beispiel gesehen. Die drei Schritte im BDD-Format formieren sich um die Keywords *Gegeben sei*, *Wenn* und *Dann*. Sehen Sie sich als Beispiel Listing 10–1 an, das Google nach dem Begriff ATDD durchsucht und eine Menge Ergebnisse erwartet.

```
1  Feature: Google-Suche
2     Als Internet-User möchte ich Google zur Recherche nutzen
3
4     Szenario: Suche nach ATDD
5        Gegeben sei die vorgegebene Google-Seite
6        Wenn ich nach "ATDD" suche
7        Dann finde ich jede Menge Ergebnisse
```

Listing 10–1 *Eine einfache Suche im Internet*

Im *Gegeben sei*-Teil drücken Sie alle Vorbedingungen für das jeweilige Verhalten aus. Die Vorbedingungen ergeben den Zusammenhang für die Beschreibung des Verhaltens. Dies schließt alle jeweiligen Einstellungen ausdrücklich mit ein. In diesem kleinen Beispiel sorgt die Vorbedingung oberhalb der Zeile 5 dafür, dass ich den Browser geöffnet und die vorgegebene Google-Seite geöffnet habe. Im Flughafen-Beispiel hatten wir keine *Gegeben sei*-Zeilen, da alle Tests auf dem Parkgebührenrechner, der im geöffneten Browserfenster lief, basierten. Im Allgemeinen drücken Sie im *Gegeben sei*-Teil das aus, was vom Ausgangszustand abweicht. Bei einem System mit verschiedenen Typen von Accounts drücken Sie an dieser Stelle aus, an welchem Account Sie im vorliegenden Beispiel arbeiten. Bei einem System mit unterschiedlichen Webseiten möchten Sie dort vielleicht sagen, auf welcher jeweiligen Seite das Beispiel durchgeführt wird. Bei einem Workflow in einer Anwendung möchten Sie eventuell alle relevanten Daten zum Ausdruck bringen und dass Sie davon ausgehen, dass alle vorherigen Schritte des

Workflows schon durchlaufen sind, bevor der jetzt beschriebene Teil des Workflows zur Anwendung kommt. Es kann auch mehr als einen *Gegeben sei*-Teil geben. Sie werden in der Regel mit aufeinanderfolgenden Und-Statements verbunden.

Im *Wenn*-Teil beschreiben Sie das Verfahren und alle seine Parameter, die Sie auslösen. Im obigen Beispiel wollen wir im Grunde eine Eingabe in ein Textfeld auf einer Webseite und das Klicken des vorgegebenen Buttons auslösen. Andere Beispiele könnten darin bestehen, dass Sie sagen wollen, dass Sie ein Konto wieder mit Geld aufgefüllt haben, dass Sie diverse Werte in ein Webformular eingegeben haben oder dass Sie zur Kasse eines Online-Shops gegangen sind, nachdem Sie Ihren Warenkorb gefüllt haben. Im *Wenn*-Teil beschreiben Sie also etwas, das passiert [dFAdF10]. Das können ein bestimmter Auslöser durch einen User oder ein Vorgang außerhalb des momentanen Subsystems, wie etwa asynchrone Mitteilungen eines Third-Party-Internet-Services oder aber auch eine Funktion im Computersystem wie eine Zeitüberschreitung sein. Außerdem sollte in jeder Szenario-Beschreibung genau ein Arbeitsschritt enthalten sein. Auf diese Weise bleibt das Szenario ausreichend eng fokussiert.

Der *Dann*-Teil beschreibt die Bedingungen nach der Durchführung des Arbeitsschrittes im *Wenn*-Teil. In den meisten Fällen sind das Assertions gegen das System. Im Beispiel der Google-Suche überprüfen wir in Zeile 7, ob es viele Suchergebnisse gibt. Ein anderes Beispiel wäre nach der Wiederauffüllung eines Kontos die Überprüfung der Höhe eines Bonus, den das System liefern soll. Bei einem Webformular wie dem Parkgebührenrechner können Sie die berechneten Gebühren entsprechend Ihrer eigenen Erwartungen geltend machen. Wie die *Gegeben sei*-Statements können auch die *Dann*-Statements durch *Und* aneinandergereiht werden.

Eines der Mantras der Verhaltensgetriebenen Entwicklung ist die Entwicklung von außen nach innen. BDD bevorzugt die Entwicklung des Codes auf der Grundlage der Anforderungen vom Kunden. Der Sinn des Workshops beim Flughafenparkplatz-Beispiel war, den Anwendungsbereich des Rechners auf den Geschäftszielen basieren zu lassen. Das Team hat das erreicht, indem es die Anforderungen formuliert hat, ohne die Implementierung des Codes durch eine vorgefertigte Lösung zu verderben. Der Zweck der Erhebung der Anforderungen besteht darin, sich den Raum der möglichen Lösungen zu erschließen [GW89], also alle möglichen Lösungen zu finden, die die Geschäftsbedingungen erfordern. Im Prozess des Entwurfs sucht der Designer nach dem besten Kompromiss zwischen den möglichen Lösungs-Parametern. Das Team bei der Flughafenbetreibergesellschaft Blaport AG hat dies durch Abstraktion der gefundenen Beispiele erreicht, ohne eine bestimmte Implementierung der Benutzeroberfläche im Hinterkopf zu haben. Durch Austausch der Step-Definitionen konnten die Beispiele sogar mit einer neuen Benutzeroberfläche verknüpft werden. Da die Geschäftsbedingungen sich unter einer neuen Benutzeroberfläche vermutlich nicht geändert

hätten, sind die Beispiele, die auf den Geschäftsbedingungen beruhen immer noch gültig.

Tabellenformate

Einer der populärsten Ansätze, die in Tabellenform zum Einsatz kommen, ist das Framework for Integrated Tests, FIT [MC05]. Robert C. Martin hat 2008 die Simple List Invocation Method (SLiM) vorgestellt [Mar08b], die mit einer ähnlichen Tabellenstruktur zum Ausdruck der Anforderungen arbeitet. Bei beiden Ansätzen gibt es drei verbreitete Stile: Tabellen, die Eingaben annehmen und die Ausgaben aus dem System überprüfen, Tabellen, die das System durchsuchen und Collections von Werten mit denen aus dem getesteten System vergleichen, und Tabellen, die Workflows unterstützen.

Entscheidungstabellen Die Tabellen der ersten Gruppe nehmen Eingabewerte in das System auf, führen irgendetwas im System aus und vergleichen dann einige Werte aus dem System mit den angegebenen. Solch ein Beispiel hatten wir beim Ampel-Beispiel in SLiM, wo diese Entscheidungstabellen genannt wurden. In FIT heißen diese Tabellen ColumnFixture und in FitLibrary, einer Erweiterung von FIT, heißen sie CalculateFixture.

Die meisten Anforderungen können Sie im Entscheidungstabellen-Format festhalten. Die Tabellen für den Flughafen-Parkplatz sind beispielsweise Entscheidungstabellen. Die Parkdauer ist die Eingabe in das getestete System, die Parkgebühren stellen den Ausgabewerte aus dem System dar, die der Test-Runner überprüft. Listing 10–2 zeigt die Beispiele für das Valet-Parking als Entscheidungstabelle formatiert. Kommt Sie Ihnen bekannt vor?

```
1  !|Parkgebuehren fuer   |Valet-Parking   |
2  |Parkdauer             |Parkgebuehren?  |
3  |30 Minuten            |12,00 €         |
4  |3 Stunden             |12,00 €         |
5  |5 Stunden             |12,00 €         |
6  |5 Stunden, 1 Minute   |18,00 €         |
7  |12 Stunden            |18,00 €         |
8  |24 Stunden            |18,00 €         |
9  |1 Tag, 1 Minute       |36,00 €         |
10 |3 Tage                |54,00 €         |
11 |1 Woche               |126,00 €        |
```

Listing 10–2 *Die Valet-Parking-Tests als Entscheidungstabelle in SLiM dargestellt*

