INF FAKULTÄT FÜR INFORMATIK

Bachelorarbeit

Reduzierung des Lernaufwandes für test-first Strategien (ein kontrolliertes Experiment)

Marius Bozem
Matrikel Nr.: 186424

13. September 2013

Gutachter:
Prof. Dr. rer. nat. habil. Gunter Saake

Betreuerin:
Dr. Janet Siegmund

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Marius Bozem
Magdeburg, 13.09.2013

Inhaltsverzeichnis

Verzeichnis der Auflistungen

Einführung von Abkürzungen

TDD Test-Driven Development
BDD Behavior-Driven Development

Zusammenfassung

Die Entwicklung von kommerzieller und auch von Open-Source-Software ist ein komplexer, aber strukturierter Prozess. Die Komplexität entsteht, da von einem Team, bestehend aus verschiedenen Personen und verschiedenen Rollen, eine neuartige Sache geschaffen werden soll. Um die Herausforderungen bei der Entwicklung von Software zu meistern, wird der Prozess in unterschiedliche Phasen eingeteilt. In jeder dieser Phasen werden Komponenten der Software entwickelt. Dazu können die Erstellung eines Pflichtenhefts, die Qualitätssicherung oder das Schreiben des eigentlichen Programmcodes gehören. Neuere Ansätze der Softwareentwicklung versuchen dabei, den Prozess möglichst flexibel und schlank zu gestalten. Dabei wird die Software in iterativen Schritten entwickelt, um somit auch schon bei einem nur teilweise fertig gestellten Produkt Rückmeldungen von den Nutzern einholen so können. Diese aus den Rückmeldungen gezogenen Schlüsse fließen in die anschließend getroffenen Entscheidungen ein. Es ist daher wichtig, auf sich ändernde Bedingungen reagieren zu können.

Dies wird erreicht, indem begleitend zum Programmcode eine ausführliche Sammlung von Tests geschrieben wird. Diese Tests haben die Aufgabe, Teile des Programms auszuführen und deren korrektes Verhalten sicherzustellen. Müssen im weiteren Verlauf der Entwicklung Änderungen durchgeführt werden, dann liefern diese Tests Rückmeldungen, welche Auswirkungen diese Änderungen gehabt haben. So ist es möglich, bestehenden Code zu ändern und schnell zu erfahren, ob das Programm immer noch korrekt arbeitet.

Diese Tests werden sogar vor dem Code, den sie testen sollen, geschrieben. Dieser Ansatz der Softwareentwicklung wird in dieser Arbeit behandelt. Es wird dabei die Fragestellung untersucht, vor welche Herausforderungen man gestellt wird, wenn man das Vorgehen, erst einen Test zu schreiben, bevor man den getesteten Code schreibt, lernen und anwenden möchte. Es haben sich dabei zwei hauptsächliche Herausforderungen gezeigt. Zum einen muss man bereit sein, seine gefasste Meinung zum Programmieren zu überdenken. Zum anderen muss ein Weg gefunden werden, um die benötigten Fertigkeiten für diese Art der Entwicklung auszubauen. In dieser Arbeit wird dazu als Lösung vorgeschlagen, Programmierern eine Sammlung Tests zur Verfügung zu stellen, mit deren Hilfe sie dann das Programm entwickeln sollen, zu dem die Tests sie führen. Diese Lösung wurde im Rahmen dieser Arbeit mit einem Experiment evaluiert. Als Ergebnis hat sich gezeigt, dass diese vorgegebenen Tests als hilfreiche Unterstützung beim Programmieren empfunden werden und einen möglichen Einstieg in das Schreiben von Tests bieten können.

1 Einführung

Dieses Kapitel dient als Einstieg in die vorliegende Arbeit. Dazu wird in Kapitel 1.1 beschrieben, in welchem Fachgebiet die Arbeit angesiedelt ist. Anschließend folgt in Kapitel 1.2 die Beschreibung der Motivation, die zur Entstehung der Arbeit geführt hat. Daraus wurden Ziele formuliert, die mit der Arbeit erreicht werden sollten. Deren Beschreibung und die zur Erreichung notwendigen Aufgaben werden in Kapitel 1.3 erläutert.

1.1 Hintergrund

Zur Entwicklung von Software können verschiedene Verfahren eingesetzt werden. Die als "klassisch" bezeichneten Verfahren geben dabei eine klar definierte Vorgehensweise vor und Zerlegen den Prozess in einzelne Phasen (Gloger und Häusling [5], S. 6). Typisch sind Planung, Definition, Entwurf, Implementierung, Testen, Einsatz und Wartung. Jede Phase hat dabei einen vorgegebenen Start- und Endpunkt. Die Phasen werden sequenziell abgearbeitet, und bevor eine nachfolgende Phase begonnen werden kann, muss ihr Vorgänger abgeschlossen sein. Darüber hinaus sind auch die Aufgaben der Teammitglieder vorgegeben. Jedem ist ein kleiner Bereich zugeordnet, auf dem er als Spezialist seine Arbeit zu erledigen hat (Gloger und Häusling [5], S. 6).

Aus einer solchen Vorgehensweise können sich mehrere Probleme ergeben (Gloger und Häusling [5], S. 6).

- Der Auftraggeber hat wenig Einfluss auf die Art, auf die sich das Projekt entwickelt. Lediglich zu Beginn des Projekts, beim Festlegen der Anforderungen und bei der Endabnahme des fertigen Produkts kann er seine Meinung wirklich einbringen.

- Die Qualität kann leiden und die Kosten können sich erhöhen, wenn aufgrund einer sich nähernden Frist eine Phase bald abgeschlossen werden muss, für die aber eigentlich noch Zeit benötigt würde. Um die Frist einzuhalten, können technische Mängel ignoriert oder finanzielle Mittel aufgewendet werden.

- Teammitglieder werden voneinander abgekapselt, da ihnen klar vorgegeben ist, um welchen Bereich sie sich zu kümmern haben. Durch den fehlenden Austausch von Wissen im Team kann es dazu kommen, dass Fehler nicht rechtzeitig erkannt und eliminiert werden können.

Um die Nachteile der klassischen Methoden zu überwinden, können *agile Methoden* eingesetzt werden. Hinter agilen Methoden stehen mehrere Prinzipien (Kent u.a. [4]):

- Änderungen an den Anforderungen sind willkommen.
- Die Mitarbeiter, die an einem Projekt beteiligt sind, sollen eine hohe Motivation mitbringen, und Informationen werden am besten in persönlichen Gesprächen ausgetauscht.
- Funktionierende Software soll in regelmäßigen und kurzen Zeitabständen ausgeliefert werden können. Dazu muss eine konstante Entwicklungsgeschwindigkeit aufrecht erhalten werden, die auf einer nachhaltigen Entwicklung beruht.

Einer der Faktoren, der die Geschwindigkeit von Softwareentwicklung beeinflusst, ist die Qualität des zugrundeliegenden Codes (Martin [22], S. 4). Verschlechtert sich der Zustand des Codes, aus dem die Software besteht, im Laufe ihres Lebenszyklus, so nimmt auch die Geschwindigkeit ab, mit der sie weiterentwickelt und verbessert werden kann. Dies liegt darin begründet, dass Änderungen am Code nur schwer vorgenommen werden können, da stets die Befürchtung besteht, dass durch die Änderungen an anderen Stellen im System neue Fehler verursacht werden.

Um die Prinzipien der agilen Entwicklung aufrecht erhalten zu können, muss also die Qualität des Codes auf einem hohen Level gehalten werden (Kent u.a. [4]). Die einzige Möglichkeit, um schnell Software entwickeln zu können, ist es, den Code jederzeit so sauber wie möglich zu halten (Martin [22], S. 6).

Eine Lösung zur Sicherstellung von hoher Codequalität ist eine ausführliche Sammlung von Softwaretests (Martin [22], S. 124). Diese Tests werden von Programmierern geschrieben und führen Teile der Software automatisiert aus, um sicherzustellen, dass diese Softwareteile korrekt funktionieren. Diese Tests ermöglichen es, dass Code flexibel, wartbar und wiederverwendbar gestaltet werden kann.

1.2 Motivation

Tests können im Rahmen von agilen Entwicklungsmethoden als Möglichkeit zur Aufrechterhaltung der Entwicklungsgeschwindigkeit eingesetzt werden. Da bei agilen Methoden das Projekt nicht in Phasen unterteilt wird, gibt es dementsprechend auch keine Phase, der das Schreiben der Tests zugeordnet ist. Die Tests werden stattdessen parallel zur Entwicklung der Software geschrieben. Dabei stellt sich die Frage, ob die Tests vor oder nach dem Schreiben des Codes, den sie testen sollen, geschrieben werden.

In dieser Arbeit werden zwei Ansätze untersucht, bei denen die Tests geschrieben werden, bevor anschließend der Code folgt, der getestet werden soll. Bei den zwei Ansätzen handelt es sich um *Test-Driven Development (TDD)* und *Behavior-Driven Development (BDD)*. Im Folgenden wird die Herangehensweise, erst die Tests und anschließend den getesteten

Code zu schreiben, als “test-first” bezeichnet.

Die Motivation, die zur Erstellung dieser Arbeit geführt hat, ist die Erkenntnis, dass anscheinend sowohl junge als auch erfahrene Programmierer mit Problemen konfrontiert werden können, wenn sie die Anwendung der beiden Ansätze lernen (Janzen und Saiedian [15])(Melnik und Maurer [23])(Spacco und Pugh [29])(Buffardi und Edwards [7]). Zusammengefasst haben sich in den verwandten Arbeiten die folgenden Probleme gezeigt, die auch eine Relevanz für diese Arbeit haben:

- Es ist sowohl bei jungen als auch erfahrenen Programmierern wahrscheinlich, dass sie bereit sind, “test-first” einzusetzen, wenn sie schon die Gelegenheit gehabt haben, es auszuprobieren (Janzen und Saiedian [15]).
- Die größte Hürde, Programmierer zum Einsatz von “test-first” zu bringen, ist deren anfänglich fehlende Akzeptanz dafür. Bringt man sie zum Einsatz von “test-first” trotz ihrer anfänglichen Abneigung dagegen, so können die Programmierer die Vorteile von “test-first” akzeptieren und nachvollziehen (Buffardi und Edwards [7]).

Daraus ergibt sich die Frage, wie man Programmierern eine geeignete Gelegenheit bietet, “test-first” auszuprobieren, auf eine Art, die sie ihre eventuell vorhandene anfängliche Abneigung überwinden lässt, um damit den Nutzen dieser Herangehensweise nachvollziehen zu können.

1.3 Ziele und Aufgabenstellung

Zur Beantwortung der Frage wird in dieser Arbeit der folgende Vorschlag untersucht: Man stelle Programmierern eine Sammlung an fertig geschriebenen Tests zur Verfügung, für die sie dann, ein Test nach dem anderen, den Code schreiben sollen, der erforderlich für den jeweiligen Test ist.

Die beiden Gründe, warum dieser Vorschlag untersucht werden soll, sind:

1. Er soll einen einfachen Einstieg in die “test-first”-Herangehensweise bieten.
2. Die Programmierer sollen früh zu der Erkenntnis gebracht werden, dass “test-first” Vorteile beim Programmieren bietet.

Konsequentermaßen wird für diese Arbeit als Ziel festgelegt, eine Empfehlung aussprechen zu können, ob ein zur Verfügung Stellen von vorgegebenen Tests als sinnvoller Einstieg in den “test-first” Ansatz verwendet werden kann.

Die zum Erreichen der Ziele notwendigen Aufgaben sind:

1. Durchführung eines kontrollierten Experiment zu Evaluierung des Ansatzes.

2. Die untersuchten Ausprägungen von “test-first” Entwicklung in dieser Arbeit sind Test-Driven Development (TDD) und Behavior-Driven Development (BDD). Die zweite Aufgabe ist deshalb, herauszuarbeiten, ob sich TDD oder BDD im Zusammenhang mit vorgegebenen Tests besser eignet.

2 Theoretischer Hintergrund

In diesem Kapitel wird der theoretische Hintergrund vermittelt, der zum Verständnis der restlichen Arbeit benötigt wird.

Da im Rahmen dieser Arbeit die Effekte von TDD und BDD auf Softwareentwicklung untersucht werden sollen, werden hier die beiden Ansätze erläutert. Zuerst wird TDD in Abschnitt 2.4 eingeführt, da BDD darauf basiert. BDD folgt dann in Abschnitt 2.5. Zu beiden Ansätzen folgt als erstes eine Definition, die für diese Arbeit relevant ist. Anschließend werden die Auswirkungen der Anwendung auf die Softwareentwicklung theoretisch untersucht und sowohl Vor- als auch Nachteile aufgeführt. Dies soll dabei helfen, einen Eindruck über die beiden Ansätze zu vermitteln.

Vor der Einführung von TDD und BDD werden zwei zum Verständnis benötigte Begriffe eingeführt: Refaktorieren und Unit Tests. Refaktorieren ist ein Vorgang, bei dem bereits existierender Code überarbeitet wird. Bei Unit Tests handelt es sich um eine Methode, bei der einzelne Codeeinheiten getestet werden, um zu bestimmen, ob sie eingesetzt werden können (Kolawa und Huizinga [18], S. 426).

2.1 Refaktorieren

Refaktorieren ist eine kontrollierte Technik, um das Design von bereits existierendem Code zu verbessern (Fowler [10], Preface). Es erhöht die Qualität des Codes durch Eliminierung von Duplikationen und der Verbesserung der internen Struktur.

