„Konzeption und praktische Anwendung der Testpyramide bei einer Frontend-Applikation“

Bachelor-Thesis

für die Prüfung

Bachelor of Science (B. Sc.)

der

Universität Bremen

Sebastian Cordes

1. Gutachter:	Prof. Dr. Christoph Lüth
2. Gutachter:	Dr. Sabine Kuske
Praxispartner:	engram GmbH Konsul-Smidt-Straße 8r 28217 Bremen
Abgabedatum:	26.02.2019

I Inhaltsverzeichnis

I Inhaltsverzeichnis ..... **I**

II Abkürzungsverzeichnis ..... **III**

III Abbildungsverzeichnis ..... **IV**

IV Tabellenverzeichnis ..... **V**

V Hinweise zur Bachelorarbeit ..... **VI**

0 Kurzfassung **0**

1 Einleitung **1**

1.1 Ziel und Motivation der Arbeit 1

1.2 Vorgehensweise 2

1.3 Kurzportrait der engram GmbH und des engram Learning Management Systems 3

2 Theoretische Grundlagen **4**

2.1 Klassische Testarten 4

2.1.1 Unittest 5

2.1.2 Integrationstest 6

2.1.3 End-to-End 9

2.2 Testabdeckung 10

2.3 Verfahren zur Erstellung von Testfällen 12

2.3.1 Analyse fachlicher und technischer Konzepte 12

2.3.2 Datenkombinationstest 13

2.3.3 Objektlebenszyklustest 15

2.3.4 Datenzyklustest 15

2.3.5 Szenariotests 16

3 Konzept **17**

3.1 Auswahl des konkreten Testgegenstands 18

3.2 Erste Ebene: Unittests 20

3.3 Zweite Ebene: Integrationstest 21

3.4 Dritte Ebene: End-to-End-Test 22

3.5 Kriterien von wichtigen Codestellen 23

4 Testframeworks im React-Kontext **25**

4.1 Eigenschaften und Funktionalitäten von Testframeworks 25

4.2 Framework: Mocha 26

4.3 Framework: Karma 26

4.4 Framework: Jest 26

4.5 Testwerkzeug-Bibliotheken 27

4.6 Auswahl im Hinblick auf das Konzept 28

5 Praktische Umsetzung 30
5.1 Umsetzung Unittest 30
5.2 Umsetzung Integrationstest 31
5.3 Umsetzung End-to-End-Tests 32
6 Zusammenfassung und Ausblick 34
7 Literaturverzeichnis 36
VI Anhangsverzeichnis VII

II Abkürzungsverzeichnis

Abkürzung	Beschreibung
eLMS	Engram Learning Management System
App	Applikation (eine Software-Anwendung)
UTF	Unabhängig testbares Feature
TDD	Test Driven Development (Testgetriebene Entwicklung)

III Abbildungsverzeichnis

Abbildung 1: Testpyramide mit der effizienten Verteilung der Tests 4

Abbildung 2: Beispiel eines Kontrollflussgraphen 11

Abbildung 3 CompactCourseListOverview 18

Abbildung 4 Flux/Redux Architektur 19

Abbildung 5 Visuelle Darstellung der Presentational BlockHeader 30

IV Tabellenverzeichnis

Tabelle 1 Ergebnis Codeanalyse BlockHeader 30

Tabelle 2 Ergebnis des Datenkombinationstest für den BlockHeader 31

Tabelle 3 Klickstrecken für den End-to-End-Test 32

V Hinweise zur Bachelorarbeit

Die **fett** gedruckten Begriffe in der vorliegenden Bachelor-Thesis werden im Glossar (siehe A1 Glossar, S. A-1) erläutert.

Die *kursiv* gedruckten Begriffe in der vorliegenden Bachelor-Thesis werden im Abkürzungsverzeichnis (siehe II Abkürzungsverzeichnis, S. III) erläutert.

0 Kurzfassung

In den letzten Jahren sind Javascript **Frameworks** wie React, Angular und VueJs immer populärer geworden. Die ebengenannten **Frameworks** erleichtern das Entwickeln von Frontend Anwendungen mit HTML5 und Javascript. Das Testen solcher Anwendungen bietet mehr Möglichkeiten, als zeitaufwändiges End-to-End-Testen. Allerdings haben sich die End-to-End-Tests in vielen Firmen etabliert und die meisten Projektmanager bevorzugen die Tests, wegen des Bezugs zu den realen Endnutzern. Andererseits sind die End-to-End-Tests aber auch sehr kostspielig und erfordern eine voll funktionsfähige Systemlandschaft.[1] „Im Vergleich zu Tests auf der Integrationsebene sind sie etwa viermal so teuer und dauern fünf- bis zehnmal so lang.“[2]

Die vorliegende Arbeit konzentriert sich auf die Frage, ob es möglich ist, im Frontend am Beispiel einer React-Anwendung ebenfalls eine automatisierte Testabdeckung zu schaffen, die der Theorie der Testpyramide entspricht.