Hier gibt es keine Beschränkungen auf einen Eingabewert oder bei der Überprüfung eines Ausgabewertes. Es ist sogar so, dass Sie, wenn Sie jegliche Ausgabewerte weglassen, eine spezielle Tabelle zum Aufstellen der Werte im System bekommen. Das kann sich als sehr günstig erweisen, wenn Sie drei Benutzerkonten beschreiben, mit denen Ihre Tests später arbeiten. Siehe Listing 10–3 mit

einem Beispiel. Diese besondere Form einer Entscheidungstabelle nennt man oft Setup-Tabelle oder SetUpFixture.

```
1 !|Kontogenerierer                          |
2 |Kontoname        |Kontostatus |Funktion    |
3 |Susi Service     |aktiv       |Service-User|
4 |Tim Techniker    |aktiv       |Techniker   |
5 |Arne Arbeitslos  |arbeitslos  |Service-User|
```

Listing 10–3 *Eine Setup-Tabelle, die drei verschiedene Konten vorbereitet*

Auch wenn es keine Begrenzung hinsichtlich der Spaltenanzahl Ihrer Tabelle gibt, so leidet die Lesbarkeit deutlich, wenn man mit einer Tabelle konfrontiert ist, die 10 Spalten übersteigt. Mir sind schon Tests untergekommen, die um die 30 Spalten mit hunderten von Zeilen enthielten.[1] Das Problem bei solchen Tests ist ihre schwierige Verständlichkeit. Jemand, der später die Tests wartet – und vergessen Sie nicht: Das könnten auch Sie selbst sein – würde wohl kaum in wenigen Sekunden erfassen können, worum es in einer bestimmten Zeile geht. Diesen Testsmell sollten Sie vermeiden.[2]

Abfragetabellen Abfragetabellen werden für Collections aus dem System gebraucht. In SLiM heißen sie Query-Tables, in FIT RowFixture und in FitLibrary gibt es ArrayFixture, SetFixture und SubsetFixture. Oft muss man die Reihenfolge der Einträge in der Collection überprüfen oder auf Untergruppen der Einträge in einer Collection achten. In SLiM tragen diese häufig den Präfix Subset oder Ordered, um spezielle Varianten zu ergeben. In FitLibrary gibt es unterschiedliche Klassen, von denen Sie Unterklassen ableiten müssen. In Listing 10–4 ist ein Beispiel zu sehen, das alle User in einem System überprüft.

```
1  !|Query: Benutzer im System                          |
2  |Benutzername          |Funktion       |Kontostatus |
3  |Tim Tester            |Tester         |aktiv       |
4  |Paul Programmierer    |Programmierer  |aktiv       |
5  |Petra Projektmanager  |Projektmanager |aktiv       |
```

Listing 10–4 *Eine Abfragetabelle, die die Existenz vorher eingerichteter Daten prüft*

Abfragetabellen werden oft bei der Sammlung von Daten aus dem System benutzt und enthalten eine Liste mit verschiedenen Dingen, die im System gespeichert sind. Sie erweisen sich als sehr nützlich bei einer Collection von Konten im System, etwa nach einem Such-Arbeitsschritt. Bei einer Online-Shopping-Seite können Sie den Warenkorb Ihres Kunden abfragen und den Inhalt mit einer erwarteten Liste vergleichen.

1. Um einige Bäume zu retten, habe ich dafür kein Beispiel angefügt. Auch möchte ich Sie durch ein solch abschreckendes Beispiel nicht verführen.
2. Aus »Refactoring: Improving the Design« of Existing Code [FFB+99] wo Fowler und Beck von »Design smells« sprechen, wenn mit dem zugrunde liegenden Entwurf etwas nicht ganz stimmt. Der Begriff »smell« kommt auch in »xUnit Test Patterns: Refactoring Test Code« vor [Mes07].

Sie können die Einträge in der Collection mit nur einem ihrer Merkmale, wie etwa dem Namen der Ware oder einer Kombination aus Namen, Preis und Menge bei einem Warenkorb vergleichen. Normalerweise drückt die Menge der Merkmale, die in der Tabelle angezeigt sind, aus, welche Merkmale überprüft werden. Die Detailgenauigkeit Ihrer Tabellen bestimmt dann, wie gründlich Sie die Daten des Systems überprüfen wollen.

Je mehr Daten Sie in der Tabelle hinterlegen, desto mehr Anpassungsmöglichkeiten haben Sie, falls sich der Name eines Merkmals ändert. Beim Entwurf Ihrer Tests gibt es daher immer einen Kompromiss zwischen der Gründlichkeit der Tests der Anwendung und dem Wartungsaufwand Ihrer Testsuite. Einerseits möchten Sie vielleicht weniger Details in Ihren Abfragetabellen haben, was zu einem erhöhten Risiko führt, dass Sie etwas kaputt machen, ohne dass dieser Test es bemerkt. Dieses Risiko können Sie mindern, indem Sie der Testsuite mehr Beispiele von bisher ausgelassenen Details hinzufügen oder in dem Sie etwas Zeit für exploratives Testen einplanen. Natürlich hat jede dieser Entscheidungen ihre eigenen Nebenwirkungen. Ein anderer denkbarer Weg wäre, alle nur erdenklichen Details mit aufzunehmen. Dies kann aber, etwa bei Umbenennungen der Merkmale, zu fragilen Tests führen. Zwischen diesen beiden Extremen finden sich allerdings Ansätze, die dieses Problem lösen können. Das bedeutet aber auch, dass Sie die Anzahl der Details in Ihrer Tabellenstruktur in ihrer jeweiligen Situation festlegen müssen.

Vermutlich ist es am einfachsten, wenn Sie zunächst Ihrem Bauchgefühl vertrauen und später von Zeit zu Zeit Ihre Entscheidung revidieren. Wenn Sie dann andauernd Tests in Ihrer Testsuite warten müssen, sollten Sie noch einmal auf Ihre Entscheidung zurückkommen und Ihre Tests refaktorisieren.

Script-Tabellen Mit Script-Tabellen können Sie die Workflows in Ihrem System ausdrücken. Sie können Script-Tabellen auch nutzen, um in einem Test Entscheidungs- mit Abfragetabellen zu kombinieren. Eine Alternative zum *Gegeben sei-Wenn-Dann*-Stil von BDD im Tabellenformat ist ein Muster, das man Setup-Execute-Verify oder Arrange-Act-Assert nennt. Bei Setups verwende ich oft Entscheidungstabellen ohne überprüfte Ausgaben, um jede Vorbedingung im System einzurichten. Anschließend rufe ich einen einzigen Schritt einer Script-Tabelle auf, etwa den Aufruf der Bilanz eines Kontos, das ich gerade eingerichtet habe. Als letzten Verifizierungsschritt kann ich die Abfragetabelle benutzen, um alle Bilanzen, die unter meinem Konto gespeichert sind, zu sammeln und mit den Werten nach dem Wiederauffüllen zu vergleichen. Ein vollständiges Beispiel sehen Sie in Listing 10–5.

```
1  !|script|Kontowiederauffuellung|
2
3  !|Kontogenerierer          |
4  |Kontoname       |Tarif    |
5  |Prepaid-Konto   |Prepaid  |
```

```
6
7  |ueberweise|50.00 EUR|auf|Prepaid-Konto|
8
9  !|Query:BilanzChecker  |Prepaid-Konto  |
10 |Bilanzname            |Bilanzwert     |
11 |Hauptbilanz           |50.00 EUR      |
12 |Bonusbilanz           |5.00 EUR       |
```

Listing 10–5 *Eine Script-Tabelle mit einem vollständigen Ablauf durch das System*

Script-Tabellen aus SLiM heißen in FIT ActionFixture oder DoFixture in FitLibrary. Mit ihnen können Sie in Ihrer Test-Tabelle alles Mögliche ausdrücken. Die Tools benutzen eine Konvention beim Aufruf bestimmter Funktionen in Ihrem Glue-Code basierend darauf, wie Sie Ihre Beispiele geschrieben haben. Dies hängt maßgeblich von dem jeweiligen Tool ab, das Sie benutzen. Meistens gibt es einen Weg, wie man die Parameter der Funktionen vom Text trennt. Der Name der aufzurufenden Funktion ergibt sich, indem der Text mit Binnenmajuskeln (engl.: camel case) zusammengefügt wird[3].

