HAW HAMBURG

BACHELORTHESIS
Jannik Reinefeld

Bewertung von Methoden des Test-Driven und Behaviour-Driven Development anhand eines Beispiels in Java

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Jannik Reinefeld

Bewertung von Methoden des Test-Driven und Behaviour-Driven Development anhand eines Beispiels in Java

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Wirtschaftsinformatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Ulrike Steffens
Zweitgutachter: Prof. Dr. Bettina Buth

Eingereicht am: 20. Dezember 2022

Inhaltsverzeichnis

Gender Disclaimer

Die in dieser Arbeit gewählte männliche Form bezieht sich immer zugleich auf weibliche, männliche und diverse Personen. Auf eine Mehrfachbezeichnung wird in der Regel zugunsten einer besseren Lesbarkeit verzichtet.

1 Kurzfassung

Test-Driven Development und Behaviour-Driven Development sind zwei Methoden, um mit der Hilfe von Tests qualitativeren Code zu schreiben, welcher weniger Fehler aufweist und besser lesbar ist. Dies schafft Test-Driven Development dadurch, dass hier immer erst ein Test geschrieben wird, bevor der eigentliche Produktionscode verfasst wird. Behaviour-Driven Development bildet User Storys als Testfälle da. Somit können neue Features schnellstmöglich zusammen mit den Fachexperten entwickelt und getestet werden. Weiter bieten sie dem Entwickler ein klares Bild der zu implementierenden Features dar.

Test-Driven Development hat jedoch die Schwäche, dass dieses zu viel Code testet, auch Code, welcher vorher bereits getestet wurde und wird somit häufig als zu zeitaufwendig angesehen. Behaviour-Driven Development hat die Schwäche, dass diese häufig als nicht mehr notwendig angesehen werden, sobald Ende zu Ende Tests vorhanden sind. Beide schaffen es jedoch die Codequalität zu erhöhen. Vor- und Nachteile beider Verfahren werden in dieser Arbeit genau beleuchtet und evaluiert.

1 Abstract

Test-driven development and behavior-driven development are two methods of testing to write better code with higher quality code that has fewer bugs and is more readable. Test-driven development achieves this by always writing a test before the actual production code is written. Behavior-Driven Development creates user stories as test cases. This means that new features can be developed and tested together with technical experts as quickly as possible. They also provide the developer with a clear picture of the features to be implemented.

However, test-driven development has the weakness that it tests too much code, including code that has already been tested, and is therefore often considered too time-consuming. Behavior-driven development has the weakness that once end-to-end tests are in place, these are often considered unnecessary. However, both manage to increase the code quality. The advantages and disadvantages of both methods are examined and evaluated in detail in this work.

2 Einleitung

Tests sind ein tragender Bestandteil des Programmierens, dies ist allgemein bekannt. In der Realität werden jedoch kaum Tests geschrieben, da diese mit mehr Aufwand zusammenhängen und sie am Code nichts verändern. Häufig werden Tests sogar erst geschrieben, sobald der Code bereits vollständig ist und nachdem per Hand die Fehler behoben wurden.

Zumeist werden Tests nur geschrieben, weil es gemeinhin als guter Stil gilt Tests zu haben. Wieso sollten Tests überhaupt geschrieben werden? Häufig sehen Programmierer in ihnen Zeitverlust und sie bereiten ihnen unnötiges Kopfzerbrechen. In gleicher Zeit könnte genauso gut andere Software programmiert werden, die ebenfalls fertig werden muss.

So oder so ähnlich denken viele, einschließlich mir, über das Thema Tests. Zumindest bis mir das Thema Test-Driven Development nahegelegt wurde. Erst war ich auch hier skeptisch. Tests schreiben, bevor der eigentliche Code entsteht? Kostet das nicht zu viel Zeit? Es werden ja weitere Kosten verursacht. Kosten für welchen Preis. Dies ist leider auch immer wieder ein Thema in der Industrie. Viele Firmen sagen gerne: „Verzichte doch auf die Tests, du bist ja ein guter Programmierer. Das spart Zeit und Geld.“ Aber spart dies wirklich Zeit und Geld? Ich durfte selber die Erfahrung machen, ein Programm erst ohne die Kenntnisse von Test-Driven Development und danach, 3 Monate später, das gleiche Programm nochmal komplett neu mit den Techniken des Test-Driven Development und des Behaviour-Driven Development zu schreiben. Daraus ergibt sich die Frage: Hat sich etwas an der Qualität des Codes geändert? Wie haben mir die Methoden des Test-Driven Development und des Behaviour-Driven Development hier geholfen?

Insbesondere diese und weitere vertiefende Fragen sollen in dieser Thesis geklärt werden. Zunächst wird dazu im Folgenden erörtert, was qualitativ hochwertiger Code ist. Hier werden Qualitätskriterien von Code betrachtet, um einen Maßstab zu finden,

woran dies gemessen werden kann und ob sich die Qualität von Code mithilfe von TDD und BDD verbessert. Danach werden sich die Techniken des BDD (Behaviour-Driven Development), sowie das richtige Definieren von Anforderungen genauer angeschaut und schließlich an Beispielen erläutert. Anschließend wird die Funktionsweise des TDD (Test-Driven Development) erklärt und mit Beispielen dargelegt. Im Kern soll im Fazit schließlich die Frage beantwortet werden, ob der Einsatz von TDD zu einem qualitativ hochwertigen Code.

In dieser Bachelorarbeit wird nur auf das Thema TDD und BDD in Programmen mit Java eingegangen. In der Demo kann auch ein Frontend, welches mithilfe des Frameworks Angular geschrieben wurde, betrachtet werden. Durch das Frontend kann die Anwendung danach komplett von einem Endnutzer benutzt werden und dient der Bachelorarbeit als Demonstration. Hierfür wurde jedoch kein TDD oder BDD genutzt. Das Frontend ist ganz ohne Tests und dient nur dazu, die Nutzbarkeit des Programmes, welches mit Java erstellt wurde, zu verbessern. Weiter wird in der Arbeit das Thema Ende zu Ende Tests immer wieder vorkommen. Diese werden jedoch in dieser Arbeit nicht im Detail beleuchtet. Um die Anwendung zu starte, siehe Entwicklerdokumentation Anhang 2.

Die Bachelorarbeit strukturiert sich folgendermaßen. Zuerst werden Qualitätskriterien von Code betrachtet, um ein Maßstab zu finden, woran gemessen werden kann, ob sich die Qualität von Code mithilfe von TDD und BDD verbessert. Anschließend werden User Storys erörtert, um das nötige Grundwissen für BDD zu erlangen. Die ganze Arbeit hangelt sich an einem Beispielprogramm lang, für welches hier User Storys erstellt werden. Folglich werden nun anhand dieser User Storys Tests angelegt. Hierfür wird das Thema BDD von dem Thema TDD abgegrenzt und danach werden die ersten BDD-Testfälle mithilfe des Cucumber Frameworks erstellt. Danach wird das Thema TDD genauer untersucht. Hier wird der TDD-Zyklus genauer erklärt und anschließend mithilfe dessen Tests geschrieben.

Nachdem TDD und BDD genauestens beleuchtet wurden, wird ein Meinungsbild, welches mit der Hilfe von zehn befragten Entwicklern erstellt wurde, erläutert und

anschließend evaluiert. Danach folgt die Evaluation der Kernfrage, ob TDD und BDD zu besserem Code führen. Damit überhaupt evaluiert wird ob TDD und BDD zu besserem Code führen können, wird sich im folgenden Kapitel definiert was guter Code überhaupt ist.

3 Qualitätskriterien für Code

Im folgenden Kapitel werden die Qualitätskriterien für guten Code dargestellt. Es wird dargelegt, was guten Code ausmacht und was ihn von schlechtem Code unterscheidet. Zunächst geht es also um die Frage: Was ist qualitativ hochwertiger Code? Um dieser Frage auf den Grund zu gehen, wird die Kopfstandmethode verwendet, das heißt, es wird sich das Gegenteil angeschaut. Also was ist eigentlich ein schlechter Code?

3.1 Schlechter Code

Wenn ein schlechter Code gelesen wird, kann zumeist erstmal kaum etwas verstanden werden. Der Code wird nach Hinweisen durchsucht, um heraus zu finden, was dieser eigentlich tut und wie er es tut.[1] Schlechter Code kann metaphorisch mit einem Escape Game verglichen werden, bei dem verzweifelt nach Hinweisen gesucht wird, um das Rätsel zu lösen. Es ergeben sich Fragen, wie: Welcher Code löst überhaupt welches Problem?

Dieses Chaos entsteht, wenn der Programmierer unter Zeitdruck arbeitet und nur mal schnell etwas ausprobieren möchte und wenn sich nicht vorher überlegt wird, wie was organisiert sein sollte. Dahinter steckt der Gedanke, dass dies auch noch später organisiert werden könnte.[2]

Schlechter Code oder eine „Quick and Dirty“ Lösung funktioniert im ersten Schritt natürlich, aber wenn dieser Code das Fundament für die weitere Arbeit darstellen soll, auf dem später aufgebaut werden möchte, so gerät dieses Gebilde schnell ins Wanken. Dies kann dazu führen, dass es günstiger ist, den Code zu verwerfen und alles neu zu schreiben, anstatt die Fehler zu verbessern.

[1] (vgl. Martin, 2009, S. 28)
[2] (vgl. Martin, 2009, S. 28)

Ferner ist das Verbessern von schlechtem Code nicht immer zielführend. Besonders, wenn das Fundament bereits schlecht ist. „Wenn andere schlechten Code ändern, neigen sie dazu, ihn noch schlechter zu machen.“[3]

3.2 Guter Code

> Bjarne Stroustrup, Erfinder von C++ und Autor von The C++ Programming Language:
>
> „Mein Code sollte möglichst elegant und effizient sein. Die Logik sollte gradlinig sein, damit sich Bugs nur schwer verstecken können, die Abhängigkeiten sollten minimal sein, um die Wartung zu vereinfachen, das Fehler-Handling sollte vollständig gemäß einer vordefinierten Strategie erfolgen, und das Leistungsverhalten sollte dem Optimum so nah wie möglich kommen, damit der Entwickler nicht versucht, den Code durch Ad-hoc-Optimierungen zu verunstalten. Sauberer Code erledigt eine Aufgabe gut.“[4]
>
> Big Dave Thomas, Gründer der OTI, der Pate (Godfather) der Eclipse-Strategie:
>
> „Sauberer Code kann von anderen Entwicklern gelesen und verbessert werden. Er verfügt über Unit- und Acceptance-Tests. Er enthält bedeutungsvolle Namen. Er stellt zur Lösung einer Aufgabe nicht mehrere, sondern eine Lösung zur Verfügung. Er enthält minimale Abhängigkeiten, die ausdrücklich definiert sind, und stellt ein klares und minimales API zur Verfügung. Code sollte literate sein, da je nach Sprache nicht alle erforderlichen Informationen allein im Code klar ausgedrückt werden können.“[5]

Bjarne Stroustrup spricht hier davon, dass sauberer Code eine spezielle Aufgabe gut erledigen sollte. Es ließ sich bereits feststellen, dass schlechter Code entsteht, wenn sich nicht vorher überlegt wird, wie der Code organisiert sein soll. Es muss also bereits vorher überlegt werden, wie diese bestimmte Aufgabe des jeweiligen Codes aussieht. Das heißt, die Aufgabe muss im Vorfeld definiert werden und es darf auch nur diese eine Aufgabe möglichst effizient gelöst werden.

Wie werden diese Aufgaben definiert? Dave Thomas schreibt dazu, dass guter Code über Unit- und Acceptance-Tests verfügt. Somit können Aufgaben einfach in Unit- und Acceptance Tests definiert werden. Nachfolgend wird deshalb einmal erläutert, was Unit- und Acceptance-Tests sind. Bevor jedoch erklärt werden kann, was Unit- und

[3] (Martin, 2009, S. 33)
[4] (Martin, 2009, S. 32)
[5] (Martin, 2009, S. 35)

Acceptance-Tests sind müssen für die Acceptance-Tests erst einmal Anforderungen definiert werden. Wie genau diese Anforderungen aussehen und wie diese im Zusammenhang mit User Storys stehen wird im nachfolgendem Kapitel erläutert.

4 Anforderungen erfassen

Bevor Acceptance-Tests geschrieben werden können müssen die Anforderungen, auf welche diese aufbauen, geschrieben werden. In diesem Kapitel wird das Erstellen von Anforderungen mithilfe von User Storys erläutert und Beispiele zu möglichen User Storys für das begleitende Blackjack-Programm entwickelt. Unter dem Namen Acceptance-Tests fallen viele verschiedenen Auffassungen. In der Regel werden sie jedoch von dem entwickelnden Team geschrieben, um zu definieren, wann die Software, das Feature oder das Epic fertig sind. Hierfür muss das Team sich jedoch erstmal einig sein, was „fertig“ überhaupt bedeutet. Bedeutet fertig, wenn die Software das erste Mal läuft? Wenn alle Funktionen eingebaut sind oder erst, wenn diese auch getestet wurden? Professionelle Entwickler haben hier nur eine Definition. Fertig ist eine Software, wenn der gesamte Code geschrieben ist, dieser alle Tests besteht, vom Qualitätsmanagement abgenommen und von den Stakeholdern abgesegnet wurde.[6]

Hierfür müssen natürlich alle Anforderungen umgesetzt worden sein. Bevor sich also direkt mit Unit- und Acceptance-Tests beschäftigt werden kann, sollte der Programmierer einen Schritt vom Code zurücktreten und die Anforderungen an das Programm definieren.

4.1 User Storys

Es wird mit einem leichtgewichtigen und flexiblen Anforderungsformat begonnen, welches User Storys heißt. Der Programmierer muss genau wissen, wie die Software sich verhalten soll und dieses Verhalten kann genau in diesen User Storys definiert werden. Danach kann sich angeschaut werden, was Acceptance-Tests beinhalten müssen und wie diese aufgebaut sein sollten.[7]

[6] (vgl. Martin, 2012, S. 1127)
[7] (vgl. Koskela, 2008, S. 324)

Ein wichtiger Teil beim Schreiben von User Storys ist das Team. Das Schreiben von User Storys ist eine Teamaufgabe und ein Teamprozess. In diesem Prozess wird darüber geredet, was am Ende überhaupt getestet werden soll.[8]

User Storys sind ein sehr einfacher Weg, um Anforderungen kurz und bündig darzustellen. In seiner klassischen Form ist dies ein kurzer Satz, welcher beinhaltet *wer* etwas *wie* tut und *wieso*. In der Praxis wird das *Wieso* häufig weggelassen, da sich dies bereits aus dem *wer* und *wie* ergibt. Der Grund, wieso diese Anforderungen nur einen Satz lang sind, ist der, dass diese nicht für die Dokumentation gedacht sind, sondern vielmehr für die Repräsentation der Anforderung stehen. [9]

Dabei wird sich häufig an folgendes Format gehalten: „Als (Rolle) möchte ich (Funktionalität), so dass (Mehrwert)“. Dies ist jedoch eher als Richtlinie zu verstehen. Wenn die Anforderung so nicht richtig darstellbar ist, kann hiervon auch abgewichen werden.[10]

4.1.1 User Story – Beispiel

Im Folgenden soll dieses Prinzip nun beispielhaft an einem Programm, welches das Spielen von Blackjack ermöglicht, angewendet werden. Blackjack ist wohl eines der populärsten Karten-Glücksspiele in den Casinos der Welt. Das Ziel des Spiels besteht darin, den Dealer zu schlagen. Dafür muss mit zwei oder mehr Karten näher an die 21 gekommen werden, als der Dealer.

