

Bachelorarbeit

**Testautomatisierung im Rahmen von Test-Driven-Development**

Fachhochschule Aachen

Medizintechnik und Technomathematik

Scientific Programming

Autor: **Bjarne Herrmann**  
 Leimkaul 4  
 51143, Köln  
 T.: 0176 64484501  
 E-Mail: herrmann.bjarne@outlook.de

Matrikelnummer: 3158998

Gutachter: Prof. Dr. Bodo Kraft  
Zweitgutachter: Stefan Sensler

Abgabetermin: 10.08.2020

Bearbeitungszeit: 6 Wochen

# Sperrvermerk

Die nachfolgende Bachelorarbeit enthält vertrauliche Daten und Informationen der Gothaer Systems GmbH. Veröffentlichungen, Vervielfältigungen oder die Weitergabe des Inhalts der Arbeit im Gesamten oder in Teilen sowie das Anfertigen von Kopien oder Abschriften (auch in digitaler Form) sind grundsätzlich untersagt. Ausnahmen bedürfen der schriftlichen Genehmigung der Gothaer Systems GmbH.

Die Bachelorarbeit ist nur den Korrektoren sowie den Mitgliedern des Prüfungsausschusses zugänglich zu machen.

Die Weitergabe oder Veröffentlichung ist nicht erlaubt bis zum 01.08.2070.

# Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Bachelorarbeit mit dem Thema

**Testautomatisierung im Rahmen von Test-Driven-Development**

Selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Name: Bjarne Herrmann

Köln, den 27. Juli 2020

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Unterschrift

# ATDD – Vorwort

Dieses Kapitel dient als Einblick in Acceptance Test-Driven Development (kurz: ATDD), um einen Überblick zu erhalten. Im späteren Verlauf werden kommende Themen im Detail aufgegriffen und erläutert.

Im Rahmen der stetig komplexer werdenden Softwareentwicklung rückt ATDD immer weiter in den Vordergrund. Besonders durch ständige Anpassungen im agilen Kontext, sich ändernden Anforderungen und veränderbaren Technologien muss die Qualität von Software gewährleistet werden. ATDD gewährleistet eine kontinuierliche Absicherung oben genannter Punkte. Zudem ist es wichtig zu verstehen, dass nicht nur die Erstellung von Tests bei der Entstehung von Software, sondern das Beibehalten dieser mit möglichst wenig zukünftigen Anpassungen für die Automatisierung eine hohe Relevanz besitzen. Akzeptanztests in diesem Kontext grenzen sich von den traditionellen Benutzer-Akzeptanztests[[1]](#footnote-1) oder Systemtests[[2]](#footnote-2) ab. Die Implementierung von Anforderungen folgt stets der Erstellung von Akzeptanztests. (Pugh, 2010 S. 2) Sie fungieren als Blackbox-Tests[[3]](#footnote-3), die unabhängig von der Implementierung entstehen. Weiterhin gilt: Der Zeitpunkt und die Art und Weise der Erstellung dieser Tests differenziert diese von den klassischen Tests. (Pugh, 2010 S. 3) Während klassische Tests durch die Implementierung definiert werden, gilt dies bei Akzeptanztests in ATDD nicht. Da die Absicht des Tests selbst zu priorisieren ist, gibt es auch keine Regel der festen Eingliederung in eine Testebene[[4]](#footnote-4). Solche Tests lassen sich beispielhaft als Validierungstests für Eingaben, oder ebenso als Designverifizierungstests, die eine Unit oder Komponente testet, nutzen. In beiden Fällen gilt es zu gewährleisten, dass der Akzeptanztest sicherstellt, dass eine bestimmte Anforderung des Kunden im System „akzeptiert“ ist. (Pugh, 2010 S. 3) In der klassischen Softwareentwicklung werden Tests in der Regel von Entwicklern oder Testern[[5]](#footnote-5) erstellt, um eine Kundenanforderung – meist nach der Implementierung dieser – auf dessen Gültigkeit zu prüfen. Akzeptanztests innerhalb des ATDD-Kontexts werden aber gemeinsam vom Kunden, Entwicklern und Testern entwickelt, um ein gemeinsames Verständnis über diverse Anforderungen und dessen Details zu erhalten (Pugh, 2010 S. 3,14).

Während ATDD bereits lange bekannt war, wurde dies im Rahmen von „extreme programming“[[6]](#footnote-6) wiederbelebt. Dabei beinhaltet das Manifest diverse Entwicklervorgehen:

* Acceptance Test-Driven Development (kurz: ATDD)
* Example Driven Development von Brian Marick (kurz: EDD)
* Behavior Driven Development von Dan North (kurz: BDD)
* Story Test-Driven Development von Joshua Kerievsky (kurz: SDD)
* Domain-Driven Development von von Eric Evans (kurz: DDD)
* Executable Acceptance Test-Driven Development (EATDD)

Jedes dieser vorgehen besitzt das Ziel von Erstellung qualitativ hochwertiger Software, indem Entwickler und Tester mit spezifischen Methoden unterstützt werden, den Kunden und dessen Bedürfnisse zu verstehen.

ATDD kann mitunter Methodiken aus anderen Entwicklungsphilosophien verwenden, muss dies aber nicht zwingend. Dazu gehören zum Beispiel das Konzept der Ubiquitous Language[[7]](#footnote-7) aus DDD sowie das Given-When-Then Template[[8]](#footnote-8) und die Tabellendarstellung von Tests aus BDD. (Pugh, 2010 S. 5,6) Vorher genannte Punkte werden im späteren Verlauf konkretisiert.

# ATDD Einleitung

Der primäre Vorteil von ATDD wird durch folgende Gegenüberstellung zweier Szenarien ersichtlich.

***Szenario 1***

Man nehme den letzten Tag der aktuellen Iteration in einem agilen Softwareprojekt. Der Tester überprüft mit den letzten Tests die neue Implementierung eines Entwicklers und entdeckt dabei ein unerwartetes Ergebnis. Aufgrund der Uhrzeit ist dieser nicht in der Lage den Kunden rechtzeitig vor Abschluss der Iteration zu kontaktieren, um Klarheit zu schaffen und muss folglich einen Defekt[[9]](#footnote-9) für die nächste Iteration erstellen. Die Konsequenz ist eine Verzögerung des Abschlusses der Aufgabe und somit eine Erhöhung der Projektkosten.

***Szenario 2***

Erneut ist der letzte Tag der aktuellen Iteration in einem agilen Softwareprojekt. Die aktuelle Anforderung wurde von einem Entwickler implementiert und mit den zu Beginn erstellen Akzeptanzkriterien erfolgreich getestet. Der Tester überprüft erneut auf erfolgreich absolvierte Akzeptanztests, um dies zu bestätigen. Im Anschluss erweitert dieser diese um weitere Tests, um die technische Qualität von der neuen Implementierung zu gewährleisten. Beim Review am nächsten Tag bestätigt der Kunde dies als abgenommen.