Script-Tabellen sind immer dann angebracht, wenn Sie größere Gruppen von Arbeitsschritten ausdrücken wollen. Sie können etwa das Starten eines Systems, die Eingabe von Werten, das Klicken eines Buttons und dann die Überprüfung einiger Werte vor dem finalen Herunterfahren des Systems mit Script-Tabellen verfassen. Vermutlich möchten Sie öfters eine Kombination unterschiedlicher Tabellen in Ihren Tests verwenden.

Keyword-getriebene Automatisierung

Mit Keywords können Sie Kombinationen von Arbeitsschritten in einem einzigen zusammenfassen. Der Arbeitsschritt, der mehrere Keywords kombiniert, wird dann zu einem Arbeitsschritt auf höherer Ebene. Sie können mehrere Keyword-Ebenen virtuell kombinieren, um die richtige Abstraktion für Ihre Tests zu erhalten.

Im Beispiel mit den Ampeln haben wir das Szenario »ungültige Kombination« benutzt, um ein Konzept auf höherer Ebene in einem eigenen Keyword auszudrücken (siehe Listing 10–6). Im Test-Report konnten wir die Spur dieses Keywords verfolgen. Ein weiteres, auf Keywords beruhendes Framework ist Robot Framework.[4] Robot Framework ist mit zahlreichen Test-Bibliotheken ausgestattet und bietet Erweiterungen für webbasierte Tests, SSH-Unterstützung, Datenbanken und Swing.

---

3. Camel-cased bezieht sich auf die Methode, die Fließtext in Funktionsnamen verkettet, indem die Leerezeichen weggelassen werden. Aus `Das macht Spass` wird dann als Funktionsname `dasMachtSpass`.
4. *http://robotframework.org*

```
1  ...
2  !2 Ungueltige Kombinationen
3
4  !|scenario          |ungueltige Kombination |firstLight||secondLight  |
5  |set erste Ampel    |@firstLight                                      |
6  |set zweite Ampel   |@secondLight                                     |
7  |execute                                                              |
8  |check              |erste Ampel           |Gelb blinkend             |
9  |check              |zweite Ampel          |Gelb blinkend             |
10
11 !|script|FirstLightSwitchingCrossingController|
12
13 !|ungueltige Kombination      |
14 |firstLight      |secondLight  |
15 |Grün            |Rot, Gelb    |
16 |Grün            |Grün         |
18 |Gelb            |Rot, Gelb    |
19 |Gelb            |Grün         |
20 |Gelb            |Gelb         |
21 |Rot, Gelb       |Rot, Gelb    |
22 |Rot, Gelb       |Grün         |
23 |Rot, Gelb       |Gelb         |
24 |Rot             |Rot, Gelb    |
25 |Rot             |Grün         |
26 |Rot             |Gelb         |
27 |Gelb blinkend   |Rot          |
28 |Gelb blinkend   |Rot, Gelb    |
29 |Gelb blinkend   |Grün         |
30 |Gelb blinkend   |Gelb         |
```

Listing 10–6 *Der Gebrauch einer Szenario-Tabelle als Keyword im Ampel-Beispiel (Wiederholung von Listing 8–15).*

Bei der Nutzung von Keywords verschwimmt die Grenze zwischen Testdaten-Repräsentation und Testautomatisierungs-Code. Sie haben vielleicht Keywords in einer Programmiersprache implementiert, die die Anwendung betreibt oder auf niedrigerer Ebene mit dem Betriebssystem interagiert. Sie haben gleichfalls vielleicht Keywords, die aus einer Kombination von Keywords niedrigerer Ebenen aufgebaut sind. Und Sie können Keywords höherer Ebenen auch auf neue Art und Weise kombinieren. Durch diesen Aufbau bekommen Sie mehrere Ebenen von Keywords.

Dieser Ansatz ist sehr wirkungsvoll bei der Organisation Ihrer Testdaten um eine Gruppe von Testautomatisierungs-Treibern auf niedrigen Ebenen. Durch neue Kombinationen von Keywords entwickelt sich die Sprache Ihres Testautomatisierungs-Codes weiter und wird dadurch noch wirkungsvoller. Der Nachteil besteht darin, dass dieser Ansatz zu tiefreichenden und komplexen Ebenen der Testautomatisierungs-Keywords führen kann. Als dieses Buch geschrieben wurde, standen kaum Refaktorisierungs-Tools für Testautomatisierungs-Key-

words zur Verfügung. Die Wartung mehrerer Ebenen von Keywords kann da schon mal im Chaos enden. Auch wenn es recht einfach sein sollte, Text in einigen Dateien zu durchsuchen und zu ersetzen, so macht der flüssige Stil dieser Testdaten es doch sehr kompliziert, jedes Vorkommen aufzufinden. Ein Keyword umzubenennen, zum Beispiel, oder die Parameterliste eines Keywords zu erweitern, wird dann schwierig. Die meisten modernen IDEs für Programmiersprachen unterstützen diese Refaktorisierungen jetzt auf zuverlässige Weise. Langfristig werden wir hoffentlich auch für die Keyword-getriebene Automatisierung noch mehr solcher Refaktorisierungs-Tools vorfinden – eventuell sogar zur gleichzeitigen Refaktorisierung über niedrigere und höhere Ebenen der Keywords hinweg.

Glue-Code und Support-Code

Wie Ihre Beispiele automatisiert werden, hängt stark vom benutzten Framework ab. Einige Java-Frameworks stützen sich zum Beispiel auf Java-Annotations, um den Code zur Ausführung einer gegebenen Phrase eines Beispiels zu finden. Einige der Frameworks unterstützen Bean-Methoden – Getters und Setters – für Eigenschaften in Ihren Klassen, wie Vorname und Nachname bei einem Benutzerkonto.

Unabhängig vom Framework oder Tool, das Sie zur Automatisierung Ihrer Beispiele einsetzen, sollten Sie die Entwicklung Ihres Glue-Codes berücksichtigen. Einige Teams scheinen sich der Tatsache nicht bewusst zu sein, dass sie Code entwickeln, wenn sie Glue- und Support-Code schreiben. Wenn Sie Ihre Anwendung mit den Tests verschalten, mag der Code noch einfach und leicht wartbar erscheinen. Mit der Zeit wird die anwachsende Code-Basis aber schwierig zu warten, wenn Sie nicht ähnliche Prinzipien beim Entwurf Ihres Glue- und Support-Codes anwenden, wie Sie es bei Ihrem Produktionscode tun würden.

Für mich heißt das, in der Regel von einer einfachen Code-Basis auszugehen. Ich möchte in der Lage sein, mit automatisierten Akzeptanz-Tests schnell beginnen zu können. Bei einem Kunden konnten wir einmal nach ungefähr zwei Stunden Arbeit zu zweit mit einigen automatisierten Tests auf der Akzeptanz-Ebene aufwarten. Nach zwei weiteren Stunden hatten wir eine grundsätzliche Vorstellung des Treiber-Codes für unsere automatisierten Tests sowie eine grundlegende Glue-Code-Ebene, durch die wir die meisten unserer Akzeptanztests laufen lassen konnten. Uns war bewusst, dass dieser Code später erweitert würde, sobald wir auf komplexere automatisierte Tests stoßen würden.

Wenn mit der Zeit mehr und mehr Funktionalität anwächst, denke ich aktiv an die Extraktion neuer Komponenten aus dem vorhandenen Code. Wie wir schon in Teil II beim `LightStateEditor` gesehen haben, haben wir diese Komponenten mit testgetriebener Entwicklung hinzugefügt. Wir haben, genau genommen, unseren Test-Code getestet. In einer Firma, in der ich mal gearbeitet habe, haben wir für unseren Automatisierungs-Code mehrere Code-Metriken wie statische Code-Analyse und Code-Abdeckung angewandt. In unserem Versionskont-

rollsystem hatten wir ein eigenständiges Projekt mit einem eigenen Build. Als wir die Metriken mit denen unseres Produktions-Codes verglichen, hatten wir mit unserem Test-Code den Produktionscode übertroffen. Noch zwei Jahre später war dieser Test-Code in Gebrauch und Entwicklung.

