hochschule mannheim

Re-Implementierung der Trainer Cloud App mittels testgetriebener Entwicklung

Josua Geiger

Bachelor-Thesis

zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)

Studiengang Informatik

Fakultät für Informatik

Hochschule Mannheim

29.06.2017

Betreuer

Prof. Dr. Peter Knauber, Hochschule Mannheim

Dr. Oliver Theissen, SAP SE

Geiger, Josua:
Re-Implementierung der Trainer Cloud App mittels testgetriebener Entwicklung / Josua Geiger. –
Bachelor-Thesis, Mannheim: Hochschule Mannheim, 2017. 67 Seiten.

Geiger, Josua:
Test driven re-implementation of the Trainer Cloud App / Josua Geiger. –
Bachelor Thesis, Mannheim: University of Applied Sciences Mannheim, 2017. 67 pages.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Mannheim öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Mannheim, 29.06.2017

Josua Geiger

Abstract

Re-Implementierung der Trainer Cloud App mittels testgetriebener Entwicklung

Test Driven Development existiert seit Jahrzehnten als Vorgehensweise zur Softwareentwicklung. Die Vorgehensweise besagt, dass zuerst ein automatisierter Testfall für das Inkrement verfasst werden muss, bevor mit der Implementierung begonnen werden kann. In dieser Arbeit wird zunächst ein Konzept zur testgetriebenen Entwicklung von OpenUI5-Anwendungen aufgezeigt. Darauf aufbauend wird Test Driven Development als Methodik anhand einer Re-Implementierung der Trainer Cloud App bewertet. Durch die Evaluation der Re-Implementierung konnte gezeigt werden, dass sich erhöhter Aufwand aufgrund von Test Driven Development im Bezug auf Code-Qualität, Überzeugung von funktionierender Software und risikofreiem Refactoring bezahlt macht. Im Vergleich zur alten Anwendung konnten die Code-Zeilen sowie die zyklomatische Komplexität des Projekts um circa zwei Drittel gesenkt werden.

Test driven re-implementation of the Trainer Cloud App

Test Driven Development has been around for decades as a software development approach in which tests are written before the software increment is implemented. This thesis illustrates a concept for Test Driven Development of OpenUI5 applications. Furthermore, an evaluation is conducted based on the re-implementation of the Trainer Cloud App. In this thesis it is shown that the increased effort resulted from Test Driven Development pays off in regards of improvement in code quality, reinsured confidence in working software and risk-free refactoring. Compared to the old OpenUI5 application, the code lines and cyclomatic complexity of the project are decreased by two thirds.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Man muss alsdann billig bedenken, dass derjenige, der sich auf einer gewissen Höhe behaupten will, ebenso den Umständen nachgeben muss, wie der Hahn auf dem Kirchturm, den, obgleich er von Eisen ist, jeder Sturmwind zerbrechen und herabschleudern würde, wenn er trotzig unbeweglich bliebe und nicht die edle Kunst verstände sich nach jedem Winde zu drehen.

-Heinrich Heine, Reisebilder Kapitel 115

Immer mehr Entwicklungsteams geben starre Softwareentwicklungsprozesse auf und adaptieren agiles Vorgehen [1]. Erste Ideen hierzu existieren seit den Neunzigern. Große Berühmtheit erlangte die agile Softwareentwicklung mit der Veröffentlichung des agilen Manifests 2001 [2]. Die agile Softwareentwicklung besteht aus vielen Prinzipien, Mustern und Methodiken [3]. Eine dieser Methodiken ist die testgetriebene Entwicklung. Laut Kent Beck, dem Wiederentdecker von Test Driven Development (TDD), Co-Autor des agilen Manifests und Erfinder [4] von Extreme Programming (XP), ist TDD das Mittel der Wahl, wenn es um eine Vorgehensweise zur Programmierung in agilen Softwareprojekten geht [5].

Im Zuge der agilen Transformation innerhalb der SAP SE organisieren sich viele Abteilungen um. Sie reduzieren ihre Systemlandschaft, führen Scrum als Vorgehensmodell für das Projektmanagement ein und rücken den Endanwender mehr in den Mittelpunkt. Aufgrund dieser Transformation erfährt die testgetriebene Entwicklung immer mehr Aufmerksamkeit innerhalb der Abteilungen. Derzeit gibt es kein Erfolgsrezept für die erfolgreiche agile Transformation einer Abteilung [6]. Deshalb muss jede Abteilung einen Großteil des Wandels selbst bewältigen und evaluieren, welche Methodiken zum Arbeitsumfeld der Abteilung passen.

Die von der SAP SE verwendeten Technologien zur Implementierung von User Interfaces (UIs) haben sich im Laufe der Jahrzehnte stetig gewandelt. Laut aktueller Produkt-Road Map [7] gilt das Framework SAPUI5, beziehungsweise die Open Source Variante OpenUI5, derzeit als Schlüsseltechnologie zur Implementierung grafischer Oberflächen. Die Architektur einer OpenUI5-Anwendung basiert auf dem Muster Model-View-Controller (MVC) [8]. Informationen zu TDD im Kontext von OpenUI5 sind rar und fokussieren sich nur auf die testgetriebene Entwicklung der Controller.

Diese Arbeit beschäftigt sich mit der Fragestellung, wie OpenUI5-Anwendungen testgetrieben entwickelt werden können und ob die durch TDD geweckten Erwartungen in diesem Kontext eintreffen. Eine detaillierte Beschreibung der Aufgabe ist im Kapitel *1.2. Ziele und Vorgehen* vorzufinden.

1.1. Vorstellung des Unternehmens und der Abteilung

Die SAP SE ist ein Softwarehersteller mit Sitz im baden-württembergischen Walldorf. Tätigkeitsschwerpunkt ist die Entwicklung von Software zur Abwicklung sämtlicher Geschäftsprozesse eines Unternehmens.

Diese Arbeit wird im Umfeld der IT-Abteilung Go-To-Market Solutions Education and Services verfasst. Unter anderem verantwortet die Abteilung die Software, mit der Schulungen der SAP SE organisiert und Trainingsmaterialien zu SAP-Produkten verkauft wird.

1.2. Ziele und Vorgehen

In diesem Kapitel werden die Ziele dieser Arbeit benannt. Zudem wird das Vorgehen zum Erreichen der Ziele erläutert.

Ziel 1: Benennung der Erwartungen aus der Literatur hinsichtlich TDD. Damit die Erwartungen benannt werden können, muss zunächst eine Einarbeitung in die testgetriebene Entwicklung geschehen. Darauf aufbauend sollen Erwartungen an das Paradigma formuliert werden.

Ziel 2: Realisierung einer Entwicklungsumgebung zur testgetriebenen Entwicklung. Zur Realisierung der Entwicklungsumgebung müssen Werkzeuge be-

gründet ausgesucht und komponiert werden. Durch die Zusammenstellung der Werkzeuge soll es möglich sein, automatisierte UI- und Unit-Tests für OpenUI5-Anwendungen zu schreiben.

Ziel 3: Konzeptionierung einer Teststrategie von OpenUI5-Anwendungen. Eine OpenUI5-Anwendungen besteht hauptsächlich aus den Architekturbausteinen Model, View und Controller. Aufgrund dessen soll ebenso eine Einarbeitung in das MVC-Architekturmuster stattfinden. Nach der Einarbeitung kann eine Teststrategie definiert werden. Durch die Definition der Teststrategie soll aufgezeigt werden, ob und inwiefern es notwendig ist, jede dieser Komponenten zu testen.

Ziel 4: Evaluierung der Erwartungen anhand eigener Beobachtungen während der Re-Implementierung, eines Vergleichs zwischen alter und neuer Lösung anhand von Softwaremetriken (Code-Zeilen, zyklomatische Komplexität) und einer Fallstudie. Damit die Erwartungen evaluiert werden können, muss zuvor eine testgetriebene Re-Implementierung der Trainer Cloud App stattfinden. Die Re-Implementierung muss in der bereits realisierten Entwicklungsumgebung und anhand der definierten Teststrategie durchgeführt werden. Nach der Re-Implemen-tierung sollen die Analysemethoden zur Bewertung der Erwartungen definiert werden. Die drei verschiedenen Analysemethoden (eigene Beobachtungen, Softwaremetriken, Fallstudie) zielen darauf ab, möglichst viele Erwartungen evaluieren zu können.

1.3. Aufbau der Arbeit

Ein Überblick über den Aufbau soll helfen, sich innerhalb der Arbeit zurecht zu finden.

Im Kapitel *2. Grundlagen* wird ein Grundverständnis von TDD, dem MVC-Muster, einer OpenUI5-Anwendung und der Trainer Cloud App geschaffen. Dieses Grundverständnis hilft, nachfolgende Kapitel zu verstehen. Die aus der Theorie zu TDD hervorgehenden Erwartungen werden im Kapitel *3. Erwartungen an Test Driven Development* formuliert. Die formulierten Erwartungen werden zur Evaluation von TDD herangezogen. Das Kapitel *4. Re-Implementierung der bisherigen Trainer Cloud App* beschreibt das Konzept der Entwicklungsumgebung zur testgetriebenen Entwicklung. Zudem wird eine Teststrategie aufgezeigt, anhand der TDD im Bezug auf eine OpenUI5-Anwendung und dessen MVC-Hauptbestandteile gelebt werden

kann. Abschließend werden in Kapitel *5. Analyse der Re-Implementierung* die verwendeten Bewertungsmethoden definiert, um daraufhin in Kapitel *6. Auswertung der Analyseergebnisse in Bezug auf die Erwartungen* eine Evaluation durchführen zu können.

Kapitel 2

Grundlagen

Im Grundlagenkapitel wird das Fundament für Verständnis und Nachvollziehbarkeit dieser Arbeit gelegt. Zuallererst wird die testgetriebene Entwicklung erläutert, um im Anschluss an dieses Kapitel Erwartungen formulieren zu können. Das Architekturmuster MVC ist essenzieller Bestandteil von OpenUI5-Anwendungen. Zudem hat die Realisierung des Musters innerhalb des Frameworks Einfluss auf die Teststrategie. Folglich ist es notwendig, MVC und OpenUI5 zu behandeln. Abschließend wird eine funktionale Übersicht über die Trainer Cloud App gegeben und erläutert, in welchen Systemkontext sich die Anwendung eingliedert.

2.1. Test Driven Development

TDD ist die englische Bezeichnung für testgetriebene Entwicklung. Hierbei geht es um eine planmäßige Vorgehensweise [9] zur Softwareentwicklung. Sie verfolgt den Ansatz, Module konsequent auf Basis von automatisierten Tests entstehen zu lassen.

Folgender Ablauf wird vorgeschrieben [5]:

- Schreibe zuerst einen automatisierten Test für die zu implementierende Einheit. Starte den Test und lasse ihn fehlschlagen.
- Bringe den Test durch Programm-Code zum erfolgreichen Durchlauf.
- Ist die bisherige Code-Basis schlecht strukturiert: verbessere die Struktur des Codes durch Refactoring.

2.1.1. Herkunft und Verbreitung

Zur Herkunft beziehungsweise dem Erfinder dieses Ansatzes lässt sich folgendes sagen: Bisher gibt es niemanden, der es für sich beansprucht, die testgetriebene Entwicklung erfunden zu haben. Mit Sicherheit kann jedoch gesagt werden, dass die ersten Ansätze dieser Vorgehensweise schon Ende der sechziger Jahre in Veröffentlichungen erwähnt wurden. Hier ein Beispiel aus einer Veröffentlichung aus dem Jahre 1968: „A software system can best be designed if the testing is interlaced with the designing instead of being used after the design“ [10]. In diesem Zitat wird dazu geraten, das Testen in den Designprozess einzubauen. Ein weiteres Zitat aus dem Jahr 1972: „But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand [...]“ [11]. In diesem Zitat wird folgende Empfehlung ausgesprochen: Ein Programmierer sollte den Beweis der Richtigkeit seines Programms zeitgleich mit dem Fortschritt der Implementierung entstehen lassen, um die Belastung so gering und die Motivation so hoch wie möglich zu halten. Wer schreibt die Tests gerne hinterher? Und ist der Code dann überhaupt vernünftig testbar?

Die weite Verbreitung des Ansatzes ging mit der Erfindung von XP [12] und der Veröffentlichung des Buches „Test Driven Development By Example“ von Kent Beck einher. Kent Beck bezeichnet sich selbst nur als „Neuentdecker“ dieser Methode [13], denn im Kern baut sein Buch auf den Ideen aus den vorhergegangenen Veröffentlichungen auf.

2.1.2. Definition der Vorgehensweise

Das Vorgehen während der Implementierung wird durch die drei Phasen „Rot - Grün - Refactor“ [5] bestimmt. Sie beschreiben einen TDD-Zyklus und sind als TDD-Mantra bekannt.

Rot: Schreibe einen Test für eine zu implementierende Einheit. Der Test muss automatisiert ausführbar sein und fehlschlagen. Sofern der Test durch fehlende Typen (Klassen, Interfaces etc.) nicht kompiliert werden kann, gilt dies ebenfalls als fehlschlagen.

Grün: Den Test so schnell wie möglich durch Programm-Code zum erfolgreichen Durchlauf bringen. Der Programm-Code darf schlecht strukturiert sein und Überarbeitungspotential besitzen. Um es mit den Worten von Kent Beck zu sagen: „Der Code darf stinken“.

Refactor: Bei Bedarf das Überarbeitungspotential umsetzen und Duplizierungen[1] entfernen. Den Code demnach so umbauen, dass sich das externe Verhalten nicht ändert, die interne Struktur aber verbessert wird [16].

Das Gelingen von TDD ist eng an die Entwicklungsumgebung geknüpft. Sie muss in der Lage sein die Tests innerhalb weniger Sekunden durchlaufen zu lassen. Dadurch kann sich der Entwickler zu jedem Zeitpunkt schnelles Feedback bezüglich seiner Änderungen am Code einholen. Auch bei wachsender Anzahl Tests muss diese Anforderung an die Entwicklungsumgebung erfüllt bleiben. Wiederholtes langes Warten auf den Durchlauf der Tests führt zu Verärgerung und Effizienzverlusten beim Programmierer.

Hier ein Beispiel zur Veranschaulichung der Vorgehensweise von TDD. Geschrieben ist das Beispiel in JavaScript und dem xUnit[2]-Framework QUnit.

Angenommen ein Webservice liefert den Status zu einer Schulung. Leider nicht als Text, sondern als Zeichen. Dieses Zeichen muss zur Darstellung in einen sprechenden Text umgewandelt werden. Für die Umwandlung soll die Funktion *formatEventStatus* zuständig sein. Unterstützt werden folgende Zuordnungen von Zeichenwert zu Text: f - finished, r - running, s - scheduled.

Dem TDD-Mantra zufolge muss nun zuerst ein Test geschrieben werden.

```
1 QUnit.test("formatEventStatus", function(assert) {
2     assert.equal(formatEventStatus("f"), "finished");
3     assert.equal(formatEventStatus("r"), "running");
4     assert.equal(formatEventStatus("s"), "scheduled");
5     assert.equal(formatEventStatus("x"), undefined);
6 });
```

Listing 2.1: Vorgehensweise zu TDD: rote Phase

[1] „DRY - Don’t repeat yourself“ ist ein genereller Bestandteil eines guten Softwaredesigns. Im Kontext von TDD sind zusätzlich aber auch die Duplizierungen/Wiederholungen gemeint, die zwischen Test- und Programm-Code in Form von sogenannten „magischen Zahlen“ entstanden sind [14]. Mit „magischen Zahlen“ sind Zahlen- und Zeichenwerte im Programm gemeint (auch „hard coded values“ genannt), dessen Bedeutung sich nicht unmittelbar erkennen lässt [15].

[2] „xUnit ist die Bezeichnung für verschiedene Frameworks für Modultests. Diese Frameworks erlauben das Überprüfen verschiedener Elemente (Units) von Software, wie etwa Funktionen und Klassen. Das erste xUnit-Framework wurde von Kent Beck unter dem Namen SUnit für die Programmiersprache Smalltalk entworfen“ [17].

Der Test schlägt fehl. Man befindet sich in der roten Phase. Weder die Deklaration, noch die Definition der Funktion existiert. Bis zu welchem Grad man direkt mit der eigentlichen Implementierung anfängt, also soweit es geht auf „Sünden“ in Form von Überarbeitungspotential und schlechter Strukturierung verzichtet, obliegt dem Programmierer. Kent Beck empfiehlt, je unsicherer der Programmierer bei der Implementierung ist, desto mehr „Sünden“ sollte er zunächst begehen, um sich der Lösung des Problems langsam zu nähern [5].

Um in die grüne Phase zu kommen, muss der Test so schnell wie möglich zum Durchlauf gebracht werden.

```
1  QUnit.test("formatEventStatus", function(assert) {
2      assert.equal(formatEventStatus("f"), "finished");
3      assert.equal(formatEventStatus("r"), "running");
4      assert.equal(formatEventStatus("s"), "scheduled");
5      assert.equal(formatEventStatus("x"), undefined);
6  });
7
8  function formatEventStatus(status) {
9      switch(status) {
10         case "f": return "finished";
11         case "r": return "running";
12         case "s": return "scheduled";
13         default: return undefined;
14     }
15 }
```

Listing 2.2: Vorgehensweise zu TDD: grüne Phase

Der Test läuft nun erfolgreich durch. Die Funktion *formatEventStatus* gibt lediglich die erwartete Statusbeschreibung als „hart-codierten“ Wert zurück. Der Code bedarf einer Überarbeitung. Es sind Duplizierungen beziehungsweise Abhängigkeiten zwischen Test- und Programm-Code entstanden.

Nun beginnt die Refactor-Phase. Der Programm-Code muss bereinigt und die Abhängigkeiten zum Test-Code entfernt werden.

```
1  QUnit.test("formatEventStatus", function(assert) {
2      assert.equal(formatEventStatus("f"), EventStatus.finished.text);
3      assert.equal(formatEventStatus("r"), EventStatus.running.text);
4      assert.equal(formatEventStatus("s"), EventStatus.scheduled.text);
5      assert.equal(formatEventStatus("x"), undefined);
6  });
7
8  const EventStatus = {
9      finished:  { key: "f", text: "finished"},
```

```
10      running:   { key: "r", text: "running"},
11      scheduled: { key: "s", text: "scheduled"}
12  }
13
14  function formatEventStatus(statusKey) {
15      for(var status in EventStatus) {
16          if(EventStatus[status].key === statusKey) {
17              return EventStatus[status].text;
18          }
19      }
20      return undefined;
21  }
```

Listing 2.3: Vorgehensweise zu TDD: Refactor-Phase

Ein konstantes Objekt *EventStatus* wurde eingeführt. Es besitzt die Zuordnung von Status-*key* zu Status-*text*. Die Funktion *formatEventStatus* iteriert über alle vorhandenen Status-*keys* innerhalb des *EventStatus*-Objekts. Bei Übereinstimmung des *keys* gibt sie den entsprechenden *text* zurück. Durch die Umstrukturierung wurde nicht nur die Duplizierung entfernt. Der Programm-Code ist jetzt einfach zu erweitern und lesbarer geworden. Falls sich der *key* zu einem *text* ändern sollte, muss im Programm-Code lediglich das Objekt *EventStatus* angepasst werden. In Bezug auf die Lesbarkeit ist aus einem „hart-codierten“ Wert ein sprechender Bezeichner geworden. Andere Programmierer müssten an Stellen, an denen der Schlüssel womöglich auch auftaucht, nun nicht mehr *„f“* lesen, sondern dort stünde *EventStatus.finished.key*. Man würde sofort erkennen, was hiermit gemeint wäre.

Nach dem Refactoring läuft der Test immer noch durch. Die Implementierung dieser Einheit ist beendet. Ein neuer TDD-Zyklus kann beginnen.

2.1.3. Vor- und Nachteile

TDD ermöglicht dem Programmierer vor jedem weiteren Inkrement kurz inne zu halten, um sich zu überlegen, welche Funktionalität als nächstes entwickelt werden soll. Implizit wird dieses Verhalten auch erzwungen, denn bekanntlich muss der Test für die Funktionalität zuerst geschrieben werden. Somit nähert sich der Programmierer der Lösung des großen Ganzen immer mehr durch das Lösen kleinerer Teilprobleme an. Aus dem fortwährenden Schreiben der Tests ergibt sich zusätzlich eine hohe Testabdeckung, mit der Umbauarbeiten am Programm schnell verifiziert werden können. Somit ist TDD auch ein Hilfsmittel, um konstante Sicherheit und

Überzeugung in Bezug auf funktionierenden Code, besonders nach Umbauarbeiten, zu erzeugen [5]. Die Umbauarbeiten werden vom TDD-Mantra explizit eingeplant, denn Software verändert sich stetig und genauso sollte es sich mit dem Design verhalten - in kleinen überschaubaren Inkrementen. TDD versucht die Lücke zwischen Implementierungsentscheidungen, die der Programmierer fortwährend trifft, und direktem Feedback, ob die Entscheidungen erfolgreich waren, zu schließen [5].

TDD bietet nicht nur Vorteile für den Entwickler. Durch die konstant hohe Testabdeckung:

- tauchen weniger Fehler in der Produktion auf und somit kann die Qualitätssicherung vom reaktiven zum proaktiven Verhalten übergehen [5].
- können in kürzeren Zyklen Softwareinkremente geliefert werden, wodurch eine bessere Beziehung zu den Kunden entsteht [5].
- wird das Projekt kostengünstiger, denn es treten weniger Fehler in späteren Projektphasen sowie der Produktion auf [18].
- entsteht eine aktuelle Dokumentation des Programms durch die Testfälle. Studien haben gezeigt, dass während Wartungsarbeiten an Software, mehr als die Hälfte der Zeit für das Verstehen des Quellcodes benötigt wird [19].
- verbessert sich die Qualität des Codes [20]. Ein Teil dazu trägt die stetige Refactor-Phase bei. Durch die Benutzung der eigenen Programmierschnittstelle im Test-Code fällt schlechtes Design schnell auf. Zudem erzwingt TDD die lose Kopplung von Einheiten durch die Anforderung der automatisierten Testbarkeit.