Die Entwicklung eines Testkonzeptes und die praktische Umsetzung findet in Kooperation mit der engram GmbH statt. Die Ergebnisse bilden eine Basis für die Entscheidung, wie der Entwicklungs- und Testprozess zukünftig im Frontend bei der engram GmbH, ablaufen wird.

[1] Vgl. Platz, W. 2015

[2] Platz, W. 2015

1 Einleitung

1.1 Ziel und Motivation der Arbeit

Die vorliegende Bachelorarbeit verfolgt das Ziel, dem Leser eine neue Sichtweise für das Testen einer grafischen Benutzeroberfläche, dem Frontend, zu vermitteln.

Über lange Zeit konnte das Frontend nur über das manuelle Ausführen von Testskripts (Klickstrecken-Beschreibung) überprüft werden. Solch ein Vorgehen war in der Ausführung und in der Pflege der Testskripts sehr aufwändig und wurde aufgrund dessen oftmals nicht sorgfältig angewendet.[3] Die Entwicklung von Selenium im Jahr 2004 brachte allerdings auch viele Fürsprecher für Frontendtests mit sich. Durch Selenium ist es möglich, die kostspieligen manuellen Klickstrecken-Beschreibungen zu automatisieren, dennoch existiert ein hoher einmaliger Aufwand für die Inbetriebnahme der Testautomation. [4]

In der heutigen Zeit ist es üblich, die grafischen Benutzeroberflächen mit HTML5 und Javascript zu entwickeln. Auch bei Smartphone Applikationen (*Apps*) und Desktopanwendungen wird inzwischen HTML5 und Javascript zu einer sehr populären Alternative gegenüber den altbewährten Wegen. Die am weitesten verbreiteten **Frameworks** für eine Entwicklung mit HTML5 und Javascript sind React[5], Angular[6] und VueJs[7]. Mit der Verwendung derartiger **Frameworks** entstehen nicht einfach nur Benutzeroberflächen, sondern komplett alleinstehende Applikationen die mit Unittest, Integrationstest und End-to-End-Tests getestet werden können.

Meine Motivation besteht darin aufzuzeigen, dass das Testen einer Benutzeroberfläche nicht nur über das Ausführen von Klickstrecken am Ende eines Projektes möglich ist. Bei dem Entwickeln von Backendsystemen existiert das gängige Konzept der Testpyramide. Das Konzept besagt, dass es auf der un-

---

[3] Vgl. Hansbauer, G. (o. J.)
[4] Vgl. Platz, W. 2015
[5] https://reactjs.org/
[6] https://angular.io/
[7] https://vuejs.org/

tersten Ebene eine Testbasis von 70 Prozent Unittests geben soll, darauf folgen 20 Prozent Integrationstests und 10 Prozent End-to-End-Tests. Durch die neuen Technologien im Frontend bietet sich das Vorgehen der Testpyramide auch bei einer Benutzeroberfläche an, so, dass der volle Funktionsumfang der Oberfläche über automatisierte Tests gesichert ist.

1.2 Vorgehensweise

Die Arbeit beginnt mit einem Kurzportrait der engram GmbH. Im darauffolgenden Abschnitt werden die theoretischen Grundlagen erörtert. Angefangen bei den drei Testarten, die für die Testpyramide benötigt werden, bis hin zu Techniken für die Testfallermittlung und die Erfüllung bestimmter Metriken der Testabdeckung. In dem dritten Kapitel wird die Umsetzung der Testpyramide anhand einer React Anwendung in Form eines Konzepts geplant. Der Einstieg in das Kapitel erfolgt über die konkrete Auswahl des Testgegenstandes und einer Erläuterung des Lebenszyklus einer React Anwendung. Nachfolgend entsteht eine Beschreibung entlang des Lebenszyklus, wie für alle drei Ebenen der Pyramide die Tests umzusetzen sind. Das Kapitel schließt mit einer Hilfestellung zum Identifizieren wichtiger zu testender Codestellen ab. Bevor die Umsetzung des Konzepts angegangen werden kann, wird zuerst die Recherche, zu den gängigen **Testframeworks**, in dem vierten Kapitel beschrieben. Das Kapitel beginnt mit einer Liste von Anforderungen an ein **Testframework**, stellt dann die recherchierten **Frameworks** vor und erläutert abschließend die Auswahl für die praktische Umsetzung. In dem praxisorientierten Teil der Arbeit findet die komplette Umsetzung des Konzepts statt, welches in dem fünften Kapitel dokumentiert ist. Sämtliche implementierten Tests sind im Anhang zu finden. Das Abschlusskapitel der Arbeit reflektiert das gesamte Vorgehen und beleuchtet die Sinnhaftigkeit der Methodik der Testpyramide im Frontend.