Wenn Ihnen das etwa extrem vorkommt, hören Sie mal die Geschichte, die mir 2009 zu Ohren gekommen ist. Das Team bekam Besuch von einem externen Auftragnehmer für Software-Testautomatisierung. Dem Auftragnehmer war nicht wohl dabei, immer mehr Support-Code mit komplizierter Logik hinzufügen. Also nutzte er testgetriebene Entwicklung für den komplexen Code, den er hinzufügte. Nachdem der Auftragnehmer wieder gegangen war, war die Code-Abdeckung für die ganze Anwendung von 10% auf insgesamt 15% gestiegen.

Die Entwicklung von Testautomatisierungs-Code ist Software-Entwicklung. Daher sollten Sie bei Ihrem Glue- und Support-Code die gleichen Prinzipien anwenden wie bei Ihrem Produktionscode. Früher habe ich Testautomatisierungs-Code gesehen, dem das Design fehlte, der nicht ordentlich dokumentiert und manchmal sogar fragil und fehlerhaft war. Als wir bei einer Firma einen neuen Testautomatisierungs-Ansatz entwickelten, haben wir durch schrittweisen Entwurf bei testgetriebener Entwicklung alle diese Nachteile überwinden können. testgetriebene Entwicklung hat uns beim Verständnis des Codes geholfen und uns gleichzeitig die notwendige Dokumentation verschafft. Durch TDD hatten wir auch sichergestellt, dass unser Testautomatisierungs-Code nicht unter Fehlern leidet und dadurch nicht zu falschpositiven Tests – Tests, die durchlaufen, obwohl die Software kaputt ist – oder falschnegativen Tests – Tests, die nicht durchlaufen, obwohl die Software in Ordnung ist – führt, während unsere automatisierten Tests liefen.

Die Notwendigkeit des Zufügens von Unit-Tests zu unserem Glue- und Support-Code wird klar, wenn Sie ATDD anwenden, wie wir es in Teil II getan haben. Beim Arbeiten mit den Akzeptanztests haben wir uns die Domäne der Anwendung erschlossen. Mit der Kenntnis über und der Erfahrung mit der Domäne fiel es uns leicht, den Domänen-Code vom Glue-Code zu extrahieren bzw. abzuleiten. Ohne die Unit-Tests für den Glue-Code hätten wir damals weitere Unit-Tests entweder zunächst durch Extraktion und anschließendes Hinzufügen der Tests zugefügt oder durch die Entwicklung eines neuen Domänen-Codes mit testgetriebener Entwicklung.

Das richtige Format

Ein paar Worte will ich noch über das richtige Format loswerden. Die Formate, die ich hier präsentiert habe, Verhaltensgetriebene Entwicklung, Tabellenformate und Keyword-basierte Beispiele stehen für die Vielzahl der Formate, die während der Erstellung dieses Buchs verfügbar waren. Im letzten Jahrzehnt haben die heute verfügbaren Tools eine Weiterentwicklung durchlaufen, um Tests in jedem dieser Formate zu ermöglichen. Sie können zum Beispiel ein Tabellenformat mit

dem SLiM-Framework von FitNesse mit Entscheidungs- und Abfrage-Tabellen einsetzen. Alternativ können Sie das *Gegeben sei-Wenn-Dann*-Format unter Einsatz von Script-Tabellen in SLiM benutzen. Im Robot Framework gibt es ähnliche Wege, wie man eines der drei angeführten Formate nutzen kann. Für jede bestimmte Kombination von Testdaten-Formaten – BDD, Tabellen oder Keywords – und Testautomatisierungs-Sprache existiert vermutlich mindestens ein Framework, mit dem Sie loslegen können.

Das ist natürlich toll, da Sie Ihre Aufmerksamkeit auf die Nutzung des richtigen Formats lenken, Ihre Beispiele für Ihre Anwendung im naheliegendsten Stil ausdrücken und die Entscheidung für das Tool aufschieben können. Die meisten Teams, die mit der Implementierung eines bestimmten Tools beginnen, enden genau damit: mit der Implementierung eines Tools, nicht aber der Implementierung eines erfolgreichen Ansatzes [Adz11]. Offensichtlich bedeutet ein automatisiertes Test-Tool noch lange nicht die Existenz einer entscheidenden Teststrategie [KBP01].

Eins gilt für Ihre Beispiele wie auch für Ihre Tests. Wenn sie erst einmal automatisiert sind, gibt es mehrere Parteien. Da wäre als Erstes der Programmierer des Features, der das Beispiel aus der schriftlichen Beschreibung erfassen muss. Dann gibt es noch den Programmierer, der dieses spezielle Beispiel automatisieren wird. Ihr Kunde oder Product-Owner wird den Test, wenn er erst einmal automatisiert wurde, ebenfalls durchlesen. Ihr gesamtes Team wird sich die Tests auch noch einmal ansehen, sobald die nächste Erweiterung in mehreren Iterationen geplant wird, um die Nebenwirkungen zu verstehen. Schlussendlich wäre da noch derjenige, der das System zukünftig wartet und die Testfälle, falls nötig, anpasst.

Alle diese Parteien sollen in der Lage sein, den Inhalt Ihres Beispiels und Ihrer Tests zu verstehen. Damit sie das können, müssen sie die Tests lesen und deren Zusammenhang und spezifischen Fokus erfassen. Je länger dieses Lesen und Verstehen eines Tests dauert, desto mehr Zeit wird man im Sinne des Lean-Thinking verlieren. Aus systemischer Sicht führt verschwendete Zeit zu weniger Zeit für produktives Arbeiten. Je weniger produktives Arbeiten, desto mehr Druck von außen fühlt man auf sich lasten. Je mehr Druck man verspürt, desto eher nimmt man bei den Tests Abkürzungen. Je mehr dieser Abkürzungen man nimmt, desto unverständlicher werden die Tests (siehe Abb. 10–1). An dieser Stelle hat man dann einen Teufelskreis oder eine Abwärtsspirale. Sie scheinen verloren.

Sie können diesen Teufelskreis durchbrechen, indem Sie Ihre Rolle in diesem System erkennen. Offensichtlich gibt es einen bestimmten Entscheidungspunkt in diesem Kreis, von dem abhängt, ob das beschriebene System ein Teufelskreis oder eine positive Feedback-Schleife wird, die zu einem ausgeglichenen System führt: der Entscheidung, mit der Lektüre von Akzeptanztests Zeit zu verschwenden [Wei91].

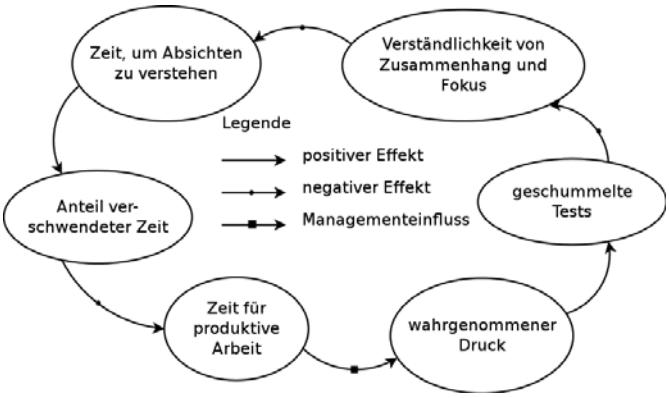

Abb. 10–1 *Ein systemischer Blick auf absichtsverratende Tests*

Wenn Sie diese Entscheidung rückgängig machen, verwenden Sie weniger Zeit auf das Verstehen Ihrer Tests. Je weniger Zeit Sie darauf verwenden, desto mehr Zeit haben Sie für produktive Arbeit. Das nimmt Ihnen den Druck von außen und verhindert, dass Sie bei Ihren Tests Abkürzungen nehmen.

Teams nutzen die Akzeptanztests als Kommunikationsmittel. Die Unterschiedlichkeit der Parteien erklärt das. Während die Anwendung in der Entwicklung ist, gibt es laufend Übergaben von einem Team-Mitglied zum nächsten. Erfolgreiche Teams sorgen daher für eine gute Lesbarkeit ihrer Tests, damit die Reibungsverluste, die solche Übergaben mit sich bringen, minimiert werden. 2009 hat mir Enrique Comba-Riepenhausen erzählt, dass ein vor Ort anwesender Kunde nicht nur die Akzeptanztests lesen konnte, sondern auch seine Unit-Tests und sogar seinen Domänen-Code. Sein Team hatte es offenbar geschafft, erfolgreich das Konzept zu implementieren, das Eric Evans einmal »allgegenwärtige Sprache« genannt hat [Eva03].