Durch die Überstellung lässt sich erkennen, dass vorher definierte Akzeptanzkriterien eine fehlerfreie Implementierung gewährleisten können. Im ersten Falle benötigt der Tester mehr Details, die Ihm durch andere Faktoren, wie zum Beispiel mangelnde Kommunikation, fehlendes Wissen oder Missverständnisse, fehlten. (Pugh, 2010 S. 10) Besonders im Rahmen von einheitlichen Verständnis diverser Anforderungen gilt es besonders zu beachten, diese vorab zu klären. Interpretationsfreiheiten und Meinungsverschiedenheiten habe oft unterschiedliches Verständnis zu bestimmten Aussagen zur Folge. Ein Kunde kann beispielsweise unter der Bedeutung von „schneller Software“ etwas anderes verstehen als der Entwickler. Während der Kunde von einem System ausgeht, welches bei jedem Klick nicht hängt, kann der Entwickler von einer speziellen Reaktionszeit beim Ausführen von spezifischen Befehlen ausgehen. Somit ist es wichtig, eine gute Kommunikation zwischen relevanten Rollen zu besitzen.

## Die Triade

Der Kern der Kommunizierenden Teilnehmer im Rahmen von ATDD ist der Kunde, der Entwickler und der Tester. Dabei sind diese nicht einzelne Personen, sondern Rollen, welche je nach Größe und Komplexität des Projekts aus mehreren Personen bestehen können.

ABBILDUNG TRIADE (S16, F2.1 The Triad)

Die Abbildung zeigt die wichtigsten Rollen für die Entstehung von Akzeptanztests. Dies bedeutet aber nicht, dass es die einzigen sind (Pugh, 2010 S. 16). Es sollte immer beachtet werden, welche Stakeholder im aktuellen Stadium eine wichtige Rolle für die Findung von Akzeptanztests, Akzeptanzkriterien oder Businessregeln besitzen. Der Vorteil befindet sich ebenfalls in der Kommunikation, die durch die verschiedenen Sichtweisen und Aufgabenverteilungen der Teilnehmer vielfältig wird. Zudem sollte beachtet werden, dass in der agilen Welt der Softwareentwicklung nicht jeder nur eine Rolle ausübt, sondern auch vermindert Rollenübergreifend handelt – ähnlich, wie in einem Basketballteam, in dem der Angreifer auch Verteidiger ist (Pugh, 2010 S. 16).

## Charakteristiken

Wie in obigen Szenarien dargestellt, können Tests in zwei Arten entwickelt werden:

ABBILDUNG POST UND PRE TEST IMPLEMENTIERUNG (S17/S19)

Bei der Pre-Testimplementierung gilt es Tests im Anschluss einer Implementierten Anforderung zu Entwickeln. Bei Fehlern muss die Implementierung korrigiert werden und erneut verifiziert werden. Wie bereits erwähnt, kann dies an ein Missverständnis oder an einer falschen Interpretation vom Kunden liegen. Durch eine Post-Testimplementierung kann vorab mit dem Kunden Tests für diverse Anforderungen erstellt werden, sodass der Test bereits beim Entwickeln vorhanden ist. Der Vorteil ist eine Vermeidung von Loopbacks[[10]](#footnote-10) sowie ein schnelles Feedback des Kunden (Pugh, 2010 S. 18).

Akzeptanztests heben sich in einigen Eigenschaften von klassischen Tests[[11]](#footnote-11) ab. Zum einen liegt der Fokus auf das Verständnis vom Kunden, der die Tests mit unter spezifizieren kann (Pugh, 2010 S. 12). Zudem gilt: Ein Akzeptanztests ändert sich nicht nach einer Änderung der Implementierung (Pugh, 2010 S. 14). Der Test wurde in der Regel mit dem Kunden erstellt oder ist in Abhängigkeit von einer Anforderung des Kunden. Das heißt, einen Akzeptanztest zu ändern, würde eine Änderung der Kundenanforderung zur Folge haben:

„Any test created after the code is written is a new requirement or a new detail on an existing requirement.“ (Pugh, 2010 S. 27)

Diese Aussage scheint zu Beginn Trivial, doch die Praxis zeigt, dass Entwickler gerne in stressigen Projektphasen zu “schnellen“ Umsetzungen tendieren – ähnlich, wie sich Entwickler versprechen, Tests schnell nach der “einfachen“ Implementierung zu schreiben. Somit ist es wichtig im Hinterkopf zu behalten, dass jede neue Anforderung einen neuen Akzeptanztest mit sich zieht und auch umgekehrt (Pugh, 2010 S. 27, 28).

Weiterhin gilt, dass absolute Akzeptanztests nicht zwingend festgesetzt sein müssen (Pugh, 2010 S. 11). Wenn ein System als Akzeptanzkriterium 5.0 Sekunden Reaktionszeit haben soll, es dennoch 5.1 Sekunden braucht und eine Verbesserung der letzten 0.1 Sekunden viele Ressourcen benötigen würde, kann unter Umständen diskutiert werden, ob dies vernachlässigt wird. Hierbei sollte Beispielsweise das Pareto Prinzip (20% effektiver Aufwand ergeben 80% effektiven Ertrag) berücksichtigt werden.

## Lean und Agil

ATDD besitzt Eigenschaften aus den Lean (2003 entworfen von Mary und Tom Poppendieck) und Agile (2001 durch „The Agile Manifesto“ veröffentlicht) Prinzipien.

Ersteres ist die Reduzierung von Müll während des Entwicklungsprozesses beispielsweise durch die Vermeidung von den vorher erwähnten Loopbacks. Weiterhin gehört das Prinzip der Integrität: Mit Akzeptanztests geht stets eine hohe Qualität einzelner Module einher. Dies geschieht durch direktes Feedback beim Entwickeln von Modulen und nicht im Nachhinein beim Abschluss des Systems. Durch die Triade und der einhergehenden Kommunikation lernen alle Beteiligten mehr über das Business sowie der damit verbundenen Entwicklung und die Herausforderung des Testens innerhalb eines Projektes – dies verschafft mehr Verständnis und Transparenz. Zudem gehört nicht nur die Initiale Zusammenarbeit der Triade zu den Prinzipien, sondern eine ununterbrochene. Dies hat zur Folge, dass die Transparenz im Verlauf des Projektes mitgenommen wird und zukünftige Änderungen oder auftretende Probleme gemeinsam gelöst werden können. Letzteres gehört das Prinzip der Messbarkeit zu ATDD. Implementierte Software ist die Primäre Messmetrik bezüglich des Fortschrittes, die durch Akzeptanztests gewährleistet wird.[[12]](#footnote-12)

## Test Strategien

Es ist wichtig zu verstehen, welche Arten von Tests existieren und welche dieser Arten sich automatisieren lassen, um eine qualitativ hochwertige Software zu gewährleisten. Diesbezüglich gibt es die sogenannte Test Matrix:

ABBILDUNG TEST MATRIX

Einfach ausgedrückt lassen sich Tests in zwei Kategorien einteilen: Funktionale Tests und Cross-Funktionale Tests. Erstere verifizieren das Verhalten einzelner Funktionalitäten der Software in Abhängigkeit auf spezifische Eigenschaften, während zweitere verschiedene Aspekte hinsichtlich des Verhaltens des Systems über multiple Funktionalitäten verifizieren. (Meszaros, 2007 S. 50,51)

Wie bereits vorher erwähnt sind Akzeptanztests keine klassischen Tests direkt, sondern bilden die Bedingungen und Basis für die daraus resultierenden Tests – diese lassen sich also in mehreren Tests beziehungsweise Akzeptanzkriterien gliedern (kommendes Kapitel gibt spezifischere Auskunft diesbezüglich). Wichtig zu wissen ist, dass Akzeptanztests in jeglicher Form auftauchen können und jeder Teil der Triade mit seinem Wissen einen Testbereich abdeckt, sodass gemeinsam an allen Tests entwickelt werden kann.