Um das Beispiel einfacher zu halten, werden die Optionen des Verdoppelns und des „Splitt“ nicht mit einbezogen. Mit der Grundlage zu dem Spiel können nun die User Storys definiert werden.

- Als Spieler möchte ich eine neue Karte ziehen können, um meinen Kartenwert zu erhöhen.

[8] (vgl. Koskela, 2008, S. 324)
[9] (vgl. Koskela, 2008, S. 325)
[10] (vgl. Koskela, 2008, S. 325)

- Als Spieler möchte ich in meinem Zug die Möglichkeit haben, keine Karte zu ziehen, damit ich meinen Kartenwert behalte.
- Als Spieler möchte ich aufgeben können, um die Hälfte meines Einsatzes zurück zu bekommen.
- Als Dealer möchte ich den Kartenwert bestimmen können.
- Als Dealer möchte ich erkennen, wenn ein Spieler das Spiel verloren hat
- Als Dealer möchte ich erkennen, wenn ein Spieler gewonnen hat.
- Als Dealer möchte ich am Start einer Runde Karten austeilen

4.2 Acceptance Tests

Acceptance Tests sind Spezifikationen des gewollten Verhaltens und der Funktion. Sie geben für eine gegebene User Story an, wie das System bestimmte Situationen behandelt, mit welchen Eingaben es dies tut und wie das Ergebnis aussehen soll.[11]

Acceptance-Tests sollten einer Person gehören, meistens dem Kunden. Somit kann dieser genau einschätzen, ob sein Verhalten bereits in der Software implementiert wurde. Jedoch muss diese Person nicht zwingend auch den Acceptance-Test schreiben. Am sinnvollsten ist es, wenn die Person, welche den Test schreibt, ein Experte in diesem Thema ist. In dem Beispiel vom Blackjack Spiel, sollte dies also zum Beispiel ein professioneller Spieler, ein erfahrener Croupier oder am besten noch, der Erfinder des Spiels selbst sein. Bei Acceptance-Tests geht es nicht um das „wie wird etwas technisch umgesetzt“ sondern „wie funktioniert etwas fachlich“. Hier kann es oft hinderlich sein, wenn der Entwickler selber die Acceptance-Tests schreibt.[12] Dieser hat möglicherweise nicht das komplette fachliche Bild im Kopf, sondern konzentriert sich direkt auf die Umsetzung. Das heißt, Acceptance-Tests sollten von der fachlich fähigsten Person zu dem Thema geschrieben werden.

Auch wenn der Experte der Besitzer der Tests sein soll, sollte dieser beim Schreiben der Tests nicht alleine gelassen werden. Besonders wenn das Schreiben von Acceptance-

[11] (vgl. Koskela, 2008, S. 327)
[12] (vgl. Koskela, 2008, S. 329)

Tests neu für denjenigen ist. Tests sollten immer im Team geschrieben werden. Dies fördert die Kommunikation und führt durch mögliche Diskussionen zu besseren Tests.[13]

Weiter muss darauf geachtet werden, dass die Tests nicht das „wie wird es geschehen" sondern das „was soll geschehen" bearbeiten.[14] Zu sagen: „Der Spieler drückt die Taste *E,* um eine neue Karte zu ziehen, bis er nicht mehr ziehen kann", ist nicht zielführend. Hier ist das *Wie* im Fokus. Besser wäre: „Der Spieler hat die Möglichkeit eine Karte zu ziehen, solange sein Kartenwert nicht 21 überschreitet".

Es sollte weiter eine klar lesbare Sprache genutzt werden. Wenn ein Entwickler die Acceptance-Tests in seinem Entwickler-Jargon verfasst, könnte es dem Kunden schwerfallen, dies zu verstehen. Acceptance-Tests sollten so einfach und prägnant wie möglich verfasst werden. Jeder Test sollte nur einen einzigen Aspekt abdecken.[15] So können später alle Funktionen individuell getestet werden.

4.2.1 Acceptance Tests – Beispiel

Nachfolgend werden für die zuvor definierten User Storys einmal Acceptance Tests definiert.

Story	Acceptance-Test
Als Spieler möchte ich eine neue Karte ziehen, um meinen Kartenwert zu erhöhen.	Der Spieler darf nur solange eine Karte ziehen, solange sein Kartenwert unter 22 liegt.
Als Spieler möchte ich in meinem Zug die Möglichkeit haben, keine Karte zu ziehen, damit ich meinen Kartenwert behalte.	Der Spieler kann jederzeit seinen Zug selbstständig beenden, wenn sein Kartenwert unter 22 liegt. Wenn der Spieler seinen Zug beendet, ist der nächste Spieler an der Reihe.

[13] (vgl. Koskela, 2008, S. 329)
[14] (vgl. Koskela, 2008, S. 330)
[15] (vgl. Koskela, 2008, S. 331)

	Wenn der Spieler seinen Zug beendet, ändert sich sein Kartenwert in der Runde nicht mehr.
Als Spieler möchte ich aufgeben können, um die Hälfte meines Einsatzes zurückzubekommen.	Wenn der Spieler aufgibt, wird die Hälfte seines Einsatzes wieder seinem Konto gutgeschrieben.
	Wenn der Spieler aufgibt, kann er kein zusätzliches Geld in dieser Runde gewinnen.
	Wenn der Spieler aufgibt, ist der nächste Spieler an der Reihe.
Als Dealer möchte ich den Kartenwert bestimmen können.	Der Kartenwert vom Ass beträgt 11, solange der summierte Kartenwert nicht über 21 liegt.
	Der Kartenwert vom Ass beträgt 1, wenn der Kartenwert sonst über 21 liegt.
	Der Kartenwert von allen Zahlenkarten beträgt den Zahlwert der Karte.
	Der Kartenwert aller Bildkarten, außer vom Ass, beträgt 10.
Als Dealer möchte ich erkennen, wenn ein Spieler das Spiel verloren hat.	Wenn ein Spieler einen Kartenwert über 21 hat, hat dieser verloren.
	Wenn der Dealer einen Kartenwert unter 22, aber über dem des Spielers hat, dann hat der Spieler verloren.
	Wenn der Dealer und der Spieler den gleichen Kartenwert haben, gibt es ein unentschieden.

	Wenn der Dealer nach dem Austeilen der Karten 21 hat, hat der Spieler verloren.
Als Dealer möchte ich erkennen, wenn ein Spieler gewonnen hat.	Wenn der Dealer einen Kartenwert über 21 hat, hat der Spieler gewonnen.
	Wenn der Dealer einen niedrigeren Kartenwert als der Spieler hat und der Kartenwert von dem Spieler unter 22 liegt, hat der Spieler gewonnen.
Als Dealer möchte ich am Start einer Runde Karten austeilen.	Der Dealer bekommt die erste Karte offen.
	Der jeweils nächste Spieler bekommt eine offene Karte, bis der Dealer an der Reihe ist.
	Der Dealer bekommt nach der ersten Karte eine zweite verdeckte Karte und prüft, ob der Kartenwert genau 21 ist. Der jeweils nächste Spieler bekommt seine zweite Karte offen.

Nachdem nun User Storys gebaut und dazu Acceptance-Tests entworfen wurden, sollen diese natürlich getestet werden. Diese Tests haben jedoch nicht viel mit den Tests des Test-Driven Development zu tun, obwohl sie doch gewisse Gemeinsamkeiten teilen. Um hier eine klare Trennung der beiden Begrifflichkeiten zu schaffen, werden im anschließenden Kapitel die Unterschiede und Gemeinsamkeiten der beiden Methoden erläutert.

5 Unterschiede und Gemeinsamkeiten von Behaviour-Driven Development und Test-Driven Development

Test-Driven Development und Behaviour-Driven Development sind recht unterschiedlich und fungieren auf anderen Ebenen im Code. In diesem Kapitel werden die Unterschiede und Gemeinsamkeiten der beiden Methoden herausgearbeitet.

Behaviour-Driven Development (BDD) baut auf die Grundlagen des Test-Driven Development (TDD) auf, indem es dieselben Praktiken wie TDD nutzt. Gute TDD Entwickler arbeiten jedoch von außen nach innen, sprich es wird mit BDD angefangen und mit TDD beendet. TDD wird im Verlauf dieser Arbeit noch weiter erläutert. BDD bringt, im Gegensatz zum TDD, den Vorteil gut lesbare Beispiele definieren zu können und diese dann nicht nur direkt mit den Stakeholdern zu diskutieren und Feedback zu bekommen, sondern auch die Möglichkeit diese Tests direkt automatisiert auszuführen.[16] TDD hingegen ist für Außenstehende wie zum Beispiel Stakeholder schwer verständlich, da diese sehr kleinteilig und detailliert sind sowie sehr nah am Programmcode liegen. BDD beschreibt häufig das große Ganze, es beinhaltet eine komplette User Story, somit können BDD Tests das komplette Programm testen, wobei TDD Tests häufig aus Unit-Tests bestehen, welche nur eine einzelne Unit testen.

[16] (vgl. Wynne & Hellesøy, 2012, S. 4)

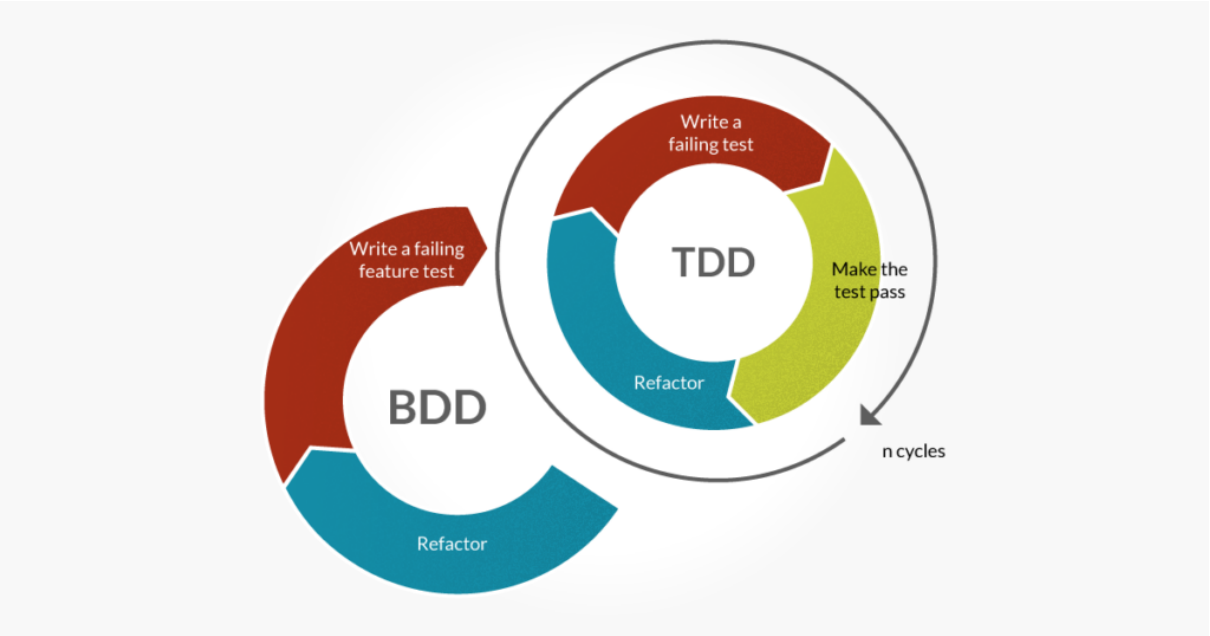

Abbildung 1 - Der BDD und TDD Zyklus[17]

In Abbildung 1 sind die Grundlagen für TDD und BDD anschaulich dargestellt. Beim BDD ist der Einstiegspunkt ein Feature- oder Acceptance-Test. Dieser Test wird immer zuerst geschrieben. Da dieses Feature jedoch natürlich noch nicht eingebaut ist, wird dieser Test fehlschlagen. Dies ist auch gewollt, denn das Ziel ist es einen fehlschlagenden Test zu schreiben.

Nachdem nun der Acceptance-Test geschrieben wurde, wird der TDD Zyklus erläutert. Auch dieser beginnt mit einem fehlschlagenden Test. Dies sind nun schon sehr technische Tests, im Beispiel von Java meist Unit-Tests. Beim TDD ist es wichtig, dass ausschließlich Code geschrieben wird, welcher diesen Test zum Erfolg bringt. Hierbei sollte nicht direkt die perfekte Lösung kreiert werden, sondern die möglichst einfachste Lösung, welche den Test erfüllt.

Soll zum Beispiel eine Methode geschrieben werden, welche Primzahlen erkennt, dann wird erst ein Test geschrieben, welcher eine Primzahl an die Methode übergibt und erwartet, dass die Methode den booleschen Wert „Wahr“ wiedergibt.

[17] Quelle: https://www.techiexpert.com/wp-content/uploads/2022/12/BDD.jpg

Dieser Test schlägt natürlich fehl, da die Methode noch nicht geschrieben wurde. Nun wird die einfachste Methode geschrieben, welche den Test zum Erfolg bringt. Sprich eine Methode, die bei jeder Zahl, welche sie bekommt, den booleschen Wert „Wahr“ zurückgibt. Der gestandene Programmierer würde an dieser Stelle natürlich sagen: „Halt! Die Methode ist doch komplett falsch, sie erkennt doch noch gar keine Primzahlen! Ich weiß doch, wie man Primzahlen erkennt und kann es doch direkt komplett hinschreiben“. Das ist natürlich richtig. TDD erzeugt auch die richtige Methode, jedoch durch den Ansatz, dass der Code bereits vor dem Schreiben schon getestet wird. Um also die Methode weiter auszubauen, wird ein weiterer Test geschrieben. Diesmal wird ein Test geschrieben, welcher der Methode keine Primzahl übergibt. Dieser Test schlägt nun logischerweise auch fehl. Die Methode gibt ja zum jetzigen Zeitpunkt immer den booleschen Wert „Wahr“ zurück. Also wird nun die Methode so umgebaut, dass sie diesen Test auch besteht. Hierbei muss jetzt natürlich darauf geachtet werden, dass vorherige Tests weiterhin erfolgreich sind.

Nachdem ein Test erfolgreich durchlaufen wurde, darf mit der Refrakturierung der Methode begonnen werden. Hierbei darf sich allerdings die Logik der Methode nicht ändern.

An dieser Stelle wurde der komplette TDD Zyklus einmal durchlaufen. Die eben beschriebenen Schritte des TDD werden nun so oft wiederholt, bis auch der Acceptance-Test erfolgreich ist. Danach kann der Acceptance-Test refaktoriert und dann der nächste Acceptance-Test geschrieben werden.