Refaktorieren funktioniert nach dem Prinzip, mehrere kleine nicht-verhaltensändernde Transformationen durchzuführen, die zwar einzeln betrachtet nicht erscheinen, als ob sie die benötigte Zeit wert wären, aber einen kumulativen Effekt haben, der sie effektiv macht. Die Transformationen werden in kleinen Schritten durchgeführt, um das Risiko, einen Fehler einzuführen, so gering wie möglich zu halten.

Für das Refaktorieren sollte im Idealfall nicht extra Zeit eingerichtet werden, sondern es sollte kontinuierlich und in kleinen Phasen durchgeführt werden. In Einzelfällen können jedoch charakteristische Situationen auftreten, in denen es nötig ist, gesondert Zeit zum Refaktorieren einzuplanen (Fowler [10], S. 49 ff).

Eine typische Situation dafür ist, wenn ein neues Feature implementiert werden soll. Um dieses neue Feature programmieren zu können, muss meistens schon existierender Code bis zu einem gewissen Maß verändert werden. Hier wird Refaktorieren angewendet, um beim

Verständnis des Codes zu helfen, der modifiziert werden muss. Es kann an diesem Punkt auch vorkommen, dass durch das derzeitige Design des Programms das neue Feature nur schwierig implementiert werden kann. Diese Schwächen im Design können dann durch Refaktorieren beseitigt werden.

Eine ähnlicher Umstand ist das Beheben eines Bugs. Auch hier hilft Refaktorieren dabei, sich ein Verständnis des Codes zu erarbeiten, welches benötigt wird, um das Problem zu beheben.

2.2 Unit Tests

Unit Tests gehören zu einer Gruppe von Tests, die beim Anwenden von TDD geschrieben werden. Sie werden in der Regel zeitgleich mit dem Produktionscode entwickelt (Hamill [13], S. 1). Ein einzelner Unit Test sollte ein bestimmtes Verhalten im Produktionscode testen (Hamill [13], S. 2). Um dies sicherzustellen, muss der Test unabhängig von Einflüssen von außen laufen. Dazu wird in den Tests eine neutrale Umgebung oder ein neutrales Szenario eingerichtet, das andere Einflüsse eliminiert. Zu diesen Einflüssen zählen auch die anderen Tests. Es muss für den Ausgang eines Tests egal sein, in welcher Reihenfolge er gestartet wird und welche Tests bereits vorher gelaufen sind.

Werden in den Tests beispielsweise Datenbankabfragen ausgeführt, dann muss vorher sichergestellt werden, dass die Tabellen der Datenbank in einen im Test definierten Zustand versetzt werden.

Unit Tests nutzen die Objekte aus dem Produktionscode, existieren selbst aber innerhalb eines Unit Test Frameworks. Diese Frameworks führen die Unit Tests aus. Dadurch ergeben sich die folgenden Vorteile (Hamill [13], S. 2):

- Der Produktionscode wird nicht mit dem Test Code vermischt.
- Die Größe des Programms im kompilierten Zustand wird kleiner gehalten.
- Die Tests können unabhängig vom Programm laufen.
- Die Objekte können isoliert getestet werden.

Unit Tests gehören zur Kategorie der “white box” Tests (Hamill [13], S. 2). Sie haben Zugriff auf die internen Abläufe des Produktionscodes, den sie testen. Sie stehen damit im Gegensatz zu “black box” Tests, die nur von außen ersichtliches Verhalten testen. Da es bei objekt-orientierter Programmierung Zugriffsbeschränkungen auf Methoden innerhalb von Klassen gibt, wird in den Tests auch nur mit dem öffentlichen Interface der Klassen interagiert. Dadurch haben die Unit Tests Einfluss auf das Design des Produktionscodes. Um in den Tests mit den Objekten und Klassen des Produktionscodes zu interagieren, müssen

diese leicht von außen zugänglich sein. Dadurch können diese Objekte dann innerhalb des Produktionscodes von anderen Objekten ebenso leicht genutzt werden.

2.3 Beispiel zu Refaktorieren und Unit Tests

In Auflistung 1 ist der Code einer Klasse abgebildet, die zur Berechnung der Punkte beim Bowling Spielen eingesetzt wird. Die `Game`-Klasse bietet die `throwPins()` Methode, der die geworfenen Pins übergeben werden. Anschließend können über die Methoden `score()` oder `scoreForFrame()` die Punkte berechnet werden. `score()` berechnet dabei die aktuellen Gesamtpunkte während bei der `scoreForFrame()` Methode auch die Punkte zu früheren Zeitpunkten berechnet werden können.

```
1  public class Game {
2     private int[] theThrows = new int[21];
3     private int totalThrows = 0;
4     public void throwPins(int pins) {
5        theThrows[totalThrows++] = pins;
6     }
7     public int score() {
8        return scoreForFrame(10);
9     }
10    public int scoreForFrame(int theFrame) {
11       int sum = 0, throwsTillFrame = theFrame * 2;
12       for(int i = 0; i < throwsTillFrame; i++) {
13          if(theThrows[i] == 10) {
14             sum += theThrows[i] + theThrows[i+1] + theThrows[i+2];
15             throwsTillFrame--;
16          }
17          else if(theThrows[i] + theThrows[i+1] == 10) {
18             sum += theThrows[i] + theThrows[i+1] + theThrows[i+2];
19             i++;
20          }
21          else {
22             sum += theThrows[i] + theThrows[i+1];
23             i++;
24          }
25       }
26       return sum;
27    }
28 }
```

Auflistung 1: *Vor dem Refaktorieren: Berechnung der Punkte eines Bowling Spiels*

Die Klasse berechnet zwar die Punkte korrekt, allerdings zeigt ihr Design Schwächen, welche das Verständnis der Klasse schwierig gestalten. So ist die `scoreForFrame()` mit

Bevor der Code refaktoriert wird, sollte eine solide Sammlung an Tests geschrieben werden für die Funktionalität, die überarbeitet werden soll (Fowler [10], S. 17). Diese Tests sind notwendig, da auch bei kleinen und strukturierten Refaktorierungen Fehler eingeführt werden können. Sie dienen als Kontrolle beim Refaktorieren und liefern Rückmeldung, ob der Code immer noch korrekt funktioniert.

In Auflistung 2 ist ein Auszug der Testsammlung zu sehen. Abgebildet sind drei Tests, die

jeweils unterschiedliche Situationen der `Game`-Klasse testen. Hier wird getestet, dass das einfache Werfen von Pins funktioniert und dass die Berechnung von Strikes (Umwerfen aller 10 Pins mit einem Wurf) und Spares (Umwerfen aller 10 Pins mit zwei Würfen eines Durchgangs) korrekt erfolgt.

```
1  public class TestGame {
2     private Game game;
3     @Before
4     public void setUp() {
5        game = new Game();
6     }
7     @Test
8     public void TwoSimpleThrows() {
9        game.throwPins(5);
10       game.throwPins(4);
11       assertEquals(9, game.score());
12    }
13    @Test
14    public void Spare() {
15       game.throwPins(7);
16       game.throwPins(3);
17       game.throwPins(6);
18       assertEquals(16, game.scoreForFrame(1));
19       assertEquals(22, game.score());
20    }
21    @Test
22    public void Strike() {
23       game.throwPins(10);
24       game.throwPins(3);
25       game.throwPins(6);
26       assertEquals(19, game.scoreForFrame(1));
27       assertEquals(28, game.score());
28    }
29    // ...
30 }
```

Auflistung 2: *Unit Tests für die Bowling Spiel Klasse*

Die Tests laufen dabei alle nach folgendem Prinzip ab:

- Instanziiere die `Game`-Klasse vor jedem Test neu, damit die Tests unabhängig voneinander sind. Dazu dient die `setUp()` Methode, die einmal vor jedem Test aufgerufen wird.
- Werfe beliebig häufig Pins, um den Zustand zu erreichen, der getestet werden soll.
- Überprüfe, ob die von der Klasse berechneten Punkte mit der erwarteten Punktezahl übereinstimmen.

In Auflistung 3 ist die refaktorierte Version der `Game`-Klasse zu sehen. Angewendet wurden die folgenden Refaktorierungsmethoden:

Methodenextraktion Es werden Codefragmente zu einer Methode zusammengefasst. Der Name erläutert die Absicht der Methode (Fowler [10], S. 89). Im Beispiel wurden Methoden eingeführt zur Überprüfung, ob es sich um einen Strike oder Spare handelt (`isStrike()` und `isSpare()`). Auch die Berechnung der Punkte in den jeweili-

gen Situationen (Strike, Spare oder keins der beiden) wurde in eigene Methoden extrahiert.

Variablen umbenennen Der Code sollte dem Leser klar vermitteln, welchem Zweck er dient (Fowler [10], S. 22). Die Namen der Variablen sind dabei von zentraler Bedeutung. Im Beispiel wurde die Schleifenvariable `i` umbenannt in `currrentThrow`.

```
1  private int currentThrow;
2  public int scoreForFrame(int theFrame) {
3     int sum = 0, throwsTillFrame = theFrame * 2;
4     for(currentThrow = 0; currentThrow < throwsTillFrame; currentThrow++) {
5        if( isStrike() ) {
6           sum += getStrikePoints();
7           throwsTillFrame--;
8        }
9        else if( isSpare() ) {
10          sum += getSparePoints();
11          currentThrow++;
12       }
13       else {
14          sum += getPoints();
15          currentThrow++;
16       }
17    }
18    return sum;
19 }
20 private int getPoints() {
21    return theThrows[currentThrow] + theThrows[currentThrow+1];
22 }
23 private int getStrikePoints() {
24    return 10 + theThrows[currentThrow+1] + theThrows[currentThrow+2];
25 }
26 private int getSparePoints() {
27    return 10 + theThrows[currentThrow+2];
28 }
29 private boolean isStrike() {
30    return theThrows[currentThrow] == 10;
31 }
32 private boolean isSpare() {
33    return theThrows[currentThrow] + theThrows[currentThrow+1] == 10;
34 }
```

Auflistung 3: *Nach dem Refaktorieren: Der Code wurde mittels Refaktorierungsmethoden überarbeitet*

2.4 Test-Driven Development

Im folgenden Abschnitt wird TDD eingeführt. Dazu folgt als erstes eine kurze Übersicht, in der beschrieben wird, wie TDD entstanden ist. Es wird außerdem erklärt, welche Schritte und Regeln zu befolgen sind, wenn man TDD anwenden möchte. Es existieren zwar verschiedene Interpretationen von TDD, allerdings sind die Kernkonzepte überall die gleichen. Abschließend folgt eine Auflistung der Vor- und Nachteile von TDD, welche zum Teil verantwortlich für die Entstehung von BDD waren.

2.4.1 Einführung

TDD ist der zentrale Teil der agilen Softwareentwicklung, welche sich aus *Extreme Programming* (Beck und Andres [3]) abgeleitet hat. Mit TDD sollen mehrere positive Effekte erzielt werden. So soll es

- Testbarkeit garantieren
- zu einer extrem hohen Testabdeckung führen (Astels [1])
- den Entwicklern zu hohem Vertrauen verhelfen
- zu starker Kohäsion führen
- zu lose gekoppelten Softwaresystemen führen

Trotz des Namens ist TDD keine Technik, um Software zu testen, sondern vielmehr eine Entwicklungs- und Designtechnik (Beck [2]). Dabei werden Tests vor der Implementierung geschrieben und sobald die Tests erfolgreich sind wird der Code refaktoriert, um die interne Struktur zu verbessern. Erfolg oder Fehlschlagen der Tests wird beim automatisierten Ausführen selbiger festgestellt. Hier erhält man zu jedem Test, der ausgeführt wurde eine Rückmeldung, ob alle im Test festgelegten Erwartungen erfüllt wurden oder nicht. Dieses schrittweise Vorgehen wird wiederholt, bis alle benötigten Funktionen implementiert wurden (Astels [1]). Ein TDD Zyklus besteht aus den sechs folgenden Schritten (Beck [2]) (vgl. Abbildung 1):

1. Schreibe einen Test
2. Stelle sicher, dass der Test fehlschlägt
3. Schreibe Produktionscode, bis der Test erfolgreich ist
4. Lasse alle Tests laufen, um sicherzustellen, dass keine Fehler eingeführt wurden.
5. Refaktoriere
6. Lasse alle Tests laufen, um sicherzustellen, dass keine Fehler eingeführt wurden.

Im ersten Schritt wird der Testcode geschrieben, der die gewünschte Funktionalität testet. Der zweite Schritt ist notwendig, um sicherzustellen, dass der Test korrekt ist, er also zu diesem Zeitpunkt fehlschlägt, da die Implementierung noch nicht existiert. Sollte der Test nicht fehlschlagen, wird entweder nicht das gewünschte Verhalten getestet oder es wurden nicht die TDD Schritte befolgt. Im dritten Schritt wird der Produktionscode geschrieben. Hier sollte beachtet werden, dass nur soviel Code geschrieben wird, wie benötigt wird, um alle Tests in einen erfolgreichen Zustand zu versetzen (Astels [1]). Anschließend lässt

man alle Tests laufen, um zu kontrollieren, dass nicht an anderen Stellen Fehler entstanden sind. Zur Erhöhung der internen Code Qualität wird jetzt der Code refaktoriert. Bei TDD wird versucht, nachdem man einen Test geschrieben hat, diesen so schnell wie möglich erfolgreich zu machen. In diesem Schritt wird nicht unbedingt direkt eine perfekte Implementierung entwickelt. Erst nachdem der Test erfolgreich ist, erfolgt die Verbesserung der Code Qualität. Refaktorierung ist demnach einer der zentralen Schritte im TDD Zyklus. Nach der Refaktorierung müssen immer noch alle Tests erfolgreich sein.