Aus allen genannten Vorteilen ergibt sich die Erkenntnis, dass der geringste Effekt von TDD der ist, dass die getestete Einheit funktioniert [3].

Bisher wurden nur die positiven Aspekte beleuchtet, doch auch TDD besitzt einige Stolperfallen. Zunächst müssen die Programmierer umdenken. Erst Tests, dann Code. Eine Studie hat gezeigt, dass Junior-Programmierer es vorziehen, sofort mit der Programmierung anzufangen [21], weil ihnen die Erfahrung fehlt, wie komplex große Softwaresysteme werden können. Ein weiteres Problem sind Altsysteme, in denen Komponenten so entwickelt wurden, dass sie nur schwer testbar sind. Selbst bei Neuentwicklungen innerhalb dieser Altsysteme kann es gut sein, dass durch die enge Kopplung die Tests nur unter erschwerten Bedingungen geschrieben werden können. Hektische Phasen in Projekten bieten beste Voraussetzungen, damit

sich Test- und Programm-Code nicht mehr die Waage halten. Einmal übersprungene Tests verleiten leicht dazu, dies auf Dauer zu tun. Zu den bisherigen Schwierigkeiten kommen Aufgaben wie Datenbankdesign und nebenläufige Programme hinzu, die nur schwer testgetrieben entwickelt werden können. Die Richtigkeit des Programm-Codes wird bei TDD mit automatisierten Tests sichergestellt. Nur weil die Tests durchlaufen, heißt das im Umkehrschluss noch lange nicht, dass das Produkt auch den Anforderungen des Kunden in seiner spezifischen Domäne gerecht wird [22].

2.1.4. Varianten

Die bisherigen Kapitel und das Buch von Kent Beck *Test Driven Development By Example* suggeriert, dass TDD lediglich durch xUnit-Tests gelebt werden kann. Einher geht das Verständnis, wie man mittels TDD Backend-Logik entwickeln könnte. Doch wie sieht es mit Anwendungen aus, die eine grafische Benutzerschnittstelle besitzen?

Bisher wurde nur der Bottom-Up-Ansatz beschrieben, TDD funktioniert aber auch Top-Down. Beispielsweise mit automatisierten UI-Tests. Kent Beck sagt selbst, dass TDD eine Technik ist, die auf unterschiedlichsten Abstraktionsebenen ausgeübt werden kann [23]. Somit könnte TDD auch gelebt werden, indem die Entwicklung einer Anwendung mit grafischer Oberfläche, von automatisierten UI-Tests getrieben wird. Bottom-Up oder Top-Down sind somit nur Varianten der Vorgehensweise TDD.

2.1.5. Abgrenzung zu Acceptance Test Driven Development

Die Varianten von TDD sind nicht zu verwechseln mit Acceptance Test Driven Development (ATDD). Aufgrund der Bezeichnung könnte fälschlicherweise der Eindruck entstehen, dass man durch automatisierte Akzeptanz- beziehungsweise UI-Tests folglich auch die Methodik lebt.

ATDD verfolgt jedoch das Ziel eines gemeinsamen Verständnis der Domäne des Kunden und einer beidseitigen Übereinkunft bezüglich der zu liefernden Funktionalität. Dieses Ziel versucht man mit der gemeinsamen Formulierung von automa-

tisierten Akzeptanztests durch den Kunden, die Programmierer und die Tester zu erreichen [23].

Genau hier liegt der Unterschied: TDD setzt zu dem Zeitpunkt an, an dem die Anforderungen schon fest stehen und es um deren Umsetzung geht. Ob die Anforderungen mithilfe von ATDD, und somit durch gemeinsam formulierte Akzeptanztests spezifiziert wurden, spielt dabei keine Rolle.

2.1.6. Erfolgsfaktor: To-do-Liste?

Während Kent Beck in seinem Buch *Test Driven Development By Example* die testgetriebene Entwicklung anhand mehrerer Beispiele erklärt, benutzt er begleitend dazu eine To-do-Liste. Die To-do-Liste erfährt neben ihrer Benutzung im Buch jedoch keine weitere Erklärung. Sie wird zwar verwendet, aber zählt nicht zum Kern von TDD.

Aus dem Kapitel *2.1 Test Driven Development* ist bekannt, dass die erfolgreiche Anwendung von TDD mehrere Phasen durchläuft, die sich mit jedem neuen Inkrement wiederholen. Das stetige Wiederholen der Phasen bietet viel Potential, Dinge zu vergessen, die noch zu erledigen sind, um sich der Lösung des großen Ganzen anzunähern. Vor allem bei größeren zeitlichen Pausen, sowie sich wechselnden Aufgaben aus verschiedenen Projekten. Wo genau habe ich letztes Mal aufgehört? Was genau wollte ich eigentlich machen? Was sind die nächsten Dinge, die zu erledigen sind? Beispielhafte Fragen, die sich ein Programmierer bei der Wiederaufnahme seiner Tätigkeit stellt.

Das Führen einer To-do-Liste wird keineswegs in Kent Beck's Erklärung von TDD vorgeschrieben, dennoch kann sie zum Erfolg von TDD beitragen, indem sie hilft den Überblick zu behalten.

2.2. Model-View-Controller

MVC ist ein Architekturmuster und gibt vor, wie Zuständigkeiten in Anwendungen mit grafischer Benutzeroberfläche aufgeteilt werden sollen. Zudem schreibt die technologieabhängige Implementierung des Musters vor, welche Komponenten in MVC getestet werden müssen und welche außer Acht gelassen werden können.

Aufgrund der Tatsache, dass die Trainer Cloud App mit diesem Architekturmuster entwickelt wurde und es zudem Einfluss auf Testen und Testbarkeit hat, wird dem Architekturmuster dieses Kapitel gewidmet.

Es widmet sich den Fragen: Woher kommt das Muster? Warum gibt es dieses Muster? Wieso herrscht soviel Verwirrung [24] bezüglich des Musters? Wie sieht die Urform des Musters aus?

2.2.1. Ursprung und Absicht

Entstanden ist das Entwurfsmuster 1976 mit der Sprache Smalltalk-76 und der Entwicklung von grafischen Benutzeroberflächen. „Our experiences with the Smalltalk-76 programming system showed that one particular form of modularity–a three-way separation of application components–has payoff beyond merely making the designer's life easier. This three-way division of an application entails separating (1) the parts that represent the model of the underlying application domain from (2) the way the model is presented to the user and from (3) the way the user interacts with it.“ [25]. Es wird von einer vorteilhaften Aufteilung einer grafischen Anwendung in drei Bereiche gesprochen:

- Abstrahierung der Anwendungsdomäne in Form eines Models.
- die Darstellung des Models für den Benutzer.
- die Interaktion mit dem Model durch den Benutzer.

Mit dem Satz „Model-View-Controller (MVC) programming is the application of this three-way factoring [...]“ [25], der die Anwendung dieser Aufteilung als MVC-Programmierung bezeichnet, kam das Entwurfsmuster zu seinem Namen.

In dem ersten Zitat aus dem Artikel *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System* wird von „making the [...] life easier“ gesprochen. Die angesprochene Erleichterung wird mit folgendem Zitat begründet: „When building interactive applications, as with other programs, modularity of components has enormous benefits. Isolating functional units from each other as much as possible makes it easier for the application designer to understand and modify each particular unit, without having to know everything about the other units.“ [25]. Das Leben der Programmierer wird also durch Isolierung funktional

zusammengehöriger Einheiten[3] und der Modularität[4] der Komponenten erleichtert. Diese Vereinfachung erlaubt es den Programmierern, eine spezielle Einheit zu modifizieren, ohne viel von den anderen verstehen zu müssen.

2.2.2. Unterschiedliche Interpretationen

Somit wäre geklärt, woher das Entwurfsmuster stammt und warum es entstanden ist. Die Frage nach der verbreiteten Verwirrung und den unterschiedlichen Interpretationen [25] lässt sich mit einem Zitat von Martin Fowler klären. „Take Model-View-Controller as an example. It's often referred to as a pattern, but I don't find it terribly useful to think of it as a pattern because it contains quite a few different ideas. Different people reading about MVC in different places take different ideas from it and describe these as 'MVC'. If this doesn't cause enough confusion you then get the effect of misunderstandings of MVC that develop through a system of Chinese whispers." [27]. Laut Martin Fowler ist MVC also mehr eine Zusammenstellung verschiedener Entwurfsmuster, als ein einziges alleine. Unterschiedliche Personen lesen verschiedene Interpretationen dieser Zusammenstellung und folglich ergibt sich kein einheitliches Verständnis von MVC. Martin Fowler lässt mit seiner Aussage dennoch offen, woher die unterschiedlichen MVC-Interpretationen kommen. Es hilft zu verstehen, dass Architekturmuster meist geprägt von Designentscheidungen sind, die im Kontext einer bestimmten Technologie getroffen werden. Im Ursprung, als MVC im Rahmen von Smalltalk entwickelt und veröffentlicht wurde, versuchte man die Darstellungs- und die Datenmodel-Zuständigkeiten strikt voneinander zu trennen. Aktuelle Technologien, Sprachen und die darauf aufbauenden Frameworks benötigen den Controller nicht mehr zwingend, um die Benutzereingaben abzufangen und an das Datenmodell weiterzuleiten. Somit fällt diese Ursprungsverantwortlichkeit des Controllers weg. Strikte Anwendung des MVC-Musters in der reinen Form würde somit die Entwicklung nur verkomplizieren. Aus dieser Erkenntnis heraus lässt sich schließen, dass die Beschreibung des MVC-Musters eine generelle ist und es bei der Umsetzung zu technologieabhängigen Anpassungen kommt. Dadurch, dass diese Informationen in den Beschreibungen von MVC für eine bestimmte Programmiersprache fehlen, kommt es zu unterschiedlichen Interpretationen, deren Ursache sich der Leser nicht bewusst wird [28].

[3] Eigene Interpretation: Man will auf hohe Kohäsion der Einheiten hinaus.

[4] Bezeichnet die Aufteilung eines Ganzes in Teile, die Komponenten genannt werden können. Diese Bauteile können entlang definierter Schnittstellen zu einem System zusammengebaut werden [26].

2.2.3. Die Urform des Musters

Die erste Veröffentlichung [25] zu MVC beschreibt das Muster und die damit verbundene Aufgabenteilung. Das Schaubild in Abbildung 2.1 entstammt dieser Veröffentlichung.

Der Controller nimmt Eingaben des Benutzers entgegen, die durch Sensoren registriert wurden. Diese Eingaben werden bearbeitet und entsprechende Aufgaben ausgeführt. Dies könnte eine Änderung des Models oder der View sein. Das Model kann auch unabhängig von Benutzereingaben den Controller oder die View über Zustandsänderungen benachrichtigen. Zudem kann sich die View selbstständig Informationen aus dem Model holen, beziehungsweise an das Model senden.

In der Beschreibung der Urform wird auch erwähnt, dass es abstrakte Klassen von Model, View und Controller geben sollte, die Standardaufgaben übernehmen und von denen abgeleitet werden soll.

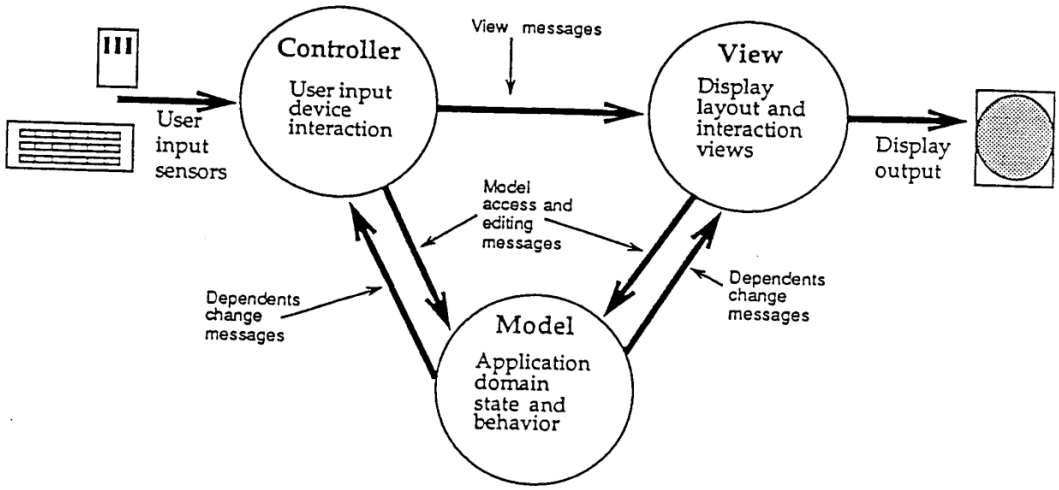

Abbildung 2.1.: MVC in der Smalltalk-Veröffentlichung (Quelle: [25])

2.3. OpenUI5

Da die zu re-implementierende Anwendung mit OpenUI5 entwickelt wird, folgt nun ein Einblick in die Technologie.

OpenUI5 ist ein Open Source Framework zur Erstellung webbasierter Geschäftsanwendungen. Im Kern basiert das Framework auf JavaScript, jQuery und Less [29]. Mit dem Framework lassen sich sogenannte „One-Page-Webapps“ bauen. Das Rendering findet beim Client im Browser statt. Daten für die Anwendungen werden über Representational state transfer (REST)-Schnittstellen bezogen.

Die Bibliothek wird mit Architekturbausteinen für das MVC-Muster ausgeliefert, welche unter anderem Aufgaben übernehmen wie:

- Data-Binding zwischen Model und View.
- verschiedene View-Formate (Extended Markup Language (XML), Hyper Text Markup Language (HTML), JavaScript Object Notation (JSON), JavaScript).
- sehr gute Unterstützung von OData-Datenquellen für das Model.

2.3.1. Model-View-Controller Implementierung

Essenzieller Bestandteil von OpenUI5 sind die MVC-Komponenten und deren Zusammenspiel. Die Model-, View- und Controller-Komponenten bilden das Grundgerüst einer jeden OpenUI5-Anwendung.

Aus dem Kapitel *2.2 Model-View-Controller* ist bekannt, dass es unterschiedliche Implementierungen des Musters gibt und dass es eine wichtige Rolle in Bezug auf Testen spielt. Deshalb folgt nun eine Erläuterung der Umsetzung von MVC bei OpenUI5.

View

Innerhalb des Frameworks OpenUI5 wird die View durch drei Bausteine (siehe Abbildung 2.2) realisiert:

- die Beschreibung der View zur Designzeit als Template.
- Die Instanz der View, die während der Laufzeit existiert und auf Basis des Templates instanziiert wurde,
- oder aber auch das zur Laufzeit von der View-Instanz gerenderte HTML.

.

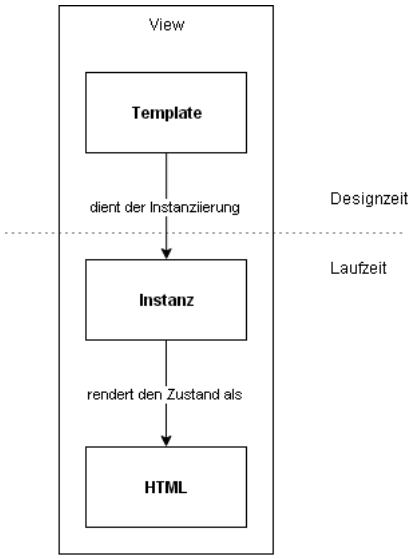

Abbildung 2.2.: Aufteilung der View bei OpenUI5 in drei Komponenten

Model

Neben der Beschreibung der View wird im Template auch das Mapping zu Attributen eines bestimmten Models definiert. Abbildung 2.3 veranschaulicht dies. Während der Prozessierung des Templates werden auf Basis des definierten Mappings Bindings erstellt. Die Bindings sind Eventhandler, die Änderungsereignisse in beide Richtungen propagieren können. Änderungen der Instanzattribute über das gerenderte HTML werden mittels Binding an das Model berichtet. Umgekehrt werden Änderungen der Attribute im Model an die View-Instanz propagiert. Die View-Instanz rendert sich bei Bedarf neu. Das Model bekommt seine Daten durch den Anschluss verschiedener Webservice-Schnittstellen, die im XML- und JSON-Format antworten. Die Bibliothek liefert wiederverwendbare Model-Implementierungen.

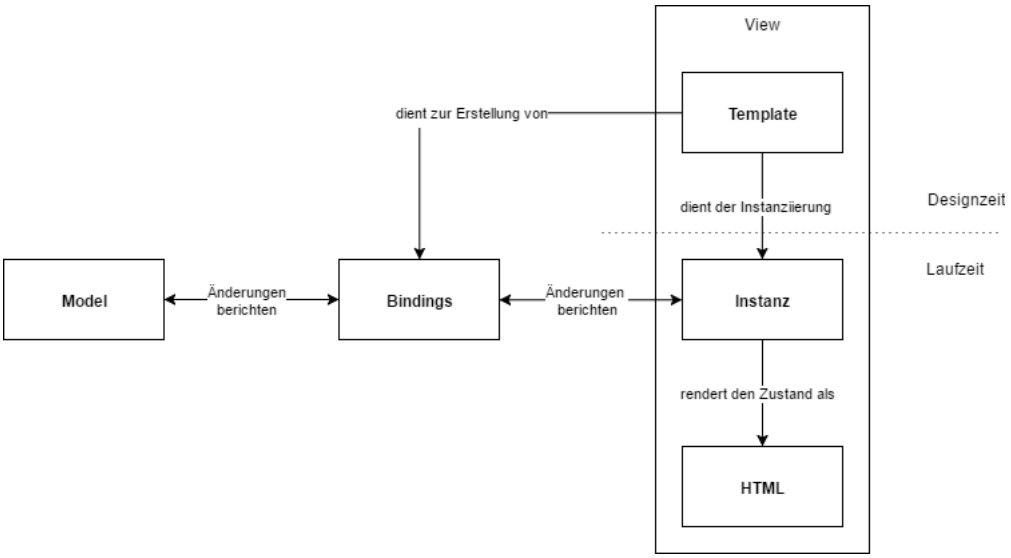

Abbildung 2.3.: Datenaustausch zwischen Model und View bei OpenUI5

Controller

Der Controller (siehe Abbildung 2.4) wird der View-Instanz über das Template bekannt gemacht. Durch den Verweis im Template auf einen bestimmten Controller, wird dieser beim Parsen des Templates instanziiert und der View-Instanz zugewiesen. Der Controller behandelt Benutzerinteraktionen, die im gerenderten HTML ausgelöst werden und von der View-Instanz als Ereignis an alle Interessenten berichtet wird. Zudem kann er den Zustand der View-Instanz durch angebotene Methoden verändern und somit eine Änderung des gerenderten HTML auslösen. Der Controller kann aber auch auf Änderungen des Models reagieren, sofern er sich für ein bestimmtes Attribut registriert hat und somit ein Binding erstellt wurde. Änderungen an Attributen des Models kann er direkt am Model durchführen. „Gemappte“ Bindings reagieren entsprechend.

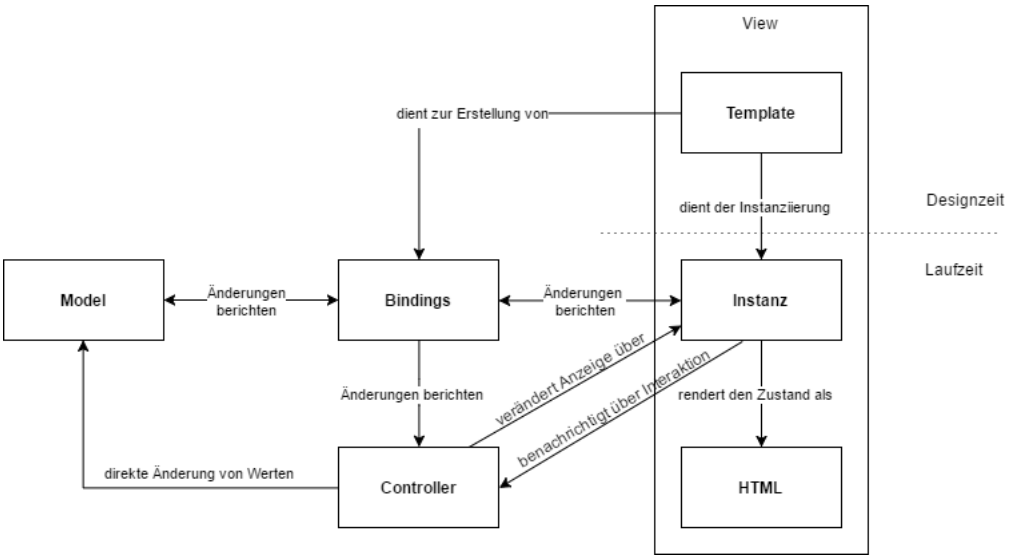

Abbildung 2.4.: Aufgaben des Controllers bei OpenUI5

2.3.2. HeyJoe-Beispiel

Folgendes HeyJoe-Beispiel vermittelt einen ersten Eindruck vom Framework, den beschriebenen MVC-Komponenten und deren Interaktion miteinander. Das Beispiel ist in der Lage, Daten eines Model auf dem UI anzuzeigen und mittels Interaktion durch einen Button, Daten aus dem Model auszulesen.