TDD ist also hier nur als eine Erweiterung von BDD zu sehen. Ein BDD Zyklus beinhaltet meist viele TDD Zyklen. Diese Zyklen sind jedoch nicht voneinander abhängig. Nach TDD kann auch ohne BDD entwickelt werden und andersherum. Jedoch testet TDD die kleinstmögliche Einheit als Code. BDD testet wiederum komplexe fachliche Zusammenhänge. Diese komplexen fachlichen Zusammenhänge müssen natürlich nicht nur als Acceptance-Tests definiert werden, sondern auch im Code wiederzufinden sein. Wie diese im Code übernommen werden und letztendlich als automatische Testresultate angezeigt werden können, wird im nächsten Kapitel erläutert.

6 Automatisierte Acceptance-Tests

Nachdem sich angeschaut wurde, wie das generelle Vorgehen beim BDD und TDD ist, wird jetzt veranschaulicht, wie die Acceptance-Tests automatisiert und im Code ausführbar gemacht werden können. Dies hat den Vorteil, dass die Tests bei jeder Änderung im Code mitlaufen können und so direkt gesehen werden kann, wenn ein Test durch die Änderungen fehlschlägt.

Hierfür wird in diesem Beispiel Cucumber genutzt. Cucumber ist ein Command-Line Tool, welches reine Textdokumente, sogenannte Feature-Files liest und die hier drin definierten Szenarios gegen das Programm laufen lässt. Jedes dieser Szenarios ist eine Liste von Schritten, welche Cucumber durcharbeitet. Damit Cucumber diese Feature-Files auch versteht, müssen diese in einer bestimmten Syntax vorliegen. Der Name dieser Syntax bei Cucumber ist Gherkin.[18]

6.1 Die Syntax Gherkin

Gherkin wird genutzt um die Acceptance-Tests, welche vorher definiert wurden, in für die Software lesbaren Code zu verwandeln.

Eine Gherkin-Datei nutzt die Dateiendung *.feature*. Da es sich hier um einfache Textdateien handelt, lassen sie sich durch einfachste Textbearbeitungs-Programme verändern. Somit ist Gherkin ähnlich zu Formaten wie zum Beispiel YAML, XML, Markdown oder Textile.[19]

Gherkin ist in vielen verschiedenen Sprachen verfügbar. Generell sollte darauf geachtet werden, dass dieselbe Sprache genutzt wird, die die fachlichen Experten verwenden, um Verwirrungen zu vermeiden und die Lesbarkeit zu erhöhen. [20]

Wir nutzen aus diesem Grund die Sprache Deutsch für Gherkin.

[18] (vgl. Wynne & Hellesøy, 2012, S. 7)
[19] (vgl. Wynne & Hellesøy, 2012, S. 28)
[20] (vgl. SmartBear Software, 2020)

Gherkins Struktur wird definiert mit verschiedenen Schlüsselwörtern. Diese Schlüsselwörter sind folgende:

- Funktionalität
- Grundlage
- Szenario
- Szenariogrundriss
- Beispiele
- Angenommen
- Wenn
- Dann
- Und
- Aber

6.1.1 Die Gherkin Schlüsselwörter

Jede Gherkin-Datei startet mit dem Schlüsselwort Funktionalität. Dieses Schlüsselwort verändert nicht die Funktionalität der Tests, sondern ist vielmehr eine Beschreibung vom Test.[21]

Dies sieht dann folgendermaßen aus:

```
Funktionalität: Dies ist die Beschreibung der Funktionalität
    Die Beschreibung kann Mehrzeilig sein und endet beim nächsten
Schlüsselwort.
```

Die Funktionalität ist somit gleichzusetzen mit der User Story.

Damit die Gherkin-Datei valide ist, muss nach dem Schlüsselwort Funktionalität eins der folgenden Wörter folgen.

[21] (vgl. Wynne & Hellesøy, 2012, S. 29)

- Grundlage
- Szenario
- Szenariogrundriss

Ein Feature-File hat meistens mehrere Szenarios. Szenarios werden später einzeln von Cucumber getestet. Ein Szenario sagt also genau aus, wie das Programm sich später verhalten soll. Wenn Cucumber dieses Szenario ausführt und der Code sich nicht so verhält, schlägt der Test fehl. Ansonsten ist er erfolgreich.[22]

Ein Szenario kann folgendermaßen aussehen:

```
Szenario: Erfolgreich Geld von meinem Bankkonto abheben
    Angenommen ich habe 100€ auf meinem Bankkonto
    Wenn ich 20€ abhebe
    Dann habe ich noch 80€ auf meinem Bankkonto
```

Hier wird nun also das *Angenommen* Schlüsselwort genutzt, um den Kontext festzulegen, das *Wenn* Schlüsselwort, um mit dem System zu interagieren und das *Dann* Schlüsselwort, um das Ergebnis zu prüfen.

In diese Schritte können weitere Schritte eingebaut werden. Hierfür sind die Schlüsselwörter *Und* und *Aber* zuständig.

Beispiel:

```
Szenario: Geld mit einer gestohlenen Karte abheben
    Angenommen ich habe 100€ auf meinem Bankkonto
    Aber meine Karte ist invalide
    Wenn ich 20€ abheben möchte
    Dann wird meine Karte einbehalten
    Und ich werde aufgefordert die Bank zu kontaktieren
```

Dies sind auch schon die Basics, um Cucumber Funktionalitäten zu definieren. Wichtig hierbei ist, dass die Szenarios in sich geschlossen sein müssen. Es sollte darauf geachtet werden, dass zum Beispiel nicht im ersten Szenario 20€ von dem Bankkonto abgezogen

[22] (vgl. Wynne & Hellesøy, 2012, S. 30)

werden und im nächsten Szenario hiermit weitergerechnet wird. Dies ist zwar möglich, wird aber als „Bad Practise“ betrachtet.[23]

6.2 Cucumber Tests Beispiele

Nun werden die bereits definierten Acceptance-Tests in Cucumber überführt, um diese dann automatisiert laufen zu lassen.

```
Funktionalität: Als Spieler möchte ich eine neue Karte ziehen um meinen
Kartenwert zu erhöhen

  Szenario: Durch das Ziehen einer Karte erhöht sich mein Kartenwert
    Angenommen ich habe eine Karo 10 auf der Hand
    Wenn ich eine Karo 6 ziehe
    Dann liegt mein Kartenwert bei 16

  Szenario: Ich ziehe eine Karte und mein Kartenwert liegt über 21
    Angenommen ich habe eine Karo 10 auf der Hand
    Wenn ich eine Karo 8 ziehe
    Und ich eine Karo 9 ziehe
    Dann liegt mein Kartenwert über 21
    Und ich darf keine weitere Karte ziehen
```

```
Funktionalität: Als Spieler möchte ich in meinem Zug die Möglichkeit haben,
keine Karte zu ziehen, damit ich meinen kartenwert behalte

Szenario: Der Spieler kann jederzeit seinen Zug selbstständig beenden, wenn
sein Kartenwert unter 22 liegt
    Angenommen ich habe die Karten Karo 10 und Herz König
    Und ich bin am Zug
    Wenn ich keine Karte mehr ziehen möchte
    Dann kann ich meinen Zug beenden

Szenario: Wenn der Spieler seinen Zug beendet, ist der nächste Spieler an der
Reihe
    Angenommen ich habe die Karten Karo 10 und Herz König
    Und ich bin am Zug
    Wenn ich meinen Zug beende
    Dann ist der nächste Spieler an der Reihe
    Und ich bin nicht mehr am Zug

Szenario: Wenn der Spieler seinen Zug beendet, ändert sich sein Kartenwert in
der Runde nicht mehr
```

[23] (vgl. Wynne & Hellesøy, 2012, S. 32)

```
    Angenommen ich habe die Karten Karo 10 und Herz König
    Und ich bin am Zug
    Wenn ich meinen Zug beende
    Dann behalte ich den Kartenwert 20 für den Rest der Runde
```

```
Funktionalität: Als Dealer möchte ich den Kartenwert bestimmen können

Szenario: Der Kartenwert vom Ass beträgt 11, solange der summierte Kartenwert
nicht über 21 liegt
    Angenommen ich ziehe eine Herz 6
    Wenn ich ein Herz Ass ziehe
    Dann beträgt mein Kartenwert 17

Szenario: Der Kartenwert vom Ass beträgt 1, wenn der Kartenwert sonst über 21
liegt
    Angenommen ich ziehe eine Herz 6
    Und ich ziehe eine Piek 10
    Wenn ich ein Herz Ass ziehe
    Dann beträgt mein Kartenwert 17

Szenario: Der Kartenwert von allen Zahlenkarten beträgt die Zahl selber
    Angenommen ich ziehe eine zufällige Zahlenkarten
    Wenn ich den Kartenwert ermittle
    Dann beträgt der Kartenwert den Wert der Zahlenkarte

Szenario: Der Kartenwert von allen Bildkarten beträgt 10, außer vom Ass
    Angenommen ich ziehe eine zufällige Bildkarten
    Und diese Karte ist kein Ass
    Wenn ich den Kartenwert ermittle
    Dann beträgt der Kartenwert den Wert 10
```

Dies sind die ersten drei User Storys überführt in Cucumber Tests. Die weiteren Tests sind auf der beiliegenden CD zu finden.

6.3 Ausführen von Acceptance-Tests

Obwohl die Acceptance-Tests jetzt geschrieben sind, können diese dennoch noch nicht ausgeführt werden. Es wurde lediglich die fachliche Grundlage für die Tests geschaffen. Es muss nun noch technisch definiert werden, wie genau welcher Schritt funktionieren soll. Die Schritte sind mit allen zuvor verwendeten Schlüsselworten definiert. In der Fachsprache werden diese Step Definition genannt. Bei dem anschließenden Cucumber Feature sind die Schritte wie folgt definiert.

```
Szenario: Durch das Ziehen einer Karte erhöht sich mein Kartenwert
    Angenommen ich habe eine Karo 10 auf der Hand mit dem Wert 10 #erster
Schritt
    Wenn ich eine Karo 6 ziehen #zweiter Schritt
    Dann liegt mein Kartenwert bei 16 #dritter Schritt
```

Um dies ausführen zu können, ist es notwendig das Projekt mit Gradle zu bauen. Hierfür kann folgender Befehl im in dem Projektverzeichnis genutzt werden.

```
./gradlew build
```

In dem vorliegenden Beispiel wird IntelliJ Idea genutzt. Jetzt kann die Entwicklungsumgebung erkennen, dass es sich hier um ein Cucumber Feature handelt. Sobald dieses Szenario ausgeführt wird, generiert Cucumber die Step Definition und gibt diese in der Konsole aus. Das Ergebnis ergibt sich, wie folgt:

```
You can implement missing steps with the snippets below:

@Angenommen("ich habe eine Karo {int} auf der Hand")
public void ich_habe_eine_auf_der_Hand(Integer int1) {
    // Write code here that turns the phrase above into concrete actions
    throw new cucumber.api.PendingException();
}

@Wenn("ich eine Karo {int} ziehe")
public void ich_eine_ziehe(Integer int1) {
    // Write code here that turns the phrase above into concrete actions
    throw new cucumber.api.PendingException();
}

@Dann("liegt mein Kartenwert bei {int}")
public void liegt_mein_Kartenwert_bei(Integer int1) {
    // Write code here that turns the phrase above into concrete actions
    throw new cucumber.api.PendingException();
}
```

Dieses Ergebnis kann nun in eine Java Klasse kopiert werden. Es ist sinnvoll, sich ein neues Package für die ganzen Step Definitons anzulegen. Nachfolgend wird eine neue Klasse in diesem Package erzeugt mit dem Namen *SpielerZiehtKarteStepDef.java*. Hier wird der eben generierte Code reinkopiert. Schließlich sieht diese Klasse dann folgendermaßen aus.

```
import io.cucumber.java.de.Angenommen;
import io.cucumber.java.de.Dann;
import io.cucumber.java.de.Wenn;

public class SpielerZiehtKarteStepDef {

  @Angenommen("ich habe eine Karo {int} auf der Hand")
```

```
  public void ich_habe_eine_Karo_auf_der_Hand(final Integer int1) {
    // Write code here that turns the phrase above into concrete actions
    throw new cucumber.api.PendingException();
  }

  @Wenn("ich eine Karo {int} ziehe")
  public void ich_eine_Karo_ziehen(final Integer int1) {
    // Write code here that turns the phrase above into concrete actions
    throw new cucumber.api.PendingException();
  }

  @Dann("liegt mein Kartenwert bei {int}")
  public void liegt_mein_Kartenwert_bei(final Integer int1) {
    // Write code here that turns the phrase above into concrete actions
    throw new cucumber.api.PendingException();
  }

  @Dann("liegt mein Kartenwert über {int}")
  public void liegt_mein_Kartenwert_über(final Integer int1) {
    // Write code here that turns the phrase above into concrete actions
    throw new cucumber.api.PendingException();
  }

  @Dann("ich darf keine weitere Karte ziehen")
  public void ich_darf_keine_weitere_Karte_ziehen() {
    // Write code here that turns the phrase above into concrete actions
    throw new cucumber.api.PendingException();
  }

}
```

Im Beispiel erkennt die Entwicklungsumgebung bereits die Datentypen. Dies wird genutzt, um in Zukunft die Methoden mit mehreren Daten aufzurufen. So können in diesen Methoden zum Beispiel ganze Datentabellen durchlaufen und die Methode muss hierfür nicht umgeschrieben werden.

Um die Tests nun laufen zu lassen, fehlt lediglich eine Runner Class. Diese wird jetzt im selben Package wie die StepDefs angelegt.

```
import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;
import org.junit.runner.RunWith;

@RunWith(Cucumber.class)
@CucumberOptions(features =
{"src/test/feature/resource/spieler_zieht_karten.feature",
```

```
    "src/test/feature/resource/bestimmen_kartenwert.feature"}
)
public class CucumberRunner {

}
```

Hier werden mit der @CucumberOptions Annotation die Feature Files eingebunden, welche durchlaufen sollen. Wenn die Step Definitions nicht im selben Package abgelegt werden sollen, kann hier mit der Option *glue = „Pfad zur Step Definition“* der Pfad für die Step Definitions definiert werden. Wird jetzt die Runner Class ausgeführt, erscheint folgender Output:

```
cucumber.api.PendingException: TODO: implement me

      at
test.feature.stepDef.BestimmenKartenwertStepDef.ich_ziehe_eine(BestimmenKarten
wertStepDef.java:12)
      at *.ich ziehe eine "Herz"
6(file:src/test/feature/resource/bestimmen_kartenwert.feature:5)
```

Wie zu sehen ist, wird an dieser Stelle ein Fehler geworfen. Dies ist auch richtig, da die Step Definitions noch nicht implementiert wurden.