1.3 Kurzportrait der engram GmbH und des engram Learning Management Systems

Seit mehr als 20 Jahren berät und unterstützt die engram GmbH Unternehmen bei der Integration von ganzheitlichen E-Learning-Lösungen. Neben dem Entwurf und der Erstellung von stationären, als auch von mobilen Web Based Trainings, hat sich engram darauf spezialisiert, das OpenSource LMS Moodle übersichtlich und nutzerfreundlich zu gestalten und um unternehmensspezifisch entwickelte Zusatzmodule zu erweitern.

Ein Produkt der engram GmbH ist das engram Learning Management System (eLMS). Das eLMS ist eine Software zur Gestaltung digitaler Aus- und Weiterbildungsprozesse in Unternehmen. Die Betreuung, Weiterentwicklung und Wartung der Software ist der Hauptaufgabenbereich der eLearning Abteilung.

2 Theoretische Grundlagen

Das folgende Kapitel schlüsselt die theoretischen Grundlagen auf, die es bedarf, eine Anwendung zu testen. Dabei stützt sich die Arbeit auf die Vorgehensweisen und Methodiken aus der objektorientierten Anwendungsentwicklung, da der spätere Testgegenstand durch das **Framework** React äußerst modular aufgebaut ist und diese Struktur unweigerlich an eine objektorientierte Programmiersprache erinnert. Zunächst kommen einige Erklärungen zu klassischen Testarten, anschließend folgen Metriken zur Testabdeckung und Verfahren zum Identifizieren von Testfällen.

2.1 Klassische Testarten

„Das Testen objektorientierter Software lässt sich in drei Stufen unterteilen, angefangen bei einzelnen Klassen bis zur Integration und der Überprüfung möglicher Wechselwirkungen.“[8] Gut veranschaulichen lassen sich die drei Stufen mit dem Konzept der Testpyramide.

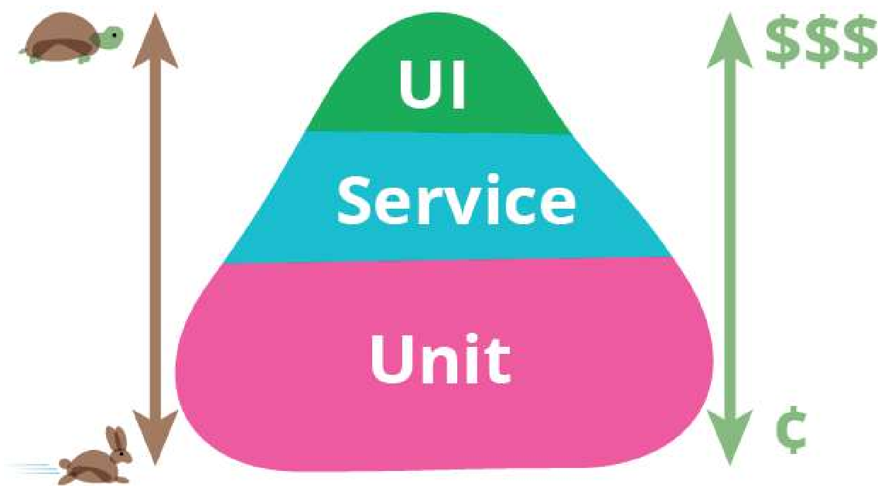

Abbildung 1: Testpyramide mit der effizienten Verteilung der Tests[9]

Auf der Ebene „Unit“ gibt es sehr viele schnelle und günstige Unittest die eine Basis bilden. Darüber kommen die Integrationstests, um alle Kommunikationswege zwischen den einzelnen Units zu überprüfen. Die Spitze der Pyramide

[8] Pezzè, M. ; Young, M. 2009: S.315
[9] Fowler, M. 2012

besteht aus wenigen langsamen und teuren End-to-End-Tests, um das Softwaresystem als Ganzes zu testen.