Zunächst wird eine HTML-Seite benötigt (siehe Listing 2.4). Sie referenziert die OpenUI5-Bibliothek und hängt die View nach dem Start des Frameworks auf der Seite ein.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta http-equiv='X-UA-Compatible' content='IE=edge' />
5      <title>Hey Joe</title>
6
7      <script id='sap-ui-bootstrap'
8          src='/resources/sap-ui-core.js'
9          data-sap-ui-theme='sap_belize'
10         data-sap-ui-libs='sap.m'
11         data-sap-ui-compatVersion='edge'
12         data-sap-ui-resourceroots='{
13             "de.sap.itservices": "./"
14         }'
15         data-sap-ui-preload='async'>
16     </script>
17
18     <script>
19         //warte bis UI5-core hochgefahren ist
20         sap.ui.getCore().attachInit(function () {
21             //erstelle View und haenge sie im HTML ein
22             new sap.ui.xmlview({
23                 viewName: "de.sap.itservices.HeyJoe"
24             }).placeAt("content");
25         });
26     </script>
27 </head>
28 <body class='sapUiBody'>
29     <div id='content'></div>
30 </body>
31 </html>
```

Listing 2.4: Platzieren der View im Grundgerüst einer HTML-Seite

Bevor man die Seite erfolgreich aufrufen kann, wird die HeyJoe-View, der HeyJoe-Controller und eine Model-Instanz benötigt. Die Beschreibung der View lässt die Verbindung zum Controller (siehe Listing 2.5 Zeile 3) erkennen und registriert sich

auf das Event, wenn der Button gedrückt wird, mit einer Methode des Controllers. Zusätzlich wird der anzuzeigende Text auf ein Attribut im Model zugeordnet. Ein sogenanntes „Binding“ zwischen View und Model wird erstellt.

```
<mvc:View xmlns="sap.m"
          xmlns:mvc="sap.ui.core.mvc"
          controllerName="de.sap.itservices.HeyJoe">
    <Button press="onButtonPress" text="Say Hey to"/>
    <Text text="{/person/name}"/>
</mvc:View>
```

Listing 2.5: Beschreibung des Templates einer View im XML-Format

Zum Schluss fehlt noch eine konkrete Implementierung des HeyJoe-Controllers. Im Lebenszyklus-Ereignis *onInit* wird eine Model-Instanz erzeugt und an die Komponente gehängt. Dieser Mechanismus wird normalerweise in einer eigenen Manifest-Datei definiert. Einfachheitshalber werden dem Model auch schon konkrete Daten mitgegeben. Selbstverständlich lassen sich hier auch die Endpunkte eines Webservices konfigurieren. Zusätzlich bekommt der Controller eine zweite Methode, den Eventhandler für das Klicken des Buttons. In dem Event-Handler wird ein natives Browser-Popup erzeugt und Daten aus dem Model angezeigt.

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel"
], function (Controller, JSONModel) {
    "use strict";

    return Controller.extend("de.sap.itservices.HeyJoe", {
        onInit: function() {
            this.getOwnerComponent()
                .setModel(new JSONModel({
                    person: {
                        name: "Joe"
                    }
                }));
        },

        onButtonPress: function () {
            alert("Hello " + this.getOwnerComponent()
                                 .getModel()
                                 .getProperty("/person/name"));
        }
    });
});
```

Listing 2.6: Implementierung eines Controllers

2.4. Trainer Cloud App

Bei der Trainer Cloud App handelt es sich um die zu re-implementierende Anwendung. Dieses Kapitel bietet einen funktionalen Überblick der Anwendung und zeigt auf, in welchen Systemkontext sich die Anwendung eingliedert.

2.4.1. Gesamtübersicht der Funktionalität

Wie viele Softwarefirmen bietet die SAP SE Schulungen an. Diese Schulungen decken eigene Produkte, sowie aktuelle Technologien und Methodiken ab. Die Schulungen können auf unterschiedlichste Weisen stattfinden. Neben virtuellen Schulungen gibt es, ganz klassisch, auch die sogenannten Klassenraum-Schulungen. Jede Klassenraum-Schulung wird von einem oder mehreren Referenten gehalten. Die Referenten wiederum können interne Mitarbeiter der SAP SE sein, oder in einem externen Beschäftigungsverhältnis zur SAP SE stehen.

Damit sich die Referenten einen Überblick über die ihnen zugewiesenen Schulungen verschaffen können, wurde die Trainer Cloud App entwickelt. Die Anwendung kann von internen (SAP Mitarbeitern) und externen (Vertragspartner) Referenten benutzt werden. Folgende Funktionalitäten stehen den Referenten durch die App zur Verfügung:

- bevorstehende Schulungen auf einen Blick überprüfen.
- innerhalb aller zugewiesenen Schulungen suchen.
- Suchergebnisse nach bevorstehenden beziehungsweise vergangene Schulungen filtern.
- aktuelle Details wie Teilnehmer, Ressourcen, Beurteilungsinformationen und Veranstaltungsort zu den Schulungen einsehen.
- Teilnehmer mittels E-Mail kontaktieren.

Die Trainer Cloud App wurde als responsive Webanwendung entwickelt. Das heißt, dass sich die Darstellung der Anwendung der Größe des Endgeräts anpasst. Abbildung 2.5 zeigt die Desktop-, sowie die mobile Version der Anwendung.

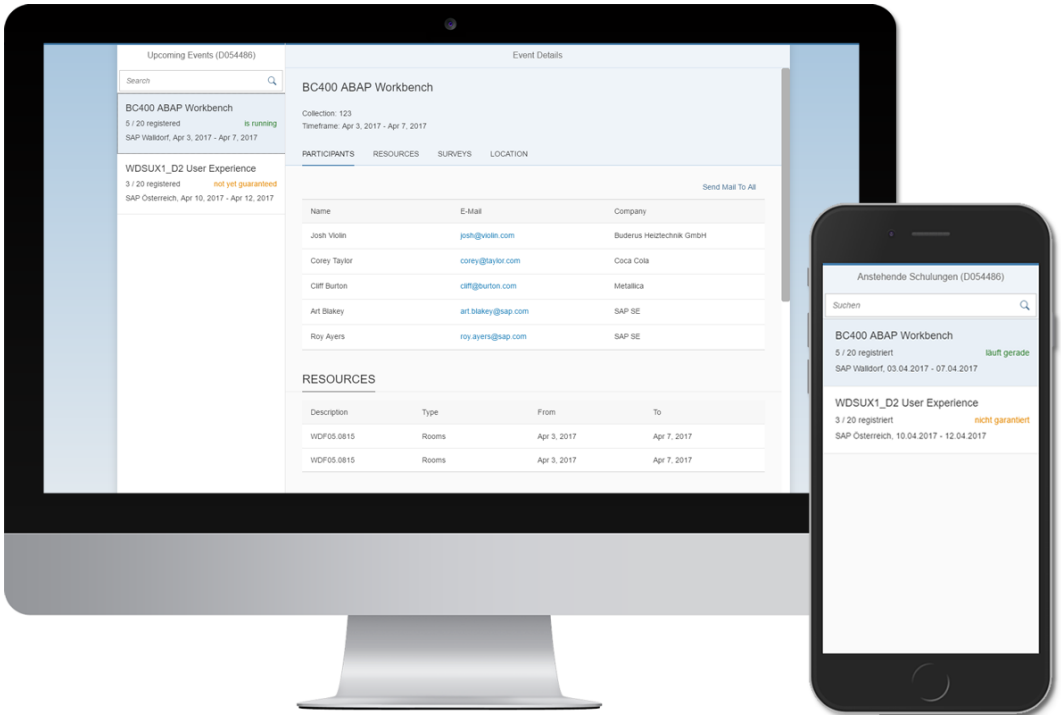

Abbildung 2.5.: Trainer Cloud App in der Desktop- sowie der mobilen Version

2.4.2. Systemkontext

Der aufgezeigte Systemkontext der Trainer Cloud App (siehe Abbildung 2.6) gibt einen Überblick, in welche Systemlandschaft sich die Anwendung eingliedert.

Die SAP SE bietet eine eigene Platform as a Service (PaaS)-Cloud-Lösung an. Auf ihr wird die Webanwendung gehostet. Somit wird die Anwendung über die Cloud-Lösung im Internet betrieben.

Das *SAP Cloud Identity System* ist ein Identity Provider. Die SAP Cloud Plattform (SCP) integriert diesen Dienst zur Benutzerauthentifizierung.

Die Daten der Referenten liegen im Intranet der SAP SE. Um vom Internet ins Firmen-Intranet zu gelangen, wurde ein *SAP Cloud Connector (SCC)* aufgesetzt und an die Cloud-Platform angeschlossen. Er leitet Aufrufe der Trainer Cloud App ins Intranet weiter.

Die weitergeleiteten Aufrufe des *SCC* gehen gegen ein *Gateway*. Das Gateway-System dient als Load-Balancer für das *Learning Solution* System.

Die *Learning Solution* stellt die eigentliche Datenbezugsquelle dar. Sie ist das Backend, über das alle Schulungen der SAP SE organisiert werden und somit auch die Informationen für die Referenten enthält.

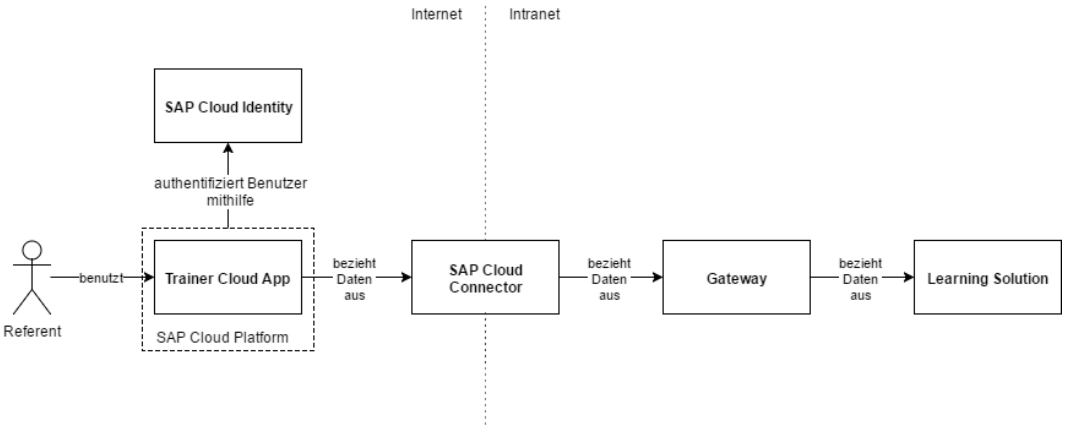

Abbildung 2.6.: Systemkontext der Trainer Cloud App

2.4.3. Komponentensicht

Das System der Trainer Cloud App besteht aus mehreren Komponenten. Neben der eigentlichen Webanwendung kommen noch die Destinations *int_pg_proxy_tca*, *int_pg_tca*, sowie ein eigener Proxy *PG-Proxy* hinzu.

Das Konzept der Destinations wird von der SCP vorgeschrieben. Durch sie können Anwendungen, die auf der SCP laufen, andere Endpunkte erreichen. Im Beispiel der *Trainer Cloud Webanwendung* ermöglicht die Destination *int_pg_proxy_tca* den Aufruf zum *PG-Proxy*.

Der *PG-Proxy* agiert als Stellvertreter, der alle Aufrufe entgegen nimmt, bevor sie an das Gateway mithilfe des SAP Cloud Connectors weitergeleitet werden. Der Proxy übernimmt die Aufgabe, den Aufruf mit Informationen über den aktuell angemeldeten Benutzer anzureichern. Dies wird benötigt, denn die Authentifizierung gegenüber dem Gateway findet mittels technischem User statt. Ohne die Anreicherung des Proxys würde die Information über den angemeldeten Benutzer verloren gehen.

Damit der *PG-Proxy* das Gateway erreicht bedient er sich der Destination *int_pg_tca*.

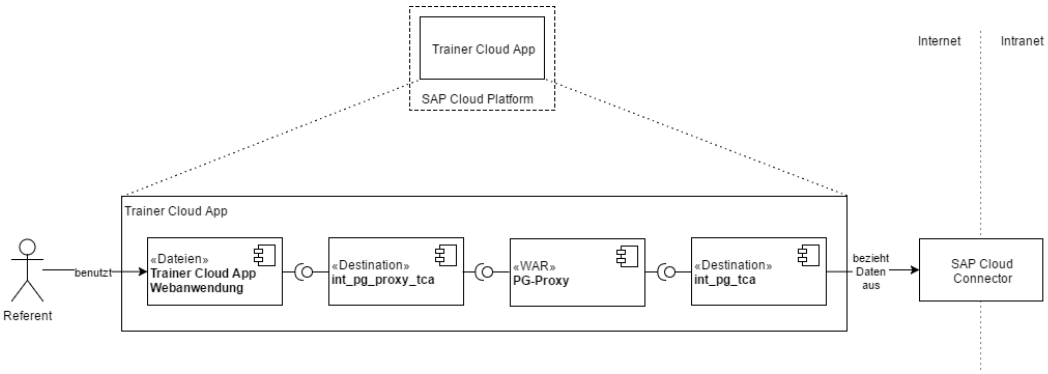

Abbildung 2.7.: Komponentensicht des Trainer Cloud App Systems

Wenn im weiteren Verlauf der Arbeit von „Trainer Cloud App“ gesprochen wird, ist die Komponente *Trainer Cloud App Webanwendung* gemeint.

2.4.4. Verteilungssicht

Zur besseren Abgrenzung im nachfolgenden Kapitel *4. Re-Implementierung der bisherigen Trainer Cloud App* wird die Verteilungssicht der Anwendung durch Abbildung 2.8 aufgezeigt.

Durch die Aufzählung aller relevanten Artefakte kann definiert werden, welche Artefakte Bestandteil der Re-Implementierung sind. Zudem trägt die Information der verwendeten Kommunikationsprotokolle zum Gesamtverständnis bei. Der SCP-Teil mit all seinen Artefakten wurde im Kapitel *2.4.3 Komponentensicht* erläutert.

Damit Aufrufe ins Firmennetz abgesetzt werden können, wird eine getunnelte SSH-Verbindung zwischen SCP und SCC hergestellt. Damit der vom *PG-Proxy* angereicherte Aufruf sein Ziel erreicht, muss im SCC noch der entsprechende Host und die benötigten Pfade konfiguriert werden.

Das Gateway stellt den Webservice durch die *Trainer Cloud App Service-Registrierung* bereit und agiert als Hypertext Transfer Protocol (HTTP)-Schnittstelle zur Learning Solution. Gateway-System und Learning Solution kommunizieren über eine konfigurierte Remote Function Call (RFC)-Verbindung.

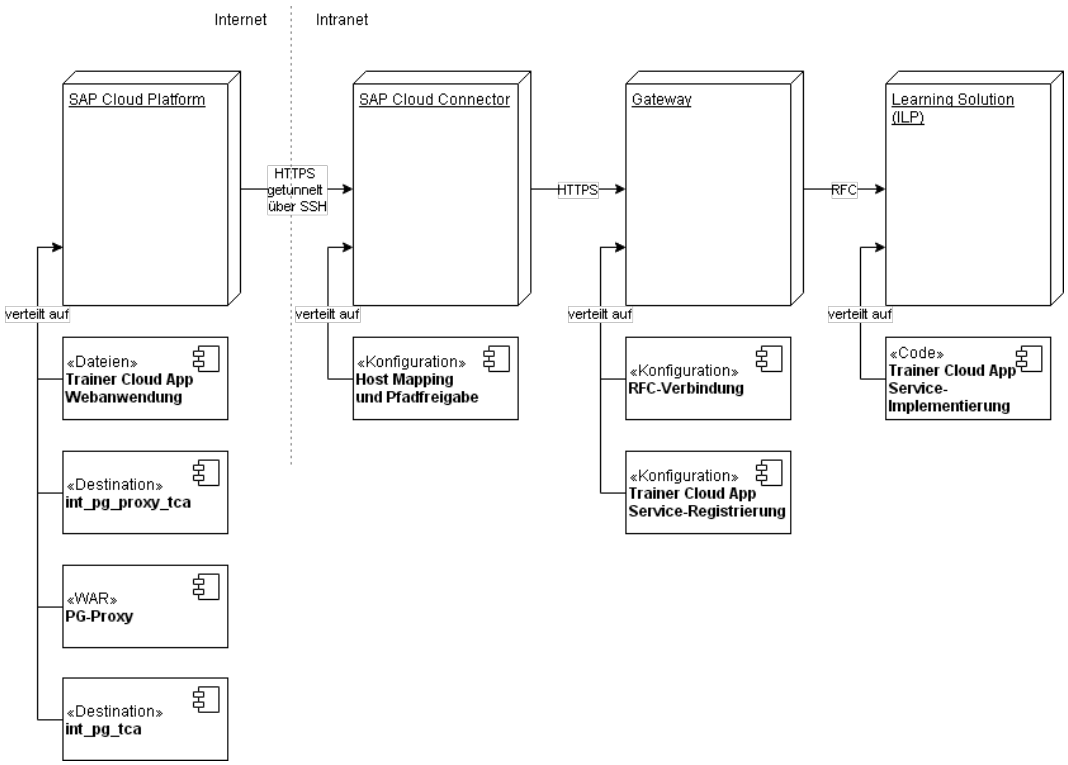

Abbildung 2.8.: Verteilungssicht aller Artefakte, die zur erfolgreichen Inbetriebnahme der Trainer Cloud App notwendig sind

Kapitel 3

Erwartungen an Test Driven Development

In diesem Kapitel werden die Erwartungen des Autors an TDD geäußert, um später im Kapitel *6. Auswertung der Analyseergebnisse in Bezug auf die Erwartungen* auf deren Bewahrheitung einzugehen. Die Erwartungshaltung baut maßgeblich auf dem Kapitel *2.1. Test Driven Development* und den Büchern *Test-Driven Development By Example* von Kent Beck und *Agile Software Development, Principles, Patterns, and Practices* von Robert C. Martin auf. Zudem werden die Erwartungen klassifiziert und Teilgebieten der Softwareentwicklung zugeordnet [30]. Die anschließend erwarteten Schwierigkeiten sind das Resultat eigener Überlegungen.

Jede Erwartung wird mit einem Identifikator versehen, um im weiteren Verlauf der Arbeit auf einzelne Erwartungen dieses Kapitels referenzieren zu können.

Implementierung

(EI1) Der TDD-Zyklus erzwingt die Aufteilung des Problems in kleine Teilprobleme. Man muss sich der Lösung in überschaubaren Inkrementen annähern.

(EI2) Die Verwendung einer To-do-Liste unterstützt den uneingeschränkten Fokus auf die jeweilige Entwicklungseinheit.

(EI3) Die stetige Test-Feedback-Schleife generiert Überzeugung von funktionierender Software.

(EI4) Die Qualität des Codes nimmt zu.

(EI5) Es entsteht testbare Software.

(EI6) Fehler fallen früher auf und sind deswegen günstiger in deren Behebung.

Entwurf

(EE1) Die Qualität des Designs fällt durch Benutzung der eigenen Schnittstellen im Test-Code auf.

(EE2) Ausgiebige Verfeinerungen des Designs während der Konzeption werden vermieden. Das Design wird, falls nötig, während der Implementierung in der Refactoring-Phase angepasst.

Dokumentation

(ED1) Begleitend zur Entwicklung entstehen kompilierbare Code-Beispiele die zum Verständnis bezüglich Benutzung der eigenen Schnittstellen beitragen.

(ED2) Die Tests dokumentieren die Zuständigkeiten einzelner Einheiten.

(ED3) Die Tests repräsentieren stetig aktuelle Programm-Dokumentation.

Testen

(ET1) Weniger Fehler durch vorhandene Tests.

Lieferung

(EL1) Durch die Absicherung von automatisierten Tests entstehen kürzere Lieferzyklen.

Wartung

(EW1) Die Code-Beispiele innerhalb der Tests helfen den Programm-Code zu verstehen.

Schwierigkeiten

(ES1) Erhöhter Aufwand für das Aufsetzen der Entwicklungsumgebung.

(ES2) Zusätzliche Kosten für das Projekt aufgrund des erhöhten Aufwands für das Schreiben der Tests.

(ES3) Viele Mock-Objekte wirken sich negativ auf die Lesbarkeit der Tests aus.

(ES4) Wenn man nicht weiß, welche Abhängigkeiten für das gewünschte Verhalten imitiert werden müssen, ist es schwierig einen Test zu verfassen.

(ES5) Duplizierungen sowie Abhängigkeiten zwischen Testdaten und UI-Tests.

Kapitel 4

Re-Implementierung der bisherigen Trainer Cloud App

Aus dem Kapitel *2.4 Trainer Cloud App* geht hervor, dass die Anwendung aus Back- und Frontend besteht. Re-Implementiert wird ausschließlich das mit OpenUI5 entwickelte Frontend. Backend und die dazugehörige Open Data Protocol (OData)-Schnittstelle bleiben unverändert.

Die Re-Implementierung der Trainer Cloud App bringt viele Neuerungen mit sich: Werkzeuge, Technologien und die Vorgehensweise der testgetriebenen Entwicklung. Dieses Kapitel beschäftigt sich mit der neu entstandenen Entwicklungsumgebung, einer Teststrategie bezüglich der entstehenden Komponenten (Model, View, Controller) und der auf die Gegebenheiten zugeschnittenen Adaption von TDD.