5.3.1 Step Definitions schreiben

Jetzt werden die ersten Designentscheidungen getroffen. Die Step Definitions werden zuerst geschrieben, hiermit wird bereits vor dem Schreiben des Codes entschieden, wie die Klassen aufgerufen werden sollen. Die Klasse mit der fertigen Step Definition sieht nun wie folgt aus.

```java
public class SpielerZiehtKartenStepDef {

    PlayDeck deck = Mockito.spy(new PlayDeck(8));

    @Spy
    Hand dealerHand = new Hand(deck, true);

    @Spy
    Hand playerHand = new Hand(deck);

    @InjectMocks
    Game game = new Game();

    @Angenommen("ich habe eine Karo {int} auf der Hand")
    public void ichHabeEineKaroAufDerHand(int arg0) {
        given(this.deck.drawCard())
                .willReturn(Card.builder()
                        .color(CardColor.Diamond)
                        .face(getFaceByValue(arg0))
                        .build());
        this.dealerHand = Mockito.spy(new Hand(deck, true));
        this.playerHand = Mockito.spy(new Hand(deck));
        this.game = new Game(dealerHand, playerHand);
        this.playerHand.draw();
        assumeThat(this.playerHand.getCards().size()).isEqualTo(1);

    }

    @Wenn("ich eine Karo {int} ziehe")
    public void ichEineKaroZiehe(int arg0) {
        given(this.deck.drawCard())
                .willReturn(Card.builder()
                        .color(CardColor.Diamond)
                        .face(getFaceByValue(arg0))
                        .build());
        this.playerHand.draw();
    }

    @Dann("liegt mein Kartenwert bei {int}")
    public void liegtMeinKartenwertBei(int arg0) {
        assertThat(this.playerHand.value()).isEqualTo(arg0);
    }

    @Dann("liegt mein Kartenwert über {int}")
    public void liegtMeinKartenwertÜber(int arg0) {
        assertThat(this.playerHand.value()).isGreaterThan(arg0);
    }

    @Und("ich darf keine weitere Karte ziehen")
    public void ichDarfKeineWeitereKarteZiehen() {
        assertThat(this.playerHand.isBusted()).isTrue();
    }
}
```

Hier wird in Zeile 16 die Mockito Bibliothek genutzt. Diese ist dafür da, dass ein bestimmtes Verhalten von einer Methode erzwungen wird, umgangssprachlich wird

dies als *mocken* bezeichnet. Die Methode *draw()* soll später eine zufällige Karte aus dem Deck ziehen. Dies wäre in diesem Test jedoch nicht praktikabel, da der Test ja vorsieht, eine bestimmte Karte zu ziehen. Die Alternative wäre, eine Methode im *PlayDeck* zu schreiben, welche eine angeforderte Karte zieht. Da diese Methode aber ausschließlich für die Tests genutzt werden würde, wäre diese Methode unnötiger Code und somit unsauber. Indem die Klasse „gemocked“ wird, kann das Verhalten auf die Tests angepasst werden. So wird hier erzwungen, dass, wenn die Methode *draw()* aufgerufen wird, immer eine Karte mit der Farbe Karo mit dem übergebenen Wert gezogen wird.

Um den übergebenen Integer-Wert aus dem Enum zu ermitteln, wird eine weitere Klasse ausgewiesen, welche TestUtil heißt. Hier finden wir viele Methoden, welche das Testen erleichtert. Diese wird hier in Zeile 19 genutzt. In Zeile 17 wird jetzt direkt festgelegt, dass eine Builder Methode genutzt wird, um die Lesbarkeit zu erhöhen. Nachdem die Karte dann in Zeile 15 gezogen wurde, wird mit einer Assumtion nochmal überprüft, ob wirklich die richtige Karte gezogen wurde. Der Vorteil von *Assumptions* gegenüber einer *Assertion* ist, dass wenn diese fehlschlägt, der Test nur übersprungen wird und nicht direkt fehlschlägt, wie in Zeile 25 zu sehen ist.

Allgemein wird hier die Bibliothek *AssertJ* für die *Assertions* genutzt, um die Lesbarkeit weiterhin zu erhöhen. Durch die Acceptence Tests ergibt sich nun folgende Klassenstruktur, welche im nächsten Kapitel mit den Techniken des TDD implementiert werden.

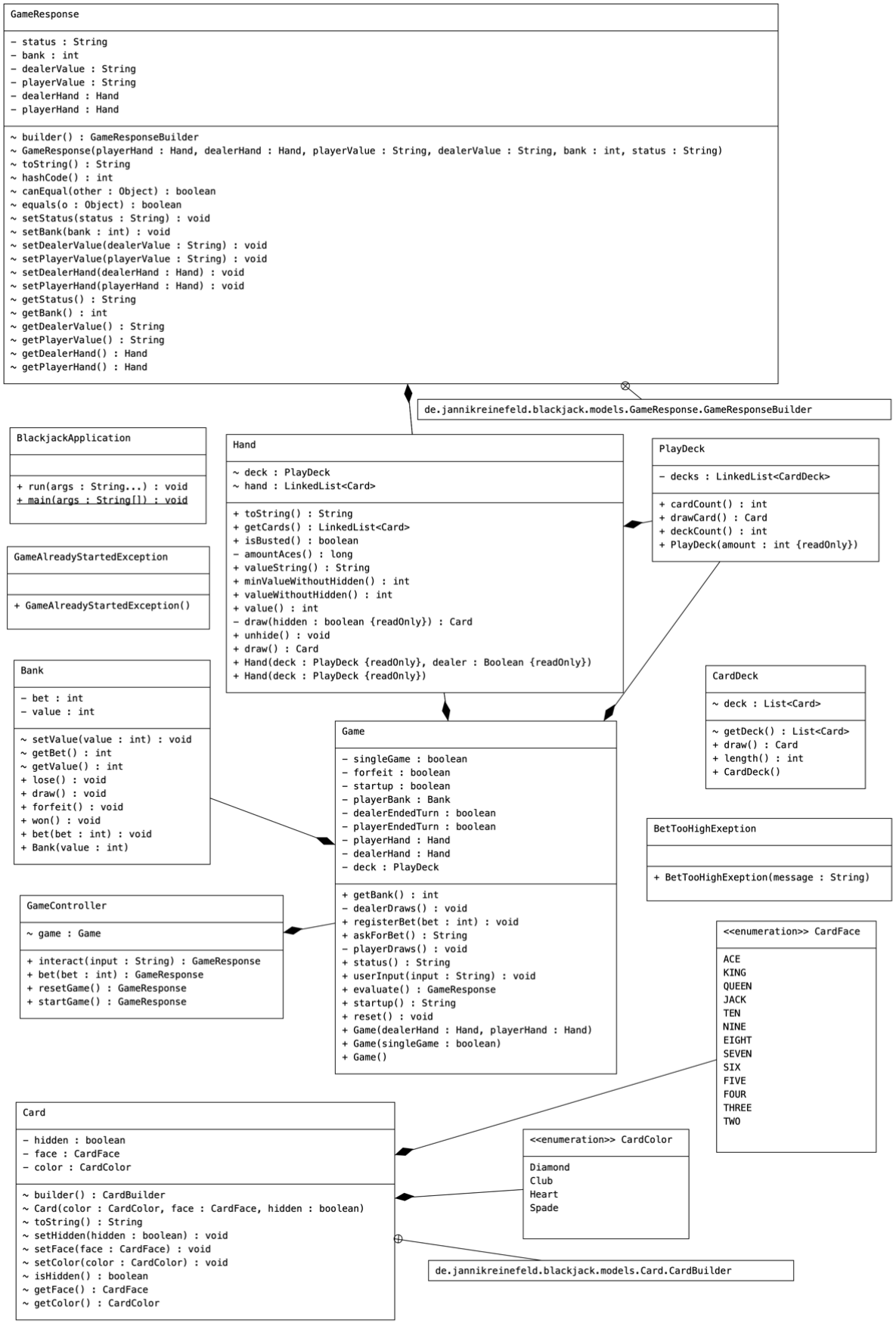

Abbildung 2 - UML Diagramm des vorigen Codes

7 Test-Driven Development

In folgendem Kapitel wird die zuvor erstelle Klassenstruktur nun erstellt. Der Zyklus des TDD wurde bereits im vorherigen Kapitel erläutert. TDD ist jedoch viel mehr als nur der oben beschriebene Zyklus. Hierfür werden an dieser Stelle einmal die drei Gesetze des TDD betrachtet:

> Erstes Gesetz – Sie dürfen Produktionscode erst schreiben, wenn Sie einen scheiternden Unit-Test geschrieben haben.
>
> Zweites Gesetz – Der Unit-Test darf nicht mehr Code enthalten, als für das Scheitern und ein korrektes Kompilieren des Tests erforderlich ist.
>
> Drittes Gesetz – Sie dürfen nur so viel Produktionscode schreiben, wie für das Bestehen des gegenwärtig scheiternden Tests ausreicht. [24]

Diese drei Gesetze erzwingen den Zyklus des TDD, dieser kann sogar auch nur 30 Sekunden dauern, wichtig ist aber, dass die Tests und der Produktionscode immer zusammengeschrieben werden. Hierbei werden die Tests oft nur wenige Sekunden vor dem Code geschrieben.[25]

7.1 Schreiben des Tests

In diesem Beispiel wird mit dem Schreiben des Klasse *PlayDeck* begonnen. Hierfür wird eine Testklasse *PlayDeckShould* angelegt. Es wird sich für die Benennung der Klasse mit der Endung *Should* entschieden, um die Lesbarkeit bei den einzelnen Tests zu erhöhen. Deshalb kann der erste Test den Namen *containEightDecks()* tragen. Zusammen mit dem Klassennamen entsteht damit ein logischer Satz, welcher direkt vermittelt, was diese Methode testen soll. In diesem Beispiel ergibt sich der Satz *PlayDeck should contain eight decks*, womit verdeutlicht wird, dass diese Methode testet, dass ein PlayDeck acht Kartendecks enthält.

Nun wird der entsprechende Unit-Test geschrieben.

```
@Test public void containEightDecks() {
    //given
```

[24] (Martin, 2009, S. 160f.)
[25] (vgl. Martin, 2009, S. 161)

```
    final PlayDeck deck = new PlayDeck(8);
    //when
    final int count = deck.deckCount();
    //then
    Assertions.assertThat(count).isEqualTo(8);
}
```

Wird dieser Test ausgeführt, schlägt er zunächst einmal fehl. Damit ist der erste Teil des TDD Zyklus abgeschlossen.

7.1.1 Given-When-Then

Wie in dem vorangegangenen Beispiel zu sehen ist, ist dieser Test in drei Sektionen eingeteilt. Die erste Sektion, das *Given,* gefolgt von der zweiten Sektion, dem *When* und abgeschlossen von der dritten Sektion, dem *Then.*

Das Given-When-Then Format ist ein übliches Format, welches bereits in frühen BDD Artikeln beschrieben wurde. Das *Given* gibt hier eine Vorabbedingung an, welche gelten muss. In dem obigen Beispiel muss so etwa eine Instanz von PlayDeck erzeugt werden. Das *When* gibt nun die Schritte an, welche getestet werden sollen. In dem Beispiel wird hier die Anzahl der Decks mit der Methode *deckCount()* abgefragt. Im *Then* Teil findet dann letztendlich die Prüfung statt, ob das richtige Ergebnis erhalten wurde.[26]

Es handelt sich hier um dasselbe Format, welches bereits in den hier behandelten Cucumber Szenarios angewandt wurde. Dort wurde jedoch die deutsche Übersetzung Angenommen-Wenn-Dann genutzt. Um hier die Lesbarkeit zu erhöhen und eine einheitliche Struktur zu gewährleisten, wird hier das gleiche Prinzip angewendet.

7.2 Schreiben des Codes

Nach dem Schreiben des Tests, wird der erste Programmcode geschrieben. Hierfür wird laut dem Test eine Klasse *PlayDeck()* mit einem Konstruktor gebraucht, welcher einen

[26] (vgl. Adzic, 2012, S. 129)

Integer-Wert übergeben bekommt, sowie eine Methode *deckCount()*, welche einen Integer-Wert liefert. Hier wird das dritte Gesetz des TDD angewendet. Sprich, es wird nur soviel Code geschrieben, wie benötigt wird, um den Test zum Erfolg zu bringen.

```
public PlayDeck(final int amount) {

}

public int deckCount() {
  return 8;
}
```

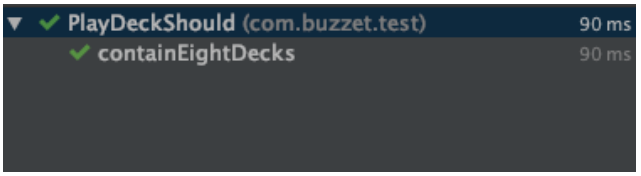

Abbildung 3 - Erfolgreicher Test

Mit diesem geschriebenen Code wird nun der Test erfolgreich ausgeführt. Jeder gestandene Programmierer sieht hier natürlich direkt, dass der implementierte Code nicht richtig ist. Der übergebene Wert aus dem Test wird nicht berücksichtigt und es wird immer nur die Zahl *Acht* als Rückgabewert geliefert. Dieses Verhalten ist jedoch genau das Verhalten, welches der Test fordert und welches erreicht werden sollte.

Der nächste Schritt im TDD ist der des *Refactorings.* Nun dürfen der Code und die Tests refaktoriert werden, jedoch darf sich das Verhalten nicht ändern. In dem folgenden Beispiel wurden *static imports* hinzugefügt.

```
import static org.assertj.core.api.Assertions.assertThat;

public class PlayDeckShould {
    @Test public void containEightDecks() {
        //given
        final PlayDeck deck = new PlayDeck(8);
        //when
        final int count = deck.deckCount();
        //then
        assertThat(count).isEqualTo(8);
    }
```

Jetzt wiederholt sich der Zyklus des TDD, es wird ein weiterer Test geschrieben.

```
@Test
public void containEightDecks() {
```

```
  //given
  final PlayDeck deck = new PlayDeck(8);
  //when
  final int count = deck.deckCount();
  //then
  assertThat(count).isEqualTo(8);
}

@Test
public void containTenDecks() {
  //given
  final PlayDeck deck = new PlayDeck(10);
  //when
  final int count = deck.deckCount();
  //then
  assertThat(count).isEqualTo(10);
}
```

Der zweite Test schlägt nun fehl, also wird der Code so angepasst, sodass beide Tests erfolgreich sind.

```
int amount;

public PlayDeck(final int amount) {
  this.amount = amount;
}

public int deckCount() {
  return amount;
}
```

Da dieses Verhalten nun implementiert wurde, kann mit dem nächsten Test fortgefahren werden.

```
@Test
public void returnACardOnDraw() {
  //given
  final PlayDeck deck = new PlayDeck(8);
  //when
  final Card card = deck.drawCard();
  //then
  assertThat(card).isInstanceOf(Card.class);
}
```