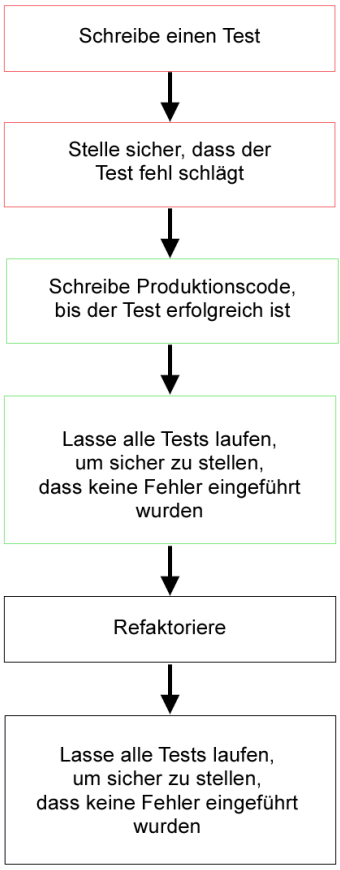


Abbildung 1: *Die sechs Schritte des TDD Zyklus. Da der neu geschriebene Test in den Schritten eins und zwei fehlschlägt, sind diese rot markiert. In den nächsten beiden Schritten sind alle Tests erfolgreich, deshalb die grüne Markierung. Rot und Grün sind die Farben, die in der Regel von Testing-Frameworks eingesetzt werden, um anzuzeigen, ob ein Test fehlschlägt oder erfolgreich ist.*

Es existieren verschiedene Interpretation von TDD. Streng betrachtet muss man sich an die folgenden drei Regeln halten (Martin [21]), um gemäß TDD zu entwickeln:

- **Regel 1**: Es darf kein Produktionscode geschrieben werden, solange es keinen fehlschlagenden Unit-Test gibt
- **Regel 2**: Es darf nur die minimal notwendige Menge an Testcode geschrieben werden, der ausreicht, um den Test fehlschlagen zu lassen. Dabei gilt es schon als Fehlschlag, wenn der Test nicht kompiliert.
- **Regel 3**: Es darf nicht mehr Produktionscode geschrieben werden als nötig, um den

derzeit fehlschlagenden Test erfolgreich zu machen.

Dies führt zu einem kurzen Zyklus, in dem ständig hin und her gewechselt wird zwischen dem Schreiben der Tests und dem Schreiben des nötigen Produktionscodes. Die Tests und der Produktionscode werden zusammen geschrieben, wobei die Tests nur kurze Zeit vor dem Produktionscode geschrieben werden.

Folgt man ständig diesen Schritten, dann ist erkennbar, dass TDD zu einer extrem hohen Codeabdeckung führt. Die Codeabdeckung gibt an, wie viele Zeilen des Produktionscodes von den Tests ausgeführt werden und berechnet sich nach der Formel:

$$Codeabdeckung = \frac{\#\, Lines\, of\, Code\, executed\, by\, Tests}{\#\, Total\, Lines\, of\, Code}$$

Dementsprechend ergibt sich im Laufe der Entwicklung eine beachtliche Menge an Tests. Hier stellt sich die Frage, ob durch den Einsatz von TDD nicht mehr Zeit gebraucht wird, um das Programm fertig zu stellen. Hierauf wird in Kapitel 2.4.2 zu den Vor- und Nachteilen von TDD näher eingegangen.

2.4.2 Vor- und Nachteile

Im folgenden Abschnitt werden verschiedene Vor- und Nachteile von TDD aufgezeigt. Der Ansatz, Tests zu schreiben vor der eigentlichen Implementierung, entspricht nicht dem klassischen Verlauf von Softwareentwicklung, bei dem Tests in der Regel erst vor der Übergabe an die Produktion geschrieben werden. Daher ist es sinnvoll, bei der Entscheidung, ob man TDD anwenden möchte, die potentiellen Vor- und Nachteile zu kennen.

Der besseren Übersicht halber werden hier die Auswirkungen, die dann anschließend erläutert werden, aufgelistet:

1. Auswirkungen auf den Zeitbedarf zur Entwicklung der Software
2. Dauer zur Ausführung der Tests
3. Auswirkungen auf das Design der Software
4. Tests als Dokumentation für Programmierer

1. Ein Kritikpunkt von TDD ist die mögliche negative Auswirkung auf den Zeitbedarf zur Entwicklung der Software. Programmierer haben bei TDD zusätzlich zum Schreiben des Produktionscodes noch die Aufgabe, eine vollständige Sammlung an Tests zu schreiben, die die Funktionen ihres Programmcodes verlässlich verifiziert. Dadurch erhöht sich die gesamte Menge an Code, die ein Programmierer zu schreiben hat. Außerdem müssen die

Tests genau wie der Produktionscode gewartet werden. Auch die Tests müssen refaktoriert und angepasst werden, sollte die Implementierung geändert werden.

TDD kann allerdings an anderen Stellen auch zu Zeiteinsparungen führen. Aufgrund der Tatsache, dass bei TDD immer in kurzen Schritten neue Funktionalität hinzugefügt wird, befindet man sich häufig an einem Punkt, an dem alle Tests erfolgreich sind, also ein Punkt, an dem das Programm korrekt funktioniert. So muss man bei richtiger Anwendung von TDD nur wenig Zeit zum Debuggen aufwenden. Probleme können schneller gefunden und leicht behoben werden (Shore und Warden [28], S. 301).

2. Eine der Anforderungen an die Sammlung von Tests in diesem Zusammenhang ist, dass die Ausführung der Tests nur wenige Sekunden in Anspruch nehmen darf (Shore und Warden [28], S. 297). Die Tests müssen beim Programmieren regelmäßig in kurzen Zeitabschnitten ausgeführt werden, um als hilfreiche Rückmeldung dienen zu können. Muss zu lange auf die Tests gewartet werden, besteht die Gefahr, dass der Programmierer in der Zwischenzeit abgelenkt wird und mehr Zeit als nötig verloren geht, bis die Arbeit wieder aufgenommen wird. Die komplette Sammlung von Tests sollte aus diesem Grund weniger als 10 Sekunden zum Durchlaufen benötigen (Shore und Warden [28], S. 297).

3. Ein positiver Effekt, entstanden durch den Einsatz von TDD, besteht darin, dass die Software flexibel designed werden kann und sich somit an verändernde Anforderungen leicht anpassen lässt (Martin [20], S. 90). Einer der Gründe, warum die Qualität von Software im Laufe ihrer Existenz sinkt, liegt darin begründet, dass sich Anforderungen und Spezifikationen ändern. Neue Funktionen müssen hinzugefügt und bereits bestehende geändert werden. Diese Änderungen können meist nicht im Voraus im ursprünglichen Design der Software vorhergesehen und beachtet werden. Häufig müssen diese Anpassungen schnell durchgeführt werden, und dies oft von Programmierern, die mit der ursprünglichen Designphilosophie nicht vertraut sind. Obwohl die Designänderungen anschließend zwar die geforderten Ansprüche erfüllen, verletzen sie das ursprüngliche Design. Sammeln sich mehrere dieser Verletzungen an, sinkt die Qualität des Designs stark ab.

Da sich die Anforderungen an Software bei den meisten Projekten unweigerlich ändern, muss eine Möglichkeit gefunden werden, das Design widerstandsfähig gegenüber Änderungen zu machen. Mithilfe von TDD lässt sich das Design einfach und sauber halten, gestützt von einer Sammlung von Tests. Dadurch bleibt es flexibel und wird gegenüber Änderungen offen gehalten (Martin [20], S. 90).

4. Ferner stellt eine Sammlung von Tests eine Form der Dokumentation für Programmierer dar. Um herauszufinden, wie Funktionen aufgerufen oder Objekte erstellt wurden, kann es hilfreich sein, sich die entsprechenden Tests anzuschauen, in denen diese Funktionalität ausgeführt wird. Die Tests dienen damit als Beispiele bzgl. der Benutzung des Codes. Der Vorteil gegenüber klassischer Dokumentation besteht darin, dass die Tests immer aktuell bleiben bzw. direkt ersichtlich ist, wenn Test und Code nicht mehr synchronisiert sind, da

der Test dann nicht mehr erfolgreich ist (MARTIN [20], S. 24).

Siehe Auflistung 4 für ein Beispiel.

```
1  public class AccountTest  {
2     public void testAccountAmortizesCorrectly()  {
3        double value = Any.value();
4        int term = Any.term();
5        int yearToWriteOff = Any.yearUpTo(term);
6
7        Account testAccount = new Account(value, term);
8        double expectedAmount = max(value/term, 100.00);
9
10       double actualAmount = testAccount.amortize(yearToWriteOff);
11
12       assertAreEqual(expectedAmount, actualAmount, 1);
13    }
14 }
```

Auflistung 4: *Ausführbare Dokumentation* KOLSKY UND BAIN *[19]: Beispiel, wie Tests als Dokumentation für Entwickler dienen können.*

Anhand des Tests in Auflistung 4 können die folgenden Informationen hergeleitet werden:

- Es gibt ein Objekt namens *Account* (Zeile 7), das sich selbst amortisieren kann (Zeile 10).
- *Account* erhält über seinen Konstruktor zwei Argumente, *value* und *term* (Zeile 7).
- *value* ist vom Typ "double" (Zeile 3), *term* ist ein Integer (Zeile 4). Keiner der Werte ist beschränkt. *Any* ist eine Klasse, die als Hilfe beim Erstellen der Tests dienen soll. Sie kann genutzt werden, um Zufallswerte von bestimmten Datentypen zu erhalten. Dadurch wird bei jedem Durchlaufen der Tests ein zufälliger Wert genutzt. Dies verdeutlicht, dass die Werte nicht beschränkt sind und die Tests mit beliebigen Werten funktionieren müssen.
- Amortisieren bedeutet in diesem Zusammenhang eine Abschreibung eines Gegenstandes (Zeile 10).
- Alle Jahre schreiben sich auf die gleiche Weise ab. Erkennbar durch die Nutzung der *Any* Klasse, die ein zufälliges Jahr zurück gibt, welches kleiner oder gleich *term* ist (Zeile 5).
- Zur Abschreibung nutzt man die *amortize()* Methode und übergibt ihr das Jahr der Abschreibung (Zeile 10).
- Der Wert der Abschreibung sollte *value/term* betragen, allerdings nicht mehr als 100 (Zeile 8).
- Abweichungen unter einem Wert von *1* werden ignoriert (Zeile 12, drittes Argument der *assertAreEqual()* Methode)

Mit Hilfe solcher ausführlichen Testfälle können am Projekt beteiligte Entwickler sich mit dem existierenden Code vertraut machen. Sie können lernen, wie das System funktionieren sollte und wie es strukturiert ist.

2.4.3 Zusammenfassung

TDD ist eine Praxis, die sich im Rahmen von *Extreme Programming* entwickelt hat. TDD wird von Softwareentwicklern betrieben und der Hauptgrund, mit TDD zu entwickeln, ist die Hilfe, die es beim Programmieren darstellt. Obwohl verschiedene Ansichten von TDD existieren, stimmen alle in den folgenden Punkten überein:

1. Schreibe einen Test für eine noch nicht existierende Funktionalität
2. Schreibe Code um den Test erfolgreich zu machen
3. Refaktoriere den Produktionscode und die Tests

Ein Problem von TDD ist die fehlende Einbeziehung von Nicht-Programmierern, die am Projekt beteiligt sind, wie z.B. Produktmanager. Somit entsteht durch das Anwenden von TDD kein Vorteil, von dem das ganze Team profitieren würde.

Die ungewöhnliche Herangehensweise von TDD führt oft bei Programmierern zu Unverständnis und Ablehnung. So kann es beispielsweise fragwürdig erscheinen, wie etwas getestet werden kann, das noch nicht existiert. Tests werden traditionell angewendet, um Fehler aufzuzeigen und werden deshalb erst am Ende der Produktion eingeführt.

Als Konsequenz entstand BDD, worauf im folgenden Kapitel eingegangen wird.

2.5 Behavior-Driven Development

BDD war ursprünglich eine Weiterentwicklung von TDD und ist mittlerweile ein ganzheitlicher Prozess zum Entwickeln von Software in iterativen Schritten. Nach der Erklärung des Ursprungs von BDD folgt eine konkrete Beschreibung, wie BDD anzuwenden ist. Danach wird beschrieben, welche Auswirkungen BDD auf die Entwicklung von Software hat.

2.5.1 Einführung

BDD basiert auf den folgenden drei Prinzipien (North u.a. [8], S. 148):

- **Enough is enough**: Es müssen die Erwartungen der Stakeholder (Kunden, Aktionäre etc.) erfüllt werden, aber es sollte vermieden werden, mehr als nötig zu tun.

- **Deliver stakeholder value**: Alle Anstrengungen, die beim Entwickeln unternommen werden, sollten darauf gerichtet sein, den Stakeholdern einen vorzeigbaren Mehrwert zu liefern

- **It's all behavior**: Nicht nur die Anwendung selbst kann als Verhalten aus Sicht der Stakeholder beschrieben werden, sondern auch das Verhalten von low-level Code kann aus Sicht von anderem Code, der diesen nutzt, beschrieben werden.