4.1. Entwicklungsumgebung zur testgetriebenen Entwicklung

Zur testgetriebenen Re-Implementierung der Anwendung muss es möglich sein, automatisierte UI- und Unit-Tests innerhalb der Entwicklungsumgebung zu verfassen und auszuführen. Zudem muss die Entwicklungsumgebung bestimmte Kriterien erfüllen.

Die Definition der Entwicklungsumgebung wird von folgenden Fragen geleitet:

- Wie können die Tests in wenigen Sekunden ausgeführt werden?

- Wie lässt sich eine Schnittstelle zur Ausführung der Tests für die Kommandozeile definieren?
- Mit welchem Framework lassen sich UI-Tests schreiben?
- Sind die UI-Tests für Dritte (Tester, Kunde etc.) lesbar?
- Kann vermieden werden, dass sich ein Browser-Fenster bei der Ausführung der UI-Tests öffnet?
- Wie können Unit-Tests ausgeführt und definiert werden, wenn Abhängigkeiten zur OpenUI5-Bibliothek bestehen?
- Mit welchem Framework lassen sich die Unit-Tests formulieren?
- Mit welchem Framework können Mock-Objekte erstellt werden?

4.1.1. Zusammenstellung der Toolchain

Im Folgenden werden die Komponenten der Entwicklungsumgebung beschrieben. Sie liefern Antworten auf die zuvor formulierten Leitfragen. Neben der Beschreibung der Komponenten wird ebenfalls deren Auswahl begründet.

NodeJS [31]: Der Browser repräsentiert die Laufzeitumgebung der Trainer Cloud App. Zur testgetriebenen Entwicklung wird eine weitere Laufzeitumgebung benötigt. Die zusätzliche Laufzeitumgebung soll es ermöglichen, wiederkehrende Aufgaben zu automatisieren. Die Wahl fällt auf NodeJS. NodeJS ist eine vom Browser unabhängige Laufzeitumgebung für JavaScript. Durch NodeJS bleibt JavaScript die einzige Sprache innerhalb des Projekts. Es entsteht keine zusätzliche Komplexität in Bezug auf die Sprachenvielfalt des Projekts. Anwendung und Tests können in der selben Sprache entwickelt werden.

Grunt [32]: Die Automatisierung der Ausführung wiederkehrender Aufgaben verlangt nach einem sogenannten „Taskrunner“ für NodeJS. Wiederkehrende Aufgaben sind beispielsweise die Ausführung von Tests, Erstellung statischer Code-Analysen und Starten eines Webservers für die Entwicklung. Der Taskrunner Grunt bringt einen eigenen Kommandozeilen-Client sowie viele vordefinierte Tasks mit. Neben Grunt gibt es noch die Konkurrenzlösung Gulp. Jedoch lassen sich, aus Sicht des Autors, die Tasks mit Grunt leichter definieren.

Express [33]: Zur Auslieferung der Trainer Cloud App während der Entwicklung muss ein Webserver eingeführt werden. Durch den Webserver werden Sicherheitsfehler im Browser bezüglich der same-origin-policy[1] verhindert. Die UI- und Unit-Tests benötigen zur Ausführung ebenfalls einen Webserver. Aufgrund dessen muss es möglich sein, den Webserver aus der NodeJS Laufzeitumgebung zu starten. Die Wahl des Webservers fällt auf den verbreitetsten im NodeJS-Umfeld: Express.

PhantomJS [34]: Durch einen Browser mit Schnittstelle zur Steuerung sollen die UI-Tests automatisiert ausgeführt werden. Zudem sollte er keine grafische Oberfläche besitzen. Ein solcher Browser bietet Geschwindigkeitsvorteile und kann auf Umgebungen wie einem Server ausgeführt werden, die selbst auch keine grafische Oberfläche besitzt. Die Wahl fällt auf PhantomJS, denn der Browser besitzt keine grafische Oberfläche und implementiert die WebDriver-Api. Mithilfe der implementierten WebDriver-Api kann der Browser über HTTP gesteuert werden.

Selenium WebDriver [35]: Um die von PhantomJS angebotene Implementierung der WebDriver-Api anzusteuern, wird eine Bibliothek benötigt. Durch die Zunahme einer Bibliothek als Wrapper der WebDriver-Api müssen die HTTP-Anfragen nicht von Grund auf erstellt werden. Der Selenium WebDriver ist eine solche Bibliothek, die zudem in der Sprache JavaScript verfügbar ist.

CodeceptJS [36]: Durch NodeJS, Grunt, Express, PhantomJS und dem Selenium WebDriver steht das Grundgerüst zum automatisierten Testen der Oberfläche. Das Framework zur Formulierung der UI-Tests muss die WebDriver-Api Schnittstelle beziehungsweise Selenium WebDriver unterstützen. CodeceptJS bringt eine solche Unterstützung mit. Es ist ein Framework zur Formulierung von Aktzeptanztests für Webanwendungen. Zudem lassen sich die Tests mit CodeceptJS in JavaScript formulieren. Damit findet kein weiterer „Sprachenbruch“ statt und alle Artefakte können weiterhin nur mit JavaScript entwickelt werden. Das Formulieren der Tests findet auf hohem Abstraktionslevel statt. Dadurch bleiben die Tests für Nicht-Entwickler lesbar.

QUnit [37]: Bezüglich Unit-Tests gibt es bei Webanwendungen die Besonderheit, dass sie im Browser laufen und im Falle einer Anwendung, die mit einem Framework entwickelt wurde, auch Abhängigkeiten zu eben diesem Frame-

[1] Die same-origin-policy besagt, dass Browser mittels XMLHttpRequest nur Ressourcen des eigenen Hosts beziehen dürfen.

work bestehen. Somit fällt ein xUnit-Framework für NodeJS weg. Mit NodeJS können keine JavaScript-Bibliotheken referenziert und verwendet werden. Bei dieser Arbeit sind das zum Beispiel Teile des OpenUI5-Frameworks, die zur Ausführung der Unit-Tests benötigt werden. Demnach sind xUnit-Tests nur in Kombination aus einer HTML-Seite zum Referenzieren der benötigten Bibliotheken, den eigentlichen Unit-Tests, einem Browser als Laufzeitumgebung und einem Webserver zur Auslieferung der HTML-Seite möglich. Bis auf ein xUnit-Framework bietet das bisherige Setup alles. Die Wahl des xUnit-Frameworks fällt auf das in der OpenUI5 enthaltene QUnit-Framework. Somit muss keine zusätzliche Bibliothek geladen werden.

SinonJS [38]: Um Abhängigkeiten bei der Formulierung von Unit-Tests zu imitieren, wird das bekannteste JavaScript Mocking-Framework SinonJS verwendet.

timemachine [39]: Mit der JavaScript-Bibliothek timemachine lässt sich die Information über die aktuelle Uhrzeit des nativen Date-Objekts verändern. Somit können Datümer der Testdaten und das aktuelle Browser-Datum während der Testausführung in Einklang gebracht werden.

Mocha [40]: Damit die Unit-Tests automatisch ausgeführt und ausgewertet werden können, wird ein Test-Runner im NodeJS-Umfeld benötigt. Die Wahl fällt auf den im NodeJS-Umfeld bekannten Test-Runner Mocha. Der Test-Runner ermöglicht es, bevor die Tests ausgeführt werden, einen Webserver hochzufahren. Bei der Ausführung eines Tests wird PhantomJS mit dem Selenium WebDriver angesteuert und die QUnit-Test-Seite aufgerufen. Sobald die QUnit-Tests durchgelaufen sind wird das Ergebnis ausgewertet.

Das Zusammenspiel der wichtigsten Komponenten zur Ausführung der UI- und Unit-Tests wird durch folgende zwei Abbildungen aufgezeigt.

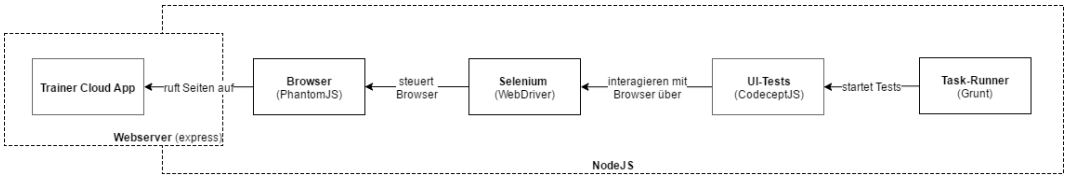

Abbildung 4.1.: Zusammenspiel der wichtigsten Komponenten der Entwicklungsumgebung zur Ausführung der UI-Tests

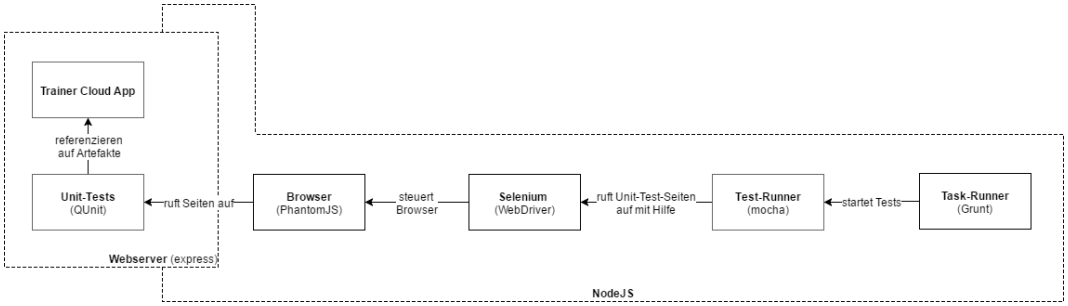

Abbildung 4.2.: Zusammenspiel der wichtigsten Komponenten der Entwicklungsumgebung zur Ausführung der Unit-Tests

4.1.2. Ausführung der UI-Tests

Der Ausführungsablauf der Unit-Tests wird durch Abbildung 4.3 dargestellt. Zur Ausführung der UI-Tests werden zwei Grunt Tasks definiert. Der Task *grunt desktop-acceptance-tests* startet die Tests der Desktop- und *grunt mobile-acceptance-tests* die der mobilen Version. Wie bei der *Ausführung der Unit-Tests* werden zu Beginn ein Webserver und PhantomJS gestartet. CodeceptJS, das verwendete Framework zur Formulierung der UI-Tests, kümmert sich um die Ausführung der einzelnen Tests, die Interaktion mit dem Browser und die Auswertung der grafischen Oberfläche. Sobald alle UI-Tests ausgeführt wurden, wird das Ergebnis auf der Kommandozeile ausgegeben. Da die Ausführungszeit aller UI-Tests mit größer werdender Anzahl der Tests ansteigt, kann bei beiden Grunt Tasks ein Parameter übergeben werden, mithilfe dessen einzelne Tests ausgeführt werden können.

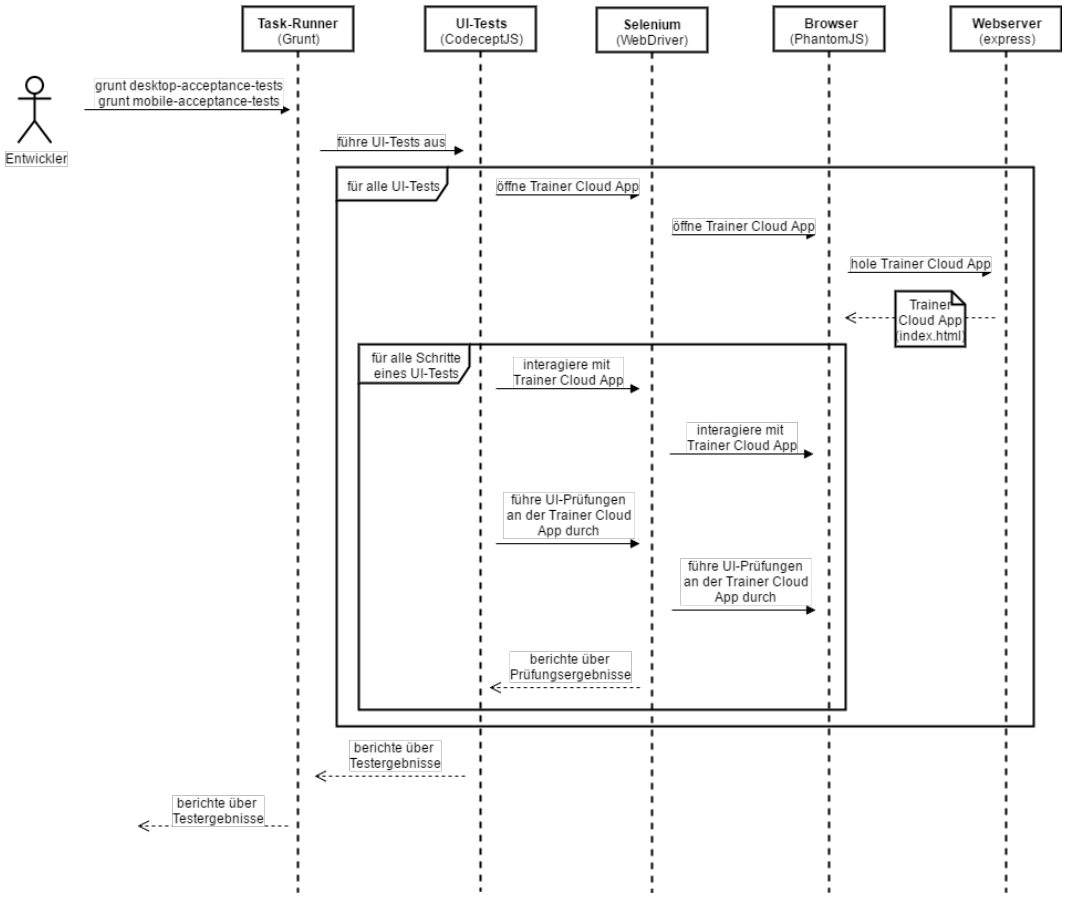

Abbildung 4.3.: Zusammenspiel der Komponenten zur Ausführung und Auswertung der UI-Tests

4.1.3. Ausführung der Unit-Tests

Der Ausführungsablauf der Unit-Tests wird durch Abbildung 4.4 dargestellt. Die automatisierte Ausführung aller QUnit-Tests wird durch den eigens hierfür definierten Grunt Task (*grunt unit-tests*) gestartet. Anschließend wird der Webserver hochgefahren um die QUnit-Testseiten ausliefern zu können. Sobald der Webserver hochgefahren ist, wird der „kopflose“ Browser PhantomJS gestartet. Mittels Selenium WebDriver wird der Browser angesteuert. Nun werden die QUnit-Testseiten nacheinander aufgerufen und das Ergebnis der QUnit-Tests ausgelesen. Sobald alle QUnit-Test ausgeführt wurden wird das Gesamtergebnis aller Tests auf der Kommandozeile ausgegeben.

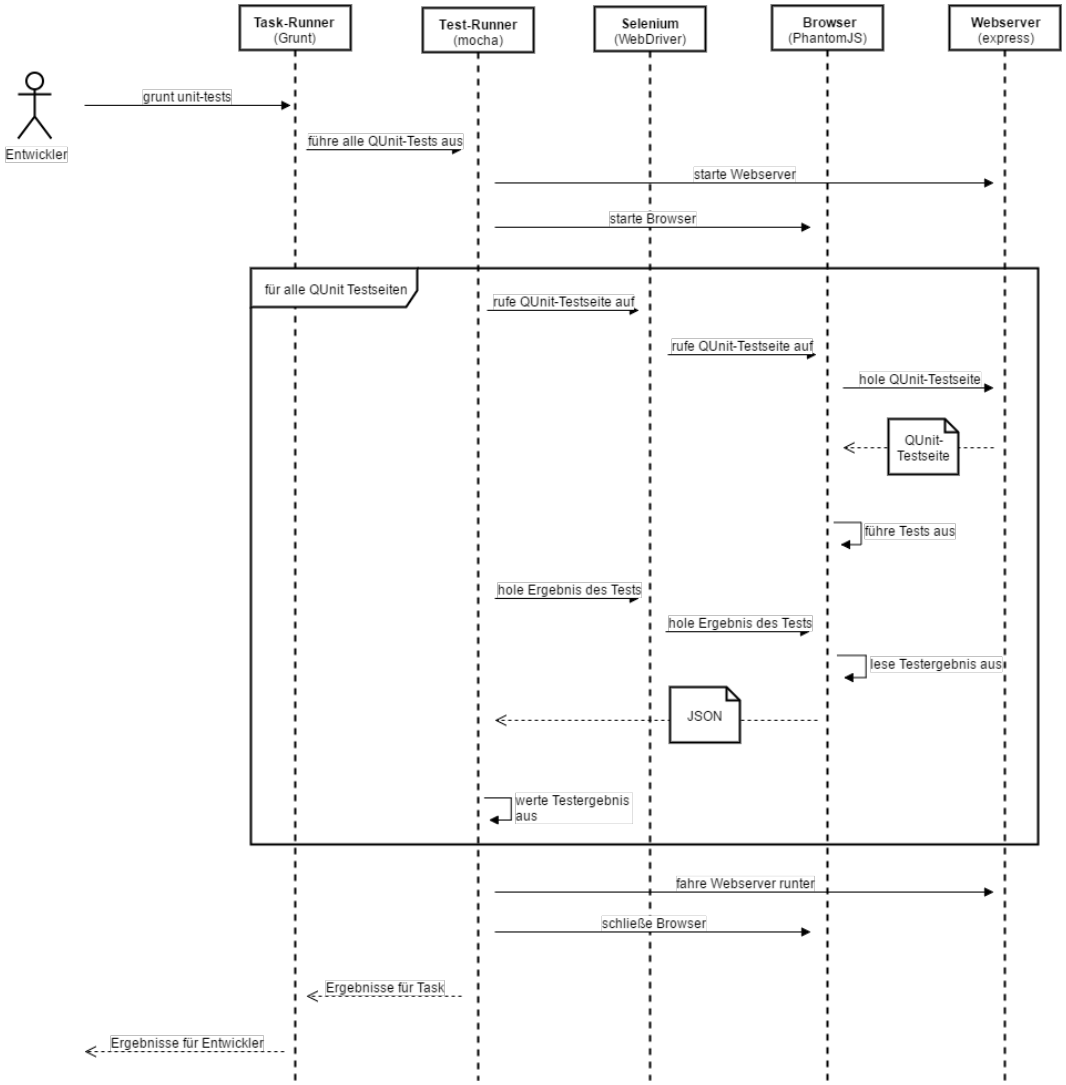

Abbildung 4.4.: Zusammenspiel der Komponenten zur Ausführung und Auswertung der Unit-Tests

4.2. Strategie zum Testen der Model-View-Controller Komponenten

Dieses Kapitel befasst sich mit der Notwendigkeit und der Ausprägung des Testens, bezogen auf jede MVC-Komponente des OpenUI5-Framework.

4.2.1. Model

Das Datenmodel der Trainer Cloud App liegt im Backend. Mit dem Backend der Trainer Cloud App kann nur über eine OData-Schnittstelle kommuniziert werden. OData ist eine Microsoft Erweiterung des REST-Protokolls. REST baut auf HTTP auf. Typisch für OData-Schnittstellen ist eine Manifest-Datei. Sie enthält die notwendigen Informationen zur Benutzung der Schnittstelle. Aufgrund der spezifischen OData-Schnittstelle wird bei der Re-Implementierung deshalb die Model-Implementierung *ODataModel* verwendet. Das Framework bietet zusätzlich einen *MockServer*. Mit Hilfe des *MockServer* und der Beschreibung der Schnittstelle durch die Manifest-Datei, kann die Datenbezugsquelle imitiert werden. Somit richten sich die OData-Aufrufe nicht gegen das Backend, sondern werden vom *MockServer* beantwortet. Der *MockServer* antwortet mit Testdaten, die als JSON-Dateien im Projekt abgelegt sind. Dadurch können Tests schneller und unabhängig von der Verfügbarkeit des Backends ausgeführt werden. Es findet keine Kommunikation über das Netzwerk statt. Zudem können sich die Tests auf die vorhandenen Testdaten beziehen und sind von keiner korrekten Pflege im Backend abhängig.

Die Beschreibung des verwendeten Models und der Mechanismus für den Bezug der Testdaten veranschaulicht, dass keine Eigenentwicklung hierfür notwendig ist. Somit müssen keine Tests zur Entwicklung eines Models geschrieben werden. Für die wiederverwendeten Komponenten gibt es bereits Tests innerhalb des OpenUI5-Framework. Ob die Testdaten auf dem UI erscheinen, wird durch die UI-Tests geprüft.

4.2.2. View

Die View wird zur Designzeit in Form eines Templates beschrieben. Zur Laufzeit wird die Beschreibung der View verarbeitet, eine Instanz der View erstellt und die View gerendert.

Während der Re-Implementierung werden ausschließlich mitgelieferte Controls aus dem OpenUI5-Framework verwendet. Diese Controls werden, wie auch das *OData-Model*, innerhalb des Frameworks durch Testfälle geprüft. Demnach ist es überflüssig, ein Rendering beziehungsweise Instanziierung der Views zu testen, um dadurch die Entwicklung zu treiben.

Die Entwicklung der Views (Beschreibung der Templates) wird durch UI-Tests getrieben. Ziel der UI-Tests ist es, einzelne Features der Trainer Cloud App zu testen. Die UI-Tests prüfen, ob bestimmte Interaktionselemente vorhanden sind, ein vorgegebenes Seitenlayout erfüllt wird und mit der gerenderten View auf bestimmte Art und Weise interagiert werden kann. Ob sich die Views auch in den mobilen Ansichten richtig verhalten, wird mit den UI-Tests für mobile Endgeräte getestet.

4.2.3. Controller

Der Controller übernimmt innerhalb der Trainer Cloud App Aufgaben wie Formatierung von Werten, Behandlung von Interaktionen auf dem UI, Änderungen vorhandener Bindings und Navigation zwischen Views.