Darauf wird der Programmcode geschrieben und diese Schritte wiederholt, bis die Klasse PlayDeck zufriedenstellend ist. Dabei ergeben sich möglicherweise weitere Klassen, für welche dann wiederrum vor dem Implementieren eigene Tests geschrieben werden müssen. Die fertige Testklasse für PlayDeck sieht danach wie folgt aus.

```
package com.buzzet.test;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.ArgumentMatchers.anyInt;
import static org.mockito.BDDMockito.given;

import com.buzzet.blackjack.Card;
import com.buzzet.blackjack.CardColor;
import com.buzzet.blackjack.CardDeck;
import com.buzzet.blackjack.CardFace;
import com.buzzet.blackjack.PlayDeck;
import java.util.LinkedList;
import org.junit.Before;
import org.junit.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

public class PlayDeckShould {


  @Mock
  public LinkedList<CardDeck> decks = new LinkedList<>();

  @Mock
  public CardDeck cardDeck;

  @InjectMocks
  public PlayDeck mockDeck = new PlayDeck(6);

  @Before
  public void setup() {
    MockitoAnnotations.initMocks(this);
  }

  @Test
  public void containEightDecks() {
    //given
    final PlayDeck deck = new PlayDeck(8);
    //when
    final int count = deck.deckCount();
    //then
    assertThat(count).isEqualTo(8);
  }
```

```
  @Test
  public void containTenDecks() {
    //given
    final PlayDeck deck = new PlayDeck(10);
    //when
    final int count = deck.deckCount();
    //then
    assertThat(count).isEqualTo(10);
  }

  @Test
  public void returnACardOnDraw() {
    //given
    final PlayDeck deck = new PlayDeck(8);
    //when
    final Card card = deck.drawCard();
    //then
    assertThat(card).isInstanceOf(Card.class);
  }

  @Test
  public void returnADiamond10OnDraw() {
    //given
    Card givenCard =
Card.builder().color(CardColor.Diamond).face(CardFace.TEN).build();
    given(this.cardDeck.draw())
        .willReturn(givenCard);
    given(this.decks.get(anyInt())).willReturn(this.cardDeck);
    given(this.decks.size()).willReturn(6);
    //when
    Card card = this.mockDeck.drawCard();
    //then
    assertThat(card.getFace()).isEqualByComparingTo(givenCard.getFace());
    assertThat(card.getColor()).isEqualByComparingTo(givenCard.getColor());
  }

  @Test
  public void returnAHeartAceOnDraw() {
    //given
    Card givenCard =
Card.builder().color(CardColor.Heart).face(CardFace.ACE).build();
    given(this.cardDeck.draw())
        .willReturn(givenCard);
    given(this.decks.get(anyInt())).willReturn(this.cardDeck);
    given(this.decks.size()).willReturn(6);
    //when
    Card card = this.mockDeck.drawCard();
    //then
    assertThat(card.getFace()).isEqualByComparingTo(givenCard.getFace());
    assertThat(card.getColor()).isEqualByComparingTo(givenCard.getColor());
```

```
  }

  @Test
  public void returnFullCardCount() {
    //given
    final PlayDeck deck = new PlayDeck(8);
    //when
    final int cardCount = deck.cardCount();
    //then
    assertThat(cardCount).isEqualTo(8 * 52);

  }

  @Test
  public void returnLessCardsAfterDraw() {
    //given
    final PlayDeck deck = new PlayDeck(8);
    //when
    deck.drawCard();
    //then
    final int cardCount = deck.cardCount();
    assertThat(cardCount).isEqualTo(8 * 52 - 1);
  }
}
```

Und die fertige PlayDeck Klasse sieht folgendermaßen aus.

```
package com.buzzet.blackjack;

import java.util.LinkedList;
import java.util.Random;
import java.util.stream.Stream;

public class PlayDeck {

  private LinkedList<CardDeck> decks = new LinkedList<>();

  /**
   * New Playdeck with the amount of decks you want in it
   *
   * @param amount - amount of decks you want to play with
   */
  public PlayDeck(final int amount) {
    for (int i = 0; i < amount; i++) {
      this.decks.add(new CardDeck());
    }
  }

  /**
   * Gets a deck Count
```

```
   *
   * @return amount of decks in the PlayDeck
   */
  public int deckCount() {
    return this.decks.size();
  }

  /**
   * Draws a Card
   *
   * @return Card that was drawn
   */
  public Card drawCard() {
    int randomInt = new Random().nextInt(this.decks.size());
    return this.decks.get(randomInt).draw();
  }

  /**
   * Counts all Cards in the PlayDeck
   *
   * @return count of all Cards in the PlayDeck
   */
  public int cardCount() {
    final Stream<CardDeck> stream = this.decks.stream();
    return stream.map(x -> x.length()).reduce(0, Integer::sum);
  }
}
```

Hier ist besonders die Größe der Testklasse im Vergleich zu dem eigentlichen Programmcode auffällig. Für recht wenige Zeilen Programmcode wurden mehr als das doppelte an Testcode produziert. Hieraus könnte bereits ein Meinungsbild zu TDD erstellt werden. So sollte dies jedoch nicht der einzige Anhaltspunkt für eine Wertung von TDD sein. Was sagen denn zum Beispiel andere Entwickler, welche bereits mit TDD gearbeitet haben zu diesem Thema? Es wurden zehn Entwickler aus zwei verschiedenen Unternehmen zu dieser Frage und mehr befragt und ein Meinungsbild erstellt, welches im nächsten Kapitel vorgestellt wird.

8 Meinungsbild zu TDD und BDD

Im Nachfolgenden wird versucht der Antwort auf die Kernfrage, ob TDD und BDD zur einer besseren Code-Qualität führen, durch ein kleines Meinungsbild näher zu kommen. Im Folgenden soll die Umfrage kurz erläutert werden bevor im nächsten Kapitel die Ergebnisse dargelegt werden.

Zunächst wurden diese nach ihrer aktuellen Erfahrung mit TDD und BDD gefragt. Danach sollten sie zehn Minuten den Sourcecode des oben beschrieben Blackjack Programms evaluieren. Fünf von Ihnen bekamen den Sourcecode ohne Tests und fünf von ihnen mit Tests. Anschließend sollten sie einschätzen, wie verständlich der Sourcecode ist und wie sicher sie sich objektiv fühlen, eine Änderung an diesem Sourcecode unter Zeitdruck vorzunehmen. Im Weiteren wurde sie nach ihrer Meinung gefragt, was ihnen helfen würde, um mehr Sicherheit bei eventuellen Veränderungen zu erlangen.

Die letzten Fragen zielten auf vermeintliche Stärken und Schwächen von TDD und BDD ab. Zunächst wird die Frage gestellt, ob TDD zu viel Zeit kostet. Anschließend sollten die Probanden beantworten, ob TDD die Code-Qualität steigert. Weiter wurde erfragt, wie sehr sie TDD und BDD schätzen. Ferner sollten die Befragten ihre Meinung dazu äußern, ob BDD zu viel Zeit kostet und ob BDD noch notwendig ist, wenn bereits Unit-Tests vorhanden sind. Abgeschlossen wurde das Meinungsbild mit einer Frage nach Schwächen von TDD und BDD.

8.1 Ergebnisse der Umfrage

Im Folgenden werden die Ergebnisse der Umfrage zusammengefasst vorgestellt. Die genauen Antworten können dem Anhang 1 entnommen werden. Von den zehn befragten Entwicklern hat jeder von Ihnen bereits von TDD gehört. 70% von ihnen haben auch schon einmal mit TDD gearbeitet. 80% aller befragten Entwickler haben auch bereits zuvor von BDD gehört. Mit BDD gearbeitet haben jedoch nur 40% der Befragten.

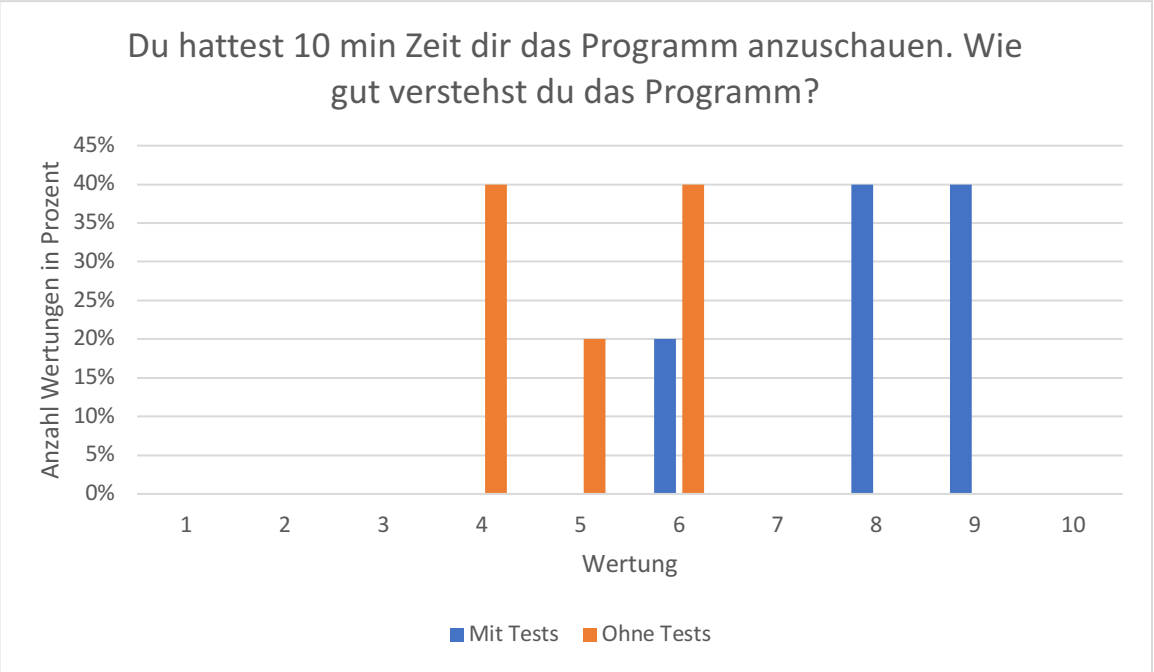

Abbildung 4 - Umfrageergebnis: Wie gut verstehst du das Programm in 10 Minuten.

Das Verständnis der Entwickler für das Programm, die dieses mit Tests evaluiert haben, kann mit einer durchschnittlichen Wertung von 8 als sehr gut bezeichnet werden. Wobei die Wertung 1 für kein Verständnis und die Wertung 10 für komplettes Verständnis steht. Die Entwickler, welche das Programm ohne Tests evaluierten, vergaben eine durchschnittliche Wertung von 5 (mittelmäßig).

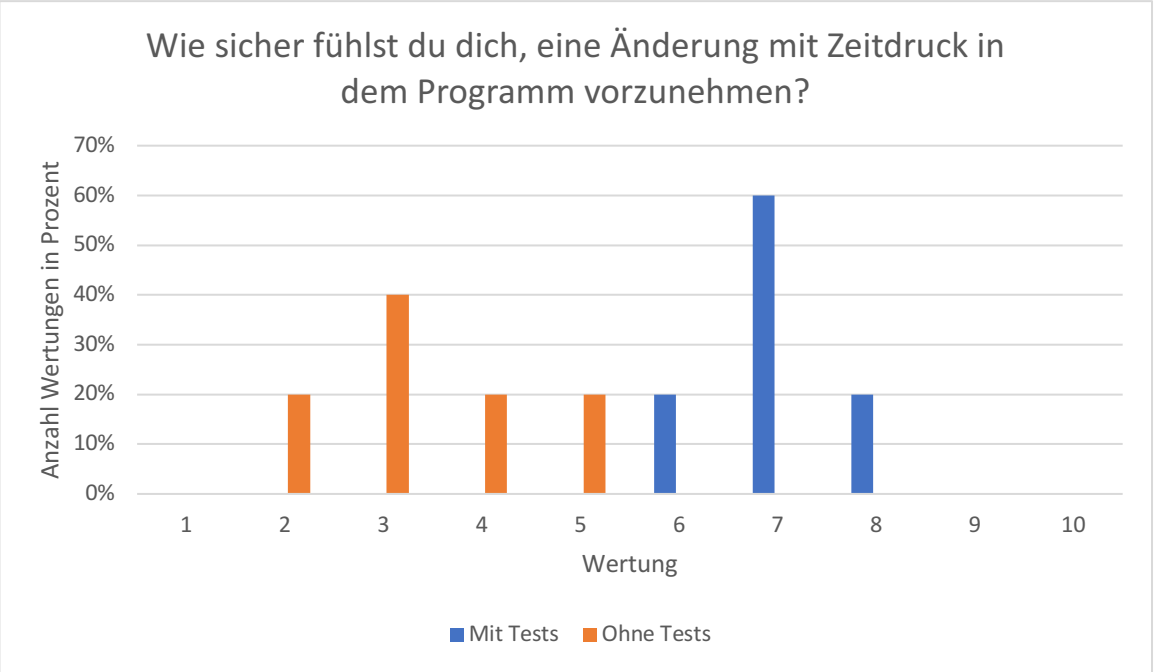

Abbildung 5 - Umfrageergebnis: Wie sicher fühlst du dich, eine Änderung mit Zeitdruck in dem Programm vorzunehmen?

Bei der Frage, wie sicher sich die Entwickler fühlten, um eine Änderung unter Zeitdruck im Programm vorzunehmen, lag die durchschnittliche Wertung bei Entwicklern ohne Tests bei 3,4 (schlecht) und bei Entwicklern mit Tests bei 7 (gut).

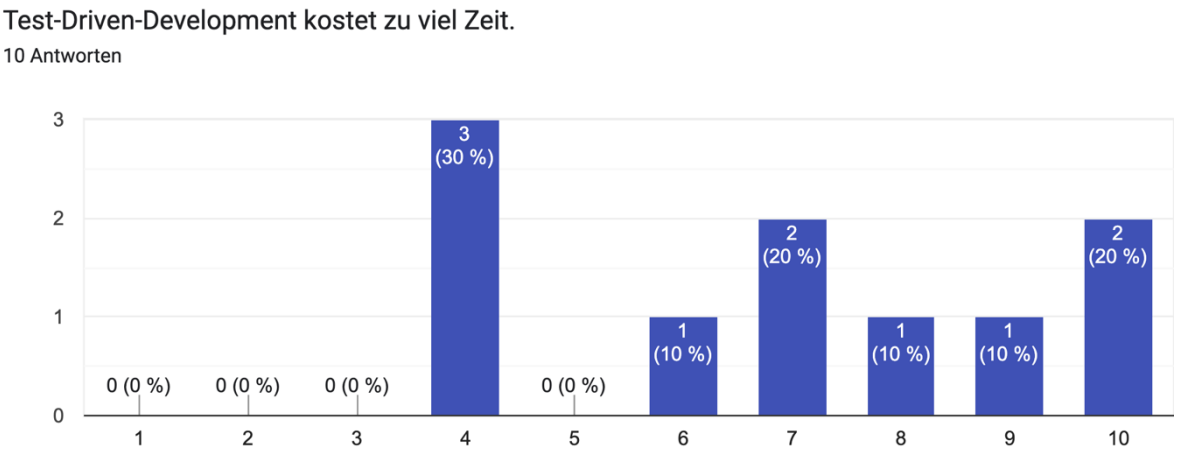

Abbildung 6 - Umfrageergebnis: TDD kostet zu viel Zeit.

Die Antwort auf die Fragestellung, ob TDD zu viel Zeit kostet, liegt durchschnittlich bei 6,9 bei allen gefragten Entwicklern (gut).