### Customer Tests

Customer Tests stellen die Basis für die Akzeptanztests dar. Durch die Triade können Customer Tests in Form von Akzeptanztests erstellt werden, um sicherzustellen, dass das System die Anforderungen vom Kunden erfüllt. Typischerweise besitzen diese Tests Bezug zu individuellen Features, User Stories oder Use-Cases. Für die Automatisierung solcher Tests muss stehts berücksichtigt werden, dass der Kunde stets ein Verständnis für diese Tests besitzt. (Meszaros, 2007 S. 51) (Pugh, 2010 S. 23)

### Unit Tests

Unit Tests verifizieren einzelne Klassen auf deren Verhalten – sie dienen somit lediglich zur Unterstützung für den Entwickler selbst und für die technische Dokumentierung. Einzige Ausnahme hier sind wichtige Business-Regeln, die in einzelnen Units getestet werden, da diese für den Kunden eine hohe Relevanz spielen. (Meszaros, 2007 S. 51) (Pugh, 2010 S. 25)

### Component Tests

Component Tests verifizieren eine Gruppe von Klassen, die gemeinsam ein Service bilden, der wiederum eine spezielle Anforderung erfüllt. Sie werden auch „integration tests“ genannt und befinden sich zwischen der Unit Test- und Customer Test Ebene. Component Tests resultieren meist aus den Akzeptanztests. (Meszaros, 2007 S. 52) (Pugh, 2010 S. 25)

### Usability Tests

Usability Tests prüfen das System hinsichtlich dessen Anwendbarkeit. Da es sich um subjektive Meinungen handelt, lassen sich diese Test nicht automatisieren und müssen von Benutzern selbst ausgeführt werden – getreulich nach: „I’m not sure what I’d like, but I’ll know it when I see it.“ (Pugh, 2010 S. 24).

### Exploratory Tests

Exploratory Tests müssen hauptsächlich manuell durchgeführt werden, da hierbei der Tester verschiedene Aspekte beachten muss, die sich nicht automatisieren lassen. Dabei handelt es sich um eine Kombination aus Test-Design, Ausführung, Interpretation von Ergebnissen und folglich das Erlangen von Erkenntnissen. Es wird versucht das System auf dessen Selbstkonsistenz zu überprüfen. Automatisieren lassen sich lediglich der Vorbedingungen zum ausführen spezieller Tests. (Meszaros, 2007 S. 53) (Pugh, 2010 S. 24)

### Property Tests

Property Tests sind Performance-Tests, die in Form von Response-Tests, Capacity-Tests und Stress-Tests implementiert werden und somit Eigenschaften der Sicherheit, Performance und Skalierbarkeit überprüfen. Diese müssen in der Regel automatisiert werden, da meist die Vorbedingungen für diese Tests – zum Beispiel das Ausführen eines Service in sehr höher Anzahl – nicht manuell erfüllt werden können. Vorteil dieser Tests ist, dass sie früh erstellt werden können, sobald die Rohanwendung mit deren Grundfunktionalitäten vorhanden ist. (Meszaros, 2007 S. 52) (Pugh, 2010 S. 23)

## Entwicklungsumgebungen

Akzeptanztests selbst können auf multiplen Plattformen laufen. Dabei besitzen die Größe und Komplexität des Projekts eine hohe Relevanz. Bei kleineren Projekten kann eine lokale Maschine ausreichen, die sämtliche Tests ausführt. Befindet sich jedoch das Projekt in einen größeren Kontext – wie zum Beispiel eine erhöhte Anzahl an Entwicklern und verschiedene in Verbindung stehenden Instanzen –, sollten mehrere Entwicklerplattformen verwendet werden:

ABBILDUNG 3.2 WHERE TESTS RUN

Bei einem Projekt, an dem mehrere Entwickler arbeiten bietet sich eine Build-/Integration-Plattform an. Dort wird der Code von allen Entwicklern als ganzes getestet, um sicher zu stellen, dass Entwickler sich nicht gegenseitig behindern und die Software fehlerfrei ist. Während Unit Tests, Component Tests und Customer Tests auf der lokalen Entwicklerplattform sowie der Build-/Integration-Plattform ausgeführt werden, führt die anschließende Test-Plattform zusätzlich Property-/Exporatory- und Usability-Tests aus. Weiterhin ist anzumerken, dass erst ab der Test-Plattform externe, Systemunabhängige Schnittstellen verwendet werden und Unit-Tests oft ausgeschlossen werden, da hier das System als Ganzes arbeitet und in der Regel auf der Ebene Daten nicht vorgetäuscht[[13]](#footnote-13) werden.

Gothaer Interne Plattformen hier erläutern

## Ein einfacher Akzeptanztest

Ein einfacher Akzeptanztest kann zum Beispiel eine Business-Regel sein. Durch den Austausch innerhalb der Triade wird diese aufgegriffen und könnte vom Kunden wie folgt lauten:

Wenn ein Käufer regelmäßig Produkte einkauft und der Warenkorb maximal 10€ aufweist,

dann wird kein Rabatt gegeben,

sonst wird ein Rabatt in Höhe von 1% gegeben.

Wenn ein Käufer täglich Produkte einkauft,

dann wird ein Rabatt in Höhe von 1% gegeben.

Wenn der Warenkorb minimal 50€ aufweist,

dann wird ein Rabatt in Höhe von 5% gegeben.

Zu Beginn erscheint die Regel als Logisch und selbstverständlich. Doch aus technischer Sicht ergeben sich direkt Fragen, die optimalerweise direkt geklärt werden können. Die Entwickler und Tester können nun aus gegebener Regel folgende Information darstellen:

|  |  |  |
| --- | --- | --- |
| **Rabattberechnung** | | |
| *Warenkorb* | *Käuferstatus* | *Rabatthöhe* |
| 10.00€ | Regelmäßig | 0% |
| 10.01€ | Regelmäßig | 1% |
| 50.00€ | Regelmäßig | 1% (?) |
| 00.01€ | Täglich | 1% |
| 49.99€ | Täglich | 1% |
| 50.00€ | Täglich | 5% (?) |

*Tabelle 1: Rabattberechnung**Quelle 1: In Anlehnung an Pugh, 2010, S30*

Das Beispiel zeigt ein einfaches Interpretationsprobleme, die häufig nicht direkt ersichtlich sind. Für den Kunden der Triade scheint die Regel verständlich, dennoch kann dies für andere nicht eindeutig erscheinen. Betrachtet man die letzte Aussage der Regel, dann zeigt sich, dass es nicht zwingend selbstverständlich sein muss, unabhängig vom Kundenstatus einen Rabatt von 50€ zu gewähren. Durch die entsprechende tabellarische Darstellung wird diese Frage für alle Teilnehmer transparent. Zeile fünf und acht sind entsprechend in der Spalte “*Rabatthöhe*“ mit einem Fragezeichen zur Verdeutlichung markiert. (Pugh, 2010 S. 30)

Obwohl die Regel nun abgeschlossen sein zu scheint, können sich durch den Tester der Triade weitere Fälle ergeben:

|  |  |  |
| --- | --- | --- |
| **Rabattberechnung** | | |
| *Warenkorb* | *Käuferstatus* | *Rabatthöhe* |
| -00.01€ | Regelmäßig | 0% |
| -00.01€ | Täglich | 1% (?) |

*Tabelle 2: Rabattberechnung-Ergänzung  
Quelle 2: In Anlehnung an Pugh, 2010, S31*

Im Falle von einem negativen Warenkorbertrag – zum Beispiel durch einen Coupon – wird laut der oben aufgegriffenen Businessregel bei einem Käuferstatus “täglich“ 1% Rabatt gewährt, was fachlich gesehen unlogisch ist. Folglich muss die Businessregel erweitert werden:

Wenn ein Käufer einen negativen Warenkorb aufweist,

dann wird kein Rabatt gegeben.