Bezogen auf das gesamte Frontend befindet sich der überwiegende Teil des JavaScript-Codes innerhalb der Controller. Im Gegensatz zu den Views, bei denen nur die für die Funktionalität wichtigen Elemente getestet werden, wird bei den Controllern jede einzelne Methode testgetrieben entwickelt.

Funktionen wie die der Werteformatierung sind einfach zu testen. Sie bilden Eingabe- auf Ausgabewerte ab und besitzen keine Abhängigkeiten zu anderen Objekten. Zustandsverändernde Methoden dagegen sind schwerer zu testen. Veränderung eines Zustands könnten ein geändertes Binding oder die Navigation zu einer anderen View sein. Auch wenn diese Methoden keinen Rückgabewert besitzen, müssen sie dennoch getestet werden. Der Fokus eines Tests für zustandsverändernde Methoden liegt auf eben diesen Seiteneffekten. Es muss also geprüft werden, ob die gewünsch-

ten Seiteneffekte nach Aufruf der Funktion eingetreten sind. Zur Überprüfung der gewünschten Seiteneffekte werden Mock-Objekte verwendet.

4.3. Praktizierte Variante von Test Driven Development

Im Kapitel *2.1 Test Driven Development* wird die Vorgehensweise theoretisch definiert. Dieses Kapitel beschäftigt sich mit der Auslegung und Umsetzung der testgetriebenen Entwicklung während der Re-Implementierung der Trainer Cloud App.

Bei der testgetriebenen Re-Implementierung wird die im Kapitel *2.1.4 Varianten* beschriebene Top-Down-Variante angewandt. Eine Neuentwicklung Bottom-Up wäre auch möglich. Aus Sicht des Autors ist es jedoch intuitiver, zuerst einen UI-Test zu verfassen und dann die benötigten Module aufgrund von Unit-Tests zu realisieren. Zusätzlich wird eine To-do-Liste über den gesamten Zeitraum der Re-Implementierung eingesetzt. Das Aktivitätsdiagramm in Abbildung 4.5 veranschaulicht den Ablauf während der Entwicklung.

Zu Beginn wird immer ein UI-Test für eine Funktionalität geschrieben. Dieser Test wird ausgeführt. Sofern er fehlschlägt, kann mit der Anpassung des Templates der View begonnen werden. Falls für die gewünschte Funktionalität noch ein JavaScript-Modul benötigt wird, muss ein entsprechender Unit-Test verfasst und fehlschlagen gelassen werden. Nun kann mit der Implementierung des Moduls begonnen werden. Sobald der Unit-Test durchläuft, es keines Refactorings mehr bedarf und keinerlei JavaScript-Module mehr benötigt werden, wird der UI-Test noch einmal ausgeführt. Läuft dieser ebenfalls durch und es ist keine Anpassung der View mehr notwendig, ist die Funktionalität fertig implementiert.

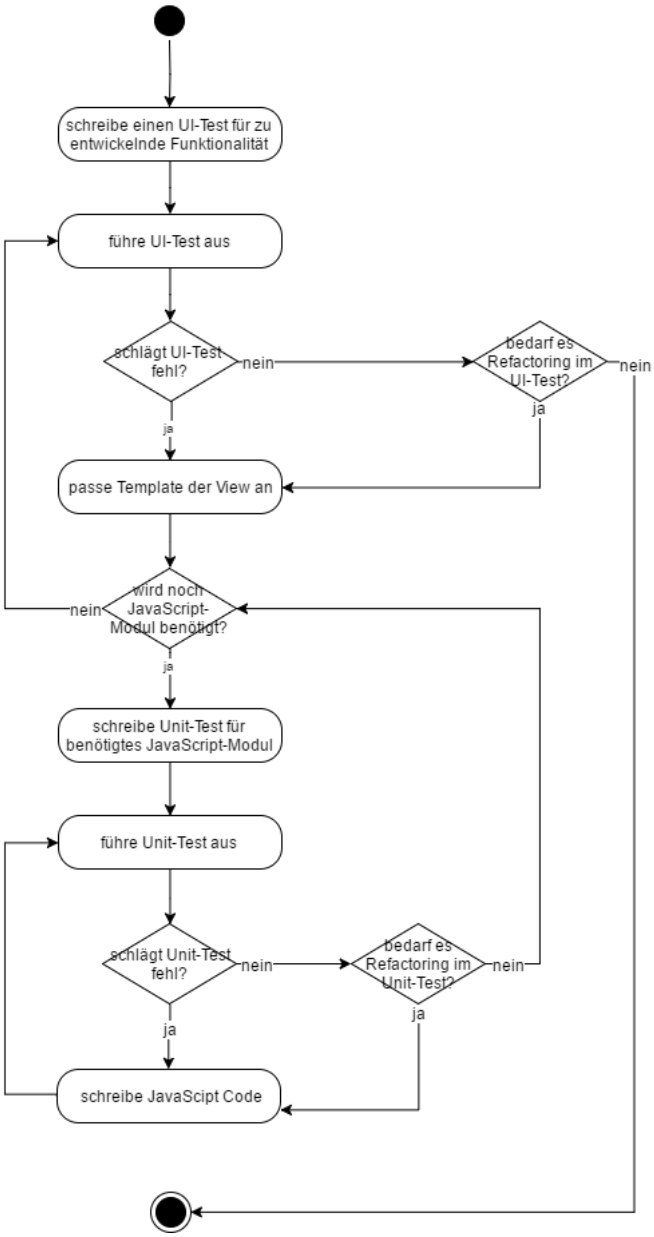

Abbildung 4.5.: Aktivitätsdiagramm zur praktizierten Variante von TDD

Kapitel 5

Analyse der Re-Implementierung

Dieses Kapitel beschreibt die verwendeten Analyse-Methoden zur Evaluierung der Annahmen im Kapitel *3. Erwartungen an Test Driven Development* und geht der Frage nach, wie evaluiert wird. Um den Vergleich zwischen neuer und alter Trainer Cloud App ziehen zu können, werden die Analyse-Methoden auf beide Lösungen angewandt. Neben Softwaremetriken wird eine subjektive Bewertung des Autors und eine Fallstudie zur Evaluierung herangezogen. Die Ergebnisse der Analysen werden im Kapitel *6. Auswertung der Analyseergebnisse in Bezug auf die Erwartungen* behandelt.

5.1. Softwaremetriken

Die erste Analyse wird anhand von Softwaremetriken durchgeführt. Softwaremetriken sind Funktionen, die Eigenschaften von Software auf Zahlenwerte abbilden [41]. Je nach gewählter Metrik lassen sich verschiedenste Aussagen bezüglich des Softwaresystems machen. Für diese Arbeit werden die Anzahl der Code-Zeilen und die zyklomatische Komplexität berechnet. Die Begründung für die jeweilige Metrik ist im entsprechenden Unterkapitel zu finden.

Um keinen unfairen Vergleich zwischen beiden Lösungen zu ziehen, werden bestimmte Datei-Typen von der Berechnung der Metriken ausgeschlossen. Ein OpenUI5-Projekt besteht hauptsächlich aus XML-Dateien (Beschreibung der Views), JavaScript-Dateien (Implementierung der Controller) und JSON-Dateien (Konfiguration der Anwendung und Testdaten). Zur Berechnung der Metriken werden bei beiden Lösungen die XML- und JSON-Dateien ausgeschlossen. Zum einen kann

keine zyklomatische Komplexität über XML- und JSON-Dateien berechnet werden. Andererseits lässt die berechnete Anzahl der Zeilen den Vergleich hinken, denn in der neuen Lösung gibt es übermäßig viele JSON-Dateien die Testdaten enthalten. Die Anzahl der Zeilen über den XML-Dateien ist ebenfalls nicht relevant, da die Beschreibungen der Views in beiden Lösungen nahezu identisch sind. Neben dem Ausschluss der XML- und JSON-Dateien werden zudem auch die JavaScript-Dateien mit Test-Code in der neuen Lösung ausgeschlossen.

Der Ausschluss verfolgt das Ziel, den Kern (JavaSript-Code) beider Lösungen anhand von Metriken vergleichen zu können. Test-Code, Test-Daten und zu ähnliche Beschreibungen der Views würden den Vergleich in ein falsches Licht rücken.

5.1.1. Zyklomatische Komplexität

Die zyklomatische Komplexität gibt die Anzahl der Entscheidungslogiken eines Softwaremoduls an [42]. Sie steht in engem Verhältnis zu Test- und Wartbarkeit des Moduls [43].

„Um ein Programm zu testen, muß es mindestens so viele verschiedene Testfälle geben, wie es Pfade durch das Programm gibt, damit jeder Pfad mindestens einmal durchlaufen wird. Für die Wartbarkeit wird allgemein angenommen, daß der Wartungsaufwand in enger Beziehung zu den Pfaden eines Programms steht. Anschaulich kann man sagen, daß die Anzahl der Pfade den Wartungsaufwand bestimmt, da bei Änderungen um so mehr Abhängigkeiten beachtet werden müssen, je mehr Pfade es gibt.“ [43]. Dem Zitat ist hinzuzufügen, dass hier Pfade als Synonym für Zweige verwendet werden. Demnach ist hier von Zweigabdeckung die Rede. Vollständige Pfadabdeckung ist, vorallem in Modulen mit Schleifen, unrealistisch.

Die Berechnung der zyklomatischen Komplexität pro Methode ist von der Programmiersprache (Welche Schlüsselwörter bezüglich Entscheidungslogik gibt es?) und dem verwendeten Werkzeug zur Berechnung (Welche Schlüsselwörter werden als Entscheidungslogik gezählt?) abhängig.

Generell gilt folgende Formel zur Berechnung:

$$Zyklomatische\ Komplexitaet = 1 + Anzahl\ der\ Entscheidungslogiken$$

Laut ursprünglicher Definition der zyklomatischen Komplexität stellt die Metrik die Anzahl der linear unabhängigen Pfade im Programm dar [11]. Diese Definition resultiert bei der Berechnung im Zählen der Entscheidungslogiken und dem Hinzuaddieren von eins. Die eins repräsentiert den bei jeder Methode vorhandenen ersten Pfad im Code.

Bei der Berechnung werden in dieser Arbeit folgende JavaScript-Schlüsselwörter als Entscheidungslogik gezählt: *if*, *?*, *while*, *dowhile*, *for*, *forin*, *switch*, *&&*, ||, *catch*.

Es folgt ein Beispiel zur Veranschaulichung der Komplexitätsberechnung. Innerhalb der Funktion *foo* in Listing 5.1 befinden sich zwei Entscheidungslogiken: *if* und ||. Es wird eins hinzu addiert und somit besitzt *foo* eine zyklomatische Komplexität von drei.

```
1 function foo(a, b) {
2     if (a || b) {
3         return 1;
4     }
5     return 2;
6 }
```

Listing 5.1: Beispiel für eine Funktion mit der zyklomatischen Komplexität von drei

Es mag auffallen, dass bei den Schlüsselwörtern, die als Entscheidungslogik zählen, das *case* fehlt, *switch* jedoch aufgezählt wurde. Die Aussagekraft bezüglich Wartbarkeit beziehungsweise Verständlichkeit anhand der zyklomatischen Komplexität kann durch triviale *switchcase*-Blöcke verringert werden.

Angenommen, ein *switch* würde nicht zählen, dafür aber jedes *case*. Folgende Funktion hätte demnach die Komplexität von vier.

```
1 function foo(a, b) {
2     switch (a) {
3         case 1: return 1;
4         case 2: return 2;
5         case 3: return 3;
6     }
7 }
```

Listing 5.2: Verminderung der Aussagekraft bezüglich Wartbarkeit durch trivialen *switchcase*-Block

Somit wäre obige (siehe Listing 5.2) Funktion, von der Komplexität her gleich auf mit der deutlich schwerer zu verstehenden Funktion unten (siehe Listing 5.3).

```
function foo(a, b) {
    if (a >=3 ) {
        return 1;
    } else if (b > 2) {
        return 2;
    } else if (a < 1) {
        return 3;
    }
}
```

Listing 5.3: Gegenüberstellung zu trivialem *switchcase*-Block in Listing 5.2 bezüglich Aussagekraft des berechneten Komplexitätswertes.

Bei beiden Lösungen, alter und re-implementierter, sind nur triviale *switchcase*-Blöcke zu finden. Aus diesem Grund wird bei dieser Analyse die *switchcase*-Blöcke nur als „+1“ bezüglich der Komplexität des Moduls gezählt.

Somit wäre geklärt, wie sich die zyklomatische Komplexität einer Methode bei dieser Analyse berechnet. Im Anhang wird die Komplexität aber als Aggregat[1] auf Datei-Ebene angegeben, um den Vergleich der Controller (alter versus reimplementierter) besser ziehen zu können.

Der Inhalt einer solchen Datei könnte wie Listing 5.4 aussehen:

```
sap.ui.define([], function() {
    "use strict";

    return {
        foo: function() {
            return 0;
        },
        bar: function(a) {
            if(a){
                return 1;
            }
            return 2;
        }
    };
});
```

Listing 5.4: Beispiel zur Berechnung der aggregierten zyklomatischen Komplexität

Die anonyme Funktion in der ersten Zeile besitzt eine Komplexität von eins. Sie gibt lediglich ein Objekt in JSON-Notation zurück. Dieses Objekt besitzt zwei weitere

[1] Für die gesamte Komplexitätsberechnung wird die NodeJS Bibliothek complexity-report verwendet. Unter folgendem Link erklärt ein zuständiger Entwickler in eigenen Worten, wie das Komplexitäts-Aggregat berechnet wird: https://github.com/philbooth/complexity-report/issues/30

Funktionen. Die Funktion *foo* besitzt eine Komplexität von eins und die Funktion *bar* eine von zwei.

Zu der bisherigen Komplexität der anonymen Funktion werden die Komplexitäten aller inneren Funktionen hinzu addiert. Die Komplexitätswerte jeder inneren Funktion werden vor der Aufaddierung jedoch um eins verringert. Somit wird die aggregierte Komplexität der anonymen Funktion durch das „Aufklappen“ jeder weiteren inneren Funktion berechnet. Demnach ergibt sich ein Komplexitätsaggregat von zwei für das Beispiel in Listing 5.4. Die definierte Regel zur Berechnung der Komplexität pro Modul - in diesem Fall ist mit Modul die ganze Datei gemeint - gilt also weiterhin: eins zu der Anzahl der als Entscheidungslogik definierten Schlüsselwörter addieren.

Durch simple Addition der Komplexitäten jeder einzelnen Funktion würde das Aggregat verfälscht werden. Das standardmäßige Hinzuaddieren von eins würde bewirken, dass jede Funktionsdefinition auch als Entscheidungslogik-Schlüsselwort gelten würde. Somit wäre die Definition der zyklomatischen Komplexität, nämlich dass sie die Anzahl der Entscheidungslogiken eines Moduls widerspiegelt, nicht mehr erfüllt.

5.1.2. Code-Zeilen

Anhand der Code-Zeilen lässt sich der Einfluss der Eliminierung von Duplizierungen beziehungsweise die Refactoring-Phase des TDD-Mantras bewerten. Leerzeilen werden bei dieser Zählung ignoriert.

Es wird zwischen physischer- und logischer Code-Zeile unterschieden. Mit physischen Code-Zeilen ist jede nicht leere Zeile Code gemeint, die Kommentarzeilen inbegriffen. Im Gegensatz zur physischen-, werden bei der logischen Code-Zeile ausschließlich die Anweisungen gezählt.

Folgendes Code-Beispiel in Listing 5.5 hätten demnach neun physische

```
1 // this is a comment
2 function foo() {
3  this.is()
4  .chaining();
5
6  var a = 1; var b = 1 + 1; var c= 1 + 1 + 1;
7
8  if (foo) {
```

```
9   this.is().chaining();this.is().chaining();
10  }
11 }
```

Listing 5.5: Beispiel für das Zählen physischer Zeilen

und sieben logische Code-Zeilen. Bei der Zählung der logischen Zeilen (siehe Listing 5.6) wird eine Anweisung, die sich über mehrere Zeilen erstreckt (siehe Zeile 3 und 4), als eine Zeile gezählt. Mehrere Anweisungen (siehe Zeile 6 und 9) in einer Zeile werden separat gezählt.

```
1  // this is a comment
2  function foo() {
3   this.is()
4   .chaining();
5
6   var a = 1;  var b = 1 + 1;  var c = 1 + 1 + 1;
7
8   if (foo) {
9   this.is().chaining();  this.is().chaining();
10      }
11 }
```

Listing 5.6: Beispiel für das Zählen logischer Zeilen

Die physische Zeilenzählung birgt die Problematik, dass Kommentare und alleinstehende schließende Klammern gezählt werden. Die Information über mehrere Anweisungen in einer Zeile geht verloren.

Aufgrund dieser Problematiken wird die Analyse anhand der Anzahl der logischen Zeilen durchgeführt. Sie begegnet den Schwachstellen der physischen Zeilenzählung und gibt so ein unverfälschteres Bild der Software wider. Sie ignoriert die Kommentare und zählt ausschließlich die Anzahl der Anweisungen, ungeachtet dessen, in welcher Zeile diese stehen.

5.2. Eigene Beobachtungen

Bei den eigenen Beobachtungen werden die Erwartungen aus dem Kapitel *3. Erwartungen an Test Driven Development* während der Re-Implementierung durch Selbstreflexion und gemachte Erfahrungen untersucht. Aufbauend auf den gemachten Erfahrungen und gewonnen Erkenntnissen wird im Kapitel *6. Auswertung der*

Analyseergebnisse in Bezug auf die Erwartungen auf deren Erfüllung eingegangen. Zum Teil wurden gemachte Erfahrungen formlos als Notizen festgehalten. Eine Sammlung dieser Notizen ist dem *Anhang: Notizen zu eigenen Beobachtungen* beigefügt.

5.3. Fallstudie mit Entwicklern

Nicht alle Erwartungen können subjektiv vom Autor selbst evaluiert werden. Durch die Involvierung Dritter und Durchführung einer Fallstudie können explizit die Erwartungen evaluiert werden, die sich auf Dokumentation und Wartbarkeit der Anwendung beziehen. Die Fallstudie wird mit zwei Entwicklern durchgeführt und dient einer punktuellen Untersuchung. Bei der Fallstudie werden den zwei Probanden Aufgaben gestellt, die sie in der alten und der re-implementierten Trainer Cloud App lösen sollen.

Damit die Fallstudie Aussagekraft besitzt, müssen bestimmte Rahmenbedingungen eingehalten werden. Keiner der Probanden darf die alte Lösung der Trainer Cloud App kennen. Die Probanden müssen Grundkenntnisse des gesamten Technologie-Stacks[2] der Trainer Cloud App mitbringen. Um den Vergleich zwischen herkömmlicher- und testgetriebener Programmierung während der Fallstudie ziehen zu können, darf nur innerhalb der neuen Lösung testgetrieben entwickelt werden.

Bevor die Probanden mit der Bearbeitung der Aufgaben beginnen, erhält jeder eine kurze Einführung in TDD und einen Überblick über die Projektstruktur der Trainer Cloud App.

Während der Durchführung der Fallstudie werden formlos Notizen festgehalten. Die Notizen orientieren sich an folgenden Fragen:

- Waren die Tests hilfreich, um die einzelnen Module zu verstehen?
- Wurden die Tests während Änderungen am Program-Code entsprechend angepasst?
- Wie wurde geprüft, ob Änderungen erfolgreich waren?
- Wurde der TDD-Zyklus erfolgreich umgesetzt?
- Wie lange betrug die Bearbeitungsdauer der einzelnen Aufgaben?

[2] Die komplette Beschreibung des Technologie-Stacks ist im Kapitel *4.1 Entwicklungsumgebung zur testgetriebenen Entwicklung* zu finden.

- Wie oft und zu was wurde auf Nachfrage der Probanden Hilfestellung gegeben?
- Welche Methodik ziehen die Probanden vor und warum?

Die Notizen der formlosen Beobachtungen sind im *Anhang: Notizen der Fallstudie mit Entwicklern* zu finden. Den Probanden werden in Summe drei Aufgaben gestellt. Die Aufgaben sind nach Schwierigkeitsgrad sortiert. Mit der einfachsten Aufgabe wird begonnen. Die Aufgaben werden bei der alten Trainer Cloud App mittels integrierter Entwicklungsumgebung auf der SCP und vorbereiteten Testdaten im Backend bearbeitet. Zur Bearbeitung der Aufgaben innerhalb der neuen Lösung wird die Entwicklungsumgebung aus Kapitel *4.1 Entwicklungsumgebung zur testgetriebenen Entwicklung* gemeinsam aufgesetzt.

5.3.1. Aufgabe: Refactoring

Die Aufgabe mit dem niedrigsten Schwierigkeitsgrad ist die Refactoring-Aufgabe. Sie bezieht sich auf die Formatierungsfunktion der Benutzernamen für die Beurteilungen einer jeden Schulung. Ziel der Aufgabe ist es, eine künstlich erhöhte zyklomatische Komplexität der Formatierungsfunktion durch Refactoring zu verringern.