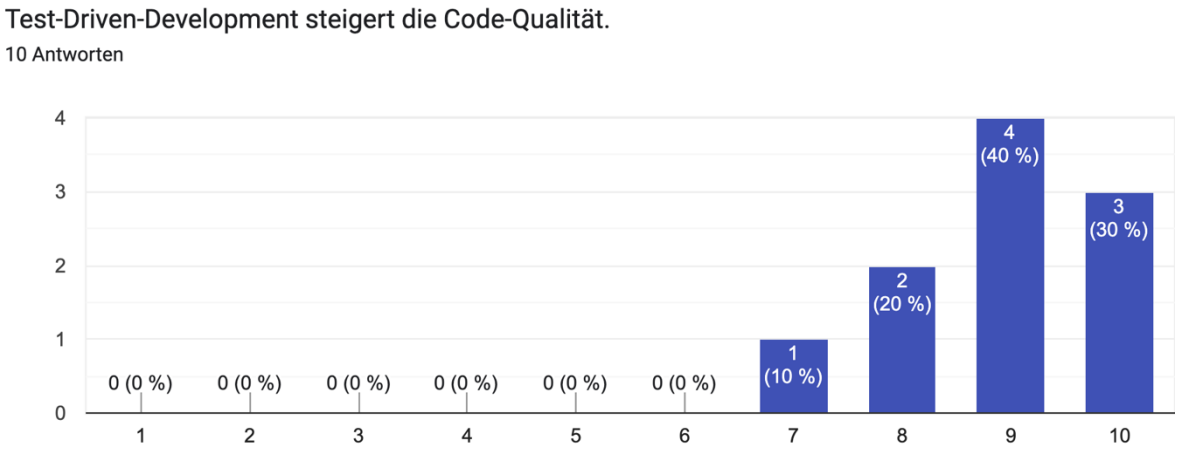

Abbildung 7 - Umfrageergebnis: TDD steigert die Code-Qualität.

Alle befragten Entwickler wurden nach der Steigerung von Code-Qualität durch TDD befragt, hierbei ergab sich eine durchschnittliche Antwort von 8,9. Der Durchschnitt schätzt durch die Anwendung von TDD die Steigerung der Code-Qualität als sehr gut ein.

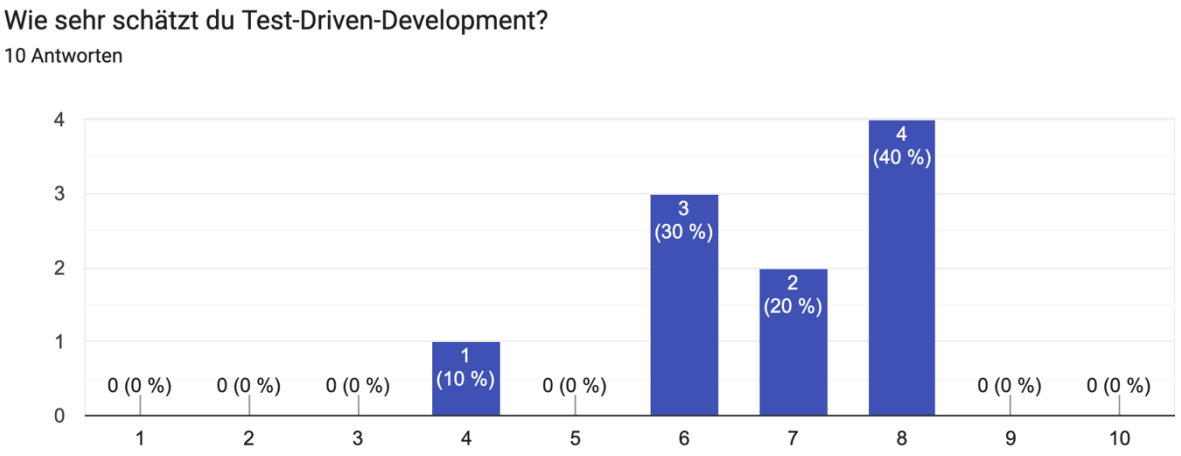

Abbildung 8 - Umfrageergebnis: Wie sehr schätzt du TDD?

Die Befragten schätzen TDD durchschnittlich mit einer Wertung von 6,8 ein. Die Wertung kann somit als gut bezeichnet werden.

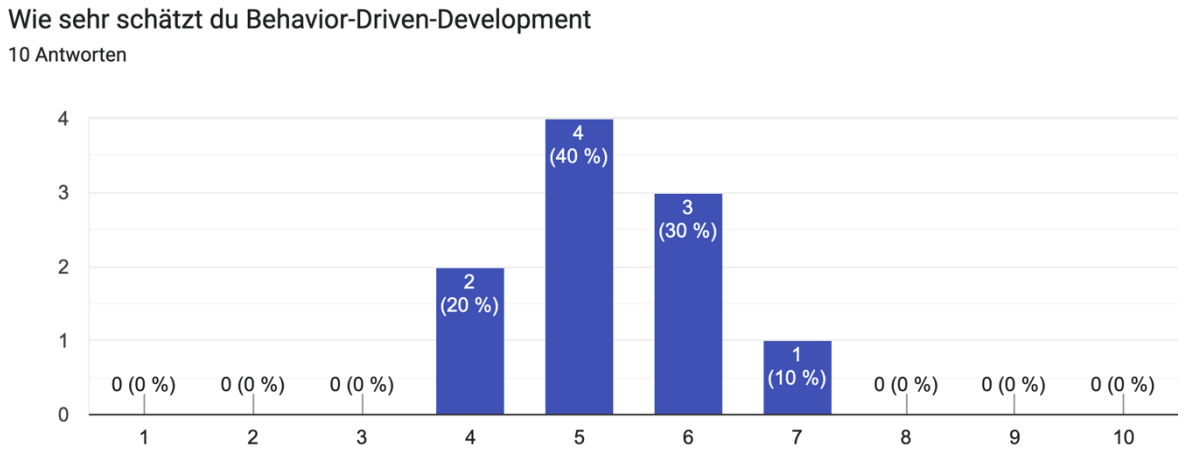

Abbildung 9 - Umfrageergebnis: Wie sehr schätzt du BDD?

Bei der Frage, wie sie BDD schätzen liegt der Durchschnittswert bei einer Wertung von 5,3 und kann damit als mittelmäßig beschrieben werden.

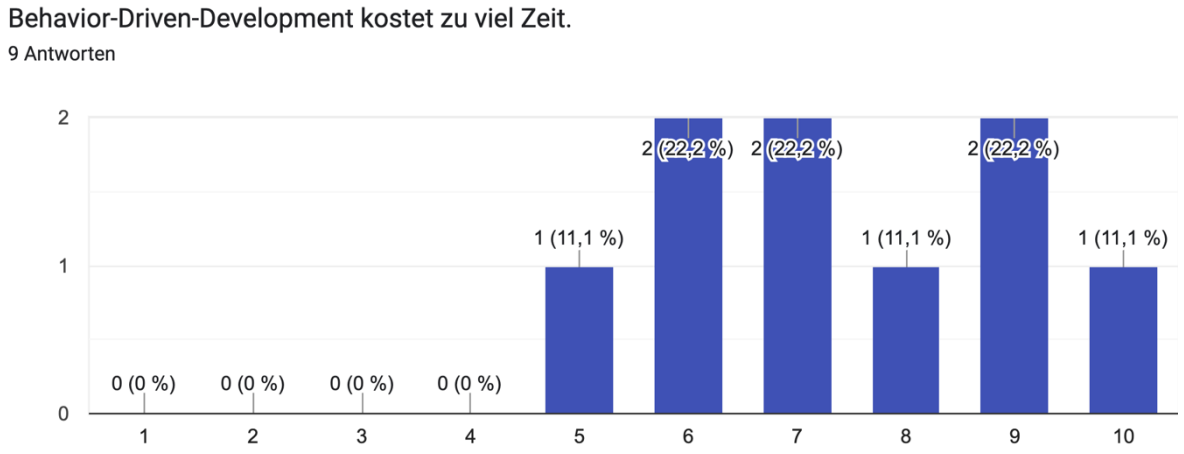

Abbildung 10 - Umfrageergebnis: BDD kostet zu viel Zeit.

Ob BDD zu viel Zeit kostet, konnten nur 9 Entwickler beantworten. Bei diesen 9 Antworten liegt der Durchschnittswert bei 7,44 (gut).

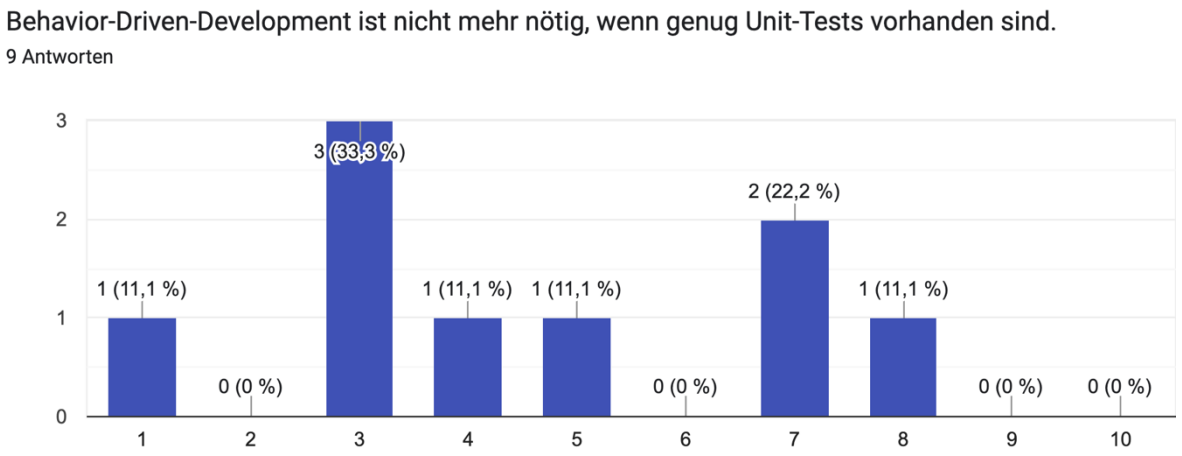

Abbildung 11 - Umfrageergebnis: BDD ist nicht mehr nötig, wenn genug Unit-Tests vorhanden sind.

Letztendlich wurde noch die Frage gestellt, ob BDD nicht mehr nötig ist, wenn genug Unit-Tests vorhanden sind. Auch diese Frage konnten nur 9 Entwickler beantworten. Die Durchschnittwertung liegt hier bei 4,55 und kann damit mit mittelmäßig bewertet werden.

Auf die Ergebnisse aus diesem Meinungsbild wird im nächsten Kapitel genauer eingegangen. Anschließend werden diese eruiert.

9 Evaluierung der Umfrageergebnisse

Neben der abschließenden Betrachtung der Umfrageergebnisse in Unterkapitel 9.2 wird in diesem Kapitel anhand der drei Gütekriterien Objektivität, Reliabilität und Validität zunächst die Frage erörtert, wie hoch die Qualität der Umfrage ist.

9.1 Gütekriterien

9.1.1 Objektivität

Die Objektivität wird unterteilt in Durchführungsobjektivität, Auswertungsobjektivität und Interpretationsobjektivität.

Ein hohes Maß an Durchführungsobjektivität war während der Umfrage gegeben. Die Probanden haben alle Hardware für die Umfrage gestellt bekommen, worauf nur das Programm und die Umfrage zugänglich waren. Weiter wurde die Interaktion mit dem Versuchsleiter auf ein Minimum beschränkt. Allen Probanden wurde ausschließlich erklärt, wo sie was finden sowie der zeitliche Ablauf vorgegeben. Weitere Fragen wurden nicht beantwortet.

Es wurde auch ein hohes Maß an Auswertungsobjektivität erreicht, da für die Auswertung der Umfrage ein externes Programm, Google Forms, genutzt wurde, welche nur Wertungen auf einer gewissen Skala zuließ. Jedoch wurden in der Umfrage viele subjektive Entscheidungen abgefragt.

Die Interpretationsobjektivität weist in dieser Umfrage kein hohes Maß auf, da die Antworten ausschließlich vom Versuchsleiter interpretiert wurden. Um hier ein hohes Maß zu erreichen, müsste ein größeres Team an der Interpretation der Auswertung arbeiten. Dies ist im Rahmen dieser Bachelorarbeit nicht möglich gewesen. Jedoch wurde im Rahmen der Umfrage mit Skalen gearbeitet, welche keinen großen Interpretationsspielraum zuließen. Hier wurde versucht durch diese Skalen ein höchstmögliches Maß an Interpretationsobjektivität zu gewährleisten.

9.1.2 Reliabilität

Das Kriterium der Reliabilität konnte in dieser Umfrage nur mäßig berücksichtigt werden. Durch die Frage nach dem subjektiven Empfinden zu TDD und dem Programm war hier keine gute Messgenauigkeit möglich. Bessere Reliabilität hätte erreicht werden können, wenn die Probanden die Aufgabe bekommen hätten, alle ein identisches Programm schreiben zu müssen. Hierbei würden die Probanden in zwei Gruppen eingeteilt. Eine Gruppe schreibt das Programm mit Tests und die zweite Gruppe dasselbe Programm ohne Tests. Danach könnten die Programme automatisch nach Fehlern überprüft und mit einer Fehlerquote bewertet werden können. Dies würde dann die Frage besser beantworten, ob TDD und BDD zu weniger Fehlern im Code führt, jedoch nicht ob die Code-Qualität hierdurch steigt, da Code-Qualität eher ein subjektiveres Kriterium darstellt. Weiter müssten bei diesem Test alle Entwickler auf demselben Wissenstand sein. Ein hohes Maß an Reliabilität wäre jedoch auch so nicht zu erreichen, da dieser Test in dieser Form und nach gewisser Zeit nicht einfach wiederholbar wäre, es sei denn, die Probanden würden sich in der Zwischenzeit nicht weiterbilden. Somit ist das Kriterium der Reliabilität in Bezug auf die leitende Fragestellung nur schwer umzusetzen.

9.1.3 Validität

Ein hohes Maß an Validität konnte in der Umfrage erreicht werden, da genau die Themen abgefragt wurden, die für die Ergebnisfindung relevant waren. Es wurden nicht die Kompetenzen der Entwickler sondern die subjektive Wahrnehmung zu dem Programm sowie zu TDD und BDD abgefragt.

9.2 Auswertung der Umfrageergebnisse

Nachfolgend werden die Ergebnisse der Umfrage interpretiert. Das Blackjack Programm wurde mit Tests definitiv besser verstanden als ohne Tests. Die Entwickler, welche Tests vorliegen hatten, konnten sich anhand dieser Tests durch den Fluss und die Logik des Programmes hangeln und somit nicht nur die Endpunkte sondern durch sprechende Testnamen auch die Logik verstehen. Hierdurch wurde auch das Vertrauen gestärkt,

Änderungen im Code vorzunehmen. Dieses Vertrauen war bei der Gruppe mit Tests deutlich höher als bei der Entwicklergruppe ohne Tests.

Die Auswertung der Frage ob TDD zu besserem Code führt, ist ziemlich eindeutig. Alle Teilnehmenden haben ausgesagt, dass TDD zu besserem Code führt.