Jedes Projekt hat seine gemeinsame, eine allgegenwärtige Sprache. Je mehr Übersetzungen zwischen verschiedenen Sprachen in einem Projekt notwendig werden, desto mehr Potenzial für Verwirrung gibt es. Wenn die Business-Domäne mit den gleichen Begriffen in den Akzeptanztests zum Ausdruck kommt, sorgen Sie auf diese Weise dafür, dass Ihr Kunde die Tests ebenfalls versteht. Wenn Sie dann auch noch den Domänen-Code um die gleiche Sprache aufbauen, stellen Sie sicher, dass jeder zum gleichen Verständnis kommt und Missverständnisse vermieden werden, bevor es zu spät ist.

Verfeinern Sie Ihre Beispiele

Kernbeispiele von Ihrem Kunden oder Product-Owner zu bekommen, hilft Ihnen, dass Sie mit der Funktionalität in die Gänge kommen. Leider liefert Ihnen eine Besprechung wie in Teil I nicht immer alle Beispiele, die Sie benötigen, um die

Software zu bauen. Sie müssen dann Ihre Beispiele, nachdem Sie eine erste Gruppe ausgemacht haben, verfeinern [Adz11].

Die Verfeinerung der Beispiele kann auf mehrfache Weise geschehen. Im Regelfall wissen die Tester, wie man versteckte Annahmen und Grenzbedingungen aus den ersten Beispielen ausarbeitet. Je nach Domäne der Anwendung können Beschränkungen in der String-Länge unterschiedliche Validierungsregeln nach sich führen oder sich gegenseitig ausschließende Geschäftsbedingungen auftreten. Wenn Sie eine dieser Bedingungen im ersten Spezifikationsworkshop verpassen, lohnt es sich, den Tester zu bitten, die Beispiele zu verfeinern. Ein anderer Fall der Verfeinerung von Beispielen kann entstehen, wenn es eine Geschäftsbedingung gibt, über die sich der Product-Owner noch unsicher ist. Wenn der Product-Owner die zugrunde liegende Geschäftsbedingung geklärt hat, tun sich Programmierer und Tester zusammen, um die Beispiele aus der ersten Sitzung zu verfeinern. Mit der Zusatzinformation sind sie dann in der Lage, die grob gehaltenen Beispielen zu vollständigerer Funktionalität zu erweitern.

Als Tester wenden Sie wie selbstverständlich Grenzbedingungen, Domänen-Partitionierung oder Äquivalenz-Klassen zur Verfeinerung der Beispiele an. Eigentlich kann jede Test-Technik beim Verfeinern der Beispiele und der Suche nach Lücken hilfreich sein. Im nächsten Abschnitt folgt eine kurze Abhandlung von Test-Techniken, die Sie vielleicht in Betracht ziehen möchten, wenn Sie Ihre Beispiele verfeinern. Eine vollständige Diskussion dieser Techniken übersteigt den Rahmen dieses Buchs bei Weitem. Für weitere Einzelheiten über Test-Techniken lesen Sie bitte *A Practitioner's Guide to Software Test Design* von Lee Copeland [Cop04] oder *Testing Computer Software* von Cem Kaner [KFN99].

Domänen-Testen

Beim Domänen-Testen werden die Tests ausgehend von der Domäne in kleinere Unterdomänen unterteilt. Die Werte, mit denen das System gespeist wird, werden in unterschiedliche Klassen partitioniert, innerhalb derer sich das System gleich oder äquivalent verhält. Diese Klassen nennt man daher Äquivalenz-Klassen. Im Flughafen-Beispiel konnten wir das beobachten. Die erste Äquivalenzklasse bestand in der Aufteilung in die diversen Parkplätze. Die nächsten Klassen leiteten sich aus den Gebührenordnungen ab. Meist hatten wir eine Klasse für die ersten paar Stunden, eine Klasse für die ersten paar Tage und schließlich eine Klasse für mehrere Wochen. Innerhalb dieser Klassen hatten wir uns ein Beispiel an der Grenze, eins genau in der Mitte und eins genau an der nächsten Grenze herausgesucht und sie zu einer Gruppe von Beispielen zusammengestellt.

Denken wir noch einmal an das Beispiel des Langzeit-Parkplatzes im Freien zurück. Die Gebührenordnung sah 2,00 € für die erste Stunde, 10,00 € als Tageshöchstsatz und 60,00 € pro Woche vor. Die erste Äquivalenz-Klasse innerhalb dieses Parkplatzes ist das Verhalten bis zur fünften Stunde des ersten Tages. Wir wählten Beispiele für eine, drei und fünf Stunden aus. Von der fünften Stunde an

betrachten wir die zweite Äquivalenz-Klasse bis zum Ende des ersten Tages. Wir hätten alternativ fünf Stunden und eine Minute, 10 Stunden und 24 Stunden aus dieser Klasse wählen können. Für den zweiten Tag hätten wir ähnliche Werte heraussuchen können und gesehen, ob die Bedingungen auch dann gelten. Dies wäre dann eine Kombination der ersten beiden Äquivalenz-Klassen gewesen. Allerdings hätte dann das Verhalten am zweiten, dritten oder vierten Tag in dieser Klasse das gleiche sein sollen. Wenn wir schließlich am Ende des sechsten Tages das wöchentliche Maximum erreicht hätten, wäre eine neue Äquivalenzklasse für den ganzen siebenten Tag nötig gewesen, die wiederum mit mehreren Wochen kombiniert worden wäre.

Wenn wir uns noch einmal die Ergebnisse nach dem Workshop anschauen, gaben die Werte diese Partitionierung wieder (siehe Tab. 10–1). Die ersten fünf Werte fallen in die erste Äquivalenzklasse der stündlichen Tarife. Die nächsten vier Darstellungen stehen für Beispiele aus der zweiten Äquivalenz-Klasse, manchmal in Kombination mit der ersten der stündlichen Tarife. Die letzten sechs Darstellungen stehen für Beispiele aus der dritten Äquivalenz-Klasse, manchmal in Kombination mit den ersten beiden.

Parkdauer	Parkgebühren
30 Minuten	2,00 €
1 Stunde	2,00 €
5 Stunden	10,00 €
6 Stunden	10,00 €
24 Stunden	10,00 €
1 Tag, 1 Stunde	12,00 €
1 Tag, 3 Stunden	16,00 €
1 Tag, 6 Stunde	20,00 €
6 Tage	60,00 €
6 Tage, 1 Stunde	60,00 €
7 Tage	60,00 €
1 Woche, 2 Tage	80,00 €
3 Wochen	180,00 €

Tab. 10–1 *Beispiele des Langzeitparkens auf dem Parkplatz im Freien nach Ende des Workshops*

Grenzwerte

Das Testen der Grenzwerte wird sehr viel leichter, wenn es zuvor mit der Domänen-Analyse kombiniert wird. Die Motivation dahinter ist die, dass Fehler an den Grenzen der Äquivalenzklassen wahrscheinlicher auftreten. Der klassischen The-

orie nach sollte man daher einen Wert genau vor der, einen auf der und einen von der Grenze entfernt testen. Das sieht mir nach einfacher Mathematik aus, und wenn ich mir die Beispiele von vorher so anschaue, scheinen wir das mit etwas Spielraum ganz vernünftig angewandt zu haben.

Denken wir noch einmal an das Flughafen-Beispiel zurück. Im Falle der Valet-Parking-Gebühren gab es zwei Äquivalenz-Klassen – eine, bei der Sie 18,00 € gezahlt hätten und eine weitere, bei der Sie 12,00 € zahlen mussten. Der Grenzwert lag bei fünf Stunden. Mit dieser Information sollten wir also einen Test für genau fünf Stunden (dem Grenzwert), einen für weniger als fünf Stunden (z.B. vier Stunden und 59 Minuten) und einen knapp über diesem Grenzwert (z.B. fünf Stunden und eine Minute) erstellen (siehe Tab. 10–2).