Das Szenario zeigt erneut die Relevanz der Triade und deren Kommunikation: Es ist wichtig, dass alle wichtigen Rollen teilhaben und Beispiele erzeugt werden, sodass jeder Aspekt der Regel aus unterschiedlichen Sichtweisen betrachtet wird und fehlende oder Missverständliche Szenarien aufgedeckt werden. Kurz gesagt: es wird mehr Transparenz geschaffen. (Pugh, 2010 S. 31, 37)

Weiterer Vorteil der gemeinsam aufgegriffenen Businessregel ist deren Automatisierung. Solche tabellarisch dargestellten Informationen lassen sich gut in Testfälle integrieren, um eine Bestätigung einer funktionierenden Anforderung zu erhalten. Dabei gilt auch wieder: die resultierenden Tests können auf verschiedenen Test-Ebenen implementiert werden. (Pugh, 2010 S. 37)

## Implementierung

Das Kapitel “Test Strategien“ hat diverse Strategien für die Implementierung von Tests gezeigt. Jede dieser Strategie besitzt eine eigene Basis an zu verwendenden Technologien und unterschiedliche Herangehensweisen für deren Umsetzung. Allgemein gilt immer zu beachten, dass – unabhängig der verwendeten Methoden – die Anwendung umfangreich auf allen Ebenen und von unterschiedlichen Sichtweisen getestet werden sollte – meist reicht eine Methode beziehungsweise ein Konzept allein nicht aus (Pugh, 2010 S. 37).

### Test Skript

In diesem Falle wird ein Skript geschrieben, das nacheinander die Werte in das Interface einpflegt und die Ausgabe überprüft. Diese Tests eignen sich gut, wenn High-Level-Tests geschrieben werden sollen, die die Anwendung als Ganzes testet[[14]](#footnote-14). Dazu kann erneut das Ergebnis tabellarisch dargestellt werden und mit dem Kunden besprochen werden:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Rabattberechnung** | | | | |
| *Warenkorb* | *Käuferstatus* | *Rabatthöhe* | *Rabattpreis* | *Notizen* |
| 10.00€ | Regelmäßig | 0% | 00.00€ |  |
| 10.01€ | Regelmäßig | 1% | 00.10€ | Abgerundet |
| 50.00€ | Regelmäßig | 1% | 00.50€ | Abgerundet |
| 00.01€ | Täglich | 1% | 00.00€ | Abgerundet |
| 49.99€ | Täglich | 1% | 00.50€ |  |
| 50.00€ | Täglich | 5% | 02.50€ | Abgerundet |

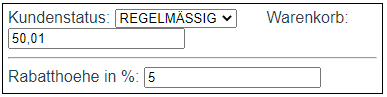
*Tabelle 3: Rabattberechnung-Testergebnis  
Quelle 3: In Anlehnung an Pugh, 2010, S32*

Nachteil hier ist aber die Abhängigkeit vom User Interface. Besonders in früheren Stadien der Entwicklung ändert sich dieses eventuell öfters oder es ist noch kein fertiges Konzept dazu vorhanden. Dies sollte stets berücksichtigt werden. (Pugh, 2010 S. 31)

* BEISPIEL GOTHAER SELENIUM; GEPR

### Test User Interface

Alternativ kann, anstatt das originale User Interface zu verwenden, ein Test User Interface (kurz: TUI) gebaut werden. Dieses hebt sich von dem originalen durch spezielle Eigenschaften ab, die für das Testen relevant sind. Dazu gehört zum Beispiel die direkte Eingabe von Werten, die für den Test relevant sind:



*Abbildung 1: Test User Interface Beispiel für die Berechnung einer Rabatthöhe  
Quelle 4: Eigene Darstellung*

Zudem ist zu beachten, dass das TUI in Produktion nicht mehr vorhanden ist, da es für die Originalanwendung nicht gebraucht wird. Durch dieses Konzept wird eine Unabhängigkeit vom User Interface geschaffen, wodurch der Test weiterhin funktioniert, nachdem dieses angepasst oder entfernt wird. Nachteil ist selbstverständlich der Aufwand zur Erzeugung des TUI (da es aber lediglich zum Verifizieren gilt, ist die Optik bzw. das Design nicht primär zu beachten). (Pugh, 2010 S. 33)

* BEISPIEL GOTHAER BARBARA HAMACHER HOST FACETTENTESTS

### xUnit Test

Generell bietet das xUnit-Framework für viele Sprachen die Möglichkeit, Unit Tests zu implementieren (für Java wird JUnit verwendet). Wie bereits im Kapitel “Test Strategien“ erwähnt, lassen sich durch Unit Tests Klassen einzeln unabhängig testen:



*Abbildung 2: Unit-Test Beispiel für die Berechnung einer Rabatthöhe  
Quelle 5: In Anlehnung an Pugh, 2010, S35*

Zu beachten ist, dass das Testen von Businessregeln mit den Daten vom Kunden auf dieser Ebene ungünstig ist. Grund dafür ist, dass bei Änderung der Daten für die Businessregel eine Änderung des Unit Tests mit sich zieht. Logischerweise spricht das Verfahren gegen eine Automatisierung – es sollte lediglich die technische Logik für den Entwickler selbst der Unit getestet werden (zum Beispiel mithilfe von vorgetäuschten Daten). (Pugh, 2010 S. 35)

### Automatisierter Akzeptanztest

Eine wichtige Alternative für die Automatisierung von Akzeptanztests bieten diverse Frameworks an. Abhängig vom genutzten Framework lassen sich Akzeptanztests, die beispielsweise in einer Tabellenform formuliert worden sind, ohne weitere Konvertierungen als automatisierten Test ausführen. Somit ist es durch die Tools möglich, zum einen eine verständliche Darstellung für den Kunden zu haben und gleichzeitig diese für die Entwickler sowie Tester zum Implementieren des Features zu nutzen. Es ist zu empfehlen, bei dieser Herangehensweise sich vorab auf ein einheitliches Format in Kombination mit passendem Tool zu einigen, um Aufwand zu vermeiden und mehr Transparenz zu schaffen. (Pugh, 2010 S. 35) Kommende Kapitel geben weitere Auskünfte diesbezüglich.