2.1.1 Unittest

Seit den 1970er Jahren ist das Unit-Testing unter Entwicklern bekannt und erweist sich fortwährend als eine der optimalen Möglichkeiten ein tieferes Verständnis für die funktionalen Anforderungen einer Klasse oder Methode zu entwickeln und die Code-Qualität zu steigern.[10] Ein einzelner Unittest konzentriert sich immer auf eine sogenannte Work-Unit. „Eine ‚Work-Unit' ist das kleinste Inkrement, um das ein Softwaresystem wächst oder sich ändert, die kleinste Unit, die im Projektzeitplan oder Budget auftaucht, und die kleinste Einheit für die vernünftigerweise eine Suite aus Testfällen erstellt werden kann."[11] Eine weitere wichtige Eigenschaft solch eines Tests ist die Unabhängigkeit. Der Testgegenstand eines Unittests nennt sich auch „Unabhängig testbares Feature" (*UTF*), dabei handelt es sich um eine Funktionalität, die unabhängig von anderen Funktionalitäten des Softwaresystems getestet werden kann.[12] Mit der Einhaltung der Kriterien ist es möglich mittels eines Unittests die „[…]funktionale Korrektheit und Vollständigkeit einzelner Programmeinheiten, z.B. von Funktionen, Prozeduren oder Klassenmethoden, von Klassen oder Komponenten des Systems[…]"[13] zu zeigen.

Die Entwicklungsmethode TDD (Test Driven Development[14]) spiegelt wider, wie bedeutungsvoll Unittests während des Entwicklungsprozesses sind. Beim TDD gilt es, zuerst den Test zu entwickeln und danach die Funktion. Auf diese Weise entwickelt der Programmierer bessere Kenntnisse für die geforderten Anforderungen und hat im Nachhinein die Gewissheit, dass seine Funktion den Anforderungen gerecht wird. Auch, wenn nicht die Entwicklungsmethode TDD

[10] Vgl. Osherove, R. 2015: S.25
[11] Pezzè, M. ; Young, M. 2009: S.192
[12] Vgl. Pezzè, M. ; Young, M. 2009: S.192
[13] Rätzmann, M. 2004: S.107
[14] Übersetzung: Testgetriebene Entwicklung

verwendet wird, zählt es zu den Best Practices beim Entwickeln, die Unittests unmittelbar vor oder nach dem Fertigstellen einer Unit zu schreiben.[15]

Die am meisten verfolgten Ziele beim Software testen, sind Fehler zu finden und Code-Qualität zu steigern, dafür sollte ein Unittest gewisse Eigenschaften besitzen. Die Eigenschaften lassen sich wie folgt zusammenfassen: Ein guter Unittest sollte einen geringen Implementierungsaufwand haben und jeder sollte ihn schnell per Knopfdruck ausführen können. Des Weiteren sollte der behandelte Testfall auf Dauer relevant bleiben und unabhängig von dem restlichen System getestet werden können. Darüber hinaus ist es erforderlich, dass der Test automatisierbar, wiederholbar und konsistent in der Auswertung ist.[16]

2.1.2 Integrationstest

Für eine klare Trennung von Unit- und Integrationstest ist es hilfreich, sich die Unterschiede an einem Auto vorzustellen:

Bewegt sich ein Auto nicht mehr, kann der Fehler in einem der Teilsysteme liegen, zum Beispiel dem Motor, welcher eine Unit darstellt. Der Fehler kann allerdings auch in dem Zusammenspiel von Motor und Getriebe liegen, dieses Zusammenspiel ist die Integration der einzelnen Units.

In dem Szenario wäre die Bewegung des Autos der bestmöglichste Integrationstest und der Beweis, dass alle Teilsysteme mit einander funktionieren. Schlägt der Test fehl, versagen alle Teilsysteme zusammen. Es ist schwer feststellbar, ob ein einziges Teilsystem fehlerhaft ist, oder ob die Kommunikation zwischen den Teilsystemen gestört ist.[17]

Integrationstest sind **High-Level-Tests** und sollten in einem Software System, wegen des eben genannten Problems, nur die zweite Verteidigungsreihe beim Testen darstellen. Denn die Fehler in den Teilsystemen können ausgeschlossen werden, wenn in der ersten Verteidigungsreihe genügend gute Unittests

[15] Vgl. Bindick, S. 2018
[16] Vgl. Osherove, R. 2015: S.28
[17] Vgl. Osherove, R. 2015: S.29-30

vorhanden sind.[18] Das Ziel von Integrationstests ist somit, dass Zusammenspiel von Teilsystemen zu testen und sicherzustellen, dass die Interaktionen der Teilsysteme fehlerfrei ablaufen.