Parkdauer	Parkgebühren
4 Stunden, 59 Minuten	12,00 €
5 Stunden	12,00 €
5 Stunden, 1 Minuten	18,00 €

Tab. 10–2 *Grenzwerte beim Valet-Parking*

Natürlich gibt es noch mehr (versteckte) Grenzwerte, wie etwa die Parkdauer von null Minuten. Das besondere Verdienst von Testern bei Spezifikationsworkshops zeigt sich beim Erkennen solcher versteckter Grenzwerte und Äquivalenz-Klassen, die ansonsten bei der Entwicklung vermutet wird, aber sich im Business so nicht wiederfindet.

Paarweises Testen

Das paarweise Testen gründet sich auf der Beobachtung, dass bei einer Kombination von zwei unterschiedlichen Eingabewerten oft Fehler passieren. Wenn Ihre Anwendung eine Eingabe A mit den Werten a1, a2 und a3, sowie eine Eingabe B mit den Werten b1, b2 und b3 aufweist, kann man mit den Testfällen in Tabelle 10–3 einen hohen Grad von Zuversicht erzielen.

Wenn wir unserem System eine dritte Variable C mit den Werten c1 und c2 hinzufügen, bekommen wir die Testfälle in Tabelle 10–4.

A	B
a1	b1
a1	b2
a1	b3
a2	b1
a2	b2
a2	b3
a3	b1
a3	b2
a3	b3

Tab. 10–3 *Beispiele für paarweises Testen*

A	B	C
a1	b1	c1
a1	b2	c2
a1	b3	c2
a2	b1	c1
a2	b2	c1
a2	b3	c2
a3	b1	c2
a3	b2	c1
a3	b3	c1

Tab. 10–4 *Paarweises Testen mit drei Variablen*

Wochen	Tage	Stunden
0	0	0
1	0	1
3	0	3
1	1	0
0	1	1
0	1	3
3	1	5
3	3	0
0	3	1
1	3	3
3	6	1
0	6	5
1	6	6
1	7	5
0	7	6
3	7	0
3	0	6
1	0	5
3	1	6
0	3	5
1	3	6
3	6	0
0	6	3
3	7	1
1	7	3

Tab. 10–5 *Beispiele für paarweises Testen beim Parkplatz-Beispiel*

Der Algorithmus sorgt dafür, dass alle möglichen Kombinationen jedes Paares als Testfall vorkommen. Bei kombinatorischen Problemen hilft dieser Ansatz bei der Reduktion der Tests, die Sie laufen lassen müssen und trotzdem eine hohe Abdeckung erreichen.

Beim Flughafenparkplatz-Beispiel konnten wir diesen Ansatz auf die Äquivalenzklassen anwenden. Zum Beispiel könnte ich die Beispiele für 0, 1 und 3 Wochen, für 0, 1, 3, 6 und 7 Tage und für 0, 1, 3, 5 und 6 Stunden nehmen. Der paarweise Algorithmus ergibt dann die Testfälle in Tabelle 10–5.

Sie werden vielleicht festgestellt haben, dass wir noch die Ausgabewerte für die Parkgebühren berechnen müssen. Das überlasse ich dem Leser gern als Übung.

Kürzen Sie die Beispiele

Mit der Zeit werden Ihre Testsuiten größer und größer. Anscheinend gibt es zwei magische Grenzen, je nach dem, wie schnell Ihre Testsuiten durchlaufen.

Die erste Grenze ist erreicht, wenn die Gesamtdauer mehr als einige Minuten beträgt. Bevor es soweit ist, fühlen Sie sich gut, da Sie von Ihrer Testsuite regelmäßig Rückmeldung erhalten. Sie führen Ihre Tests fast so häufig wie Ihre Unit-Tests durch. Wenn dann aber Ihre Akzeptanztests so langsam in den zweistelligen Minutenbereich kommen, beginnen die langen Laufzeiten langsam, Sie zu stören. Sie lassen die Tests dann unregelmäßiger laufen, meistens beim Check-in, aber selten öfter. Dadurch verschleppen Sie aktiv die Rückmeldung Ihrer Tests. Dies ist mit dem Risiko von Bugs in Ihrer Code-Basis verbunden, die für einige Zeit unerkannt bleiben können. Bei der Arbeit in einem Entwickler-Team kann es einige Verwirrung stiften, wenn jemand fehlerhaften Code in seine oder ihre neuesten Änderungen einbaut.

Die zweite magische Grenze besteht bei einer Laufzeit von zwei oder drei Stunden für Ihre Regressionstestsuite. Lisa Crispin hat einmal zu diesem Thema gesagt, dass ihr Team bestrebt sei, seine automatisierten Nicht-Unit-Tests unter einer 45-Minuten-Grenze zu halten. Anderenfalls schenkten die Programmierer den Testergebnissen nicht die nötige Aufmerksamkeit. Bis zu dieser Grenze kann man die Tests sicher ausführen, darüber hinaus scheint eine Degenerierung der Tests einzusetzen. Immer mehr Tests versagen dann im Versionskontrollsystem [DMG07]. Von nun an lassen Sie langsam immer mehr fehlgeschlagene Tests aus dem nächtlichen Build als Bestandteil Ihrer täglichen Arbeit erneut durchlaufen. Das frisst dann immer mehr Ihrer Zeit und lässt Ihnen immer weniger Zeit für neue Tests oder das Beheben von Problemen in den bereits existierenden. Als würde man ganz langsam Frösche kochen, werden Sie immer besessener von der umgekehrten Augenbinde (engl.: Backward Blindfold) [Wei01, S. 125].

Der Fisch ist immer der Letzte, der das Wasser sieht.

Um das zu verhindern, sollten Sie nach Möglichkeiten suchen, Ihre Tests zu beschleunigen. Das könnte bedeuten, die Akzeptanztests so umzustrukturieren, dass Sie anhand einer Schnittstelle auf niedriger Ebene ausgeführt werden, beispielsweise mit dem Modell einer MVC-Architektur statt einer Benutzerschnittstelle oder -ansicht. Sie könnten versuchen, einige Teile der langsameren Subsysteme, wie etwa Datenbanken oder Third-Party-Komponenten, rauszumocken. Das würde zwar eine ganze Weile ganz gut funktionieren, wird aber durch zwei schwerwiegende Nachteile erkauft. Erstens wird die Kluft zwischen der Anwendung, die von Ihren Tests geschärft wird, und der Anwendung, mit der der

Benutzer konfrontiert wird, immer größer. Im nächsten Abschnitt »Denken Sie an die Lücken« werden wir dieses Risiko abhandeln. Der zweite Nachteil besteht darin, dass Sie immer weniger Möglichkeiten bekommen, Ihre Beispiele das nächste Mal zu kürzen, wenn Sie wieder einmal an eine Grenze Ihrer Testlaufzeit stoßen. Wenn Ihnen die Optionen beim Beschleunigen Ihrer ausführbaren Beispiele ausgegangen sind, bleibt Ihnen womöglich nur noch die der Roten Königin in *Alice im Wunderland* von Lewis Carroll: Runter mit dem Kopf [Car65].

Auch wenn Ihnen das jetzt extrem vorkommt, so findet Regressionstesten mitunter nur 23% der Probleme [Jon07]. Automatisierte Regressionstests finden somit nur einen kleinen der Teil der Bugs [KBP01]. Und da Ihre Beispiele möglicherweise als Regressionstests dienen, geht es vermutlich in Ordnung, dass Sie 23% halten können, wenn darin alle für Sie relevanten Regressionen enthalten sind. Natürlich besteht das Problem auch darin, dass Sie in den seltensten Fällen wissen können, welche Tests Ihnen die 23% Regressions-Fehlerrate liefern werden, bevor Sie sie implementieren. Aus diesem Grund bauen einige Teams große Testsuites auf. Das mag einem zwar für einige Zeit ein wohliges Gefühl der Sicherheit geben, sobald man aber die zweite magische Grenze der Laufzeiten mit der Regressionstestsuite durchbrochen hat, stehen sie einem im Weg.

Ich erinnere mich an ein Projekt, bei dem unsere Testsuite ungefähr 36 Stunden lief. Wir konnten nicht genau sagen, wie lange sie lief, da wir sie nicht in einem einzigen Durchlauf ausführen konnten. Die Zahl der Regressionstests schien schon überwältigend, mit der schieren Anzahl der Tests war niemandem mehr gedient. Nach zwei Monaten hatten wir die Regressionstests auf diejenigen 10% gestutzt, die uns schnelle Rückmeldung brachten, wenn etwas kaputt gegangen war, und dadurch unsere Feedback-Schleife drastisch verkürzt. Ohne diesen Schritt wären wir schlichtweg verloren gewesen.