# Entwicklungsprozess von ATDD

Akzeptanztests spielen innerhalb des ATDD-Kontextes auf allen Projektebenen eine Rolle. Meist ist das Erstellen eines Akzeptanztests in Projekten nicht so simpel und erfordert einen überschaubaren Prozessablauf – besonders dann, wenn sich ein Projekt in der initialen Phase befindet. Im groben kann der Prozessablauf wie folgt beschrieben werden:



*Abbildung 3: Prozessablauf ATDD*

*Quelle 6: Eigene Darstellung*

Die Idee ist, das Projekt mithilfe eines Charters zu initialisieren. Dabei wird das Projektziel und diverse Projektrelevante Eigenschaften ersichtlich. Ist der Grundbaustein gesetzt, kann mithilfe des Charters High-Level-Anforderungen erstellt werden. Diese werden mit Stories konkretisiert, sodass spezifische Anforderungen vorhanden sind. Weiterhin werden für die Anforderungen Szenarien erstellt, welche verschiedene Anwendungsfälle repräsentieren. Auf Basis dieser Szenarien können verschiedene Tests entwickelt werden. Zu beachten ist, dass diese Tests keine Akzeptanztests direkt sein müssen, sondern für Akzeptanztest relevante Verifizierungstests sind (z.B. Unit-/Integrations-Tests). Die Akzeptanztests innerhalb des ATDD Kontextes ergeben sich auf jeder Ebene des Entwicklungsprozesses. Diese begleiten den Prozess kontinuierlich, um Transparenz für die Triade zu schaffen und die Ziele des Kunden verfolgen zu können. Die klassischen Tests unterstützen Entwickler und Kunden um die Qualität der Anwendung in verschiedenen Aspekten (siehe Kapitel: Test Strategie) zu gewährleisten. Weiterhin gilt: Je weiter der Prozess sich in Richtung “Test“ bewegt, desto technischer werden die Details und damit verbundenen Tests. Folglich sollte dies für die Teilnahme relevanter Rollen und Stakeholder an Meetings und Abstimmungen beachtet werden. (Pugh, 2010 S. 39)

## Exkurs: Kommunikation

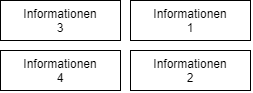
Grund für diesen Exkurs ist die Relevanz von Kommunikation. Es ist wichtig für jeden – besonders in Kollaborationen – zu verstehen, dass jeder Mensch Prozesse individuell erstellt und verarbeitet. (Pugh, 2010 S. 68) Was für einem beispielshaft positiv erscheint, mag für den nächsten negativ sein. Der Austausch von Informationen spielt in ATDD eine wichtige Rolle und sollte deshalb möglichst in Meetings in Form von physischer Anwesenheit stattfinden. Es mag zunächst logisch erscheinen, dass Kommunikationsaustausch in digitaler Form ausreicht, dennoch gehen bei physikalischen Meetings einige Vorteile einher.

Zunächst gehören das “aktive Zuhören“ sowie “aktive Schreiben“ dazu. Ziel ist es damit, die Meinungen und Motive des Gesprächspartners zu sammeln und zu verstehen. Weiterhin können diese für Diskussionen, Anregungen und weitere Ideen genutzt werden. Dazu gehört es auch, die Kommunikation durch eine empathische und offene Grundhaltung inklusive Akzeptanz und positive Beachtung des Gesprächspartners zu unterstützen. (Knauer, 2019)

Bestätigtes Wissen, Unverständlichkeiten oder Fragwürdigkeiten werden mit Gestiken untermauert. Dies erzeugt ein direktes Feedback und sorgt wiederum für Transparenz der Kommunikation. (Pugh, 2010 S. 68)

Physische Anwesenheit unterstützt weiterhin die Kreativität durch fördernde Mittel – beispielsweise ein Whiteboard – die verwendet werden können. Auch unterschiedliche Typen kommen dadurch mehr zum Vorschein. Es gibt intuitivdenkende (engl.: intuition) Menschen, die das große Bild sehen und sensitivdenkende (engl.: sensing), die umgekehrt denken – das große Bild folgt den Fakten (Myers Briggs). Zudem folgt die Eigenschaft der Extrovertiertheit und Introvertiertheit: Ergibt sich die Kreativität und Energie aus verbalen Diskussionen mit anderen Menschen oder entspringen die Ideen aus dem Inneren (Pugh, 2010 S. 67)?

Ein einfaches Beispiel für die Relevanz zeigt folgende Abbildung:



*Abbildung 4: Kommunikationsbeispiel  
Quelle 7: Eigene Darstellung*

Wenn die Frage wäre, in welcher Reihenfolge die oben gezeigten Informationen in einer Präsentation gehalten werden müsste, was wäre dann die Antwort? Während einer die Informationen zeilenweise verarbeiten würde, würde der nächste spaltenweise vorgehen:

„Communication is about more than just words, it’s about how you organize those words.“ (Pugh, 2010 S. 68)

## Charter

Ein Projektcharter (Projektauftrag) dient zur Übersicht wichtiger Eigenschaften eines Projektes. Für die Klärung spezieller Punkte des Charters sollte der Kunde und eventuell relevante Stakeholder, wie zum Beispiel Sponsoren an den Initialen Meetings teilhaben. (Pugh, 2010 S. 40) Während in kleineren Projekten Entwickler und Tester selbst teilnehmen, sollte in größeren Projekten spezifische Rollen dies übernehmen (Projektleiter, einzelne Entwickler/Tester, die eventuell verminderte fachliche Kenntnisse besitzen).

In einem Charter sollten mindestens folgende Informationen enthalten sein:

* **V**ision
* **M**ission
* **Z**iele
* **P**rinzipien

Die Vision beschreibt einen Endzustand des Projektes, der nach Abschluss erreicht werden soll – somit der Hauptgrund für das Projekt. Die Mission wiederum ebnet den Weg, der gegangen wird, um die vorher genannte Vision zu erreichen. Zusammen mit beiden werden die Ziele definiert. Dabei handelt es sich um Projektziele und somit befindet sich diese auf einen hohen abstrakten, nicht technischen Level – zum Beispiel: In drei Monaten sollen angestellte 50% schneller Vertragsdaten vom Kunden abrufen können, um diese verwalten zu können. Letztlich dienen Prinzipien eine Form von Statements und somit Grundlagen für die Unterstützung von Entscheidungsfindungen, die im Verlauf des Projektes getroffen werden. (Pugh, 2010 S. 40)

Selbstverständlich ist es keine Pflicht, dies als zwingende Vorlage zu verwenden und vor allem größere Projekte sollten weitere Informationen beinhalten, wie zum Beispiel Stakeholder-Analysen, wichtige Annahmen, Einschränkungen, Risikoanalysen oder Kommunikationswege. (InLoox GmbH, N.A.)

### SMART-Ziele

Es ist wichtig, dass Ziele verfolgt werden können und im laufe der Zeit nicht verloren gehen oder vernachlässigt werden. Um sicherzustellen, dass Ziele deren erforderlichen Qualität entsprechen, sollten diese SMART definiert werden – Dabei ist SMART ein Akronym für Spezific, Mesasurable, Achievable, Relevant, Time boxed. Spezifisch zeichnet die genaue Definition des Ziels aus, was erreicht werden soll. Ein unklares Ziel hinsichtlich dem zu erreichenden Zustand kann nicht erfüllt werden. Weiterhin sollte der Output des Ziels messbar sein, um Richtwerte zu besitzen, die man erreichen möchte. Als nächstes müssen Ziele realistisch erreichbar sein. Wenn ein Ziel unrealistisch erscheint, ist das ein Zeichen dafür, dies in mehreren Zielen zu unterteilen oder es anzupassen. Es folgt die die Relevanz des Ziels, dass wiederum den Bezug zur Vision besitzt: Unterstützt das Ziel die Vision des Projekts? Als letztes sollten Ziele terminiert sein – unklare Terminierungen haben in der Regel Verschiebungen dieser Ziele zur Folge.