Die meisten Entwickler schätzen TDD, viele sagen jedoch, dass TDD zu viel testet. Hier wurde häufig angesprochen, dass durch TDD sehr viel Boilerplate code, sprich Code-Segmente, welche immer wieder verwendet werden können, mit getestet wird. Obwohl sich die Entwickler hier sicher sein können, dass dieser funktioniert. Beispiel hierfür wären REST-Controller, JPA-Repositorys oder auch einfacher Getter und Setter Funktionen. Diesem Gedanken entstammen wahrscheinlich auch die Aussagen, dass TDD zu viel Zeit kostet.

BDD wird von wenigen der Entwickler geschätzt, hier haben jedoch auch weniger der Entwickler bisher mit BDD gearbeitet. Insgesamt werden von den befragten Entwicklern jedoch Ende zu Ende Tests, sprich Tests welche die Geschäftsabläufe der Anwendung und verbundenen Anwendungen ganzheitlich testet[27], bevorzugt (vgl. Anhang 1, Probanden 1, 3, 8, 9). Gespalten waren die Aussagen bei der Frage, ob BDD nötig ist, wenn genug Unit-Tests vorliegen. Fünf Entwickler haben diese Frage mit einer Wertung im unteren Viertel beantwortet und vier Entwickler mit einer Wertung von 5-8. Aus der Auswertung der Umfrage sowie den eigenen Erkenntnissen kann nun ein aussagekräftiges Ergebnis evaluiert werden, welches im nächsten Kapitel näher beschrieben wird.

[27] (vgl. E2E-Testing: Was sind End-to-End-Tests?, 2022)

10 Gesamtevaluierung

Allein gestützt auf das Meinungsbild, führt TDD definitiv zu besserem Code. Der Code ist strukturierter und per Definition getestet. Auch ich komme nach dem Arbeiten mit TDD zu demselben Ergebnis. Die Kritikpunkte, welche besagen, dass TDD zu viel Code testen, stehe ich kritisch gegenüber. Ja TDD testet auch bereits zuvor getesteten Code, teilweise werden auch Frameworks mit getestet, jedoch zwingt dies dem Entwickler sich mit diesem Code noch einmal auseinander zu setzen und diesen zu verstehen, bevor er den Code schreibt. Teilweise werden so nochmal einfachere Lösungen erwogen. Weiter dienen gut strukturierte Testfälle der weiteren Dokumentation, wie im achten und neunten Kapitel ersichtlich wurde. So wurde das Programm mit TDD deutlich besser verstanden als das Programm ohne Tests. Wie bereits im ersten Kapitel herausgearbeitet wurde, ist eine Eigenschaft von gutem Code die, dass guter Code gut lesbar und schnell verständlich ich. Somit gibt es klare Anzeichen, dass TDD zu besserem Code führt.

Bei BDD ist diese Aussage jedoch nicht so einfach zu fällen. BDD führt definitiv zu einem besseren Verständnis des Programms, da hier die User Storys direkt in der Codebase gesammelt liegen. Entwickler können so im Code direkt die Akzeptanzkriterien des Programms sehen und verstehen und müssen diese nicht noch auf einer externen Seite wie zum Beispiel in Jira suchen. Jedoch müssen diese Akzeptanzkriterien auch gepflegt und gefüllt werden. Wenn dies nicht geschieht, gibt es keinen allgemeingültigen Datenbestand und so können Akzeptanzkriterien falsch verstanden werden, da sich gegebenenfalls andere Informationen auf anderen Plattformen finden lassen. Somit führt BDD dann sogar zu einer Verschlechterung des Code-Qualität. Auch in der Umfrage sprechen sich die gefragten Entwickler eher gegen BDD aus (siehe Anhang 1). Diese setzen lieber auf gut gepflegte „Ende zu Ende“-Tests. BDD kann also zu einer besseren Code-Qualität und Lesbarkeit führen, wenn richtig eingesetzt, gut gepflegt und durch die Verantwortlichen gelebt. Diese könnten Wird aber nur einer dieser Punkte nicht richtig umgesetzt, kann es im schlimmsten Fall sogar zu einer Verschlechterung der Code-Qualität kommen.

11 Zusammenfassung und Ausblick

Im Folgenden sollen die Ergebnisse der Umfrage sowie meines Praxisbeispiels zusammengefasst dargestellt werden, um die Kernfrage dieser Arbeit beantworten zu können. Des Weiteren soll ein Ausblick auf mögliche anschließende Arbeiten gegeben werden. Insgesamt kann durch die Umfrage festgestellt werden, dass es keine eindeutige Antwort auf die Frage, ob TDD und BDD zu besser Code Qualität führen, gibt. Jedoch gibt es eine starke Tendenz in die Richtung das TDD die Code Qualität verbessert. Um hier eine eindeutige Antwort zu finden, müsste, wie schon in Kapitel 9.2 erläutert, ein Versuch mit deutlich mehr Entwicklern durchgeführt werden. Dabei sollte es dann nicht nur um das subjektive Empfinden der Entwickler gehen, sondern es müsste auch der resultierende Code bewertet werden. Zudem müssten hierfür die Entwickler TDD und BDD sehr gut beherrschen, um diese Techniken richtig anwenden zu können und alle Entwickler müssten auf einem gleichen Leistungsniveau stehen. Somit könnte ein größeres Maß an Reliabilität erreicht werden. Ein höheres Maß an Interpretationsobjektivität kann hier durch ein größeres Team, welche die Umfrage Interpretiert, erreicht werden. Die Bedingungen für einen solche Untersuchung sind durch die eben genannten Gründe kaum umzusetzen, so dass die vorliegende Arbeit mit dem Schwerpunkt auf dem Programm sowie der Umfrage schon eine gute Tendenz in Bezug auf die Frage aufzeigt.

Jedoch konnte in dieser Arbeit herausgefunden werden, dass ein riesiger Vorteil in der Verwendung von TDD darin besteht, dass in dem Code ohne Probleme eine Zeile geändert werden kann, ohne großes Risiko die Logik des Programmes zu beschädigen. Weiter ist es möglich mit TDD eine hundertprozentige Testabdeckung zu erreichen, sprich eines Tests, welche alle Teile des Produktionscodes mindestens einmal durchläuft. Es kann direkt geprüft werden, ob nach einer Veränderung noch alles funktioniert und wenn nicht, kann direkt erkannt werden, welcher Teil des Codes nicht mehr funktionstüchtig ist. Gerade bei Software, welche wächst, spart dies auf lange Sicht sehr viel Entwicklungszeit ein. Wenn solch eine Änderung jedoch einmal nötig ist, weil sich Anforderungen geändert haben, reicht es nicht, den Produktionscode zu ändern, im Zweifel müssen hier direkt viele weitere Tests angepasst werden. Weiter

wird durch TDD recht viel Code getestet, welcher in vorherigen Projekten bereits getestet wurde. Hier kann es schnell vorkommen, dass sich Frust bei den Entwicklern breit macht und diese das Gefühlt bekommen nicht mehr voranzukommen. Eine gute Alternative zum Standard-Ablauf des TDD-Zyklus wäre es hier, TDD auf das Testen der Geschäftslogik herunterzubrechen. Hierbei würden dann keine externen Schnittstellen (REST-Controller, Repositorys oder ähnliches) mehr mit getestet werden, sondern lediglich jene Codeteile, welche Geschäftslogik enthalten.

Die Frage ob TDD zu besserem Code führt, kann ich persönlich, aus meinen hier gewonnen Erfahrungen, nur mit Ja beantworten. Wenn einmal Code ohne TDD geschrieben und dann der gesamte Code nochmal neu mit TDD geschrieben wird, kann ein klarer, qualitativer Unterschied ausgemacht werden. Der neue Code ist in der Regel viel kürzer und sauberer, denn durch TDD wird immer nur so viel geschrieben, wie auch nötig ist. Ferner geschieht dies in kleinen iterativen Schritten. So kann immer der Überblick über das große Ganze behalten werden. Logik oder Berechnungen sowie Validierungen sollten immer getestet werden. Boilerplate Code kann aber gerne auch ungetestet bleiben, um die Effizienz nicht zu verlieren.

Führt BDD zu besserem Code? Diese Frage lässt sich wiederum nicht so einfach beantworten und in dieser Ausarbeitung kann kein deutliches Ergebnis aufgezeigt werden. Dies kann, wie bereits erwähnt, nur durch eine größere Studie gezeigt werden. Den Code direkt beeinflusst BDD kaum. Lediglich das Schreiben der Step Definitions kann helfen, die Struktur des Codes mitzubestimmen. Jedoch hilft BDD beträchtlich dabei, das richtige Programm zu schreiben und Fortschritte in der Entwicklung einfacher zu kommunizieren und zu verifizieren. Ferner fördert BDD die Kommunikation im Team, indem das Team bei BDD stets darüber beraten muss, was gefordert und bereits umgesetzt wurde.
Auch wenn TDD am Anfang frustrierend und langsam sein kann, TDD zu lernen, ist ein richtiger Weg zu besserem Code.

Der komplette Code kann auf dem beiliegenden Datenträger eingesehen werden.

Abkürzungsverzeichnis

API ..... *Application Programming Interface*
BDD ..... *Behaviour-Driven Development*
REST ..... *Representational State Transfer*
TDD ..... *Test-Driven Development*

Literaturverzeichnis

Martin, R. C. (2009). Clean Code. In R. C. Martin, *Clean Code: Refactoring, Patterns, Testen und Techniken für sauberen Code.* Heidelberg: Mitp.

Martin, R. C. (2012). *The Robert C. Martin Clean Code Collection.* New Jersey: Pearson Education.

Koskela, L. (2008). *Test-driven: practical TDD and acceptance TDD for Java developers.* Greenwich: Manning.

Wynne, M., & Hellesøy, A. (2012). *The cucumber book: behaviour-driven development for testers and developers.* Dallas, Texas, USA: Pragmatic Bookshelf.

Adzic, G. (2012). *Specification by example: how successful teams deliver the right software.* Shelter Island: Manning.

SmartBear Software. (o.D.). *Gherkin Reference*. Retrieved 08 2022, from Gherkin Reference - Cucumber Documentation: https://cucumber.io/docs/gherkin/reference/#spoken-languages

E2E-Testing: Was sind End-to-End-Tests? (2022, 9 20). Retrieved 09 2022, from Testautomatisierung.org: https://www.testautomatisierung.org/lexikon/e2e-testing/

Abbildungsverzeichnis

Anhang 1: Umfrageergebnisse

Proband Nummer 1

Hast du schon einmal von Test-Driven-Development gehört?	Hast du schon einmal mit Test-Driven-Development gearbeitet?	Hast du schon einmal von Behavior-Driven-Development gehört?	Hast du schon einmal mit Behavior-Driven-Development gearbeitet?	Du hattest 10 min Zeit dir das Programm anzuschauen. Wie gut verstehst du das Programm?	Wie sicher fühlst du dich, eine Änderung mit Zeitdruck in dem Programm vorzunehmen?	Hatte das Programm, das du evaluiert hast, Tests?	Was würde dir helfen, mehr Sicherheit bei Veränderungen zu erlangen?
Ja	Ja	Ja	Ja	8	7	Ja	Eine Schnittstellendokumentation

Test-Driven-Development kostet zu viel Zeit.	Test-Driven-Development steigert die Code-Qualität.	Wie sehr schätzt du Test-Driven-Development?	Wie sehr schätzt du Behavior-Driven-Development	Behavior-Driven-Development kostet zu viel Zeit.	Behavior-Driven-Development ist nicht mehr nötig, wenn genug Unit-Tests vorhanden sind.	Wo siehst du die Schwächen von Test-Driven-Development?	Wo siehst du die Schwächen von Behavior-Driven-Development?
6	9	8	6	6	7	Es wird auch viel Boilerplatecode getestet. Es sollte nur Logik getestet werden.	Wird häufig von den POs nicht verstanden. Bevorzuge häufiger E2E Tests ohne ein weiteres Framework oder mit Postman / Newman.

Proband Nummer 2

Hast du schon einmal von Test-Driven-Development gehört?	Hast du schon einmal mit Test-Driven-Development gearbeitet?	Hast du schon einmal von Behavior-Driven-Development gehört?	Hast du schon einmal mit Behavior-Driven-Development gearbeitet?	Du hattest 10 min Zeit dir das Programm anzuschauen. Wie gut verstehst du das Programm?	Wie sicher fühlst du dich, eine Änderung mit Zeitdruck in dem Programm vorzunehmen?	Hatte das Programm, das du evaluiert hast, Tests?	Was würde dir helfen, mehr Sicherheit bei Veränderungen zu erlangen?
Ja	Nein	Ja	Nein	6	6	Ja	Mehr Erfahrung mit Java

Test-Driven-Development kostet zu viel Zeit.	Test-Driven-Development steigert die Code-Qualität.	Wie sehr schätzt du Test-Driven-Development?	Wie sehr schätzt du Behavior-Driven-Development	Behavior-Driven-Development kostet zu viel Zeit.	Behavior-Driven-Development ist nicht mehr nötig, wenn genug Unit-Tests vorhanden sind.	Wo siehst du die Schwächen von Test-Driven-Development?	Wo siehst du die Schwächen von Behavior-Driven-Development?
10	10	8	7	6	3	Es wird viel wissen benötigt, wie man ordentlich Tests schreibt	BDD ist teilweise schwer in Pipelines zu integrieren.

Proband Nummer 3

Hast du schon einmal von Test-Driven-Development gehört?	Hast du schon einmal mit Test-Driven-Development gearbeitet?	Hast du schon einmal von Behavior-Driven-Development gehört?	Hast du schon einmal mit Behavior-Driven-Development gearbeitet?	Du hattest 10 min Zeit dir das Programm anzuschauen. Wie gut verstehst du das Programm?	Wie sicher fühlst du dich, eine Änderung mit Zeitdruck in dem Programm vorzunehmen?	Hatte das Programm, das du evaluiert hast, Tests?	Was würde dir helfen, mehr Sicherheit bei Veränderungen zu erlangen?
Ja	Ja	Ja	Ja	8	8	Ja	Mehr mit dem Produkt arbeiten

Test-Driven-Development kostet zu viel Zeit.	Test-Driven-Development steigert die Code-Qualität.	Wie sehr schätzt du Test-Driven-Development?	Wie sehr schätzt du Behavior-Driven-Development	Behavior-Driven-Development kostet zu viel Zeit.	Behavior-Driven-Development ist nicht mehr nötig, wenn genug Unit-Tests vorhanden sind.	Wo siehst du die Schwächen von Test-Driven-Development?	Wo siehst du die Schwächen von Behavior-Driven-Development?
7	8	6	5	8	7	Kostet zu viel Zeit für den Impact	E2E Tests sind einfacher zu schreiben

Proband Nummer 4

Hast du schon einmal von Test-Driven-Development gehört?	Hast du schon einmal mit Test-Driven-Development gearbeitet?	Hast du schon einmal von Behavior-Driven-Development gehört?	Hast du schon einmal mit Behavior-Driven-Development gearbeitet?	Du hattest 10 min Zeit dir das Programm anzuschauen. Wie gut verstehst du das Programm?	Wie sicher fühlst du dich, eine Änderung mit Zeitdruck in dem Programm vorzunehmen?	Hatte das Programm, das du evaluiert hast, Tests?	Was würde dir helfen, mehr Sicherheit bei Veränderungen zu erlangen?
Ja	Nein	Nein	Nein	9	7	Ja	Einfach Änderungen vornehmen und dann zu Testen.