BDD wurde ursprünglich von Dan North auf Basis von TDD entwickelt (North [26]). Beim Anwenden und Lehren von TDD stieß er bei mehreren Projekten in verschiedenen Umgebungen immer wieder auf die gleichen Probleme. Programmierer wollten wissen, wo sie anfangen müssen, was getestet werden soll und was nicht, wie viel auf einen Schlag getestet werden muss, wie die Tests benannt werden sollen und welche Schlüsse man aus einem fehlschlagenden Test ziehen kann.

Um diese Fragen zu beantworten, entwickelte North BDD. Es entstand auf Basis von etablierten agilen Methoden. Es zielt darauf ab, diese Methoden zugänglicher und effizienter zu machen für Teams, die noch wenig Erfahrung mit agiler Softwareentwicklung haben.

Ausgehend von dieser Prämisse unternahm North am TDD Prozess verschiedene Erweiterungen.

- Ausdrucksstarke Benennung von Klasse von Methoden

- Einführung einer universellen Sprache

Eine Änderung war, Klassen und Methoden ausdrucksstark zu benennen, sodass sie in Verbindung miteinander lesbare Sätze bilden.

```
1  public class UserRegistrationTest {
2      @Test
3      public void shouldValidateUserInput {
4          //...
5      }
6
7      @Test
8      public void shouldSaveNewUserToDatabase {
9          //...
10     }
11 }
```

Auflistung 5: *Ausdrucksstarke Benennung: Der Name der Klasse in Verbindung mit den Namen der einzelnen Tests lässt erkennen, was in dem Test passiert.*

Nimmt man den Namen einer Klasse und verbindet diesen mit dem Namen der Methoden darin, entsteht daraus eine Art Dokumentation (siehe Auflistung 5). Werden bei der Namensgebung Begriffe aus der "Business Domain" verwendet, dann können daraus sowohl neben den Programmierern auch Benutzer, Analysten und Tester einen Nutzen ziehen.

Methoden sollten mit dem Begriff “should” beginnen, was dabei helfen soll, nur Tests für die aktuelle Klasse zu schreiben und so fokussiert zu bleiben.

Ausdrucksstarke Namen der Tests sind hilfreich, sollte einer dieser Tests aufgrund von Änderungen am Code fehlschlagen. Mittels der Benennung kann das vom Code gewünschte Verhalten ohne viel Aufwand ermittelt und die Implementierung entsprechend angepasst werden.

Mit diesen Änderungen blieb BDD ebenso wie TDD nach wie vor eine Vorgehensweise von und für Programmierer. Es wurden zwar die Zugänglichkeit und der aus der Anwendung gezogene Nutzen erhöht, allerdings hatte BDD es noch nicht geschafft, alle am Projekt Beteiligten miteinzubeziehen und auch für Nicht-Programmierer einen Mehrwert erkennbar zu machen.

Um dies zu ändern, wird bei BDD eine universelle Sprache eingesetzt, um die Anforderungen an das Produkt zu spezifizieren. Diese universelle Sprache stellt ein konsistentes Vokabular für Analysten, Tester, Entwickler und Stakeholder bereit (North [26]). Ziel war es, Mehrdeutigkeiten und Missverständnisse zu verhindern, die auftraten, wenn technische Mitarbeiter mit Mitgliedern des Managements sprachen.

Zu einem ähnlichen Schluss kam auch Eric Evans, der folgendes schreibt:

> “A project faces serious problems when its language is fractured. Domain experts use their jargon while technical team members have their own language tuned for discussing the domain in terms of design. The terminology of day-to-day discussions is disconnected from the terminology embedded in the code (ultimately the most important product of a software project). And even the same person uses different language in speech and in writing, so that the most incisive expressions of the domain often emerge in a transient form that is never captured in the code or even in writing. Translation blunts communication and makes knowledge crunching anemic. Yet none of these dialects can be a commom language because none serves all needs. (Evans [9], S. 25)“

Evans beschreibt hier die Probleme, die auftreten, wenn die Projektsprache frakturiert, also nicht einheitlich, ist. Domainexperten nutzen ihr Jargon, während Entwickler ihre Sprache angepasst haben, um effizient über das Design der Software zu sprechen. Außerdem spiegelt sich die Terminologie tagtäglicher Gespräche nicht im Code wieder. Die prägnantesten Ausdrücke der Domain tauchen nur in vergänglicher Form auf und werden nicht in Code oder Text erfasst. Die notwendige Übersetzung zwischen den verschiedenen Dialekten der Sprache führt zu abgestumpften Diskussionen und keiner der Dialekte kann als gemeinschaftliche Sprache genutzt werden, da keiner alle Bedürfnisse erfüllt.

Mittel einer bewussten Bemühung des Teams kann eine universelle Sprache geschaffen werden, die von allen am Projekt Beteiligten verstanden werden kann. Wenn konsistent vom Team in Konversationen, Dokumentation und Code genutzt, können Unstimmigkeiten bei der Übersetzung vermieden und die Wahrscheinlichkeit von Missverständnissen reduziert werden.

An dieser Stelle sei erwähnt, dass BDD eine Weiterentwicklung aus TDD ist und auch darauf aufbaut. Deshalb ist es auch nicht ungewöhnlich, dass beim Anwenden von BDD bei der Implementierung der Software die Vorgehensweise von TDD verwendet wird.

Damit ist die allgemeine Einführung in BDD abgeschlossen. Im folgenden Abschnitt wird definiert, wie BDD angewendet wird.

2.5.2 Anwendung

Obwohl BDD als einfache Umstrukturierung von TDD begann, ist es mittlerweile zu einem ganzheitlichen Softwareentwicklungsprozess gewachsen (North u.a. [8], S. 138).

> "Behavior-Driven Development is about implementing an application by describing its behavior from the perspective of its stakeholders. (vgl. (North u.a. [8], S. 138))"

Ziel von BDD ist die Implementierung einer Applikation durch Beschreibung ihres Verhaltens aus der Sicht der Stakeholder.

Aus dieser Beschreibung von BDD ergeben sich verschiedene Konsequenzen. So muss die Ansichtsweise der Stakeholder verstanden werden, um sinnvolle Ergebnisse abliefern zu können. Dazu gehören auch ein Verständnis des Geschäftsbereichs und den damit verbundenen Herausforderungen und Chancen. Hierbei hilft die universelle Sprache, die bei BDD Anwendung findet, bei der Vermittlung zwischen Stakeholdern und Entwicklern.

BDD fokussiert sich weniger darauf, wie ein Objekt aufgebaut ist oder intern aussieht, sondern eher auf das Verhalten von Objekten, da dies in der Regel von größerer Bedeutung ist. So interessieren sich Stakeholder nicht dafür, ob Daten in einer relationalen oder nichtrelationalen Datenbank abgespeichert werden. Für sie zählt das Konzept dahinter: Die Daten sind persistent gespeichert.

Durch das Fokussieren auf Verhalten ändert sich die Art und Weise, wie man das Programm entwickelt. Es steht nicht mehr so sehr die Struktur von Objekten im Vordergrund, sondern mehr die Interaktionen zwischen Personen und dem System (North u.a. [8], S. 24).

Ähnlich wie bei TDD wird beim Entwickeln mit BDD ein zyklischer Prozess angewendet. Im ersten Schritt wird ein Verhalten der Anwendung mit einem dafür geeigneten Werkzeug spezifiziert. Diese Beschreibung eines gewünschten Verhaltens wird "Automated Acceptance Test" genannt (Wynne und Hellesoy [30], S. 4). Der Sinn dieser Tests besteht nicht darin, dass die Stakeholder eine Liste mit Anforderungen erstellen. Stattdessen sollen Entwickler und Stakeholder zusammen arbeiten und jeweils die Möglichkeit haben, Feedback zu geben. Die Tests sollen anschließend das vom Stakeholder gewünschte Ergebnis wiedergeben. Sie werden Akzeptanztests genannt, da sie ausdrücken, was die Software leisten soll, um von den Stakeholdern akzeptiert zu werden.

Diese Tests unterscheiden sich von Unit Tests dadurch, dass sie nicht direkt an die Programmierer gerichtet sind, sondern sicherstellen sollen, dass das Produkt entwickelt wird, das von den Stakeholdern gewünscht wurde.

Ausgehend von der nicht technischen Natur dieser Tests muss die verwendete Syntax möglichst natürlich klingen. Im Rahmen von BDD hat sich dazu die Verwendung von einigen Schlüsselwörtern durchgesetzt. Um die Sprache so einfach wie möglich zu halten, werden die Wörter "given", "when" und "then" in den Szenarien beschrieben. Mit "given" wird ein Kontext für das aktuelle Szenario hergestellt. Im "when" Part wird meistens ein bestimmtes Ereignis ausgelöst. Hier ist hilfreich, auch die Rolle (wie z.B. Admin oder User) festzulegen, die das Ereignis auslöst. Dies soll dabei helfen, das Szenario aus Sicht der betroffenen Rolle zu betrachten und so deren Anforderungen und Bedürfnisse zu beachten. Abschließend wird "then" genutzt, um das gewünschte Resultat des Szenarien zu spezifizieren.

Tabelle 1: *Beispiel für Given-When-Then Szenario (*North u.a. *[8], S. 26)*

Scenario:	*Pay a bill*
Given	*a checking account with $50*
And	*a payee named Acme*
And	*an Acme bill for $37*
When	*I pay the Acme bill*
Then	*I should have $13 remaining in my checking account*
And	*the payment of $37 to Acme should be listed in Recent Payments*

In Tabelle 1 ist ein Beispiel für ein Feature zu sehen. In dem Feature wird ein Szenario beschrieben, in dem eine Rechnung bezahlt werden soll. Der mit dem "given" Keyword hergestellte Kontext beschreibt einen Account mit einem Guthaben von $50. Der Empfänger der Rechnung ist die Firma "Acme", die auch die Rechnung über $37 gestellt hat. Wenn die Rechnung beglichen wird, dann sollten anschließend noch $13 als Guthaben auf dem Account verbleiben. Außerdem sollte die Transaktion in den zuletzt getätigten Zahlungen aufgelistet sein.

Nachdem mit einem Akzeptanztest ein zu entwickelndes Feature festgelegt wurde, beginnt dessen Implementierung. Hier wird schrittweise vorgegangen, in der Reihenfolge, in der die Schritte im Akzeptanztest auftauchen. In Tabelle 1 würde demnach zuerst ein Konto implementiert werden, auf dem ein Guthaben verbucht werden kann. Es bietet sich an, diese Schritte mithilfe von TDD zu entwickeln. Mit Unit Tests werden Beispiele für den Schritt aus dem Akzeptanztest ausgewählt, um den Schritt zu entwickeln (North u.a. [8], S. 29).

Ein BDD Zyklus ist in Abbildung 2 zu sehen. In Punkt 1 wird ein Szenario ausgewählt, das als nächstes implementiert werden soll. Danach wird in 2. ein (noch) fehlschlagender Schritt des Akzeptanztests geschrieben. Um diesen Schritt zu implementieren, wird der "red-green-refactor" Zyklus aus TDD angewendet in den Punkten 3 - 5. In Punkt 6 wurden ausreichend Beispiele entwickelt, und der Schritt des Akzeptanztests ist nun erfolgreich. Bevor nun der nächste Schritt folgt, wird in Punkt 7 das Programm refaktoriert.

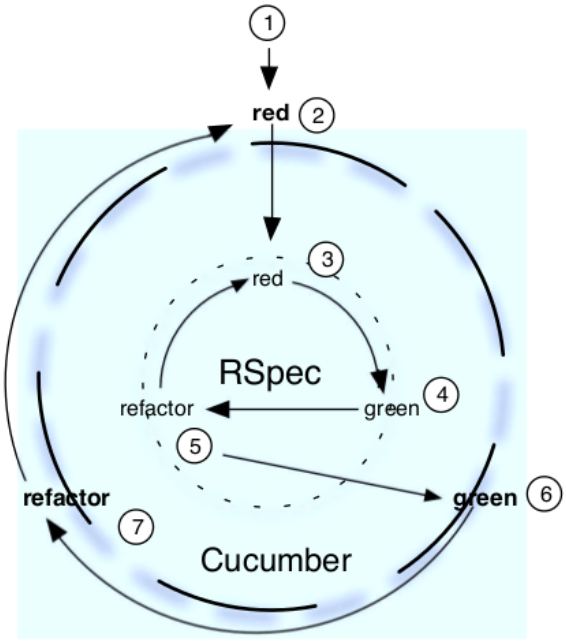


Abbildung 2: *BDD Zyklus (*North u.a. *[8], S. 29)*

2.5.3 Vor- und Nachteile

Nachdem nun erklärt wurde, was BDD ist und wie man es anwendet, sollen nun die Vor- und Nachteile untersucht werden, die BDD auf die Entwicklung von Software hat. Da sich BDD aus TDD entwickelt hat und teilweise der TDD Prozess ein Teil von BDD ist, kommen einige der Vor- und Nachteile von TDD auch bei BDD zum Tragen.