Aufgrund dessen sollten im optimalen Falle alle relevanten Instanzen der Triade – Kunde, Entwickler, Tester – an der Zielfindung teilhaben. Durch die verschiedenen Sichtweisen können die einzelnen Aspekte von SMART berücksichtigt werden. Wenn der Kunde zum Beispiel ein Ziel hat, sind Entwickler und Tester in der Lage technische High-Level-Abschätzung durchzuführen, um sicherzustellen, dass das Ziel realistisch in spezifischen Zeitfenstern erreichbar ist. (Pugh, 2010 S. 40)

Abbildung 5 zeigt ein Beispiel für ein Charter eines kleinen Projektes:



*Abbildung 5:* *Beispiel für Projektcharter*

*Quelle 8: In Anlehnung an Pugh, 2010, S41*

Die oben aufgefassten Projektziele, auch Management-Tests genannt, repräsentieren erste Projekt-Akzeptanztests, die genutzt werden, um die oberste Schicht der Anforderungen am Ende des Projektes zu überprüfen. Gute Management-Tests setzen ein zu erreichendes Ziel, spezifizieren aber nicht, wie dieses Ziel erreicht wird (Kerievsky, 2003). Weiterhin können beim Erstellen eines Projektcharters Ziele als zu groß erscheinen. Diese können folgende in Minicharter eingeteilt werden. So kann dies beispielshaft ein Charter stets an ein Release angelehnt sein, in dem das Zeitfenster bereits vorgegeben ist und somit oberste Priorität hat. (Pugh, 2010 S. 43)

## Feature

Ist das Grundgerüst gegeben, kann sich das Team – die Triade – gemeinsam treffen, um High-Level Anforderungen – Features – zu erstellen. Sie dienen zur Unterstützung der Projektziele und geben mehr Details über den zu erreichenden Zustand des Projekts. (Pugh, 2010 S. 43, 46)

Aus jedem Feature ergeben sich anschließend Akzeptanzkriterien:



*Abbildung 6: Aufteilung von Projektzielen in Features*

*Quelle 9: Eigene Darstellung*

Für die Findung von Features eignet sich oft ein kreatives Meeting, bei dem die Meinungen und Sichtweisen von mehreren Stakeholdern genutzt werden kann. Dabei kann ein Facilitator[[15]](#footnote-15) in Kombination mit speziellen Methoden (zum Beispiel Brainstorming) unterstützen, um die Kreativität zu fördern. (Pugh, 2010 S. 44)

Wurde eine Liste erstellt und auf die passenden Kern-Features für die Projektziele reduziert, können diese im Einzelnen für die dazugehörigen Akzeptanzkriterien analysiert werden. Während bei der Erstellung der Features der Tester beispielsweise eine restriktive Rolle spielt, kann dieser bei den Akzeptanzkriterien mehr unterstützen. Tester, aber auch Entwickler, haben meist bereits eine abstrakte Vorstellung für relevanten Input von Testdaten hinsichtlich Features. Wurden sich auf die Kriterien geeinigt, sollte eine klare Priorität erstellt werden. Dabei obliegt es mitunter der Entwicklungsphilosophie und dem Wert für den Kunden, ob Features beispielshaft in Iterationen innerhalb eines agilen Kontexts abgearbeitet werden oder andererseits in eine Queue gesetzt werden – wie es in Kanban der Fall ist. (Pugh, 2010 S. 45)

Abbildung 7 zeigt ein Beispiel von den aus Abbildung 5 entnommenen Projektzielen:



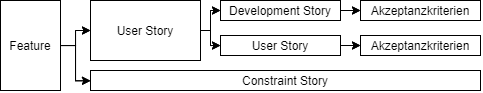
*Abbildung 7: Beispiel für Feature-Akzeptanzkriterien*

*Quelle 10: In Anlehnung an Pugh, 2010, S45*

Dabei wurden durch die Betrachtung der Projektziele drei Features erstellt. Diese besitzen jeweils ein Akzeptanzkriterium, das erfüllt werden muss, um dazugehöriges Feature als abgeschlossen werten zu können.

## Story

Features legen in der Regel keine direkten Low-Level-Aufgaben fest und müssen daher in sogenannten Stories vereinfacht werden. Erneut gilt bei den Meetings für die Stories eine mögliche Anpassung der teilnehmenden Stakeholder – beispielsweise kann hier ein Sponsor wegbleiben, da seine Wünsche in den Features bereits geäußert wurden. Generell können Stories verschiedene Arten sein, die sich nach der jeweiligen Aufgabe spezifizieren lassen. Hat ein Benutzer direkt mit einer Story zu tun, so wird diese “User Story“ genannt, während bei Stories hinsichtlich individueller Bedingungen oder Anwendungen gegenüber dem System beinhalten “Constraint Story“ genannt werden (dabei ist das Constraint zeitgleich Akzeptanzkriterium). (Pugh, 2010 S. 48) Weiterhin gibt es noch die “Developer Story“, die meist ebenfalls aus einer User Story hervorgeht und eine technische Anforderung für die Implementierung beinhaltet (Pugh, 2010 S. 54). Folglich wird für jede Story – ähnlich wie bei Features – Akzeptanzkriterien erstellt, die zum einen als Checkliste und zum anderen als erste Schätzung für den Aufwand bzw. der Größe dienen (Pugh, 2010 S. 54, 56). Abbildung 8 zeigt ein Beispiel zu dem oben genannten Thema:



*Abbildung 8: Aufteilung von Feature in Stories*

*Quelle 11: Eigene Darstellung*

Ähnlich wie bei dem Akronym SMART für Projektziele, existiert solch eins (INVEST) für die Erstellung von Stories:

* **I**ndependent
* **N**egotiable
* **V**aluable
* **E**stimable
* **S**mall
* **T**estable

Stories sollte unabhängig sein, um diese isoliert umsetzen zu können. Zweitens gehört die Verhandelbarkeit dazu, die eine Anpassung ermöglicht. Stories sollten so beschrieben werden, dass diese nicht ausspezifiziert sind – es besteht stets die Möglichkeit, kostengünstig neu über die Story zu verhandeln. Weiterhin sollten Stories einen Mehrwert besitzen: Eine Story ist kein Mittel zum Zweck, sondern einen in sich abgeschlossenen Mehrwert (z.B. Business-Value). Zudem sollte eine Story Schätzbar sein, um Prioritäten oder Schwierigkeiten ausfindig machen zu können. Darüber hinaus sollten Stories klein gehalten werden, um die beinhaltende Aufgabe möglichst in kleinen Schritten abschließen zu können. Zuletzt ist eine gute Story testbar, was innerhalb des A-TDD Kontexts eine hohe Priorität besitzt. (Schneider, 2017)

Für die Erstellung kann das eine Beschreibung aus Extreme Programming genutzt werden:

Als <***Rolle***> möchte Ich <***Funktion***>, sodass <***Grund***>

Zum Beispiel kann eine Beschreibung wie folgt aussehen:

Als ***Angestellter*** möchte Ich ***Kundeninformationen*** eintragen können, **sodass** dessen Daten für einen Vertrag genutzt werden können.