Test-Driven-Development kostet zu viel Zeit.	Test-Driven-Development steigert die Code-Qualität.	Wie sehr schätzt du Test-Driven-Development?	Wie sehr schätzt du Behavior-Driven-Development	Behavior-Driven-Development kostet zu viel Zeit.	Behavior-Driven-Development ist nicht mehr nötig, wenn genug Unit-Tests vorhanden sind.	Wo siehst du die Schwächen von Test-Driven-Development?	Wo siehst du die Schwächen von Behavior-Driven-Development?
4	7	7	6	10	5	Die Tests müssen trotzdem noch gewartet werden und jede Code Änderung erfordert eine Änderung in den Tests. Somit sind Emergency Changes langsamer. Es wird schneller mal ein Test disabled anstelle ihn richtig zu stellen.	Müsste in das Thema mich erst richtig einlesen

Proband Nummer 5

Hast du schon einmal von Test-Driven-Development gehört?	Hast du schon einmal mit Test-Driven-Development gearbeitet?	Hast du schon einmal von Behavior-Driven-Development gehört?	Hast du schon einmal mit Behavior-Driven-Development gearbeitet?	Du hattest 10 min Zeit dir das Programm anzuschauen. Wie gut verstehst du das Programm?	Wie sicher fühlst du dich, eine Änderung mit Zeitdruck in dem Programm vorzunehmen?	Hatte das Programm, das du evaluiert hast, Tests?	Was würde dir helfen, mehr Sicherheit bei Veränderungen zu erlangen?
Ja	Nein	Ja	Nein	9	7	Ja	Eine Schnittstellenbeschreibung

Test-Driven-Development kostet zu viel Zeit.	Test-Driven-Development steigert die Code-Qualität.	Wie sehr schätzt du Test-Driven-Development?	Wie sehr schätzt du Behavior-Driven-Development	Behavior-Driven-Development kostet zu viel Zeit.	Behavior-Driven-Development ist nicht mehr nötig, wenn genug Unit-Tests vorhanden sind.	Wo siehst du die Schwächen von Test-Driven-Development?	Wo siehst du die Schwächen von Behavior-Driven-Development?
4	9	4	4	7	3	Es verlangsamt den Arbeitsablauf. Ich möchte nicht bei jedem neuen Service die gleichen Tests schreiben müssen. Dann teste ich lieber wichtigen Business Code.	Behabior-Driven-Development muss auch von den Business-Analysten und Product Owner gelebt und verstanden werden. Dazu werden gute User Storys gebraucht. Diese sind jedoch teilweise nicht in der richtigen Qualität / Umfang vorhanden.

Proband Nummer 6

Hast du schon einmal von Test-Driven-Development gehört?	Hast du schon einmal mit Test-Driven-Development gearbeitet?	Hast du schon einmal von Behavior-Driven-Development gehört?	Hast du schon einmal mit Behavior-Driven-Development gearbeitet?	Du hattest 10 min Zeit dir das Programm anzuschauen. Wie gut verstehst du das Programm?	Wie sicher fühlst du dich, eine Änderung mit Zeitdruck in dem Programm vorzunehmen?	Hatte das Programm, das du evaluiert hast, Tests?	Was würde dir helfen, mehr Sicherheit bei Veränderungen zu erlangen?
Ja	Ja	Ja	Nein	4	3	Nein	Tests, Documentation

Test-Driven-Development kostet zu viel Zeit.	Test-Driven-Development steigert die Code-Qualität.	Wie sehr schätzt du Test-Driven-Development?	Wie sehr schätzt du Behavior-Driven-Development	Behavior-Driven-Development kostet zu viel Zeit.	Behavior-Driven-Development ist nicht mehr nötig, wenn genug Unit-Tests vorhanden sind.	Wo siehst du die Schwächen von Test-Driven-Development?	Wo siehst du die Schwächen von Behavior-Driven-Development?
4	10	8	5			Most of the time you test too much code that needs no testing	Never worked with it before.

Proband Nummer 7

Hast du schon einmal von Test-Driven-Development gehört?	Hast du schon einmal mit Test-Driven-Development gearbeitet?	Hast du schon einmal von Behavior-Driven-Development gehört?	Hast du schon einmal mit Behavior-Driven-Development gearbeitet?	Du hattest 10 min Zeit dir das Programm anzuschauen. Wie gut verstehst du das Programm?	Wie sicher fühlst du dich, eine Änderung mit Zeitdruck in dem Programm vorzunehmen?	Hatte das Programm, das du evaluiert hast, Tests?	Was würde dir helfen, mehr Sicherheit bei Veränderungen zu erlangen?
Ja	Ja	Nein	Nein	6	4	Nein	Mehr Zeit

Test-Driven-Development kostet zu viel Zeit.	Test-Driven-Development steigert die Code-Qualität.	Wie sehr schätzt du Test-Driven-Development?	Wie sehr schätzt du Behavior-Driven-Development	Behavior-Driven-Development kostet zu viel Zeit.	Behavior-Driven-Development ist nicht mehr nötig, wenn genug Unit-Tests vorhanden sind.	Wo siehst du die Schwächen von Test-Driven-Development?	Wo siehst du die Schwächen von Behavior-Driven-Development?
7	8	6	5	7	3	Wenn Logik getestet wird, ist TDD ein sehr gutes Tool. Wenn einfacher Controller getestet werden sollen als Unit test, ist es Zeitverschwendung	Wir nicht von der Organisation gelebt

Proband Nummer 8

Hast du schon einmal von Test-Driven-Development gehört?	Hast du schon einmal mit Test-Driven-Development gearbeitet?	Hast du schon einmal von Behavior-Driven-Development gehört?	Hast du schon einmal mit Behavior-Driven-Development gearbeitet?	Du hattest 10 min Zeit dir das Programm anzuschauen. Wie gut verstehst du das Programm?	Wie sicher fühlst du dich, eine Änderung mit Zeitdruck in dem Programm vorzunehmen?	Hatte das Programm, das du evaluiert hast, Tests?	Was würde dir helfen, mehr Sicherheit bei Veränderungen zu erlangen?
Ja	Ja	Ja	Ja	4	3	Nein	Tests würden sicherlich helfen

Test-Driven-Development kostet zu viel Zeit.	Test-Driven-Development steigert die Code-Qualität.	Wie sehr schätzt du Test-Driven-Development?	Wie sehr schätzt du Behavior-Driven-Development	Behavior-Driven-Development kostet zu viel Zeit.	Behavior-Driven-Development ist nicht mehr nötig, wenn genug Unit-Tests vorhanden sind.	Wo siehst du die Schwächen von Test-Driven-Development?	Wo siehst du die Schwächen von Behavior-Driven-Development?
10	10	8	6	9	8	Die meisten Fehler kommen bei der Environment Configuration vor und nicht im Code an sich. Dies lässt sich auch mit TDD nicht abdecken.	Gute Ende zu Ende Test sind schneller und besser.

Proband Nummer 9

Hast du schon einmal von Test-Driven-Development gehört?	Hast du schon einmal mit Test-Driven-Development gearbeitet?	Hast du schon einmal von Behavior-Driven-Development gehört?	Hast du schon einmal mit Behavior-Driven-Development gearbeitet?	Du hattest 10 min Zeit dir das Programm anzuschauen. Wie gut verstehst du das Programm?	Wie sicher fühlst du dich, eine Änderung mit Zeitdruck in dem Programm vorzunehmen?	Hatte das Programm, das du evaluiert hast, Tests?	Was würde dir helfen, mehr Sicherheit bei Veränderungen zu erlangen?
Ja	Ja	Ja	Nein	6	5	Nein	Besseres Verständis der Spielelogik

Test-Driven-Development kostet zu viel Zeit.	Test-Driven-Development steigert die Code-Qualität.	Wie sehr schätzt du Test-Driven-Development?	Wie sehr schätzt du Behavior-Driven-Development	Behavior-Driven-Development kostet zu viel Zeit.	Behavior-Driven-Development ist nicht mehr nötig, wenn genug Unit-Tests vorhanden sind.	Wo siehst du die Schwächen von Test-Driven-Development?	Wo siehst du die Schwächen von Behavior-Driven-Development?
8	9	7	4	9	1	Es muss gelebt werden und von der Management Ebene akzeptiert werden. Qualität > Quantität.	Wenn man BDD mit E2E Tests gleich setzt sind E2E Tests einfacher zu modifizieren und zu Maintainen.

Proband Nummer 10

Hast du schon einmal von Test-Driven-Development gehört?	Hast du schon einmal mit Test-Driven-Development gearbeitet?	Hast du schon einmal von Behavior-Driven-Development gehört?	Hast du schon einmal mit Behavior-Driven-Development gearbeitet?	Du hattest 10 min Zeit dir das Programm anzuschauen. Wie gut verstehst du das Programm?	Wie sicher fühlst du dich, eine Änderung mit Zeitdruck in dem Programm vorzunehmen?	Hatte das Programm, das du evaluiert hast, Tests?	Was würde dir helfen, mehr Sicherheit bei Veränderungen zu erlangen?
Ja	Ja	Ja	Ja	5	2	Nein	Mehr Zeit das Programm zu evaluieren

Test-Driven-Development kostet zu viel Zeit.	Test-Driven-Development steigert die Code-Qualität.	Wie sehr schätzt du Test-Driven-Development?	Wie sehr schätzt du Behavior-Driven-Development	Behavior-Driven-Development kostet zu viel Zeit.	Behavior-Driven-Development ist nicht mehr nötig, wenn genug Unit-Tests vorhanden sind.	Wo siehst du die Schwächen von Test-Driven-Development?	Wo siehst du die Schwächen von Behavior-Driven-Development?
9	9	6	5	5	4	Muss gelebt werden von den Kollegen. Tests nachzuziehen wenn ein Bug gefunden wird sollte keine Option sein!	Das Verständnis bei den Kollegen muss hierfür gestärkt werden durch Schulungen / Workshops.

Anhang 2: Entwicklerdokumentation

Das Programm besteht aus zwei Teilen. Teil 1 ist das Backend, welches auf der Programmiersprache Java mit dem Framework Springboot basiert. Teil 2 ist ein Beispiel für ein Frontend, welches genutzt werden kann, um mit dem Backend zu interagieren.

Starten der Anwendung

Starten über Docker

Am einfachsten ist es, die Anwendung über Docker zu starten. Hierfür muss Docker sowie docker-compose auf dem System installiert sein.
Zum Starten unter Mac oder Linux einfach das Script app-start.sh ausführbar machen.

```
chmod +x app-start.sh
```

Danach das Script starten. Das Script baut zwei Docker-Container und startet dieses.

```
./app-start.sh
```

Zum Starten unter Windows folgende Befehle verwenden:

```
docker build -f Dockerfile-backend . -t blackjack-backend
docker build -f Dockerfile-frontend . -t blackjack-frontend
docker-compose up
```

Danach ist das Frontend unter folgender URL erreichbar: http://localhost:4200/,
sowie das Backend unter folgender URL: http://localhost:8080/swagger-ui/index.html

Anwendung selber bauen

Das Backend kann ohne weitere Abhängigkeiten laufen. Um das Backend zu modifizieren, sollte eine IDE installiert sein z.B. IntelliJ IDEA. Für das Ausführen ist Java 17 zwingend erforderlich.

Das Backend kann über das Terminal mit folgendem Befehl gestartet werden, hierzu muss sich in dem Verzeichnis befunden werden:

Mac / Linux:
```
./gradlew bootRun
```

Windows:
```
./gradle bootRun
```

Das Backend startet nun einen RestController, welcher auf Port 8080 erreichbar ist. Für das Frontend muss der Node Package Manager (NPM) und node.js installiert sein.

Zum Starten des Frontends müssen erst alle Abhängigkeiten von NPM installiert werden mit folgendem Befehl:

```
npm install
```

Danach kann das Frondend mit folgendem Befehl gestartet werden:

```
npm run start
```

Danach ist das Frontend unter folgender URL erreichbar: http://localhost:4200/
Damit das Frontend auch funktioniert, muss natürlich das Backend parallel laufen.

Backend

Das Backend ist geschrieben in Java mit dem Springboot Framework.

Der Sourcecode ist in vier Packages unterteilt. Controller, Exceptions, Game und Models.
Im Controller Package stecken die REST-Controller. Hier gibt es die Klasse „GameController“, welche aller Interaktionen mit dem Spiel über REST ermöglichen.

Im Package Exceptions finden wir alle Exceptions, die das Programm wirft. Hier sind erstmal zwei Exceptions modeliert. Einmal die BetTooHighException, welche geworfen wird, wenn der Spieler ein Wettbetrag setzt, welcher sein Guthaben überschreitet und die GameAlreadyStartedException, welche geworfen wird, wenn ein Spielstart initiiert wird, dieses Spiel jedoch noch läuft.

Im Package game finden wir die ganze Logik des Spiels. Hier finden wir die Game-Klasse an sich. Dann die Hand, welche die Karten des Spielers auf der Hand darstellen. Gefolgt von dem PlayDeck, welche das Kartendeck widerspiegelt und letztendlich die Bank, welche das Geld des Spielers verwaltet.

Im Package Models finden wir alle Daten Klassen des Programms, wie zum Beispiel die Card, CardColor, CardFace und GameResponse, welche vom REST-Controller zurückgegeben wird.

Eine Schnittstellendokumentation kann bei gestarteter Anwendung unter http://localhost:8080/swagger-ui/index.html gefunden werden.

Zum Starten der Tests folgenden Befehl ausführen:

Mac / Linux:

```
./gradlew test cucumber
```

Windows:

```
./gradle test cucumber
```

Frontend

Das Frontend ist mit Angular 13 geschrieben. Im Ordner src/app/ finden wir hier den Sourcecode. Der Sourcecode ist wieder in 3 Unterordner eingeteilt. Wir haben hier lediglich eine Seite, welche die Dateien app.component.html und app.component.scss sowie app.component.ts abbilden. Dies ist die Hauptseite des Programms.

Im Unterordner Card befindet sich eine Komponente, welche eine Spielkarte darstellen kann. Im Ordner Model finden wir ein paar Klassen, welche Karten, das Spiel und die Hand des Spielers widerspiegeln und im Ordner Service finden wir den Service, welcher mit dem Backend kommuniziert.

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Dieses Blatt, mit der folgenden Erklärung, ist nach Fertigstellung der Abschlussarbeit durch den Studierenden auszufüllen und jeweils mit Originalunterschrift als letztes Blatt in das Prüfungsexemplar der Abschlussarbeit einzubinden.
Eine unrichtig abgegebene Erklärung kann -auch nachträglich- zur Ungültigkeit des Studienabschlusses führen.

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

- die folgende Aussage ist bei Gruppenarbeiten auszufüllen und entfällt bei Einzelarbeiten -

Die Kennzeichnung der von mir erstellten und verantworteten Teile der ist erfolgt durch:

Ort	Datum	Unterschrift im Original