Hier zur besseren Übersicht eine Auflistung der möglichen Auswirkungen:

1. Tests als lebende Dokumentation

2. Iterativ Entwicklung von Software

3. Flexibles Design

1. Da in den Szenarien spezifiziert wird, wie sich die Software verhalten soll, stellen sie eine Art "lebende Dokumentation" dar (Wynne und Hellesoy [30], S. 6). Diese Art von Dokumentation bietet die gleichen Vorteile wie traditionelle Dokumentation, da sie auch von Stakeholdern gelesen und verstanden werden kann. Zusätzlich dazu können sie aber auch von einem Computer ausgeführt werden und liefern dadurch eine Rückmeldung, inwieweit die Software mit den Anforderungen übereinstimmt. Dadurch wird der Nachteil traditioneller Dokumentation, die einmal geschrieben und im Laufe der Zeit immer ungenauer und unverlässlicher wird, neutralisiert. Da die Szenarien zentral an einer Stelle gespeichert werden können, bieten sie auch eine Möglichkeit, Zeit einzusparen, die sonst aufgewendet werden müsste, Anforderungsdokumente, Tests und Code synchron zu halten.

2. BDD hilft außerdem dabei, Software iterativ zu entwickeln und regelmäßig neue Features zu veröffentlichen. Wenn die Features der Software in überschaubare Arbeitspakete geteilt werden, können diese vom Entwicklungsteam in relativ kurzen Zeiteinheiten fertig gestellt werden. Außerdem können die Features von den Stakeholdern priorisiert werden, so dass immer das Feature entwickelt wird, welches den Wert der Software am stärksten erhöht.

Dies hat auch den Vorteil, dass im Laufe des Entwicklungsprozess noch Einfluss auf die Software genommen werden kann. Statt vor Projektbeginn exakt und unwiderruflich festzulegen, was entwickelt werden soll, kann auf Feedback von verschiedenen Quellen zu bereits fertig gestellten Teilen Rücksicht genommen werden.

3. Da bei einem solchen flexiblen Entwickeln von Software beim Hinzufügen von neuen Features häufig das Design angepasst werden muss, sind auch hier die ausführbaren Akzeptanztests von großem Wert. Sie können als Regressionstests eingesetzt werden. Regressionstesting bezeichnet das Speichern und Laufen lassen der Tests, nachdem Änderungen an vorhandenem Code vorgenommen wurden (Wiley [25], S. 18). Der Sinn dieser Tests besteht darin, zu verhindern, dass Änderungen, beispielsweise das Beheben eines Fehlers, nicht an anderen Stellen neue Fehler entstehen lassen (Wiley [25], S. 172).

2.5.4 Zusammenfassung

BDD bietet auf der einen Seite eine zugänglichere Methode, testgetriebenen Code zu schreiben als durch den Einsatz von TDD, und auf der anderen Seite sind nicht mehr nur die Entwickler involviert, sondern auch alle anderen am Projekt beteiligten Stakeholder.

Damit bietet BDD Vorteile gegenüber der reinen Anwendung von TDD. Allerdings ist es auch nicht unüblich, sowohl BDD als auch TDD zu kombinieren.

3 Lehren von Test-Driven & Behavior-Driven Development

Im vorherigen Kapitel wurden die Entwicklungsmethoden TDD und BDD eingeführt. Es wurde jeweils erläutert, wie diese Methoden anzuwenden sind. Außerdem wurde aufgezeigt, welche Auswirkungen die Methoden auf den Softwareentwicklungsprozess haben. Dabei hat sich gezeigt, dass sowohl TDD als auch BDD positive Effekte auf die Entwicklung von Software haben.

Diese positiven Effekte konnten auch in Studien bestätigt werden. In "An Initial Investigation of TDD in Industry " (George und Williams [11]) wurden 24 erfahrene Programmierer im Rahmen eines Experiments in zwei Gruppen eingeteilt. Beiden Gruppen wurde die Aufgabe gestellt, ein kleines Java Programm zu entwickeln. Eine Gruppe entwickelte dabei mit TDD, während die Kontrollgruppe einen klassischen Wasserfall-Ansatz verfolgte. Die Studie kam zu dem Schluss, dass die TDD Gruppe eine höhere Codequalität (18% mehr bestandene Black Box Tests) erreichte.

In einer anderen Studie ("An Experimental Evaluation of the Effectiveness and Efficiency of TDD" (Gupta und Jalote [12])) wurden die Probanden auch in zwei Gruppen eingeteilt. Eine Gruppe nutzte TDD und die andere Gruppe einen klassischen Ansatz der Softwareentwicklung, um ein moderat großes Programm zu entwickeln. Dabei hat sich gezeigt, dass TDD effizienter ist, die Gruppe also insgesamt weniger Zeit benötigte als die Kontrollgruppe. Außerdem waren die Programmierer, die TDD anwendeten, produktiver. Produktivität wurde gemessen, indem die Menge an geschriebenem Code in Verhältnis zur benötigten Zeit gesetzt wurde.

Diese Studien und deren Ergebnisse werden hier erwähnt, da sie zwei Schlussfolgerungen ermöglichen. Sie bestätigen zum einen die Annahme, dass TDD und BDD wirkungsvolle Entwicklungsmethoden sind. Zum anderen zeigen sie, dass sich Experimente eignen, um diese Entwicklungsmethoden zu analysieren und zu erforschen.

Da sich gezeigt hat, dass TDD und BDD Vorteile bieten, scheint es sinnvoll, diese Methoden Programmierern beizubringen. Im Rahmen dieser Arbeit soll darauf aufbauend untersucht werden, wie Programmierer an TDD und BDD herangeführt werden können und wie es ihnen effizient beigebracht werden kann. Dazu folgt in Kapitel 3.1 ein Überblick über die Herausforderungen, vor die man beim Lehren von TDD und BDD gestellt wird. Außerdem wird in Kapitel 3.2 eine Möglichkeit vorgestellt, die beim Lösen einiger dieser Herausforderungen dienen soll. Anschließend wird in Kapitel 4 diese Idee mithilfe eines Experiments untersucht.

3.1 Herausforderungen verbunden mit dem Lehren von TDD und BDD

Im Folgenden werden zwei Schwierigkeiten untersucht, die mit dem Lehren von TDD und BDD verbunden sind. Diese Herausforderungen werden hier vorgestellt, da sich in verwandten Arbeiten gezeigt hat, dass sie häufig in Verbindung mit TDD und BDD auftreten. Das Verständnis der mit TDD und BDD verbundenen Anforderungen ist notwendig, um anschließend eine mögliche Lösung präsentieren zu können und diese experimentell zu evaluieren.

3.1.1 Umlernen von Design und Programmieren

Ein Problem ist, Studierende (Studierende stehen hier stellvertretend für Personengruppen, die zum ersten mal mit TDD und BDD in Kontakt kommen) dazu zu bringen, ihr Verständnis von Design und Programmierfähigkeiten zu überdenken (Kabbani und Kaner [17], S. 2). In einem Kurs der Universität Auckland sollte den Teilnehmern TDD beigebracht werden (Mugridge [24]). Dabei hat sich gezeigt, dass Studierende, die sich schon eine gewisse Menge an Erfahrung im Programmieren angeeignet hatten, unglücklich waren, jetzt "wieder neu anfangen zu müssen". Das erste mal Programmieren zu lernen war schon eine herausfordernde Angelegenheit für sie gewesen. Es war unausweichlich, dass die Studierenden einige vorgefasste Meinungen zu Programmierung mitbrachten und das schränkte deren Bereitschaft ein, TDD zu lernen und anzuwenden.

Zu diesem Schluss kamen auch die Verantwortlichen des Kurses "Software Testing 2" am *Florida Institute Of Technology* (Kabbani und Kaner [17]). Ein Ziel dieses Kurses war, grundlegende Programmierfähigkeiten neu zu lernen und mit diesen neuen Kenntnissen in Zukunft Code zu schreiben, der jederzeit funktioniert. Dadurch wurde von den Studierenden Disziplin und Motivation gefordert. Manche zeigten dabei Widerstand in verschiedenen Formen. So wurden bspw. Tests erst nach der Implementierung geschrieben, also in umgekehrter Reihenfolge als eigentlich gefordert. Das hat den Nachteil, dass nicht geprüft wird, ob ein Test auch wirklich das testet, was er soll, da nicht sichergestellt wird, dass der Test fehlschlägt, wenn die Implementierung nicht stimmt. Schreibt man den Test zuerst und stellt sicher, dass er auf die richtige Art und Weise fehlschlägt und kümmert sich erst dann um die Implementierung, so erreicht man ein höheres Level an Vertrauen gegenüber Test und Implementierung.

Der Ansatz, vor der Implementierung einer Funktion die Tests für diese Implementierung zu schreiben, unterscheidet sich von klassischer Softwareentwicklung. Tests werden in der Regel nach dem Schreiben des Productionscodes entworfen und dies kann dazu führen, dass dem Testcode nicht die nötige Beachtung geschenkt wird und zu wenig Zeit zum Testen aufgewendet wird (George und Williams [11]). Im Kurs "Software Testing 2" zeigte sich, dass die Studierenden dazu neigen, die notwendigen Tests erst nach der Im-

plementierung zu schreiben. Es ist daher erforderlich, diese “test-last” Herangehensweise beim Lernen von TDD und BDD gegen eine “test-first” Herangehensweise auszutauschen.

3.1.2 Ausbauen der für TDD und BDD benötigten Fertigkeiten

In beiden vorher erwähnten Kursen (Kabbani und Kaner [17]), (Mugridge [24]) wurden die folgenden zentralen Fertigkeiten herausgearbeitet, die entwickelt werden müssen, um TDD korrekt anwenden zu können:

- Umgang mit einem Testing-Framework
- Einsetzen eines Versionierungstools
- Korrektes Anwenden von Refaktorieren bei Testcode und Produktionscode

Diese Fertigkeiten sind die Kernkompetenzen, die für den Einsatz von TDD notwendig sind. Sie können aber auch in anderen Kontexten genutzt werden (Mugridge [24]).

In einem der Kurse (Mugridge [24]) wurde versucht, diese Kompetenzen inkrementell zu vermitteln und aufzubauen. Refaktorieren wurde eingeführt, indem in einer Aufgabe erst nur die schon existierenden Methoden einer einzelnen Klasse überarbeitet werden sollten, bevor später komplexere Beispiele folgten, in denen auch Programme mit mehreren Klassen und Vererbungen refaktoriert werden sollten. Problematisch war, keine passenden Tools (automatisiert oder manuell) zur Verfügung zu haben, um den Lernprozess voran zu treiben und den Studierenden schnelle Rückmeldungen über ihren Fortschritt liefern zu können. Es wurde hier der Schluss gezogen, dass einfache Aufgaben benötigt werden, um die Kompetenzen individuell und gezielt zu entwickeln.

In dem Kurs wurde versucht, die Fertigkeiten mithilfe der den Studierenden gestellten Aufgaben zu vermitteln. Auch hier war der Mangel an unterstützenden Tools um die Aufgaben zu bewerten hinderlich, da es sich als mühsam und sehr zeitaufwendig herausstellte. Zur Bewertung war es u.a. notwendig, verschiedene Coderevisionen zu evaluieren und die Tests laufen zu lassen.

In einigen Fällen wurde nur wenig Aufwand zum Refaktorieren betrieben (Kabbani und Kaner [17]). Refaktorieren wurde in dem Kurs in der dritten Woche eingeführt. Von den Studierenden wurde ab diesem Zeitpunkt bis einschließlich der Prüfung am Ende des Kurses erwartet, die Technik einzusetzen. Es war allerdings schwierig, die Studierenden davon zu überzeugen, dass beim Refaktorieren tiefgreifendere Änderungen durchgeführt werden sollten, als nur Variablen umzubenennen und unnötige Leerzeilen zu entfernen.

3.1.3 Zusammenfassung

TDD und BDD zu lernen ist verbunden mit einem Umdenken beim Programmieren und dem Designen von Software. Es ist daher erforderlich, die Studierenden davon zu überzeugen, ihre vorgefassten Meinungen zum Programmieren zu überdenken. Dadurch können sie sich mit den Herangehensweisen TDD und BDD auseinandersetzen und lernen, diese anzuwenden. Die Bereitschaft, sich mit TDD und BDD auseinanderzusetzen ist erforderlich, um die Fertigkeiten, die für deren Anwendung benötigt werden, gezielt ausbauen zu können.

3.2 Vorgeschriebene Tests als Unterstützung beim Lernen von TDD und BDD

Nachdem nun die hauptsächlichen Herausforderungen beim Lernen von TDD und BDD erläutert wurden, soll in diesem Kapitel eine Möglichkeit beschrieben werden, um diese Herausforderungen zumindest teilweise zu meistern. Die Idee ist, Programmierern eine Sammlung von vorgeschriebenen Tests zur Verfügung zu stellen, für die sie dann, ein Test nach dem anderen, die Implementierung (den Produktionscode) schreiben und so am Ende, wenn alle Tests erfolgreich sind, ein funktionierendes Programm haben. Die Programmierer sind also von der Aufgabe, selbst Tests zu schreiben, befreit und sollen stattdessen die Tests, die sie gestellt bekommen, nutzen, um ein Programm zu schreiben.