(Pugh, 2010 S. 48)

Wichtig zu beachten ist weiterhin, dass spätestens bei den Stories Projektabhängige Begrifflichkeiten geklärt werden sollten. Dabei gilt es – zum Beispiel in der Triade – gemeinsam fachliche Begriffe zu definieren, um im Team ein einheitliches Verständnis diesbezüglich aufzubauen. Somit werden Missverstände vermieden und eine bessere Kommunikation gefördert. (Pugh, 2010 S. 54)

## Rollen und Persona

Wie bei einheitlichen Begriffen sollte ebenfalls möglichst vorab Rollen und Persona für die Stories definiert werden. Dabei dienen die Rollen für die Darstellung von verschiedenen User-Arten, die die Anwendung in spezifischen Szenarien und Prozessen bedienen und gliedern sich nach Interessen sowie Aufgaben. Rollen könnten beispielshaft ein Admin oder Standardnutzer bzw. Kunde sein. Anzumerken ist, dass Rollen keine Individuen sind, sondern allgemeine Darstellungen sind, die ein Individuum annehmen kann. (Pugh, 2010 S. 56)

Wenn Rollen gefunden wurden, lassen sich dazu Rollenattribute definieren. Diese beschreiben die Rolle im Detail für mehr Verständnis und die Wichtigkeit dieser. Dazu können Themen, wie die Nutzungsfrequenz, die Domäne-Erfahrung (Schnittstellen/Systembereiche), das technische Wissen sowie generelle Ziele notiert werden. Attribute helfen zudem beim Finden weiterer Rollen und weiterhin eignen sich Rollen gut für Exploratory Test. (Pugh, 2010 S. 49, 50)

Abbildung X zeigt eine einfache Rollenbeschreibung für eine Makleranwendung:



*Abbildung 9: Beispiel für eine Rollenbeschreibung*

*Quelle 12: Eigene Darstellung*

Um mehr Transparenz für die Aufgaben und Szenarien des Systems zu schaffen, kann man zusätzlich Persona für jede Rolle erstellen. Dabei wird das Verständnis vertieft und ein klareres Bild zu der Rolle geschaffen. Die Persona beschreibt eine imaginäre Person, deren Verhalten, Eigenschaften, Charakter und Hobbies aufgezählt werden. (Pugh, 2010 S. 50, 51)

Nimmt man sich obiges Beispiel, lässt sich dazu folgende Persona definieren:



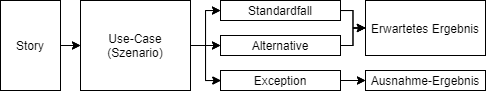
*Abbildung 10: Versicherungsmakler-Persona Beispiel*

*Quelle 13: Eigene Darstellung*

Durch die Persona, die durch die Triade erstellt werden kann, ergibt sich für die Entwickler ein besseres Bild, welche Punkte bei der Umsetzung für den Kunden zu beachten sind. Die Art und Betrachtung der Perspektive der Beschreibung liefern oft neue Erkenntnisse über weitere Akzeptanzkriterien oder Anpassungen.

## Szenario

Im Falle einer User Story gilt es anschließend passende Szenarien zu finden. Ein Szenario beschreibt einen Anwendungsfall Schritt für Schritt, in dem eine Rolle die in der Story beschriebene Aufgabe durchführt, um das Ziel zu erreichen. Allgemein können Szenarien drei Teilbereiche beinhalten, die bei der Umsetzung beachtet werden müssen: der Standardfall, mögliche Alternativwege, die durch besondere Umstände eintreten könnten sowie Exceptions, die auftreten, sobald etwas unvorhergesehenes passiert.



*Abbildung 11: Erstellung von Szenarien anhand User Stories*

*Quelle 14: Eigene Darstellung*

Durch die gemeinsame Entwicklung mit dem Kunden des Szenarios wissen vor allem die Entwickler und Tester der Triade, welche Schritte im Detail implementiert werden müssen und welche Fälle getestet werden sollten. Szenarien bieten eine gute Basis für Akzeptanztests und sollten daher gut analysiert werden. Dabei gilt es dennoch zu beachten, dass Use-Cases erst dann entwickelt werden sollten, wenn die Story benötigt wird – ansonsten könnte es zu einer Analysis Paralysis[[16]](#footnote-16) führen.

Für einen einfachen Use Case könnte folgendes Template genutzt werden:



*Abbildung 12: Einfaches Use Case Template*

*Quelle 15: In Anlehnung an Pugh, 2010, S60, S61, S62*

Eine Exception könnte zum Beispiel das Scheitern des Anlegens eines neuen Vertrages aufgrund von Verbindungsproblemen sein. Dabei würde man anfügen, was zu tun ist – zum Beispiel die Schritte für das Anlegen zweimal Wiederholen, ansonsten manuell anlegen. Alternative Wege fungieren ähnlich zu den Exceptions. Dabei ist aber stets zu unterscheiden, ob die Post-Condition trotzdem erreicht wird oder ein Sonderausgang erzeugt wird. Exceptions können auch durch das nicht einhalten einer Business Rule auftreten. Dabei wird diese zusätzlich angegeben, um Details über das Problem zu besitzen. (Pugh, 2010 S. 60)

Nachdem das Szenario erstellt wurde, können die Akzeptanztests im Detail geklärt werden. Dafür dient die Pre-Condition des Szenarios stets als Setup für den Test und die Post-Condition für die Verifizierung. Wie weiter oben erwähnt, sollten folglich der Normalfall, alle Exceptions, Alternativen und die Business Regel getestet werden. (Pugh, 2010 S. 63, 68)

Zuletzt eignen sich Use-Cases gut für die Dokumentation (zum Beispiel inklusive Use-Case-Diagramme), da externe Personen durch die Beispiele sich schnell ein geeignetes Verständnis über den fachlichen Prozess sowie die dahintersteckende Implementierung aneignen können.

## Test

Als letzten Schritt gilt es, die detaillierten Tests zu erstellen. Dabei sollte erneut die Triade im Ganzen diese erstellen, da hier alle perspektiven unterschiedliche Tests aufdecken können. (Pugh, 2010 S. 69) Dennoch gehört es zum Entstehungsprozess dazu, dass diverse Tests erst bei der Erstellung anderer Tests, während der Implementierung oder durch Exploratory Tests gefunden werden könnten. Bis zu einem gewissen Grad ist das hin und wieder legitim, wobei dessen Ursache bei erhöhtem Auftreten untersucht werden sollte (Pugh, 2010 S. 70). Weiterhin sollten die Tests an der Sprache der Domäne angelehnt werden, um ein transparentes Verständnis zu schaffen – jedes Szenario ist individuell, besitzt unterschiedliche Arten sowie Anzahlen von Tests und sollte daher möglichst einfach sprachlich dargestellt werden.