Wenn man schon am Anfang Beispiele streicht, reichen die Standard-Testtechniken aus. Rückwirkend betrachtet hätten wir mit paarweisem Testen die Beispiele bei unserer 36-Stunden-Testsuite kürzen können. Um noch einmal auf den Abschnitt »paarweises Testen« in diesem Kapitel zurückzukommen: der paarweise Ansatz beruht auf der jeweils einmaligen Kombination zweier Faktoren in allen möglichen Konstellationen. Die Anzahl der durchzuführenden Tests kann auf diese Weise drastisch reduziert werden, während man immer noch das wohlig-warme Gefühl einer guten Abdeckung der zugrunde liegenden Geschäftsbedingungen hat.

Wenn Sie die Kürzungstechniken, wie die kombinatorische Reduktion, schon ausgeschöpft haben und immer noch Laufzeiten von fünf Stunden haben, möchten Sie vielleicht noch mehr Tests auslagern. Einige Teams fangen dann an, ihre automatisierten Tests in kleinere Suiten einzuteilen und mit Tags für bestimmte Risikobereiche zu versehen. Sie könnten also einige Ihrer Tests für das Smoke-Testing taggen und bei Ihren Testsuiten einen Staged-Build [DMG07] anwenden. Beim Staged-Build führen Sie zunächst eine kleine Gruppe von Tests durch, die

die riskantesten Bereiche abdecken. Falls alle von ihnen durchlaufen, führen Sie eine größere Gruppe von Tests durch, die die Bereiche betreffen, die zuvor übersehen worden waren.

Und schließlich automatisieren einige Teams alle ihre Beispiele der Iteration parallel zum Produktionscode. Wenn die Tests dann durchgelaufen sind, löschen Sie die meisten Tests und lassen eventuell ein sehr grundlegendes Set an Regressionstests für die automatisierte Testsuite bestehen. Gojko Adzic hat mehrere solcher Fälle gefunden, als er 50 Teams 2010 über ihre Anwendung von Spezifikation anhand von Beispielen befragt hat [Adz11]. Damit haben die Teams die Laufzeiten ihrer Tests unter die erste magische Grenze drücken können. Für das schnelle Feedback hatten sie die Abdeckung aus ihren automatisierten Tests geopfert – oder aufgehört, eine solche Abdeckung ohnehin nur vorzutäuschen. Solange Sie an alle Lücken denken, geht dieser Ansatz in Ordnung.

Denken Sie an die Lücken

Bei der Testautomatisierung sollten wir daran denken, welche Bugs wir durch die Automatisierung nicht finden [KBP01]. Diese Lektion aus Lessons Learned in Software Testing handelt in erster Linie von den Opportunitätskosten der Testautomatisierung. Für jeden Test, den wir automatisieren, gibt es eine Gruppe von Tests, die wir nicht durchführen – ganz gleich, ob es sich um automatisierte oder manuelle Tests handelt.

Da wir gerade von Lücken in Ihrem ATDD-Ansatz sprechen: Sie müssen die richtige Balance aus automatisierten und manuell durchgeführten Tests finden, um Bereiche und Risiken abzudecken, die den automatisierten Tests entgehen. Die Testautomatisierung allein führt daher zu einem unausgewogenen Ansatz. Überlegen Sie zum Beispiel einmal, wie das bei einer Anwendung wäre, bei der Sie persönliche Kundeninformationen eingeben können. Ihre automatisierten Tests decken das Anlegen eines neuen Kunden, die Geschäftsbedingungen zur Anlage des Kunden und Abhängigkeiten der Felder untereinander, wie etwa ungültige Postleitzahlen und deren Paarungen mit der entsprechenden Stadt, ab.

Stellen Sie sich in diesem hypothetischen Szenario vor, Sie würden keine manuellen Tests durchführen und nach Auslieferung der Anwendung an den Kunden würde es jede Menge Beschwerden hageln, sobald sie in Betrieb ginge. Fast alle Kunden würden sich beklagen, dass Ihre Anwendung schwierig zu benutzen sei, da die Navigation durch sie unnatürlich sei. Solange man nur einen Datenwert eingibt, wird das Problem noch nicht offenkundig. Sobald man aber tausende Kundendaten eingeben muss, ist man in Schwierigkeiten.

Ich musste einmal solch eine Anwendung benutzen. Das Problem bestand in der Tabulatorfolge der Felder auf der Dateneingabe-Seite. Wenn ich den Nachnamen eingegeben hatte, sprang der Cursor in das Feld mit dem Geburtsdatum, bevor es zum Vornamen weiterging. Zuvor hatten wir eine DOS-basierte Anwendung in Gebrauch, bei der die Tabulatorfolge kein Problem war. Als wir dann auf

eine Windows-Anwendung gewechselt und 300 Accounts migrieren mussten, fiel mir das Problem auf, da mir das ziemlich bald auf die Nerven ging.

Was sollten Sie tun, wenn Sie der Anbieter einer solchen Anwendung sind? Sie könnten natürlich das Tabulatorfolge-Feature mit Beispielen beschreiben und diese automatisieren. Bei jeder Änderung der Code-Basis könnten Sie solche Bugs erwischen. Das Problem bei solch einem Test wäre allerdings, dass er stark an die jeweilige Implementierung der Benutzeroberfläche gebunden wäre. Und mit jedem neuen Feld Ihrer Eingabemaske müssten Sie wahrscheinlich diesen Test ändern.

Die Alternative dazu wäre, eine zeitlich festumrissene Sitzung explorativen Testens anzusetzen [KFN99]. In solch einer Session könnten Sie die Benutzerfreundlichkeit des Designs Ihrer Oberfläche erkunden. Bei jeder Änderung der Benutzeroberfläche würden Sie wieder eine ähnliche Sitzung für die Ermittlung der Benutzerfreundlichkeit des neuen Bildschirm-Layouts ansetzen.

Mit diesen beiden Alternativen konfrontiert, entscheiden sich die meisten Teilnehmer meiner Kurse und Workshops für die explorative Alternative. Die Gründe liegen in der Erstellung der automatisierten Tests, deren Wartungskosten und der damit verbundenen ungünstigen Rentabilität.

Kommt Ihnen das vielleicht wie ein seltenes Beispiel vor? In seinem Kurs zu Management und Leadership spricht Jurgen Appelo von dem Unknown-Unknowns-Trugschluss. So zu tun, als habe man alle Risiken abgedeckt, ist demnach eine Falle, in die viele Manager tappten [App11]. Jahrhunderte lang waren die Wissenschaftler davon ausgegangen, es gäbe nur weiße Schwäne. Die erste Sichtung eines schwarzen Schwans belehrte sie eines Besseren. Solch schwer vorhersagbare und seltene Ereignisse hinterlassen dann oft tiefe Eindrücke [Tal10]. Und der Glaube, dass die Katastrophe unmöglich sei, führt nur allzu oft zur undenkbaren Katastrophe. Jerry Weinberg hat das einmal den Titanic-Effekt genannt [Wei91].

Wenn Sie glauben, Sie hätten an alle Lücken gedacht, dann haben Sie vermutlich den schwarzen Schwan vergessen, den Unbekannten, von dem wir nichts wissen. Auch wenn Sie sich auf einen einfachen Testansatz bei Ihrer Software verlassen, so müssen Sie einen Ausgleich für die Möglichkeit finden, dass die Testautomatisierung allein nicht alle Ihre Probleme lösen wird. In Ihrem Testkonzert wird es zu Lücken kommen. Sie als Dirigent sollten allerdings dafür sorgen, dass Sie Ihrem Publikum das richtige Orchester präsentieren.

Stellen Sie Ihr Test-Orchester zusammen

Die Test-Quadranten, wie sie in *Agile Testing* von Lisa Crispin und Janet Gregory [CG09] ausführlich diskutiert und erstmals von Brian Marick [Mar03] erwähnt wurden, zeigen Ihnen möglicherweise noch zusätzliche Testaktivitäten, die Ihnen bei Ihrer Anwendung zu einem ausgewogenen Ansatz verhelfen. Kurz gesagt kann das Feld der Testaktivitäten entlang zweier Dimensionen aufgeteilt werden. Die erste Dimension teilt die Tests in solche, die das Team Ihres Projekts unterstützen, und solche, die das Produkt im Sinne des Kunden kritisch untersuchen.