Dabei bauen die Tests, wie bei TDD und BDD üblich, aufeinander auf und es werden inkrementell neue Features zu dem Programm hinzugefügt. Es muss dabei sichergestellt werden, dass immer nur ein Test nach dem anderen implementiert wird und die anderen nachfolgenden Tests sowohl vom Programmierer als auch vom Testing-Framework ignoriert werden. Die Gründe dafür sind, dass

- bei TDD und BDD immer nur ein Test (und zwar der Test, dessen Funktionalität noch nicht implementiert ist) fehlschlagen sollte (vgl. Kapitel 2.4).
- sobald man anfängt, selbst die Tests zu schreiben, auch immer nur der als nächstes geschriebene Test fehlschlägt und es noch gar keine weiteren Tests gibt.
- die Programmierer durch die vorgegebenen Tests in eine bestimmte Richtung beim Programmieren gelenkt werden können. Die Erklärung, warum das hilfreich ist, folgt im weiteren Verlauf des Kapitels.

Durch das Bereitstellen der vorgeschriebenen Test sollen die folgenden Herausforderungen adressiert werden:

1. Inkrementelles Lernen der für TDD und BDD benötigten Fähigkeiten

2. Schnelles und effizientes Feedback liefern

3. Umlernen von Softwaredesign

4. Refaktorieren mit tiefgreifenden Änderungen

5. Umgang mit Testing-Framework lernen

1. Da die Programmierer die Tests nicht selbst schreiben müssen, lernen sie zuerst, wie die Tests aufgebaut sind und welche Rückmeldungen sie liefern, wenn sie ausgeführt werden. Erst im nächsten Schritt, wenn sie schon mit Tests vertraut sind, müssen sie Tests selbst schreiben. Damit bietet diese Herangehensweise eine Möglichkeit, TDD und BDD inkrementell zu lernen.

2. Wenn die Programmierer nach und nach die vorgeschriebenen Tests implementieren und sie erfolgreich machen, bietet dies eine Möglichkeit, den Programmierern Feedback zu liefern. Durch den Einsatz eines Testing-Frameworks kann leicht die Anzahl an implementierten Tests festgestellt werden und so der Fortschritt beim Programmieren gemessen und eingeschätzt werden.

3. Ein aktives Umlernen von Softwaredesign ist nur schwierig umzusetzen (Mugridge [24]). Stellt man bspw. eine Aufgabe, bei deren Komplexität es sich anbietet, fortgeschrittene Designpatterns anzuwenden, ist noch nicht sichergestellt, dass beim Lösen der Aufgabe auch diese Lösungsansätze Verwendung finden. Durch die Vorgabe der Tests ist es möglich, ein gewolltes Design des Programms zu erzwingen. Das wird z.B. dadurch erreicht, dass in den Tests bestimmte Klassen verwendet werden, die dann vom Programmierer implementiert werden müssen.

4. Ähnlich verhält es sich beim Heranführen der Programmierer an Refaktorierungen, die tiefgreifende Änderungen am Programm mit sich bringen. Da die Tests schrittweise implementiert werden, können bewusst Situationen geschaffen werden, in denen der nächste Test nicht ohne tiefgreifende Refaktorierungen implementiert werden kann.

5. Durch die Vorgabe der Tests kann der Umgang mit einem Testing-Framework gelernt werden. Da der Test zu Beginn fehlschlägt, sind die Programmierer gezwungen, sich die Ausgabe bzw. Rückmeldung anzuschauen, die beim Ausführen des Tests erzeugt wurde. Sie erfahren dadurch, wo sie mit der Implementierung fortfahren müssen. Dadurch lernen sie auch, wie Tests designt werden müssen, um bei einem Fehlschlagen des Tests präzise schließen zu können, aus welchem Grund der Test fehlschlägt und wo sie im Produktionscode ansetzen müssen.

Zusammengefasst lässt sich vermuten, dass sich vorgeschriebene Tests als Möglichkeit, die Herausforderungen beim Lernen von TDD und BDD zu lösen, eignen. Beim Einsatz von vorgeschriebenen Test werden mehrere der Herausforderungen adressiert und gelöst. In Kapitel 4 werden diese Vermutungen im Rahmen eines Experiments untersucht.

4 Experiment

Da sich in verwandten Studien gezeigt hat, dass sich kontrollierte Experimente eignen, TDD und BDD zu analysieren, wurde im Rahmen dieser Arbeit ein Experiment durchgeführt, mit dem untersucht werden sollte, in welchem Umfang das Bereitstellen von vorgeschriebenen Softwaretests beim Lernen und Anwenden von TDD und BDD hilft. Tabelle 2 zeigt die Zusammenfassung des Experiments.

Tabelle 2: *Zusammenfassung des Experiments*

Kontext	Beschreibung	Kapitel
Zielsetzung	Evaluation vorgeschriebener Tests als Einstieg in Softwaretesting	4.1
Material	Readme; Zwei Fragebögen; Code der Tests	4.2
Aufgabenstellung	Code schreiben, damit die Tests erfolgreich werden	4.3
Probanden	Acht Bachelor Studenten	4.4
Durchführung	An acht individuellen Terminen und unterschiedlichen Orten	4.5
Auswertung	Benötigte Zeit, erfolgreiche Tests und persönliche Einschätzung	4.6
Ergebnisse	Vorgeschriebene Tests als hilfreich empfunden; TDD besser geeignet als BDD	4.7

Die Probanden wurden in zwei Gruppen eingeteilt und erhielten entweder eine Sammlung von Tests im TDD-Stil oder im BDD-Stil und mussten mit deren Hilfe eine Programmieraufgabe lösen. Untersucht wurde, wie gut die Probanden die Aufgabe lösen konnten.

Die Ergebnisse zeigen, dass die Probanden die Tests (TDD und BDD) als hilfreiche Unterstützung beim Programmieren empfunden haben. Im Vergleich hat sich TDD als die bessere Variante gegenüber BDD herausgestellt.

4.1 Zielsetzung

Das Ziel des Experiments ist, zu evaluieren, in welchem Umfang das Bereitstellen von vorgeschriebenen Tests als Unterstützung beim Programmieren hilft. In Kapitel 3.1 wurden die mit TDD und BDD verbundenen Herausforderungen ausgearbeitet. In Kapitel 3.2 wurde aufgezeigt, wie vorgeschriebene Tests dabei helfen können, diese Herausforderungen zu lösen. Deshalb wird für das Experiment die folgende Fragestellung festgelegt:

Können vorgeschriebene Tests beim Lernen von TDD und BDD helfen?

Da sich der Stil der Tests bei BDD von dem bei TDD unterscheidet, werden für das Experiment außerdem die folgenden Hypothesen festgelegt:

H1 Mit TDD haben die Probanden nach Abschluss mehr erfolgreiche Tests als mit BDD.

H2 Mit TDD sind die Probanden schneller als mit BDD.

Es wird erwartet, dass sich TDD besser eignet als BDD, da es fokussiert auf den Einsatz von Programmierern ist. Die Vorteile von BDD gegenüber TDD sollten bei dem Experimentaufbau nicht zum Tragen kommen, da die Probanden im Experiment einzeln eine Programmieraufgabe lösen sollten. Die Vorteile von BDD kommen zum tragen, wenn ein Team an der Umsetzung eines kompletten Projekts arbeiten.

Die unabhängige Variable des Experiments ist daher der Stil der vorgeschriebenen Tests mit den zwei Ausprägungen TDD und BDD. Bei unabhängigen Variablen handelt es sich um Parameter des Experiments, die bewusst von den Verantwortlichen variiert werden, um das Resultat des Experiments zu beeinflussen (Juristo und Moreno [16], S. 60).

Die zwei untersuchten abhängigen Variablen sind

- die Reaktionszeit, also die Dauer des Programmierens während des Experiments
- und die Richtigkeit, also die Anzahl der Tests, die nach dem Abschluss des Programmierens erfolgreich sind.

Die abhängigen Variablen sind das Resultat des Experiments und sind abhängig von den Varianten der unabhängigen Variable (Juristo und Moreno [16], S. 59).

4.2 Material

Im Experiment wurden folgende Materialien benutzt:

1. Readme (in HTML Format)
2. Fragebogen zur Einschätzung der Programmiererfahrung
3. Fragebogen zur persönlichen Einschätzung des Experiments
4. Einleitende Programmieraufgabe
5. Code zur Durchführung der Programmieraufgabe

1. Die Readme verschaffte den Probanden einen Überblick über den Ablauf des Experiments. Sie enthielt eine Anleitung, in welcher Reihenfolge die Aufgaben des Experiments abzuarbeiten waren. Außerdem gab es eine Erklärung des Punktesystems beim Bowling, da ein grundlegendes Verständnis der Regeln für die Programmieraufgabe nötig war (deren Beschreibung folgt im weiteren Verlauf dieses Kapitels). Die Readme wurde zur Verfügung gestellt, da die Probanden das Experiment teilweise von zu Hause aus durchgeführt haben.

2. Ein Fragebogen zur Einschätzung der Programmiererfahrung wurde den Probanden gestellt, da die Erfahrung einer der Störfaktoren des Experiments ist.

3. Ein zweiter Fragebogen wurde genutzt, um die persönliche Einschätzung der Probanden zu erfahren, wie schwierig sie die Programmieraufgabe fanden und wie hilfreich die Tests als Unterstützung beim Programmieren gewesen sind. Außerdem wurde gemessen, wie erfahren die Probanden mit TDD und BDD waren und wie gut sie sich vorher schon mit dem Punktesystem beim Bowling auskannten. Dieser Fragebogen wurde den Probanden gestellt, um zu erfahren, ob die sie Tests als Hilfe beim Programmieren einschätzten. Da die Erfahrung mit TDD bzw. BDD und den Bowlingregeln die Schwierigkeit des Experiments für die Probanden beeinflussen kann, wurde auch diese ausgewertet.

Beide Fragebögen wurden mit dem an der Universität Magdeburg entwickeltem Programm *PROPHET* (http://wwwiti.cs.uni-magdeburg.de/ feigensp/prophet/) erstellt und von den Probanden beantwortet.

4. Bevor die Probanden mit der eigentlichen Programmieraufgabe angefangen haben, wurde in einem sehr kleinen Beispiel das Prinzip erklärt, wie beim Programmieren vorzugehen ist und wie die vorgeschriebenen Tests einzusetzen sind. In dieser Einleitung erhielten sie einen einzelnen Test, der als Rückgabewert einen String erwartete. Die Probanden mussten diesen String dann an einer Stelle im Produktionscode in einer Methode zurückgeben lassen. Dieses Beispiel erläuterte den Probanden das Prinzip, wie beim Programmieren der Aufgabe vorzugehen ist.

5. Die Probanden bekamen den Code für die Programmierung der Bowling Aufgabe. Er bestand aus den jeweils für die Tests benötigten Dateien und einem Grundgerüst des Produktionscodes (Auflistung 6). Dieses Grundgerüst enthielt lediglich eine Klasse mit leeren Methoden, die bereits im ersten Test instanziiert wurde. Diese Klasse wurde bereit gestellt, damit die Tests kompiliert werden konnten und die Probanden nicht direkt am Anfang Fehler im Code zu beheben hatten.

```
1  public class Game {
2     public void throwPins(Integer pins) {
3     }
4     public Object score() {
5        return null;
6     }
7     public Object scoreForFrame(int theFrame) {
8        return null;
9     }
10 }
```

Auflistung 6: *Grundgerüst des Produktionscodes, den die Teilnehmer zur Verfügung gestellt bekamen*

Zum Code gehörten außerdem noch die vorgeschriebenen Tests. In der TDD-Bedingung waren die Tests in einer einzelnen Datei (für einen Ausschnitt siehe Auflistung 7) enthalten.

Die Tests waren mit JUnit[1], einem Test Framework für Java, geschrieben.

```
1  public class TestGame {
2     private Game game;
3     @Before
4     public void setUp() {
5        game = new Game();
6     }
7     @Test
8     public void TwoThrowsNoMark() {
9        game.throwPins(5);
10       game.throwPins(4);
11       assertEquals(9, game.score());
12    }
13    @Test
14    public void FourThrowsNoMark() {
15       game.throwPins(5);
16       game.throwPins(4);
17       game.throwPins(7);
18       game.throwPins(2);
19       assertEquals(18, game.score());
20       assertEquals(9,  game.scoreForFrame(1));
21       assertEquals(18, game.scoreForFrame(2));
22    }
23    // ...
24 }
```

Auflistung 7: *Die vorgeschriebenen Tests für die Probanden der TDD Gruppe. Hier sind die ersten beiden Tests aufgelistet, die zu implementieren waren. Insgesamt enthielt die Klasse 10 Tests.*

Für die BDD-Bedingung wurden die Tests mit Cucumber[2] geschrieben. Cucumber wurde gewählt, da es die "Given-When-Then" Syntax unterstützt. Der Code für die Tests war auf drei Dateien aufgeteilt.

In einer Datei (Auflistung 8) wurden die Szenarien, also die Testfälle, festgelegt, die von der BDD Gruppe implementiert werden sollten. Die einzelnen Szenarien deckten sich dabei mit den TDD Testfällen. Die Probanden mussten in beiden Gruppen also die gleichen Anforderungen an die Bowling Punkteberechnung im Produktionscode implementieren.