In der neuen Lösung ist die Funktion *formatSurveyUser* im *Detail.controller.js* zu finden. Die alte Lösung stellt die gleiche Funktionalität im *Formatter.js* durch die Funktion *constructSurveyUserString* bereit. Für die Refactoring-Aufgabe werden beide Implementierungen dahingehend verändert, dass sie eine höhere zyklomatische Komplexität besitzen. Die abgeänderte Implementierung in beiden Lösungen sieht wie Listing 5.7 aus:

```
1  function(amount, base) {
2      if (amount < 1) {
3          throw "to invalid parameters";
4      }
5
6      if (amount > 999) {
7          throw "invalid parameters";
8
9      }
10
11     var i = 0;
12     for (; i < amount; i++) {}
13
14     var startUser = base + "-001";
```

```
15      var endUser = "";
16
17      if(i < 10) {
18          endUser = base + "-00" + i;
19      } else if (i < 100) {
20          endUser = base + "-0" + i;
21      } else if (i < 1000) {
22          endUser = base + "-" + i;
23      }
24      return startUser + " to\n" + endUser;
25  }
```

Listing 5.7: Funktion deren Komplexität in der Refactoring-Aufgabe auf zwei gebracht werden muss

Die Aufgabe gilt als erfüllt, sobald eine Komplexität von drei erreicht ist. Die geforderte zyklomatische Komplexität kann durch folgende Funktion in Listing 5.8 erreicht werden:

```
1  function(amount, base) {
2      if (amount >= 1 && amount <= 999) {
3          return base + "-001 " + this.i18n("detail-surveys-users-to")
4                  + "\n"
5                  + base + "-" + ("000" + amount).slice(-3);
6      }
7      throw "invalid parameters";
8  }
```

Listing 5.8: Funktion die bei der Refactoring-Aufgabe die geforderte zyklomatische Komplexität liefert

	Anzupassende Dateien	**Aufgabe ist erfüllt, wenn**
Alte Lösung	Formatter.js, Funktion: constructSurveyUserString	eine zyklomatische Komplexität der Funktion von drei erreicht ist
Re-implementierte Lösung	Detail.controller.js, Funktion: formatSurveyUser	

Tabelle 5.1.: Zusammenfassung der Refactoring-Aufgabe

5.3.2. Aufgabe: Änderung

Nach der Refactoring-Aufgabe müssen die Probanden bestehende Funktionalität abändern und eine Veränderung des externen Verhaltens herbeiführen. Es geht um den Titel der *Master.view.xml*. Vor der Änderung beinhaltet dieser noch die Benutzer-ID des aktuell angemeldeten Benutzers. Nach der Änderung soll keine Benutzer-ID mehr angezeigt werden. Abbildung 5.1 zeigt die zu entfernende Benutzer-ID.

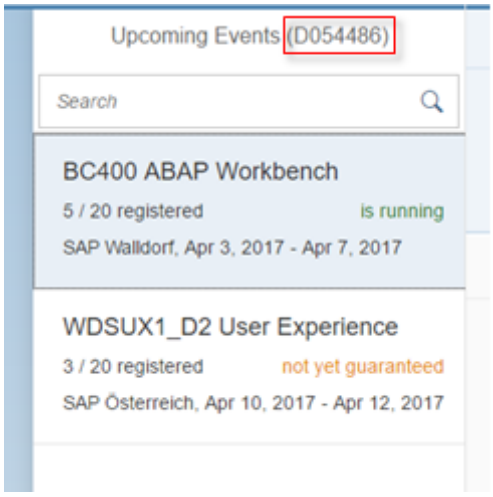

Abbildung 5.1.: Hinweis zur Änderungs-Aufgabe

In beiden Lösungen muss die *Master.view.xml* abgeändert werden. Die Titeländerungen werden in beiden Lösung im *Master.controller.js* in der Funktion *onSearch* angestoßen.

	Anzupassende Dateien	**Aufgabe ist erfüllt, wenn**
Alte Lösung	Master.controller.js	der Titel keine Benutzer-ID mehr enthält
	Master.view.xml	
Re-implementierte Lösung	Master.controller.js	
	Master.view.xml	

Tabelle 5.2.: Zusammenfassung der Änderungs-Aufgabe

5.3.3. Aufgabe: Erweiterung

Die letzte und schwerste Aufgabe befasst sich mit der Einführung einer neuen Funktionalität. Es sollen Start- und End-Uhrzeit in der *Master.view.xml* für jede Schulung angezeigt werden. Der Webservice liefert diese Informationen zu jedem Event in Form eines UTC-Zeitstempels. Die Testdaten wurden in der re-implementierten Version entsprechend angepasst.

Abbildung 5.2 zeigt, wo Start- und End-Uhrzeit auf der Benutzeroberfläche erscheinen müssen.

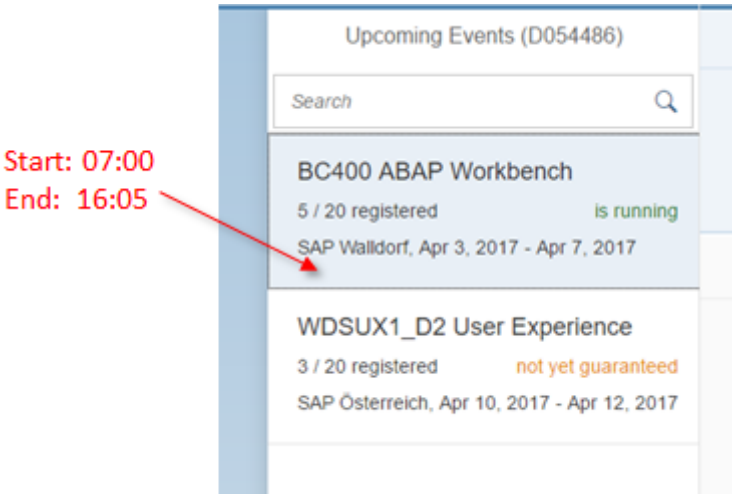

Abbildung 5.2.: Hinweis zur Erweiterungs-Aufgabe

Es soll eine Formatierungsfunktion *formatStartEnd* geschrieben werden. Diese Funktion formatiert den UTC-Zeitstempel in folgendes Format: HH:MM. Führende Nullen sollen bei einstelligen Stunden- oder Minuten-Werten verwendet werden. Für die Zeitberechnungen darf ausschließlich die native *Date*-Klasse und deren Methoden *getHours* sowie *getMinutes* verwendet werden. Die Formatierungsfunktion soll in der alten Lösung im *Formatter.js* und in der neuen Lösung im *Master.controller.js* eingebaut werden. Zusätzlich müssen die Dateien zur Internationalisierung (*i18n*) um entsprechende Bezeichner erweitert werden.

Zum Schluss muss in der *Master.view.xml* ein neues *ObjectAttribute* zum *ObjectListItem* mit entsprechendem Binding hinzugefügt werden.

	Anzupassende Dateien	**Aufgabe ist erfüllt, wenn**
Alte Lösung	Formatter.js, Funktion: formatTimestamp	die Start- und End-Uhrzeit für jede Schulung im Format HH:MM mit Bezeichnern für die englische Sprache angezeigt wird.
	Master.view.xml	
	messageBundle.properties	
Re-implementierte Lösung	Master.controller.js; Funktion: formatTimestamp	
	Master.view.xml	
	i18n_en.properties	

Tabelle 5.3.: Zusammenfassung der Erweiterungs-Aufgabe

Kapitel 6

Auswertung der Analyseergebnisse in Bezug auf die Erwartungen

In diesem Kapitel werden die Ergebnisse der durchgeführten Analysen behandelt. Die Auswertungen sind der jeweiligen Analysemethode zugeordnet. Jede Erwartung wird mindestens einmal durch eine Analysemethode überprüft.

6.1. Softwaremetriken

Anhand der berechneten Softwaremetriken wird folgende Erwartung beurteilt: (EI4) Die Qualität des Codes nimmt zu. Abbildung 6.1 und 6.2 stellt die Auswertungen der Metriken in Form von Balkendiagrammen dar. Die Ergebnisse der Auswertungen auf Dateiebene kann dem *Anhang: Metriken* entnommen werden.

Die alte Trainer Cloud App besteht in Summe aus 548 logischen Code-Zeilen. Im Durchschnitt sind das 68,50 Code-Zeilen pro JavaScript-Datei. Über alle JavaScript Dateien besteht eine aggregierte Komplexität von 77 und ein Durchschnitt von 9,63.

Die re-implementierten Trainer Cloud App besteht in Summe aus 165 logischen Code-Zeilen. Im Durchschnitt sind das 23,57 Code-Zeilen pro JavaScript-Datei. Über alle JavaScript Dateien besteht eine aggregierte Komplexität von 24 und ein Durchschnitt von 3,43.

Ein Vergleich der Daten lässt schließen, dass durch die Re-Implementierung getrieben von Tests 69,89% der logischen Code-Zeilen und 68,83% der logischen Entscheidungen eingespart wurden. Anders ausgedrückt: In der alten Trainer Cloud

App befinden sich 3,32x mehr logische Code-Zeilen und 3,20x mehr Entscheidungslogiken.

Um herauszufinden, wieso eine solch drastische Einsparung erzielt wurde, muss die alte Trainer Cloud App im Detail untersucht werden. Durch die Inspektion fällt auf, dass die JavaScript-Datei *logon.js*, mit ihren 109 logischen Code-Zeilen und einer aggregierten Komplexität von sieben, ausschließlich unbenutzten Code beinhaltet. Die Datei wurde im Zuge der Verwendung einer Master-Detail-Vorlage zu Beginn der Entwicklung erstellt und seitdem nicht mehr aus dem Projekt entfernt. Somit sind 19,89% der logischen Code-Zeilen 9,09% der logischen Entscheidungen in der alten Trainer Cloud App überflüssig. Diese Inspektion wirft die Frage auf, ob der „tote“ Code während einer Refactoring-Phase aufgefallen wäre?

Ein aussagekräftiger Vergleich auf JavaScript-Datei-Ebene kann zwischen den Controllern beider Lösungen gezogen werden. Die Zuständigkeiten sind aufgrund der selben Funktionalität gleich geblieben. Neben den Controllern *Master.controller.js* und *Detail.controller.js* existiert in beiden Anwendungen ein generischer Controller: *Controller.js* in der alten und *Base.controller.js* in der re-implementierten Lösung. Hinzu kommt die Tatsache, dass der Code der Aufgaben des *Formatter.js* innerhalb der alten Trainer Cloud App in die jeweiligen Controller der re-implementierten Lösung gewandert ist. Für den Vergleich der Controller wird eine Summe der Metriken über die Controller gebildet. Bei der alten Lösung wird bei der Summierung der *Formatter.js* neben den Controllern miteinberechnet.

Eine Summe über die Controller und dem *Formatter.js* ergibt in der alten Lösung 301 logische Code-Zeilen und eine aggregierte Komplexität von 56. Die Summe über die Controller der re-implementierten Lösung ergibt 138 logische Code-Zeilen und eine aggregierte Komplexität von 20. Daraus ergibt sich, dass sich bezogen auf Logik der Controller in der re-implementierten Lösung 54,16% weniger logische Code-Zeilen und 60,71% weniger Entscheidungslogiken befinden.

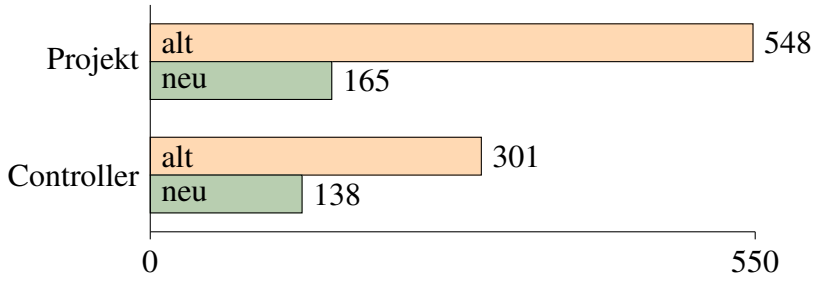

Abbildung 6.1.: Vergleich logische Code-Zeilen

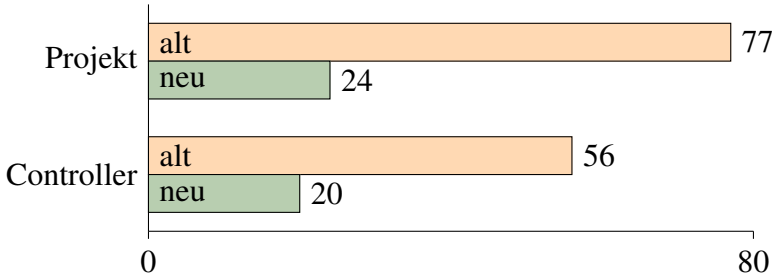

Abbildung 6.2.: Vergleich zyklomatische Komplexität

Aus dem Vergleich der Metriken über die Gesamtheit der Lösungen, die Inspektion der alten Trainer Cloud App und der Vergleich der Controller lässt sich schließen: Die Qualität des Codes ist im Bezug auf Code-Zeilen und zyklomatischer Komplexität deutlich besser geworden. Somit gilt die Erwartung (EI4) als erfüllt.

6.2. Eigene Beobachtungen

Die subjektive Bewertung geschieht nur im Kontext der Re-Implementierung der Trainer Cloud App und basiert auf eigenen Beobachtungen.

(EI1) Der TDD-Zyklus erzwingt die Aufteilung des Problems in kleine Teilprobleme. Man muss sich der Lösung in überschaubaren Inkrementen annähern. Die *Definition der Vorgehensweise* des TDD-Zyklus impliziert, dass immer nur an einem Test und dem damit verbundenen Inkrement gearbeitet werden kann. Größere Probleme werden demnach durch kleinere Inkremente gelöst. Die praktizierte Variante von TDD während der Re-Implementierung macht hier einen Abstrich. Sofern es sich bei dem zu implementierenden Inkrement um UI-Funktionalität handelte, wurden zwei Tests benötigt: ein UI- und ein Unit-Test. Nichtsdestotrotz arbeitete man an einem Inkrement, das zur Lösung des großen Ganzen beiträgt.
$\Rightarrow$ Erwartung erfüllt

(EI2) Die Verwendung einer To-do-Liste unterstützt den uneingeschränkten Fokus auf die jeweilige Entwicklungseinheit. Durch die Refactoring-Phase des TDD-Zyklus tauchten nicht auf das aktuelle Inkrement bezogene To-do's auf. Das Führen einer To-do-Liste half während der Re-Implementierung, dass Dinge nicht in Vergessenheit geraten. Sobald das To-do auf der Liste vermerkt wurde, musste

kein Gedanke mehr daran verschwendet werden und der Fokus konnte wieder komplett auf das Inkrement gerichtet werden. Die Abarbeitung der Punkte führte vor Augen, was am entsprechenden Tag geleistet wurde. Nach einer längeren Abwesenheit half die To-do-Liste, wieder den Anschluss zu finden. Zu welchem Zeitpunkt wurde die Arbeit niedergelegt? An welchem Punkt muss die Arbeit wieder aufgenommen werden?
⇒ Erwartung erfüllt

(EI3) Die stetige Test-Feedback-Schleife generiert Überzeugung von funktionierender Software. Der Durchlauf eines Tests spiegelte die Korrektheit des Inkrements wider. Neue Inkremente kamen hinzu, sofern die bisherigen Tests durchliefen. Feedback zu allen Inkrementen konnte innerhalb weniger Sekunden eingeholt werden. Längere Abwesenheit und das damit ausbleibende Feedback der Tests ließen die Überzeugung schwinden. Um nach längerer Abwesenheit wieder Überzeugung zu generieren, wurden vor Wiederaufnahme der Tätigkeiten alle Test ausgeführt. Konstantes positives Feedback wirkte sich jedoch nicht nur auf die Überzeugung von funktionierender Software aus. Der Psychologe Alfred Bandura entwickelte in den 1970er Jahren das Konzept der Selbstwirksamkeitserwartung [44]. Die Selbstwirksamkeitserwartung beschreibt die Überzeugung von den eigenen Fähigkeiten. Einflussreich zur Steigerung der Selbstwirksamkeit sind Erfahrungen, in denen Leistungserfüllungen eingetroffen sind. Während der testgetriebenen Entwicklung wurde nach jedem Durchlauf der Tests signalisiert, dass erfolgreich Leistung erbracht wurde. Demnach wird durch die Tests nicht nur das Vertrauen in die Lösung gestärkt, sondern auch die Überzeugung des Programmierers im Bezug auf seine eigenen Fähigkeiten.
⇒ Erwartung erfüllt

(EI5) Es entsteht testbare Software. Aufgrund des TDD-Zyklus bestand der Anspruch, dass jedem entwickelten Modul ein implementierter Test vorausgeht. Aufbauend auf diesem Anspruch entstanden implizit testbare Module. Die Module wurden so entworfen, dass wenig Abhängigkeiten imitiert werden mussten und der Test möglichst kurz und prägnant implementiert werden konnte.
⇒ Erwartung erfüllt

(EI6) Fehler fallen früher auf und sind deswegen günstiger in deren Behebung. Dadurch, dass sich der Umfang der Anwendung begleitend zu den Tests entwickelte, herrschte eine konstant hohe Testabdeckung. Hohe Testabdeckung und die Eigenschaft der Automatisierung ließen schnelle und ausgiebige Testphasen zu. Bevor der Programm-Code theoretisch in die Produktion gelangt, kann nun das gesamte Projekt selbstständig durch den Entwickler automatisiert und in kurzer Zeit getestet werden. Während der Re-Implementierung kam es vor, dass durch Umbauarbeiten Logik an anderen Stellen nicht mehr funktionierte. Geprüft wurde umgehend nach den Umbauarbeiten, also vor dem Transport in die Produktion. Demnach fielen die Fehler frühzeitig auf. Die Behebung geschah ausschließlich durch den Entwickler. Somit wurde keine weitere Zeit von zusätzlichen Personen (Tester, Support) in Anspruch genommen, die bezahlt werden müssten.
⇒ Erwartung erfüllt

(EE1) Die Qualität des Designs fällt durch Benutzung der eigenen Schnittstellen im Test-Code auf. Das OpenUI5-Framework bringt wiederverwendbare MVC-Architekturbausteine mit. Während der Re-Implementierung bewegte man sich im Rahmen dieser Bausteine. Spezifische Probleme, die ein eigenes Design benötigten, sind nicht aufgetreten.
⇒ Erwartung konnte nicht evaluiert werden

(EE2) Ausgiebige Verfeinerungen des Designs während der Konzeption werden vermieden. Das Design wird, falls nötig, während der Implementierung in der Refactoring-Phase angepasst. Das Design reift mit dem Fortschritt des Programms. Ganz konkret sind im Zuge der Refactoring-Phase Duplizierungen im Programm-Code aufgefallen, die durch Umbauarbeiten entfernt wurden. Beispiel hierfür ist der Controller *Base.controller.js*, von dem der *Master.controller.js* und *Detail.controller.js* ableiten. Diese Designentscheidung ist erst während der Re-Implementierung gefallen.
⇒ Erwartung erfüllt

(ET1) Weniger Fehler durch vorhandene Tests. Bei dieser Erwartung wird Bezug auf die vergangene Implementierung der alten Trainer Cloud App und das Auftauchen von Fehlern durch manuelle Tests genommen. Während der Re-Implementierung ist es nur einmal vorgekommen, dass trotz vorhandener Tests die gewünschte

Funktionalität nicht gegeben war. Das Fehlverhalten ist durch einen manuellen Test aufgefallen. Im Vergleich zur Ansammlung von Fehlern während der Implementierung der alten Trainer Cloud App ist dies eine drastische Senkung. Es liegen keine Zahlen vor, wie viele Fehler bei der Implementierung der alten Lösung aufgetreten sind. Jedoch hat der Autor bei der alten Lösung mitentwickelt und macht dadurch einen Vergleich möglich.
⇒ Erwartung erfüllt

(EL1) Durch die Absicherung von automatisierten Tests entstehen kürzere Lieferzyklen. Die Re-Implementierung wurde zu keinem Zeitpunkt ausgeliefert.
⇒ Erwartung konnte nicht evaluiert werden

(ES1) Erhöhter Aufwand für das Aufsetzen der Entwicklungsumgebung. Beim initialen Aufsetzen des Projekts musste eine Entwicklungsumgebung konzipiert und realisiert werden, in der UI- und Unit-Tests automatisch in wenigen Sekunden durchlaufen können. Diese Konzeptionierung kostete Zeit. Bei der Re-Implementierung wurden dafür rund zehn Personentage benötigt.
⇒ Erwartete Schwierigkeit ist eingetreten

(ES2) Zusätzliche Kosten für das Projekt aufgrund des erhöhten Aufwands für das Schreiben der Tests. Die Formulierung der Tests kostete Zeit. Bei dieser Re-Implementierung kommt hinzu, dass UI-Tests für die Mobile- und Desktop-Ansicht geschrieben wurden. Erhöhter Aufwand ist mit mehr Kosten verbunden.
⇒ Erwartete Schwierigkeit ist eingetreten

(ES3) Viele Mock-Objekte wirken sich negativ auf die Lesbarkeit der Tests aus. Viele Controller-Methoden haben Seiteneffekte. Diese Seiteneffekte mussten imitiert werden. Die Imitierung der Abhängigkeiten im Test-Code verschleiert die Absicht des Tests und führt zu mehr Komplexität bezüglich des Verständnisses der Tests.
⇒ Erwartete Schwierigkeit ist eingetreten

(ES4) Wenn man nicht weiß, welche Abhängigkeiten für das gewünschte Verhalten imitiert werden müssen, ist es schwierig einen Test zu verfassen. Tests

für Methoden mit Seiteneffekten ließen sich nur schreiben, sofern Kenntnisse über die zu imitierenden Abhängigkeiten vorhanden waren. War dies nicht der Fall entstand ein sogenannter „Deadlock“. Die Tests konnten nicht ohne Ausprobieren, ob die angesprochene Schnittstelle den gewünschten Seiteneffekt bewirkt, formuliert werden und Programm-Code durfte wiederum nicht ohne Tests geschrieben werden. In diesem Fall musste mit dem TDD-Mantra kurzzeitig gebrochen werden und Programm-Code zur Überprüfung des Seiteneffekts verfasst werden. Nur so konnte das Verhalten der Anwendung bei Benutzung einer bestimmten Schnittstelle verifiziert werden. Sobald die gewünschte Schnittstelle gefunden war, wurde der entstandene Programm-Code wieder konsequent entfernt und die erneute Entwicklung des Moduls durch einen Test getrieben werden.
⇒ Erwartete Schwierigkeit ist eingetreten

(ES5) Duplizierungen sowie Abhängigkeiten zwischen Testdaten und UI-Tests. Während der Re-Implementierung entstanden Duplizierungen zwischen Testdaten und UI-Tests. Diese Abhängigkeiten können nur mit nicht vertretbarem Aufwand, der Einführung zusätzlicher Werkzeuge und einer gesteigerten Komplexität der Testumgebung eliminiert werden. Aufgrund dessen blieben diese Abhängigkeiten im Projekt bestehen.
⇒ Erwartete Schwierigkeit ist eingetreten

6.3. Fallstudie mit Entwicklern

Fokus der Fallstudie lag auf der Evaluierung der Erwartungen bezüglich Dokumentation und Wartung. Im Laufe der Fallstudie zeigte sich, dass anhand der definierten Aufgaben die Erwartungen nur teilweise evaluiert werden können. Nichtsdestotrotz sind interessante Punkte zu TDD aufgekommen, die im Anschluss zur Evaluierung der Erwartungen behandelt werden.