Für die Darstellung der Tests wird die Given-When-Then-Struktur verwendet. Dabei wird der Test in drei Bereiche gegliedert: Aufbau, Durchführung, Verifizierung - für die benötigten Informationen können verschiedene Arten von Tabellen genutzt werden. (Pugh, 2010 S. 71, 81)

### Calculation Table

Folgende Struktur eignet sich für Informationen, die für Test-Berechnungen benötigt werden. Meist werden diese auch für Business Rules verwendet, da dort spezielle Berechnungsregeln oder Ja-Nein Bedingungen enthalten sind:

|  |  |  |  |
| --- | --- | --- | --- |
| **Title** | | | |
| *Input Name 1* | *Input Name 2* | *Output Name?* | *Anmerkungen* |
| Eingabewert 1.1 | Eingabewert 2.1 | Ausgabewert 1 | Beschreibung |
| Eingabewert 1.2 | Eingabewert 2.2 | Ausgabewert 2 | Beschreibung |

*Abbildung 13: Template für Calculation Table*

*Quelle 16: In Anlehnung an Pugh, 2010, S73*

Die Tabelle kann somit genutzt werden, um eine Übersicht der zu verifizierenden Ergebnisse in Abhängigkeit der Inputs zu erhalten.

### Data Table

Die Data Table stellt Testrelevante Informationen dar. Dabei muss diese nicht der originalen Datenbanktabelle entsprechen, sondern kann eine beliebige Collection sein:

|  |  |
| --- | --- |
| **Title** | |
| *Attribut 1* | *Attribut 2* |
| Attributwert 1.1 | Attributwert 2.1 |
| Attributwert 1.2 | Attributwert 2.2 |

*Abbildung 14: Template für Data Table*

*Quelle 17: In Anlehnung an Pugh, 2010, S74*

### Action Table

Eine Action Table ist hilfreich, um einen strukturierten Ablauf von mehreren Schritten innerhalb eines Tests aufzuzeichnen. Diese können zum Beispiel User Interface Aktionen beschrieben werden – in diesem Falle fungiert die Action Table als Dialogbox:

|  |  |  |
| --- | --- | --- |
| **Action Name** | | |
| *Enter* | Input Name 1 | Eingabeinput 1 |
| *Enter* | Input Name 2 | Eingabeinput 2 |
| *Press* | Bestätigen | |
| *Check* | Output Name | Ausgabewert |

*Abbildung 15: Template für Action Table*

*Quelle 18: In Anlehnung an Pugh, 2010, S75*

## Testbeispiel

Im vorherigen Kapitel wurde beispielshaft ein Feature aufgelistet:

***Suchen nach Versicherungsdaten***

* Teildaten einer Versicherung können zum Suchen angegeben werden; stelle sicher, dass bei konsistenter Eingabe passende Versicherungsdaten angezeigt werden.

Der einfachheitshalber wird keine konkrete Story bzw. Szenario erstellt, sondern direkt darauf basierend ein Test formuliert:



*Abbildung 16: Testbeispiel für Tabellenarten*

*Quelle 19: Eigene Darstellung*

Selbstverständlich ist zu beachten, dass dies ein sehr vereinfachtes Beispiel ist. Es soll lediglich die Kombination der verschiedenen Tabellenarten innerhalb eines Tests zeigen. Sollten Exceptions getestet werden, so bleibt die Struktur im Grunde gleich, außer dass das Ergebnis meist unter dem When-Aspekt erscheint, da die Exception für ein anderes Ausgangsszenario sorgt, das von der Post-Condition des Szenarios abweicht. Somit gilt es, die passende Fehlermeldung bzw. Fehleraktion zu verifizieren. Dies könnten zum Beispiel Fehlermessages sein, welche einen eindeutigen Code beinhalten. Nehme man an, dass im obigen Beispiel die Kundennummer falsch ist, so könnte ein When-Statement wie folgt aussehen:



*Abbildung 17: Beispiel für Exception-Test*

*Quelle 20: Eigene Darstellung*

Nachdem ein Test formuliert worden ist, sollte dieser im Anschluss nochmal bewertet werden. Oftmals lassen sich weitere relevante Testdaten finden und ergänzen. (Pugh, 2010 S. 77)

Weiterhin können Tests auch als reine Textform dargestellt werden. Dabei hängt dies von dem Team und den verwendeten Technologien ab. (Pugh, 2010 S. 80)

# Literaturverzeichnis

**Meszaros, Gerard. 2007.** *xUnit Test Patterns - Refactoring Test Code.* USA : Pearson Education, Inc., 2007. 978-0-13-149505-0.

**Pugh, Ken. 2010.** *Lean-Agile Acceptance Test-Driven Development.* (UTB Schlüsselkompetenzen, 3218). USA : Pearson Education, Inc., 2010. 978-0-321-71408-4.

1. Akzeptanztest in der klassischen Entwicklung (auch Abnahmetests genannt) dienen zur Überprüfung, ob die Anforderung erfolgreich implementiert worden ist. Dabei gilt es lediglich um die Erfüllung eines Tests, nicht um die Art und Weise der Abnahme und Erstellung dieser. [↑](#footnote-ref-1)
2. Tests, die in der Regel überprüfen, ob das System die Anforderungen erfüllt und richtig arbeitet. [↑](#footnote-ref-2)
3. Tests, dessen zu prüfenden Implementierungen nicht bekannt sind. [↑](#footnote-ref-3)
4. Testebene, im Sinne der vier Teststufen: Unittests, Integrationstests, Systemtests, Abnahmetests [↑](#footnote-ref-4)
5. Eine spezielle Entwicklerrolle, die auf Softwaretesten spezialisiert ist. [↑](#footnote-ref-5)
6. Entwicklermethode, in der das Lösen von Programmieraufgaben im Vordergrund steht und Anforderungen des Kunden in kleinen Schritten annähert. [↑](#footnote-ref-6)
7. Eine Kommunikation mit dem Ziel, einheitliche Begrifflichkeiten zu verwenden und ein gegenseitiges Verständnis zu gewährleisten. [↑](#footnote-ref-7)
8. Ein Format für die Erstellung von Tests in einer formalen Sprache, die für Techniker und Kunden verständlich ist. [↑](#footnote-ref-8)
9. Ein Fehlerbericht, der alle relevanten Informationen bezüglich eines Bugs oder in diesem Falle Unklarheiten beinhaltet. [↑](#footnote-ref-9)
10. Ein Rückgang zum vorherigen Prozessschritt – in diesem Falle die Entwicklung der Anforderung selbst. [↑](#footnote-ref-10)
11. Tests, die sich in Form von Unit- / Integrations- / System- / Abnahmetests eingliedern lassen. [↑](#footnote-ref-11)
12. (Pugh, 2010 S. 20) [↑](#footnote-ref-12)
13. Vortäuschen, im Sinne von festgesetzten Rückgabewerten von externen Schnittstellen, um diese nicht ausführen zu müssen und unabhängig Testen zu können. [↑](#footnote-ref-13)
14. Der Test durchläuft nicht eine Schicht unabhängig, sondern die komplette Anwendungsschicht (UI, Service, Backend). [↑](#footnote-ref-14)
15. Ein Moderator, der Kontakt, Fokus, Kreativität und Verbindlichkeiten in Gruppen fördert. [↑](#footnote-ref-15)
16. Übermäßiges Analysieren einer Situation, welches zu Hinauszögerungen oder zu Verkomplizierungen von Entscheidungen führt. [↑](#footnote-ref-16)