4.3 Aufgabenstellung

Die einzelnen Schritte, die in den Szenarien festgelegt wurden, werden beim Ausführen der Tests auf Methoden in der BowlingTestDefs.java Auflistung 9 gemappt. Die Probanden der BDD Gruppe konnten sich also mit der Feature-Datei (Auflistung 8) einen Überblick über den Test verschaffen, den sie gerade implementieren sollten. Um herauszufinden, welche Methoden des Produktionscodes sie zu implementieren hatten, mussten sie nachvollziehen, welcher Schritt des Szenarios auf welche Methode gemappt wurde, da in dieser Methode der Produktionscode aufgerufen wurde. Beispielsweise wird der Schritt `Given a player starts a new bowling game` auf die Methode `a_player_starts_a_new_bowling_game()`

[1] http://junit.org/
[2] http://cukes.info/install-cucumber-jvm.html

```
Feature: Bowling Game
  Background:
    Given a player starts a new bowling game
  Scenario: Two simple throws
    When he throws the following pins
      | pins |
      | 5    |
      | 4    |
    Then his total score should be 9
  Scenario: Four simple throws
    When he throws the following pins
      | pins |
      | 5    |
      | 4    |
      | 7    |
      | 2    |
    Then his total score should be 18
    And the score for frame 1 should be 9
    And the score for frame 2 should be 18
```

Auflistung 8: *Die vorgeschriebenen Tests für die Probanden der BDD Gruppe. Diese deckten sich inhaltlich mit den Tests der BDD Gruppe*

gemappt, in der dann die `Game` Klasse, die für die Berechnung der Punkte zuständig ist, instanziiert wird.

```
public class BowlingTestDefs {
  private Game game;
  @Given("^a player starts a new bowling game$")
  public void a_player_starts_a_new_bowling_game() throws Throwable {
    game = new Game();
  }
  @When("^he throws the following pins$")
  public void he_throws_the_following_pins(List<playerThrow> playerThrows) throws
      Throwable {
    for(playerThrow aThrow : playerThrows) {
      game.throwPins(aThrow.pins);
    }
  }
  @Then("^his total score should be (\\d+)$")
  public void his_score_should_be(int expectedScore) throws Throwable {
    assertEquals(expectedScore, game.score());
  }
```

Auflistung 9: *Die Methoden, auf die einzelne Schritte der Szenarios gemappt werden. Das Mapping erfolgt dabei mithilfe von regulären Ausdrücken.*

Die dritte Datei war nötig, um die Szenarien ausführen zu können. Diese Datei wurde den Probanden komplett bereit gestellt und musste auch nicht bearbeitet werden.

Da in beiden Gruppen ein Test nach dem anderen implementiert werden sollte, waren am Anfang in der Datei, die die TDD Tests enthielt, alle Tests bis auf den Ersten auskommentiert. Die Probanden mussten dann, wenn sie mit einem Test fertig waren, lediglich dafür sorgen, dass der nächste Test nicht mehr auskommentiert war. Bei den BDD Szenarien wurde ebenso vorgegangen, allerdings mussten die Probanden hier nur das `@Skip` Tag entfernen, das bei allen Szenarien ab dem zweiten stand und dafür sorgte, dass diese Tests beim Ausführen ignoriert werden. Das Miteinbeziehen der weiteren Tests im Ver-

lauf des Programmierens war auch die einzige Änderung, die die Probanden am Testcode durchzuführen hatten. Abgesehen davon wurde er ihnen komplett fertig zur Verfügung gestellt.

Die Tests waren so aufgebaut, dass es zu Beginn ausreichte, einfach nur die geworfenen Pins in einer Instanzvariable zu speichern und beim Aufruf der `score()` Methode zurück zu geben. Im weiteren Verlauf waren erst Spares (Umwerfen aller 10 Pins in zwei Würfen) und dann Strikes (Umwerfen aller 10 Pins in einem Wurf) von den Probanden zu implementieren. Es wurde außerdem in den Tests gefordert, nicht nur die Gesamtpunkte, sondern auch die Punkte eines einzelnen Frames (entspricht einem Durchgang beim Bowling, also zwei Würfe) zu berechnen. Die Implementierung dieser Anforderungen stellte die größte Herausforderung für die Probanden dar.

Jedes mal, wenn die Probanden mit einem der Tests fertig waren, sollten sie mit einem Versionsverwaltungstool (im Experiment wurde dazu *git*[3] verwendet, aufgrund seiner Bedienfreundlichkeit) einen Speicherpunkt ihrer Dateien anlegen. Dies hatte zwei Gründe. Zum einen konnten die Probanden ihren schon geschriebenen Code bedenkenlos überarbeiten, falls das für die Implementierung des nächsten Tests erforderlich sein sollte. Sollten sie dabei an einen Punkt gelangen, an dem sie nicht mehr weiter wussten, konnten sie einfach zum letzten Speicherpunkt zurück wechseln, an dem noch alle Tests erfolgreich waren und von vorne mit der Implementierung des nächsten Tests beginnen. Außerdem ermöglichte der Einsatz eines Versionierungstools auch die Auswertung, in welchem Umfang die Probanden ihren Code von Test zu Test refaktoriert hatten.

Beiden Gruppen wurden insgesamt 10 Tests gegeben, die von ihnen implementiert werden sollten. Die Tests waren dabei so konzipiert, dass sie aufeinander aufbauten. So wurde bevor die Berechnung eines Strikes gefordert wurde erst die Einteilung in die einzelnen Frames implementiert. Die Reihenfolge der Tests wurde so gewählt, dass die Probanden zumindest teilweise ihren bisher geschriebenen Code zu refaktorieren hatten, um den nächsten Test zu implementieren. Das wurde z.B. erreicht, indem es zu Beginn ausreichte, die Punkte der geworfenen Pins einfach nur zu addieren, dies aber später bei der Einführung der Frames nicht mehr ausreichte.

Teilweise musste in den Tests kein neues Feature implementiert werden. Diese Tests sollten hingegen aufdecken, wenn die Probanden in ihrem Code Fehler hatten, diese aber noch nicht aufgefallen waren. In Auflistung 10 könnte eine fehlerhafte Implementierung der Berechnung eines Spares die beiden Fünfer-Würfe als Spare betrachten und dann fälschlicherweise zwei Punkte zu viel zurück geben.

Es war weiterhin möglich, dass die Probanden eine Implementierung schreiben konnten, bei der die letzten zwei bis drei Tests direkt erfolgreich waren, ohne dass Änderungen am

[3] http://git-scm.com/

```
1  @Test
2  public void twoFivesThatDontMakeASpare() {
3    game.throwPins(3);
4    game.throwPins(5);
5    game.throwPins(5);
6    game.throwPins(2);
7    assertEquals(8, game.scoreForFrame(1));
8    assertEquals(15, game.scoreForFrame(2));
9    assertEquals(15, game.score());
10 }
```

Auflistung 10: *Der zweite und dritte Wurf ergeben zusammengenommen 10 Punkte. Da sie in zwei verschiedenen Frames geworfen wurden, bilden sie aber keinen Spare. Dieser Test sollte aufdecken, falls die Probanden einen Fehler in ihrer Implementierung hatten.*

Produktionscode nötig gewesen wären.

4.4 Probanden

An dem Experiment haben insgesamt acht männliche Studenten aus verschiedenen Universitäten teilgenommen. Von diesen acht Probanden wurden fünf aus dem persönlichen Umfeld des Experimentleiters rekrutiert. Die restlichen drei meldeten sich freiwillig auf öffentlich ausgehängte Ausschreibungen an der Universität Magdeburg. Die Probanden waren zwischen 20 und 25 Jahre alt. Unter allen Probanden wurden als Kompensation für die Teilnahme zwei Amazon Gutscheine über €20 verlost. Alle Probanden hatten Erfahrung im Programmieren, jedoch waren nur vier davon in Informatikstudiengängen eingeschrieben. Bei der Auswertung der Fragebögen stellte sich heraus, dass alle Probanden bis vor dem Experiment nur sehr wenig Erfahrung mit TDD und BDD gehabt haben.

Eingesetzt wurde ein *Between-Subjects* Design. Eine Gruppe arbeitete mit TDD, während die andere Gruppe mit BDD arbeitete. Ein *Within-Subject* Design wäre im Rahmen dieser Arbeit nicht durchführbar gewesen, da der Zeitaufwand des Experiments zu hoch gewesen wäre. Außerdem hätten die Probanden im ersten Durchgang zu viel über das Problem gelernt und auch schon dessen Implementierung geschrieben, sodass eine sinnvolle Interpretation der Ergebnisse nicht möglich gewesen wäre.

Die Probanden wurden pseudorandomisiert auf die beiden Gruppen aufgeteilt. Die Probanden wurden zu Beginn zufällig in eine der beiden Gruppen eingeteilt. Da jeder Proband einzeln an einem Termin teilnahm und beide Gruppen sowohl von der Größe als auch der Programmiererfahrung gleich gehalten werden sollte, wurden sie später gezielt in eine der beiden Gruppen eingeteilt.

4.5 Durchführung

Das Experiment wurde an acht Terminen im Zeitraum Juli 2013 - August 2013 durchgeführt. Mit jedem Probanden wurde ein individueller Termin vereinbart. Den Probanden wurde angeboten, das Experiment von zu Hause aus durchzuführen, wenn sie dort über die benötigten technischen Mittel verfügten. Vorausgesetzt wurde, dass die Probanden bereits vor dem Experiment die benötigte Entwicklungsumgebung installierten. Sieben Probanden nahmen dieses Angebot in Anspruch und mit einem Probanden wurde das Experiment in der Universität Magdeburg durchgeführt.

Alle Probanden wurden bei der Durchführung direkt vom Experimentator betreut. Mit den Probanden, die von zu Hause aus am Experiment teilnahmen, wurde per Skype[4] kommuniziert und dem Experimentator Zugriff auf den Computer gewährt. Die Probanden erhielten eine kurze Einführung in den Ablauf des Experiments und lasen sich die Readme durch. Danach beantworteten sie Fragebogen #1. Nachdem die einleitende Programmieraufgabe gelöst war, bearbeiteten die Probanden die eigentliche Programmieraufgabe. Zum Abschluss des Experiments wurde Fragebogen #2 beantwortet und alle benötigten Dateien dem Experimentator zugesendet.

Zur Bearbeitung der Programmieraufgabe wurde den Probanden eine zeitliche Begrenzung von 90 Minuten gesetzt, da die insgesamt für das Experiment benötigte Zeit 120 Minuten nicht übersteigen sollte. Sollten sie nach den 90 Minuten noch nicht mit allen Tests fertig geworden sein, hörten sie an diesem Punkt einfach auf, beantworteten noch den zweiten Fragebogen und waren dann mit dem Experiment fertig.

Die einzige aufgetretene Abweichung ist, dass bei einem Probanden während der Durchführung die Internetverbindung ausgefallen ist und er deshalb nicht mehr während des Experiments betreut werden konnte.

4.6 Auswertung

In H1 wurde die Annahme getroffen, dass die Probanden der TDD Gruppe mehr erfolgreiche Tests als die Probanden der BDD Gruppe haben werden. In Tabelle 3 ist zu sehen, wie viele Tests jeweils erfolgreich gewesen sind. In der TDD Gruppe waren bei allen vier Probanden alle Tests erfolgreich. Dagegen war die höchste Anzahl an erfolgreichen Tests in der BDD Gruppe nur sieben. Mit dem exakten Test nach Fisher (Bortz und Schuster [6], S. 160)) wurde untersucht, ob es sich um einen signifikanten Unterschied in der Anzahl der erfolgreichen Tests handelt. Der Test ergab einen signifikanten Unterschied bei einem p-Wert von 0,02857.

[4] http://www.skype.com/de/

Tabelle 3: *Anzahl erfolgreicher Tests je Gruppe*

Anzahl erfolgreicher Tests	Anzahl Probanden TDD	Anzahl Probanden BDD
10	4	0
9	0	0
8	0	0
7	0	1
6	0	2
5	0	1
1-4	0	0

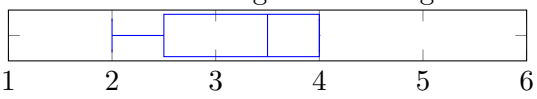
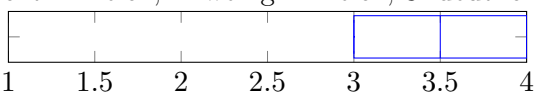
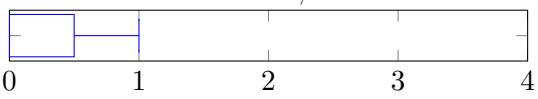
In H2 wurde die Annahme getroffen, dass die Probanden mit TDD schneller sein werden als mit BDD. Nicht alle Probanden haben innerhalb der ihnen zur Verfügung gestellten Zeit die gleiche Anzahl an Tests geschafft. Zur Untersuchung der benötigten Zeit fließen deshalb nur die Tests ein, die mindestens bei allen Probanden erfolgreich gewesen sind. Diese Tests waren die ersten Fünf. In Tabelle 4 ist aufgelistet, wie lange die Probanden jeweils gebraucht haben, bis die ersten fünf Tests erfolgreich gewesen sind. Zur Prüfung, ob die beobachteten Unterschiede in der Zeit signifikant sind, wurde der Mann-Whitney-U Test (Bortz und Schuster [6], S. 130) eingesetzt. Mit einem p-Wert von 0,4651 zeigt der Tests, dass kein signifikanter Unterschied vorliegt.