Es ist also elementar für einen effektiven Integrationstest, auf einer ausführlichen Basis von Unittest aufzubauen.[19] Allerdings spielt auch die Auswahl der Integrationsart eine wesentliche Rolle, die je nach Projekt passend gewählt werden sollte. Für ein entwicklungsbegleitendes und frühes Testen sind die verschiedenen Arten von inkrementellen Integrationstest zu empfehlen. Als inkrementelle Arten gelten die strukturierten Verfahren, wie Top-down- oder Bottom-up-Integration, aber auch die Feature-orientierten Verfahren, wie Thread- und Critical-Module-Integration.[20]

Die Top-down und Bottom-up-Integration sind schichtenorientierte Integrationsformen. Bei der Top-down-Integration wird bei der obersten Schicht (z.B. der Benutzerschnittstelle) begonnen und alle darunterliegenden Schichten werden simuliert. Nun folgt in jedem weiteren Integrationsschritt, dass die nächste simulierte Schicht gegen die richtige Implementierung ausgetauscht wird. Auf diese Weise steht sehr früh in dem Projektverlauf eine bedienbare Benutzeroberfläche zur Verfügung, die der fachlichen Projektleitung und dem Kunden als Abnahme- und Testsystem dient. Der große Nachteil des Verfahrens ist der hohe Simulationsaufwand.

Für die Bottom-up-Integration wird entgegengesetzt begonnen. Es wird mit der **Persistenz** der Daten angefangen und Schritt für Schritt die als nächste benötigte Ebene als vollständige Implementierung integriert. Somit werden keine Simulationen benötigt, sondern lediglich Testtreiber für jede neu integrierte Schicht. Aus Kostengründen ist die Bottom-up-Integration oftmals die bevorzugte Variante der beiden Verfahren, allerdings müssen die fachliche Leitung und die Kunden, bis zum Ende der Integration warten, um ein für sie testbares System zu erhalten.[21]

---

[18] Vgl. Fowler, M. 2012
[19] Vgl. Pezzè, M. ; Young, M. 2009: S. 457
[20] Vgl. Pezzè, M. ; Young, M. 2009: S. 462
[21] Vgl. Grechenig, T. ; Bernhart, M. ; Breiteneder, R. ; Kappel, K. 2010: S.311

„In der Praxis jedoch werden Softwaresysteme nur extrem selten streng top-down oder bottom-up entwickelt.“[22] Es werden häufig auch Kombinationen angewendet, in denen von beiden Seiten begonnen wird, bis die Integrationsschritte sich in der Mitte treffen. Solch eine Kombination nennt sich Sandwich-Integration. Durch den Beginn an der Benutzeroberfläche entsteht ein Prototyp, um bereits in frühen Projektphasen Nutzerfeedback einzusammeln. Der gleichzeitige Start bei der **Persistenz** der Daten, hilft dem Entwicklerteam eventuelle Risikostellen frühzeitig anzugehen.[23]

Bei heutigen Softwareentwicklungen kann es durch die neueren agilen Projektmanagementmethoden aber auch von Vorteil sein, sich für die featureorientierten Integrationsverfahren zu entscheiden. Bei der sogenannten „[…] Thread-Integration liegt das Hauptaugenmerk auf dem Zusammenspiel von Modulen im Hinblick auf eine bestimmte Funktionalität.“[24] Als Beispiel sollte am Ende eines Sprints mit der Projektmanagementmethode „**Scrum**“ ein funktionierender Teil der Software zur Verfügung stehen. Hierfür kann nicht jede Hierarchieebene als Ganzes implementiert werden, sondern es werden alle Funktionalitäten auf jeder einzelnen Ebene entwickelt, die für die bestimmten Anforderungen des Sprints benötigt werden.

Das andere featureorientierte Verfahren, die Critical-Module-Integration, ist von dem Ablauf identisch zu der Thread-Integration. Es wird also eine komplette Anforderung über alle Ebenen vollständig entwickelt, aber die Auswahl und die Reihenfolge der Anforderungen entstehen durch das Betreiben von Risikomanagement. Ziel ist es, alle kritischen Module zuerst zu integrieren, um frühzeitig auf mögliche Schwierigkeiten zu reagieren.