Die zweite Dimension steht orthogonal dazu und teilt das Feld der Testaktivitäten in solche, die sich in Richtung der Technologie orientieren und solche, die sich in Richtung des Business orientieren (siehe Abb. 10–2).

Die Testquadranten teilen also das Feld der Testaktivitäten in vier Quadranten auf. Der erste Quadrant deckt die technischen Tests ab, die dem Team helfen. Darunter fallen Unit-Tests und Integrationstests zwischen mehreren Modulen auf niedriger Ebene. Im zweiten Quadranten befinden sich Tests, die das Team unterstützen und gleichzeitig einen Geschäftsbezug haben. Das sind die automatisierten Akzeptanztests, die in diesem Buch beschrieben werden. Der dritte Quadrant befasst sich mit Business-orientierten Tests, die das Produkt kritisch untersuchen. Das sind unter anderem Alpha- und Betatests, aber auch Benutzer-Akzeptanztests, Usability-Tests und exploratives Testen. Der vierte Quadrant erinnert uns an die Tests, die das Produkt kritisch untersuchen, allerdings sehr viel technischer. Zu den bekanntesten Testtechniken gehören hier Performanz-, Last- und Stress-Tests.

Wie Sie sehen, gehören die Tests, die als Nebenprodukt von ATDD anfallen, in den zweiten Quadranten. Wenn Sie Sich allein auf die Business-Ebene, die Ihr Team unterstützt, konzentrieren, entgehen Ihnen sehr wahrscheinlich Probleme mit der Leistungsfähigkeit, der Benutzerfreundlichkeit oder auch einfach nur solche mit der Qualität Ihres Codes, was Ihnen auf lange Sicht Schwierigkeiten bereiten wird.[5]

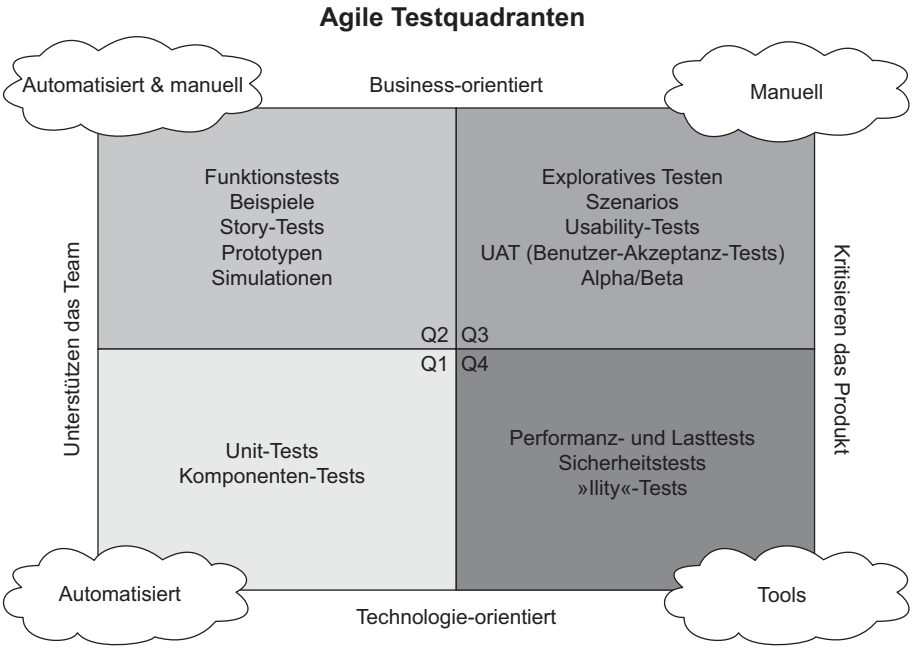

Abb. 10–2 *Die Testquadranten helfen Ihnen bei der Erkennung von Lücken in Ihrem Testansatz.*

5. In der Literatur wird dies häufig »Technische Schulden« (Technical Debt) genannt. Den Begriff hat Ward Cunningham 1992 geprägt: *http://c2.com/cgi/wiki?TechnicalDebt*

Diese drei Bereiche nicht abzudecken bedeutet, dass Ihre Anwendung und Ihr Projekt für eine Menge Risiken anfällig sind.

Falls Sie mir in diesem Punkt am Ende nicht zustimmen, dürfen Sie die Seite mit dem Parkgebührenberechner überprüfen. Dabei handelt es sich um eine echte Seite, die auf echten Anforderungen basiert. Obwohl alle Akzeptanztests durchlaufen, gibt es versteckte, aber auch ganz offensichtliche Bugs in diesem Berechner. Schauen Sie einmal, wie viele Sie davon innerhalb einer halben Stunde finden. Anschließend lesen Sie noch einmal dieses Kapitel und überlegen, ob Sie aufgrund dieser Erfahrung nicht vielleicht doch Ihre Meinung ändern.

Zusammenfassung

Falls Sie überlegen, in ATDD einzusteigen, spielen Sie einfach mit verschiedenen Wegen, Ihre Daten auszudrücken, herum. Versuchen Sie Ihre Beispiele in tabellarischer Form aufzuschreiben, und gehen Sie dann zum BDD-Stil über. Sobald Sie eine Darstellung gefunden haben, die Ihnen, Ihrem Test und Ihrem Firmenvertreter einleuchtet, machen Sie sich auf die Suche nach einem Framework, das Ihren Ansprüchen gerecht wird. Sie können die Entscheidung über das Framework auch im Team fällen. Sie sollten berücksichtigen, dass die unterschiedlichen Treiber für Ihre Anwendung – zum Beispiel Selenium oder Watir für Webseiten, SWTBot für SWT-Anwendungen – unterschiedliche Nebenwirkungen haben. Fangen Sie mit dem vielversprechendsten Framework an, implementieren Sie einen Test von Anfang bis Ende und überdenken Sie anschließend Ihre Entscheidung. Nach einigen anfänglichen Erfahrungen werden Sie Vorzüge und Nachteile erkennen und das nächste Mal eine bessere Entscheidungsgrundlage haben.

Die erste Gruppe von Beispielen reicht in der Regel noch nicht aus. Nutzen Sie Ihre Testentwurfskenntnisse, um Ihre erste Beispielgruppe zu verfeinern. Grenzwerte, paarweise Ansätze und Domänentests können Ihnen dabei helfen. Ihren Testern sollten derartige Techniken vertraut sein. Wenn Sie mehr über solche Testentwurfstechniken erfahren möchten, lesen Sie »A Practitioner's Guide to Software Test Desig« [Cop04].

Mit der Zeit sollte Ihre Testsuite anwachsen. Irgendwann werden Sie mit dem Problem konfrontiert, dass es zu lange dauert, alle Tests durchzuführen. Für eine gewisse Zeit können Sie die Tests noch in Gruppen aufteilen oder sie über Nacht laufen lassen. Aber irgendwann kommen Sie möglicher Weise zu dem Schluss, dass Sie einige Tests entfernen müssen. 90 Minuten Laufzeit scheinen dabei eine magische Grenze darzustellen. Setzen Sie sich mit dem ganzen Team zusammen und diskutieren Sie über Möglichkeiten, von Ihren Akzeptanztests ein zeitnäheres Feedback zu bekommen.

Es kann in Ihrer gesamten Teststrategie zu Lücken kommen. Denken Sie an die vier Testquadranten und überprüfen Sie, ob Sie alle Punkte in den Quadranten abgedeckt haben: Lassen Sie regelmäßig explorative Tests laufen? Wann haben Sie zum letzten Mal einen Benutzer zum Usability-Test eingeladen? Wie

sieht es mit Performanz- und Lasttests aus? ATDD kümmert sich nur um den Business-orientierten Quadranten, der Ihrem Team beim Vorankommen hilft. Akzeptanztests sind zwar ein wesentlicher Bestandteil, den die meisten Teams auslassen, aber Sie sollten ihretwegen nicht zu viel aus den anderen Quadranten opfern.