Tabelle 4: *Benötigte Zeit, bis die ersten fünf Tests implementiert waren*

Gruppe	Benötigte Zeit (in Minuten)	Rang
BDD	38	1,5
TDD	38	1,5
TDD	39	3
BDD	55	4
TDD	65	5,5
TDD	65	5,5
BDD	70	7
BDD	81	8

Um die Frage zu untersuchen, ob vorgegebene Tests beim Lernen von TDD und BDD helfen können, wurden die Antworten des zweiten Fragebogens ausgewertet. Die Antworten der Probanden sind in Tabelle 5 zu sehen. Der Mittelwert der Erfahrung der Probanden sowohl mit TDD und BDD ist 0. Ein Wert von 0 stand bei dieser Frage für “sehr unerfahren” und der höchste Wert war 4 (“sehr erfahren”). Die Schwierigkeit, mit der die Probanden die Aufgabe einstuften, wurde auf einer Skala von 0 (“sehr einfach”) bis 6 (“sehr schwierig”) gemessen. Der Mittelwert der Antworten liegt bei 3,5. Die Frage, wie hilfreich die Tests beim Programmieren gewesen sind, konnte mit 0 (“nicht hilfreich”) bis 4 (“sehr hilfreich”) beantwortet werden. Der Mittelwert der Antworten ist 3,5. Keiner der Probanden beantwortete diese Frage mit den Werten 0 oder 1, sondern alle wählten mindestens 3, also “deutlich hilfreich”, als Antwort.

Tabelle 5: *Antworten der Probanden im zweiten Fragebogen*

Frage:	Auf einer Skala von 1-6, wobei 1 sehr einfach, 6 sehr schwierig bedeutet: Wie stufst du die Schwierigkeit der Programmieraufgabe im Experiment ein?
Box Plot:	
Frage:	Wie hilfreich fandest du die Tests als Unterstützung beim Programmieren? 1: nicht hilfreich; 2: wenig hilfreich; 3: deutlich hilfreich; 4: sehr hilfreich
Box Plot:	
Frage:	Wie erfahren warst du vor dem Experiment mit Test-Driven Development? 0: sehr unerfahren; 4: sehr erfahren
Box Plot:	
Frage:	Wie erfahren warst du vor dem Experiment mit Behavior-Driven Development? 0: sehr unerfahren; 4: sehr erfahren
Box Plot:	

4.7 Diskussion

Können vorgegebene Tests beim Lernen von TDD und BDD helfen? Mit den Ergebnissen lässt sich diese Frage nicht eindeutig beantworten. Die Ergebnisse deuten aber darauf hin, dass es so ist. Alle Probanden empfanden die Tests als mindestens “deutlich hilfreich”. Keiner der Probanden stufte die Aufgabe als sehr einfach ein, sodass die Tests als Unterstützung beim Programmieren sinnvoll gewesen sind. Außerdem hat jeder Proband das Prinzip, den Code schreiben zu müssen, damit der jeweilige Test erfolgreich wird, im Durchschnitt nach knapp 15 Minuten (die durchschnittliche Zeit, nach der Test 1 erfolgreich war) verstanden. Berücksichtigt man ferner, dass alle Probanden sehr unerfahren mit TDD und BDD waren, verstärkt sich die Annahme, dass sich der Ansatz als Einstieg in die “test-first” Herangehensweise eignet.

Um die Frage eindeutig beantworten zu können, sind weitere Untersuchungen auf diesem Gebiet notwendig. Vorgegebene Tests sollen einen Einstieg in die “test-first” Welt bieten. In einem echten Softwareprojekt müssen die Programmierer ihre Tests natürlich selbst schreiben und bekommen diese nicht fertig vorgegeben, so wie im Experiment.

Bei der Frage, ob die Tests mit TDD oder BDD geschrieben werden sollten, lässt sich mit den vorliegenden Ergebnissen sagen, dass sich TDD besser zu eignen scheint. Die Probanden haben mit TDD signifikant mehr erfolgreiche Test gehabt, als bei BDD. Der Unterschied in der benötigten Zeit ist zwar nicht signifikant, aber im Durchschnitt liegt auch hier TDD vorne, mit 52 Minuten gegen 61 Minuten bei BDD.

4.8 Validitätsgefährdung

Im folgenden Kapitel werden die Gefahren beschrieben, die die Validität der Ergebnisse des Experiments beeinflussen. Dabei wird zwischen den Gefahren für die interne und die externe Validität unterschieden. Die *interne Validität* beschreibt, inwieweit der Wert der abhängigen Variable den Variationen der unabhängigen Variable zugeschrieben werden kann (Shadish u.a. [27], S. 53). Sie gibt damit an, in welchem Ausmaß das Ergebnis der abhängigen Variable durch die unabhängige Variable ausgelöst wurde. Um die interne Validität sicherzustellen, müssen Störvariablen kontrolliert werden, die sonst verhindern würden, dass das Ergebnis einzig der unabhängigen Variable zugeschrieben werden kann. Die *externe Validität* gibt an, inwieweit die Ergebnisse und Schlussfolgerungen eines Experiments auf andere Probanden und einen anderen Aufbau generalisiert werden können (Shadish u.a. [27], S. 21). Je realitätsnäher ein Experiment aufgebaut ist, desto höher ist dessen externe Validität.

4.8.1 Interne Validität

Bei der Durchführung des Experiments fiel bei einem Probanden die Internetverbindung aus. Dadurch hatte er im Anschluss daran nicht mehr die Möglichkeit, Fragen bzgl. des Experiments zu stellen. Allerdings war das Experiment zu diesem Zeitpunkt schon so weit, dass die eigentliche Programmieraufgabe zu bearbeiten war und es waren schon die ersten Tests implementiert. Dadurch konnte der Proband die Aufgabe trotzdem fortsetzen. Außerdem konnte er in der ihm zur Verfügung gestellte Readme (vgl. 4.2) Datei sehen, welche Schritte des Experiments er noch zu erledigen hatte. Deshalb konnte er das Experiment erfolgreich beenden und seine Ergebnisse wurden in der Auswertung berücksichtigt.

Da die geringe Anzahl an Probanden eine Gefahr für die interne Validität darstellen, wurden angepasste Signifikanztests zur Untersuchung der Hypothesen genutzt.

4.8.2 Externe Validität

Alle Probanden des Experiments sind zum Zeitpunkt der Teilnahme Studierende gewesen. Die Ergebnisse des Experiments sind daher nur in diesem Kontext valide. Für eine Verallgemeinerung der Ergebnisse auf professionelle Programmierer wäre es notwendig, das Experiment mit diesen durchzuführen. Dies war im Rahmen dieser Arbeit nicht möglich. Deshalb wurden im Experiment Studierende eingesetzt, die als Kompensation für ihre Teilnahme an einer Verlosung von Gutscheinen beteiligt wurden. Der Einsatz von Studierenden ist problematisch, da sie oft nicht die Erfahrung und das Wissen von professionellen Programmierern haben (Hanenberg [14]). Da es aber oft die einzige Möglichkeit

ist, Probanden zu rekrutieren, werden bei vielen Experimenten im Bereich der Softwareentwicklung Studierende eingesetzt. Deshalb sind auch die Ergebnisse dieser Arbeit in diesem Kontext sinnvoll. Da außerdem der Aufbau des Experiments wiederverwendbar ist, kann in weiteren Arbeiten das Experiment auch mit professionellen Programmierern durchgeführt werden.

Eine weitere Einschränkung ist der Umfang des im Experiment eingesetzten Quellcodes. Insgesamt bestanden die Tests zusammen mit der Implementierung der Probanden aus ca. 250 Zeilen Code. Es ist daher fraglich, inwieweit sich die Ergebnisse des Experiments auf größere Programme verallgemeinern lassen. Allerdings wächst die Komplexität der Tests nicht unbedingt proportional mit dem Umfang des Produktionscodes. Auch bei größeren Programmen werden in Tests einzelne Module möglichst unabhängig von anderen Einflüssen getestet. So können immer noch sinnvolle Tests geschrieben werden, die trotzdem einen geringen Umfang haben, daher ist diese Bedrohung minimiert.

5 Abschluss

Um bei agiler Softwareentwicklung in iterativen und regelmäßigen Schritten neue Features entwickeln zu können, muss sichergestellt werden, dass die Software nachhaltig entwickelt wird. Dadurch wird sichergestellt, dass die Entwicklungsgeschwindigkeit auf einem konstanten Level gehalten wird und neue Features regelmäßig fertiggestellt werden. Zur Nachhaltigkeit gehört die Erstellung und Wartung einer ausführlichen Sammlung von Softwaretests. Diese werden dabei parallel zur Entwicklung der eigentlichen Software geschrieben. Es bietet sich dabei an, "test-first" zu entwickeln. Erst die Tests zu schreiben, bevor danach der getestete Code folgt, führt zu einem Anstieg des Vertrauens in die Tests und hilft beim Design der Software.

In dieser Arbeit wurde untersucht, ob vorgegebene Tests beim Einstieg in das "test-first" Programmieren helfen können. Speziell wurden dabei die Ansätze TDD und BDD verglichen. Diese Untersuchungen wurden aus der Tatsache abgeleitet, dass Programmierer, die zum ersten mal mit "test-first" in Kontakt kommen, vor Probleme gestellt werden können.

Zur Untersuchung der Effekte von vorgegebenen Tests wurde ein kontrolliertes Experiment durchgeführt. Dabei hat sich gezeigt, dass sich vorgegebene Tests als Einstieg in "test-first" zu eignen scheinen und sie von den Probanden als hilfreiche Unterstützung beim Programmieren empfunden worden sind. Im Vergleich zwischen TDD und BDD hat dabei TDD etwas besser abgeschnitten. Zu den möglichen Gründen dafür gehört, dass TDD eine Methode ist, die gezielt für Programmierer entwickelt wurde. Die Vorteile von BDD können erst zum Tragen kommen, wenn in einem Team, welches nicht nur aus Programmierern besteht, an einem Projekt gearbeitet wird.

Als weitere Arbeit bleibt, Programmierer zu untersuchen, die zum ersten mal selbst die Tests schreiben müssen. Dabei könnte untersucht werden, wie Programmierer abschneiden, die bereits mit vorgegebenen Tests gearbeitet haben, gegenüber Programmierern, die direkt ihre eigenen Tests schreiben müssen. Im Fall, dass vorgegebene Tests einen sinnvollen Einstieg in "test-first" bieten, sollten diese Programmierer leichter selbst ihre Tests schreiben und damit arbeiten können.

Literatur

[1] David Astels. *Test-Driven-Development: A Practical Guide.* Prentice Hall, 2003.

[2] Kent Beck. *Test-Driven-Development: By Example.* Addison-Wesley Professional, 2002.

[3] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change, 2nd Edition.* Addison-Wesley, 2004.

[4] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. 2001.

[5] Andre Häusling Boris Gloger. *Erfolgreich mit Scrum – Einflussfaktor Personalmanagement.* Carl Hanser Verlag München, 2011.

[6] Jürgen Bortz and Christof Schuster. *Statistik für Human- und Sozialwissenschaftler. Lehrbuch mit Online-Materialien* -. Springer DE, Berlin, 2010.

[7] Kevin Buffardi and Stephen H. Edwards. Exploring influences on student adherence to test-driven development. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, ITiCSE '12, pages 105–110, New York, NY, USA, 2012. ACM.

[8] David Chelimsky, Dave Astels, Bryan Helmkamp, Dan North, Zach Dennis, and Aslak Hellesoy. *The RSpec Book - Behaviour-Driven Development with RSpec, Cucumber, and Friends.* Pragmatic Programmers, 2010.

[9] Eric J Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2003.

[10] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman, 1999.

[11] Boby George and Laurie Williams. An initial investigation of test driven development in industry. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 1135 – 1139. ACM, 2003.

[12] A. Gupta and P. Jalote. An experimental evaluation of the effectiveness and efficiency

of the test driven development. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 285 –294, 2007.

[13] Paul Hamill. *Unit Test Frameworks*. O'Reilly Media, 2004.

[14] Stefan Hanenberg. An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. *SIGPLAN Not.*, 45(10):22–35, October 2010.

[15] David S. Janzen and Hossein Saiedian. A leveled examination of test-driven development acceptance. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 719–722, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Natalia Juristo and Ana M. Moreno. *Basics of Software Engineering Experimentation*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[17] N. Kabbani and C. Kaner. Experience in teaching test-driven development course. 2010.

[18] Adam Kolawa and Dorota Huizinga. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007.

[19] Amir Kolsky and Scott Bain. Sustainable Test-Driven Development. `http://www.sustainabletdd.com`, 2012. [Online; accessed 30-April-2013].

[20] Robert C. Martin. *Agile Software Development - Principles, Patterns, and Practices*. Alan Apt Series, 2003.

[21] Robert C. Martin. Professionalism and test-driven development. 2007.

[22] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, 2008.

[23] Grigori Melnik and Frank Maurer. A cross-program investigation of students' perceptions of agile methods. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *ICSE*, pages 481–488. ACM, 2005.

[24] Rick Mugridge. Challenges in teaching test driven development. volume 2675 of *Lecture Notes in Computer Science*, pages 410 – 413. Springer, 2003.

[25] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley and Sons, 2011.

[26] Dan North. Introducing bdd. *Better Software*, 2006.

[27] William R. Shadish, Thomas D. Cook, and Donald T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference.* Houghton Mifflin, 2001.

[28] James Shore and Shane Warden. *The Art of Agile Development.* OReilly Media, Inc., 2008.

[29] Jaime Spacco and William Pugh. Helping students appreciate test-driven development (tdd). In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 907–913, New York, NY, USA, 2006. ACM.

[30] Matt Wynne and Aslak Hellesoy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers.* Pragmatic Programmers, 2012.