Die Fallstudie wurde mit zwei Probanden durchgeführt. Die in diesem Kapitel ausgewerteten Notizen sind im *Anhang: Notizen der Fallstudie mit Entwicklern* beigefügt. Um die Ergebnisse der Probanden differenziert betrachten zu können, folgen zunächst Informationen zu den Probanden und den Umständen, in denen die Fallstudie mit den Probanden durchgeführt wurde.

Die erste Fallstudie wurde am 02.06.2017 von 14:00 bis 17:00 durchgeführt. Zu Proband 1 wurden folgende Informationen notiert:

- verfügt über sehr gute OpenUI5-Kenntnisse
- hat kurzzeitig innerhalb seiner Abteilung testgetrieben entwickelt
- wirkt entspannt und konzentriert
- schreibt regelmäßig Unit-Tests
- ist das Mocking-Framework SinonJS nicht bekannt

Die zweite Fallstudie wurde am 07.06.2017 von 20:00 bis 23:00 durchgeführt. Zu Proband 2 wurden folgende Informationen notiert:

- verfügt über rudimentäre OpenUI5-Kenntnisse
- ist der Begriff TDD bekannt
- gegen Ende der Studie ließ die Konzentration nach
- schreibt unregelmäßig Unit-Tests
- ist das Mocking-Framework SinonJS nicht bekannt

(ED1) Begleitend zur Entwicklung entstehen kompilierbare Code-Beispiele die zum Verständnis bezüglich Benutzung der eigenen Schnittstellen beitragen. Die in der Fallstudie gestellten Aufgaben sehen keine komplizierte Benutzung der OpenUI5-Schnittstelle vor.
⇒ Erwartung konnte nicht evaluiert werden

(ED2) Die Tests dokumentieren die Zuständigkeiten einzelner Einheiten. Dadurch, dass die Aufgaben zuerst in der alten und anschließend in der neuen Lösung bearbeitet wurden, waren die Zuständigkeiten der zu verstehenden Einheiten schon bekannt. Kein Proband hat versucht, Einheiten, abgesehen von den zu verändernden, zu verstehen.
⇒ Erwartung konnte nicht evaluiert werden

(ED3) Die Tests repräsentieren stetig aktuelle Programm-Dokumentation. In dieser Arbeit konnte nicht evaluiert werden, ob die Tests Programm-Dokumentation repräsentieren. Falls sie jedoch eine Form von Dokumentation darstellen, wurde folgendes während der Fallstudie beobachtet: Jeder Proband war aufgrund des TDD-

Zyklus gezwungen, die Tests aktuell zu halten. Demnach lässt sich eine Aussage über die Aktualität der Tests treffen.
⇒ Erwartung erfüllt

(EW1) Die Code-Beispiele innerhalb der Tests helfen den Programm-Code zu verstehen. Bei der re-implementierte Trainer Cloud App handelt es sich um eine Anwendung mit UI. Um Änderungen innerhalb des UIs herbeizuführen, müssen Methoden mit Abhängigkeiten und Seiteneffekten aufgerufen werden. Tests solcher Methoden besitzen demnach viele Mocks (Abhängigkeiten imitierende Logik). Die Probanden, welche den Code der Anwendung zuvor noch nie gesehen hatten, taten sich schwer, den Test-Code zu verstehen.
⇒ Erwartung nicht erfüllt

Anhand der erhobenen Daten kann auf eine weitere erwartete Schwierigkeit eingegangen werden.

(ES1) Erhöhter Aufwand für das Aufsetzen der Entwicklungsumgebung. Das Eintreten der Schwierigkeit wurde im Kapitel *6.2 Eigene Beobachtungen* verifiziert und bestätigt. Mit den Zeitmessungen zur Bearbeitungsdauer (siehe Abbildungen: 6.3 und 6.4) kann dies ebenfalls durch Zahlen belegt werden. Bis auf die Refactoring-Aufgabe, bei der kein Test angepasst oder verfasst werden muss, wurde bei der Änderungs- und Erweiterungsaufgabe deutlich mehr Zeit benötigt. Anzumerken ist, dass die Probanden die jeweilige Aufgabe innerhalb der neuen Lösung bereits in der alten bewältigt hatten. Demnach herrschte ein gewisses Vorwissen bezüglich der Lösung der Aufgaben. Es stellt sich die Frage: Wieso benötigten die Probanden innerhalb der neuen Lösung mehr Zeit, obwohl die zyklomatische Komplexität geringer ist, die Code-Zeilen weniger sind und ein Vorwissen bezüglich der Lösung vorhanden war? In der alten Lösung wurde, wann immer es den Probanden sinnvoll erschien, ad-hoc getestet. Diese Ad-hoc-Tests konnten mit minimalem Aufwand durchgeführt werden. Demzufolge wurde in der alten Lösung ebenso getestet, jedoch nicht nachhaltig und umfassend. Im Gegensatz hierzu wurde in der neuen Lösung, durch den TDD-Zyklus vorgeschrieben, zusätzliche Zeit für automatisierte Tests investiert. Zudem kann die erhöhte Differenz der Bearbeitungsdauer auch durch den initialen Einlernaufwand in die Testumgebung und das Mocking-Framework entstanden sein. Es besteht die Möglichkeit, dass sich die Dif-

ferenz senkt, sobald sich die Probanden in die Lösung, die Testumgebung und in das Mocking-Framework eingearbeitet haben.

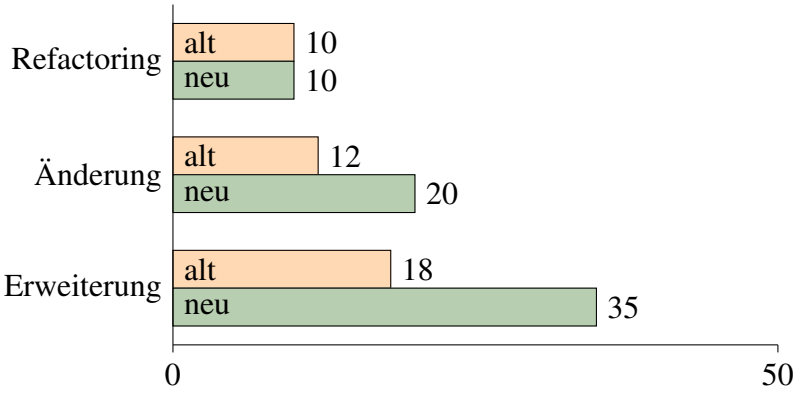

Abbildung 6.3.: Bearbeitungsdauer der Fallstudie in Minuten (Proband 1)

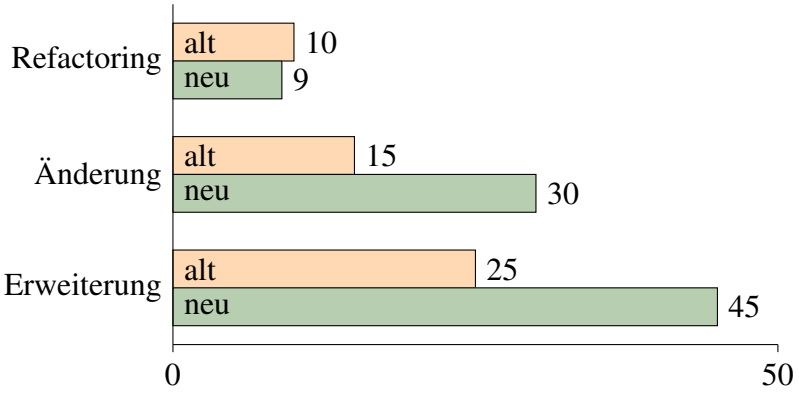

Abbildung 6.4.: Bearbeitungsdauer der Fallstudie in Minuten (Proband 2)

Während der Durchführung der Fallstudie konnten weitere Punkte bezüglich TDD festgehalten werden.

Durch Proband 1 kamen folgende Punkte auf:

- Die Abteilung von Proband 1 hat TDD im Umfeld eines Java-Backends und einer Webanwendung angewendet. Aufgrund der vielen Mocks und des erhöhten Aufwands programmiert die Abteilung wieder nach der herkömmlichen Vorgehensweise. Unit-Tests werden geschrieben, aber nur für reine Funktionen. Methoden mit Seiteneffekten werden nicht getestet.
- Die bei der Re-Implementierung praktizierte Variante von TDD scheint intuitiv zu sein. Ohne dem Probanden die Variante vorzustellen, treibt er die Entwicklung anhand dieser während der Aufgabenbearbeitung.

- Durch das Schreiben der Tests wird die Logik robuster. Es werden, im Gegensatz zur Implementierung in der alten Lösung, Grenzfälle geprüft.

Durch Proband 2 kamen folgende Punkte auf:

- Es fällt auf, dass bei der alten Lösung immer dann im UI die Implementierung manuell getestet wird, sobald ein gewisses Maß an Unsicherheit beim Entwickler aufkommt.
- Bei der Änderungsaufgabe mussten ein Unit-Test, Logik im Programm-Code und ein UI-Test angepasst werden. Die Zuordnung von Modul zu Unit-Test ist aufgrund der Tatsache, dass ein Unit-Test jeweils immer nur ein Modul testet, ersichtlich. Schwierigkeiten hatte der Proband bei der Auffindung aller relevanten UI-Tests. Wie erkennt der Entwickler die zu einem Modul zugehörigen UI-Tests und umgekehrt?
- Wie sieht das TDD-Vorgehen aus, wenn Funktionalität aus dem Programm-Code entfernt wird?

6.4. Auswertungsübersicht

Die Tabelle 6.1 gibt eine zusammenfassende Übersicht aller Ergebnisse in Bezug auf die Erwartungen und die Analysemethoden.

	Metriken	**eigene Beobachtungen**	**Fallstudie**
(EI1)		✓	
(EI2)		✓	
(EI3)		✓	
(EI4)	✓		
(EI5)		✓	
(EI6)		✓	
(EE1)		**?**	
(EE2)		✓	
(ED1)			**?**
(ED2)			**?**
(ED3)			✓
(ET1)		✓	
(EL1)		**?**	
(EW1)			✗
(ES1)		✓	✓
(ES2)		✓	
(ES3)		✓	
(ES4)		✓	
(ES5)		✓	

Tabelle 6.1.: Auswertungsübersicht der Analyseergebnisse

✓**:** Erwartung erfüllt

✗**:** Erwartung nicht erfüllt

? Erwartung konnte nicht evaluiert werden

Kapitel 7

Fazit

Ziel der Arbeit war es, das Paradigma der testgetriebenen Entwicklung im Kontext von OpenUI5-Anwendungen anhand einer Re-Implementierung des Frontends der Trainer Cloud App zu evaluieren.

Hierfür wurden zuallererst die Grundlagen der testgetriebenen Entwicklung erörtert. Es zeigte sich, dass anhand der Theorien zu TDD nicht auf die Vorgehensweise zur testgetriebenen Entwicklung von grafischen Anwendungen geschlossen werden kann. Aufgrund dessen wurde der TDD-Zyklus adaptiert und eine klare Abgrenzung zu ATDD gezogen. Anschließend folgte eine Beschreibung der Trainer Cloud App und des OpenUI5-Frameworks. Hauptbestandteil des OpenUI5-Frameworks sind die mitgelieferten Bausteine der MVC-Architektur. Um eine Teststrategie zur Re-Implementierung zu entwickeln, wurden das Architekturmuster und dessen Realisierung innerhalb des OpenUI5-Frameworks erläutert. Aufbauend auf der vorangegangenen Recherche zu TDD wurden Erwartungen an die Vorgehensweise formuliert und klassifiziert. Folgend wurde eine Umgebung zur testgetriebenen Entwicklung von OpenUI5-Anwendungen definiert und realisiert. Während der Konzeption der Entwicklungsumgebung war stets oberstes Gebot: Die Tests müssen schnell durchlaufen, um dem Entwickler innerhalb von Sekunden Feedback zu geben. Abschließend wurden nach der Re-Implementierung die Erwartungen anhand der zuvor definierten Analysemethoden evaluiert.

Die zu Beginn aufgestellten Ziele dieser Arbeit wurden demnach alle erreicht. Von insgesamt 19 Erwartungen, inklusive Schwierigkeiten, wurden 14 bestätigt, vier konnten im Rahmen der Arbeit nicht evaluiert werden, wohingegen eine Erwartung nicht bestätigt werden konnte.

Wichtige Erkenntnisse dieser Arbeit sind:

- Die Code-Qualität wird besser.
- Fehler fallen früher auf.
- Überzeugung von funktionierender Software und eigener Kompetenz wird durch dauerhafte Test-Feedback-Schleife gestärkt.
- Mit Hilfe automatisierter Tests kann der Erfolg von Refactoring-Phasen überprüft werden.
- Initiales Aufsetzen der Entwicklungsumgebung kostet Zeit.
- Die Entwicklungszeit eines Inkrements ist im Vergleich zur herkömmlichen Programmierung mit zusätzlichem Aufwand verbunden, dadurch erhöht sich die Entwicklungszeit.
- Das Imitieren von Abhängigkeiten durch Mocks macht Tests unleserlich für unerfahrene Projektmitglieder.
- Die Einhaltung des TDD-Zyklus verlangt Disziplin vom Entwickler.
- Eigene Adaptionen von TDD sind je nach Art der zu entwickelnden Software gefragt. Die Theorie kann von der praktizierten Variante abweichen.

Aufgrund der Tatsache, dass die Re-Implementierung von Grund auf neu begonnen werden konnte, also keine Altlasten und schwer testbare Module im Projekt vorhanden waren, und im Vergleich zu anderen UI-Technologien gut testbares HTML zum Einsatz kam, waren dies beste Voraussetzungen zum Gelingen von TDD. Pauschal lässt sich anhand der Arbeit keinesfalls sagen, dass TDD in jedem Projekt und zu jedem Zeitpunkt sinnvoll ist. Im Speziellen wurde mit dieser Arbeit und deren Kontext gezeigt, dass TDD im Rahmen einer OpenUI5-Anwendung mit der in dieser Arbeit vorgestellten Entwicklungsumgebung funktioniert.

Vom Team muss in Eigenverantwortung abgewogen werden, ob der Ansatz ihnen und dem Projekt einen Mehrwert bietet. TDD als Vorgehensweise verlangt viel Disziplin vom Entwickler, jedoch wird dadurch das Projekt in eine agile und professionelle Richtung gesteuert.

Kapitel 8

Ausblick

Zukünftige Tätigkeiten sehen die Ersetzung der alten Trainer Cloud App mit der re-implementierten Lösung vor. Aufgrund der unveränderten Schnittstelle zum Backend muss ausschließlich der Quellcode des Frontends ausgetauscht werden. Somit wäre die re-implementierte Lösung der Trainer Cloud App die erste produktive Anwendung innerhalb der Abteilung, welche testgetrieben entwickelt wurde. Ferner können mit der Auslieferung in die Produktion auch die Erwartungen bezüglich Lieferzyklen und Kundenzufriedenheit evaluiert werden. In die Zukunft blickend bietet die Auslieferung in die Produktion auch die Möglichkeit TDD langfristig bewerten zu können.

Durch die Ersetzung der alten Trainer Cloud App bietet dieses Projekt einen Einstiegspunkt für Abteilungskollegen, um sich mit der Vorgehensweise vertraut zu machen.

Zusätzlich birgt eine Verbreitung dieser Arbeit einen Mehrwert für die OpenUI5-Community. Sie zeigt anhand der Konzeptionierung und Beschreibung der Entwicklungsumgebung auf, wie OpenUI5-Anwendungen testgetrieben entwickelt werden können.

Abkürzungsverzeichnis

TDD Test Driven Development

XP Extreme Programming

ATDD Acceptance Test Driven Development

SCP SAP Cloud Plattform

MVC Model-View-Controller

PaaS Platform as a Service

SCC SAP Cloud Connector

RFC Remote Function Call

REST Representational state transfer

HTTP Hypertext Transfer Protocol

OData Open Data Protocol

REST Representational state transfer

JSON JavaScript Object Notation

XML Extended Markup Language

HTML Hyper Text Markup Language

UI User Interface

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Literatur

[1] C. Schwaber, G. Leganza und D. D'Silva, „The truth about agile processes“, *Forrester Research*, 2007.

[2] B. Meyer, *Agile: The good, the hype and the ugly*, 1. Aufl. Springer International Publishing, 2014, ISBN: 978-3-319-05154-3.

[3] R. C. Martin, *Agile software development, principles, patterns, and practices*, 1st. Prentice Hall, 2002, ISBN: 9780135974445.

[4] SAP, *Sap product road map*, https://www.sap.com/docs/download/2016/06/f49f0357-767c-0010-82c7-eda71af511fa.pdf, [Online; Stand 29. Mai 2017], 2017.

[5] K. Beck, *Test-driven development by example*. Addison Wesley, 2002, ISBN: 0-321-14653-0.

[6] T. J. Gandomani, H. Zulzalil und M. Z. Nafchi, „Agile transformation: What is it about?“, in *Software Engineering Conference (MySEC), 2014 8th Malaysian*, IEEE, 2014, S. 240–245.

[7] SAP, *Sap product road map*, https://www.sap.com/docs/download/2016/06/f49f0357-767c-0010-82c7-eda71af511fa.pdf, [Online; Stand 29. Mai 2017], 2017.

[8] OpenUI5, *Openui5 - features*, http://openui5.org/features.html, [Online; Stand 6. Mai 2017], 2017.

[9] Wikipedia, *Methodik — wikipedia, die freie enzyklopädie*, https://de.wikipedia.org/w/index.php?title=Methodik&oldid=159991699, [Online; Stand 20. April 2017], 2016.

[10] F. Bauer, L. Bolliet und H. Helms, „Report of a conference sponsored by the nato science committee“, in *NATO Software Engineering Conference*, Bd. 1968, 1968.

[11] E. W. Dijkstra, „The humble programmer“, *Communications of the ACM*, Bd. 15, Nr. 10, S. 859–866, 1972.

[12] K. Beck, *Extreme programming explained: Embrace change*, US ed. Addison-Wesley Professional, 1999, ISBN: 9780201616415. Adresse: http://gen.lib.rus.ec/book/index.php?md5=2F7594FD7A8A862A2B4D6AAE685359F8.

[13] K. Beck, *Why does kent beck refer to the rediscovery of test-driven development?*, https://www.quora.com/Why-does-Kent-Beck-refer-to-the-rediscovery-of-test-driven-development, [Online; Stand 20. April 2017], 2012.

[14] Wikipedia, *Test-driven development — wikipedia, the free encyclopedia*, https://en.wikipedia.org/w/index.php?title=Test-driven_development&oldid=776527278, [Online; accessed 3-May-2017], 2017.

[15] ——, *Magische zahl (informatik) — wikipedia, die freie enzyklopädie*, https://de.wikipedia.org/w/index.php?title=Magische_Zahl_(Informatik)&oldid=161032964, [Online; Stand 3. Mai 2017], 2016.

[16] M. Fowler, *Refactoring*. Addison-Wesley, 1999, ISBN: 0201485672. Adresse: http://gen.lib.rus.ec/book/index.php?md5=43F8C7D9CEA6BC9ED5C569AE56652AEA.

[17] Wikipedia, *Xunit — wikipedia, die freie enzyklopädie*, https://de.wikipedia.org/w/index.php?title=XUnit&oldid=148570354, [Online; Stand 29. April 2017], 2015.

[18] M. Soni, *Defect prevention: Reducing costs and enhancing quality*, http://www.isixsigma.com/industries/software-it/defect-prevention-reducing-costs-and-enhancing-quality/, [Online; Stand 22. April 2017], 2006.

[19] T. A. Corbi, „Program understanding: Challenge for the 1990s“, *IBM Systems Journal*, Bd. 28, Nr. 2, S. 294–306, 1989.

[20] L. Crispin, „Driving software quality: How test-driven development impacts software quality“, *IEEE Software*, Bd. 23, Nr. 6, S. 70–71, 2006, ISSN: 0740-7459. DOI: 10.1109/MS.2006.157.

[21] D. S. Janzen und H. Saiedian, „A leveled examination of test-driven development acceptance“, in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, IEEE, 2007, S. 719–722.

[22] N. Koudelia, *Acceptance test-driven development*, https://jyx.jyu.fi/dspace/bitstream/handle/123456789/37392/URN%3aNBN%3afi%3ajyu-201202161200.pdf, [Online; Stand 22. April 2017], 2011.

[23] M. Gärtner, *Atdd by example: A practical guide to acceptance test-driven development*, 1. Aufl., Ser. Addison-Wesley Signature Series (Beck). Addison-Wesley Professional, 2012, ISBN: 0321784154. Adresse: http://gen.lib.rus.ec/book/index.php?md5=3AD1FE6D9D85E7F8690F9CA1218DCC06.

[24] J. Deacon, „Model-view-controller (mvc) architecture“, *JOHN DEACON Computer Systems Development, Consulting and Training*, 2009.

[25] G. E. Krasner, S. T. Pope u. a., „A description of the model-view-controller user interface paradigm in the smalltalk-80 system“, *Journal of object oriented programming*, Bd. 1, Nr. 3, S. 26–49, 1988.

[26] Wikipedia, *Modularität — wikipedia, die freie enzyklopädie*, https://de.wikipedia.org/w/index.php?title=Modularit%C3%A4t&oldid=164699372, [Online; Stand 6. Mai 2017], 2017.

[27] M. Fowler, *Gui architectures*, https://martinfowler.com/eaaDev/uiArchs.html, [Online; Stand 07. Mai 2017], 2006.

[28] D. Greer, *Interactive application architecture patterns*, http://aspiringcraftsman.com/2007/08/25/interactive-application-architecture/, [Online; Stand 07. Mai 2017], 2006.

[29] Wikipedia, *Openui5 — wikipedia, die freie enzyklopädie*, https://de.wikipedia.org/w/index.php?title=OpenUI5&oldid=163815656, [Online; Stand 6. Mai 2017], 2017.

[30] ——, *Softwaretechnik — wikipedia, die freie enzyklopädie*, https://de.wikipedia.org/w/index.php?title=Softwaretechnik&oldid=164369794, [Online; Stand 29. April 2017], 2017.

[31] NodeJS, *Nodejs*, https://nodejs.org/en/, [Online; Stand 23. Juni 2017].

[32] Grunt, *Grunt*, https://gruntjs.com/, [Online; Stand 23. Juni 2017].

[33] Express, *Express*, http://expressjs.com/, [Online; Stand 23. Juni 2017].

[34] PhantomJS, *Phantomjs*, http://phantomjs.org/, [Online; Stand 23. Juni 2017].

[35] Selenium, *Selenium webdriver*, http://www.seleniumhq.org/docs/03_webdriver.jsp, [Online; Stand 23. Juni 2017].

[36] Codecept, *Codeceptjs*, http://codecept.io/, [Online; Stand 23. Juni 2017].

[37] jQuery, *Qunit*, http://qunitjs.com/QUnit, [Online; Stand 23. Juni 2017].

[38] Sinon, *Sinonjs*, http://sinonjs.org/, [Online; Stand 23. Juni 2017].

[39] Timemachine, *Timemachine*, https://www.npmjs.com/package/timemachine, [Online; Stand 23. Juni 2017].

[40] Mocha, *Mocha*, http://mochajs.org/, [Online; Stand 23. Juni 2017].

[41] Wikipedia, *Softwaremetrik — wikipedia, die freie enzyklopädie*, https://de.wikipedia.org/w/index.php?title=Softwaremetrik&oldid=160402025, [Online; Stand 10. Mai 2017], 2016.

[42] A. H. Watson, D. R. Wallace und T. J. McCabe, *Structured testing: A testing methodology using the cyclomatic complexity metric*. US Department of Commerce, Technology Administration, National Institute of Standards und Technology, 1996, Bd. 500.

[43] B. Müller, *Reengineering: Eine einführung*, 1. Aufl., Ser. Leitfäden der Informatik. Vieweg+Teubner Verlag, 1997, ISBN: 978-3-519-02942-7. Adresse: http://gen.lib.rus.ec/book/index.php?md5=46E2AE13FB0ADE16C94AA28153E9E72E.

[44] A. Bandura, „Self-efficacy: Toward a unifying theory of behavioral change.“, *Psychological review*, Bd. 84, Nr. 2, S. 191, 1977.

Anhang A

Anhang: Metriken

Datei	Physische Zeilen	Logische Zeilen	Aggregierte Komplexität
trainercloudappm/webapp/logon.js	203	109	7
trainercloudappm/webapp/view/Master.controller.js	186	132	21
trainercloudappm/webapp/Component.js	131	95	6
trainercloudappm/webapp/view/Detail.controller.js	131	93	11
trainercloudappm/webapp/util/Formatter.js	94	70	23
trainercloudappm/webapp/MyRouter.js	56	31	7
trainercloudappm/webapp/index.js	33	12	1
trainercloudappm/webapp/util/Controller.js	9	6	1
Summe	843	548	77
Durchschnitt	105,38	68,5	9,6

Tabelle A.1.: Metriken der alten Trainer Cloud App

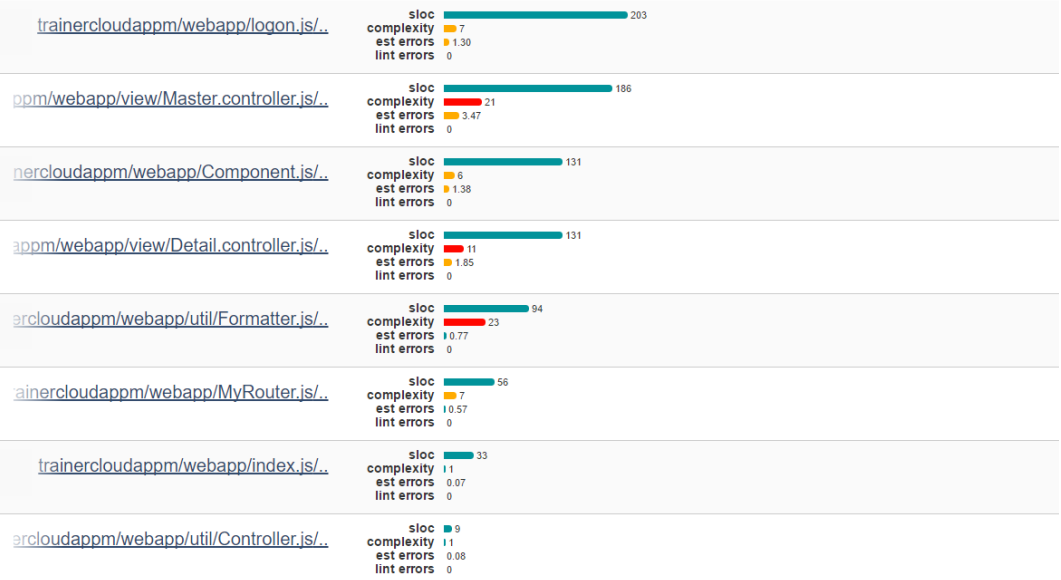

Abbildung A.1.: Balkendiagramm zur Metrik der alten Trainer Cloud App

Datei	Physische Zeilen	Logische Zeilen	Aggregierte Komplexität
trainercloudappmtdd/webapp/controller/Master.controller.js	131	78	9
trainercloudappmtdd/webapp/controller/Detail.controller.js	64	42	9
trainercloudappmtdd/webapp/controller/Base.controller.js	29	18	2
trainercloudappmtdd/webapp/Component.js	18	10	1
trainercloudappmtdd/webapp/start-trainercloudappmtdd.js	13	6	1
trainercloudappmtdd/webapp/constant/EventStatus.js	9	6	1
trainercloudappmtdd/webapp/constant/SurveyType.js	8	5	1
Summe	272	165	24
Durchschnitt	38,9	23,6	3,4

Tabelle A.2.: Metriken der re-implementierten Trainer Cloud App (exklusive Tests)

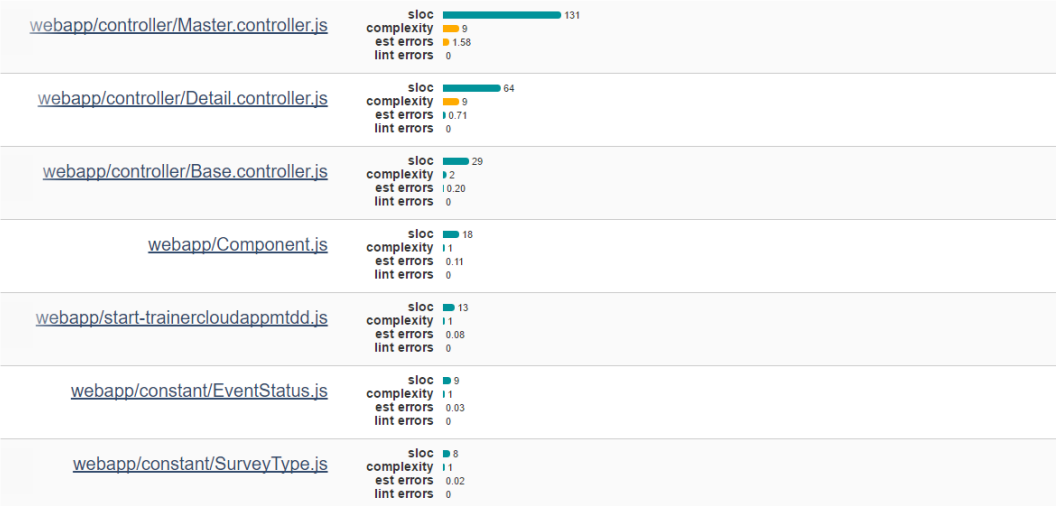

Abbildung A.2.: Balkendiagramm zur Metrik der re-implementierten Trainer Cloud App (exklusive Tests)

Datei	psysische zeilen	logische zeilen	aggregierte komplexität
trainercloudappmtdd/webapp/test/unit/Master.controller.js	384	295	1
trainercloudappmtdd/webapp/test/acceptance/desktop/master_section_desktop_test.js	241	211	1
trainercloudappmtdd/webapp/test/unit/Base.controller.js	199	118	1
trainercloudappmtdd/webapp/test/unit/Detail.controller.js	139	103	1
trainercloudappmtdd/webapp/controller/Master.controller.js	131	78	9
trainercloudappmtdd/webapp/test/acceptance/desktop/detail_section_desktop_test.js	87	75	1
trainercloudappmtdd/webapp/controller/Detail.controller.js	64	42	9
trainercloudappmtdd/webapp/test/unit/mocha-run-all-unit-tests.js	64	42	1
trainercloudappmtdd/webapp/test/acceptance/desktop/i18n_desktop_test.js	45	37	1
trainercloudappmtdd/webapp/controller/Base.controller.js	29	18	2
trainercloudappmtdd/webapp/test/acceptance/desktop/basic_structure_desktop_test.js	26	21	1
trainercloudappmtdd/webapp/test/acceptance/bootstrap.js	21	15	1
trainercloudappmtdd/webapp/Component.js	18	10	1
trainercloudappmtdd/webapp/test/acceptance/mobile/basic_structure_mobile_test.js	17	16	1
trainercloudappmtdd/webapp/start-trainercloudappmtdd.js	13	6	1
trainercloudappmtdd/webapp/test/unit/interface-qunit-mocha.js	10	7	2
trainercloudappmtdd/webapp/constant/EventStatus.js	9	6	1
trainercloudappmtdd/webapp/constant/SurveyType.js	8	5	1
trainercloudappmtdd/webapp/test/acceptance/mobile/detail_section_mobile_test.js	8	7	1
trainercloudappmtdd/webapp/test/phantomjs-print-console.js	8	6	1
trainercloudappmtdd/webapp/test/acceptance/mobile/master_section_mobile_test.js	5	5	1
Summe	1526	1123	39
Durchschnitt	72,7	53,5	1,9

Tabelle A.3.: Metriken der re-implementierten Trainer Cloud App (inklusive Tests)

Anhang B

Anhang: Notizen zu eigenen Beobachtungen

- Abhängigkeiten zwischen Akzeptanztests und Code?
- OpenUI5 generiert die IDs der HTML-Elemente. Bekommt man es hin, dennoch Elemente anhand der IDs zu selektieren?
- Zu Beginn der Entwicklung: mit welchem Akzeptanztest fängt man an? Erst einmal die grobe Struktur testen? Oder gleich eine Funktionalität?
- Aktzeptanztest schreiben, so schnell wie möglich grün werden lassen -> Layout anpassen
- Wie geht man mit Testdaten um, die zeitabhängige Werte haben (Datümer, Status: Running...)
- Man muss erst einen Rahmen (auf Basis des Frameworks) schaffen, indem a) Tests geschrieben werden können und b) die Anwendung dann auch entwickelt werden kann
- TDD hilft beim Lösen schwieriger Problem mit Hilfe einer To-Do-Liste. Man vergisst die Dinge nicht, die man noch machen wollte, man kann sich auf das Lösen des aktuellen Zwischenschritts konzentrieren
- Wie genau formuliert man die Akzeptanztests richtig? Handy vs. Desktop
- Keine URL-Parameter für die Mockdaten, strikte Trennung zwischen DEV-/PROD-Umgebung
- Refactoring ist erfolgreich, aber Tests wurden nicht richtig angepasst
- Neue Implikation: (Akzeptanz)Tests müssen auch einzeln ausführbar sein. Dauert sonst zu lange und führt zu Demotivation beim Entwickler.
- Umgebung schaffen, dass den Entwickler die Formulierung der Tests und deren Ausführung nicht demotivieren.
- Zusätzlicher Aufwand: Desktop + Mobile

- Macht es Sinn, stupide Methoden (z.B. reines Routing) zu testen? Ja, Methoden haben Seiteneffekte. Funktionen haben Rückgabewerte. Wieso die Seiteneffekte nicht testen? wurde z.B. navTo() aufgerufen?
- CodeceptJS ist umständlich, wenn es um die Überprüfung von farblichen Werten geht (z.b. state eines ListeItem).
- TDD geht flüssig, wenn man alles kann. Sobald man ausprobieren muss, sollte man so konsequent sein und den Code (sobald er funktioniert) wieder löschen, dann Test schreiben und erst danach erneut den Programm-Code.
- Grenzen von TDD: Sobald man den Browser verlässt, wie im Beispiel der Funktionalität „Send-Email-To-All“ kann man die Funktionswiese nicht mehr testen
- Keine Tests für Konstanten-Klassen!
- Anderes Gefühl von 'Done', wenn Tests dafür existieren.

Abbildung B.1.: Balkendiagramm zur Metrik der re-implementierten Trainer Cloud App (inklusive Tests)

Anhang C

Anhang: Notizen der Fallstudie mit Entwicklern

(L) **Waren die Tests hilfreich um die einzelnen Module zu verstehen?** ED1 / ED2 / EW1

(2N) ja, stubbs verwirrt, weil dann plötzlich richtige Implementierung ausgeführt

(M) **Wurden die Tests während Änderungen an Program-Code entsprechend angepasst?** %ED3

(2V) ja, bemerkt sehr rasch

(2N) UI-Test geändert (außerhalb des Codes)

(3N) Wie ist das Vorgehen beim Löschen?

(N) **Wie wurde geprüft ob Änderungen erfolgreich waren?** %EL1

(1A) kommentiert altes aus (eventuell dazu einen Schreibfehler zu erzeugen)

(1A) und in welchem Fall braucht er ?

(1A) ist auch an Testfällen gebunden

(3A) • baut erst Tests ein und schaut als Fehlermeldung
• baut Bindung ein, testet
• drückt alle Events durch, sollte Formatierung stellen

(2A) Auch wieder auf UI

(2A) schaut auch in Code, ob Änderungen erfolgreich waren

(O) **Wurde der TDD-Zyklus erfolgreich umgesetzt?** %ES3

(1A) reflektiert über Paradigma, ja, braucht kein Test

(2N) wollte ohne Code anpassen ändern

(2N) fängt mal bei UI-Tests an

(3N) • UI-Test
• beginnt sofort schon zu laufen

(P) **Wie lange betrug die Bearbeitungsdauer der einzelnen Aufgaben?** %ES1

(1A) 19:58 – 20:08 (2A) 20:11 – 20:16 (3A) 20:19 – 20:45

(1N) 21:18 – 21:27 (2N) 21:29 – 21:36 (3N) ~ 40 min

(Q) **Wie oft und zu was wurde auf Nachfrage der Probanden Hilfestellung gegeben?**

(2A) UI5 Bindung erklärt (3A) Tipps in Bezug auf View und Formatter gegeben

(3A) Tipps bzgl. Zeit-Formatierung (verwirrende SO-Beispiel mit 12-h-Format)

(2N) Stubbing erklären

(R) **Welche Methodik ziehen die Probanden vor und warum?**

• kommt aufs Team drauf an
→ nur mit denen ausprobieren
• ganz neues Projekt starten ist doch
• mit Technologie vertraut werden
• bringt es gut (nimmt sich Zeit, was die tun sollen, fängt erst an eines zu implementieren)
• klare Definition: Tests müssen laufen

Random (X)
• Proband etwas verwirrt in Bezug auf UI5
• probiert Snippet-Edits aus – etwas verwirrt
• anscheinend gibt es bei der herkömmlichen Variante immer ein Punkt, an dem getestet wird, wenn Funktionalität fertig ist (siehe (3A) Start-Reset)
• Testen (Zeitpunkt, anders zu ermitteln)
• von Stack Overflow was rauskopiert (3A)
• (3A) nicht Verbindung zu (1A) und sucht nicht
• (3A) prüft auf null und undefined
• (3A) „mach ich mal nen Zwischentest"

(X) ~~testet auch nie~~
(1A) testet auch nicht im Browser, bleibt auf und
• das mit dem wait (10) nullt
• einzeln unit-Tests
=> • wie erkenne ich, dass es für x-Funktion auch UI-Tests zum bypassen gibt? und Vice Versa
• Internet Verbindung muss da sein?
• Console als Hindernis für Junior Leute?
=> • wie ist das Vorgehen, wenn ich was zeichnen

(N) • wie doch falsch implementiert?
(1N)
• mehr integriert die Tests in der IDE
• wie besser Integration in die IDE
• fragt nach „wie handel ich „Abhängigen" und neue TODO"

+ ganz gut erst alte Version neu „zeichnen" doch sehr viel Neues für Test

(R) • kam sich viel aus Verantwortlichen zieh.
• wenn Technologien bekannt => zeichne
~~• wenn nicht: prototyp test~~
• ~~für~~ Forschung vs. Produktion
• Framework Evaluierung
lohnt dazu machen

(L) **Waren die Tests hilfreich um die einzelnen Module zu verstehen?** ED1 / ED2 / EW1

(1N): ~~Er findet Edge Cases, an die~~
~~sonst nicht gedacht wäre a > 0 || a < 999~~
liest mit Tests leicht an zu verstehen
~~bei (1A) gleiche gezielt, aber nicht aufgefallen~~

(M) **Wurden die Tests während Änderungen an Program-Code entsprechend angepasst?** %ED3

ja, alle [illegible] nicht [illegible] nachzuvollziehen

(N) **Wie wurde geprüft ob Änderungen erfolgreich waren?** %EL1

(1A) per manuellen Test, will mal per gezielt Fälle testen. Ist an Testdaten gebunden und nicht so nach [illegible]fällen

(2A) weiter manuelle Tests. (ist mit nicht wie [illegible] das weiter [illegible] könnte)

(O) **Wurde der TDD-Zyklus erfolgreich umgesetzt?** %ES3

4A Super gut! → vier ohne Tests ändern wollen
- macht auch mal schon 0700 → so schnell wie möglich
→ nicht [illegible], nicht zu hinein (Fehler [illegible])

(P) **Wie lange betrug die Bearbeitungsdauer der einzelnen Aufgaben?** %ES1

(1A) 19:16 – 19:21 (2A) 19:21 – 19:28 (3A) 19:30 – 19:38

(1N) 10 (2N) 20 (3N) 30

(Q) **Wie oft und zu was wurde auf Nachfrage der Probanden Hilfestellung gegeben?**

- [illegible] gezeigt? · manuelle: E? · welche Page [illegible]
- Verständnis zu Mocks in Tests ("[illegible]") anstatt "D054426"

(R) **Welche Methodik ziehen die Probanden vor und warum?** (X) [illegible]

→ gut

→ einleiten
erst mal verstehen, dann [illegible] anpassen

(3A) fängt bei vier an
- erst umformatiert
- testet UI
- verändert Bedingung (auf Formatt)
- baut Funktion in Formatter ein
- Testet UI → ändert Formatter
=> der Zyklus kein [illegible] Entwickeln

- mit 2, TDD Nein
- kein Unit Tests bei UI Tests zu mit gemacht → [illegible]
- Unit Test [illegible]
- UPAT zu streng

- ~~Aufgabenstellung~~ (Einleitung)
- ~~TDD~~
- Projektstruktur
- Fazit

- Einleitung
- TCA
- TDD
- Projektstruktur
- Aufgaben (alt → neu)

(X)

- Unit Tests zeigen gleich Fehler an (muss nicht im Browser testen)
- Die Funktionen mit vielen Maths etwas schwer verständlich
- Test etwas schlecht
- man kann Seitenaufruf machen
- erst an Stellen im UI geprüft

- warum mit UI-Test angefangen?
  → beinhaltet was man nicht
  → Unit Test detaillierter
- OK die Varianten-Syntax ist aktueller
  → hat er aber Erklärung angemeldet / mehr als alte angepasst
- übergreifende Test-Verbindungen in Mocking-Daten
- durch das Schreiben der Tests werden Gegebenheiten der Methoden besser durchleuchtet → Funktionsweisen erhalten Code, kein alten mehr geprüft

---

- grund noch fertig er erst
- auch lokal entwickelt

Anhang D

Anhang: Quellcode-CD