Neben den inkrementellen Verfahren existiert allerdings noch die Big-Bang-Integration. „Die Big-Bang-Integration ist eine klassische und oft durchgeführte non-iterative Integrationsform, bei der alle Systemteile gleichzeitig zu einem Gesamtsystem kombiniert werden.“[25] Die Integration in der Form besitzt den

[22] Pezzè, M.; Young, M. 2009: S. 464
[23] Vgl. Pezzè, M.; Young, M. 2009: S. 464
[24] Pezzè, M.; Young, M. 2009: S. 464
[25] Grechenig, T.; Bernhart, M.; Breiteneder, R.; Kappel, K. 2010: S.310

Vorteil, dass sehr wenige Kosten für die Integration entstehen. Durch das Entfallen der Simulationen von Modulen, dem Entwickeln von Testtreibern und dem Planen der Integrationsschritte bleiben die Kosten zunächst gering. Allerdings entsteht durch die Big-Bang-Integration das Risiko, erst am Ende der Entwicklung schwerwiegende Fehler zu entdecken, die wiederum sehr hohe Kosten in der Behebung verursachen.

2.1.3 End-to-End

Auf der Spitze der Pyramide angekommen, bezieht sich das Testen, ähnlich wie bei den Integrationstest, auf das komplette System. Der große Unterschied zwischen den beiden Schichten ist der reale Testfall. Beim End-to-End-Testen werden zunächst Testszenarien entwickelt, die exakt die Benutzung eines Endnutzers widerspiegeln. Danach können die Szenarien entweder manuell durchgeführt werden oder es wird mit entsprechenden Testwerkzeugen eine Klickstrecke des Benutzers automatisiert. Wichtig ist dabei, dass wirklich das Produkt als Gesamtes betrachtet wird. Eine Aktion des Benutzers auf der Benutzeroberfläche, sollte am Ende nicht nur einen Datenfluss bis zur Datenbank auslösen, sondern auch auf der Benutzeroberfläche eine Antwort an den Benutzer zur Folge haben.[26]

Aus fachlicher Sicht sind die End-to-End-Tests besonders beliebt. Aus der Anwendungsspezifikation kann ein Skript von Testszenarien entstehen, und bei einem erfolgreichen Durchlauf der Szenarien ist sichergestellt, dass alle spezifizierten Funktionen für den Endnutzer vorhanden sind. Allerdings ist so ein Vorgehen auch kostenintensiv und kann je nach Systemgröße auch ein komplettes Team in Anspruch nehmen, welches sich nur mit dem Entwickeln und Durchführen von Testszenarien beschäftigt. Ein weiterer großer Nachteil ist, dass End-to-End-Tests erst ganz am Ende des Entwicklungsprozesses stattfinden können. Fehler, die zu diesem Zeitpunkt entdeckt werden, können noch einmal für erheblichen Aufwand sorgen. Zunächst muss ein Entwickler genauer analysieren in welchem Modul der Fehler auftaucht und ist bei der Behebung in sei-

[26] Stäudtner, J. 2016

nen Mitteln auf die Korrekturen eingeschränkt, die an dem Rest des Systems nichts verändern.[27]

2.2 Testabdeckung

Erklärungen zu dem Begriff „Testabdeckung“ sind meist unter der englischen Bezeichnung „Code Coverage“ zu finden. Hinter dem Begriff verbirgt sich eine Maßzahl die „[…] den Anteil der tatsächlich durchgeführten Tests zu den theoretisch möglichen Tests […]“[28] angibt. Die Anzahl der „theoretisch möglichen Tests“[29] ist wiederum abhängig von der gewählten Metrik. Die gängigsten Metriken sind die Anweisungsabdeckung, die Zweigabdeckung, die Bedingungsabdeckung, die Abdeckung der Bedingungskombinationen und die Pfadabdeckung.

Bei der Anweisungsabdeckung wird eine hundertprozentige Abdeckung erreicht, wenn jede Anweisung durch einen oder mehrere Testfälle abgearbeitet wird. In Abbildung 2 stellen die Knoten die Anweisungen und die Kanten den Kontrollfluss dar. In dem zuvor erläuterten Beispiel kann bereits durch eine Kantenfolge „a, b, c, e, g, i, j, k“, also lediglich einem Testfall, eine hundertprozentige Testabdeckung erreicht werden. Dennoch befindet sich bei den Kanten d, f und h ungetesteter Code.[30]

[27] Platz, W. 2015
[28] Johner, C. 2016
[29] Ebd.
[30] Aydin, G.; Pohl, H. 2012,

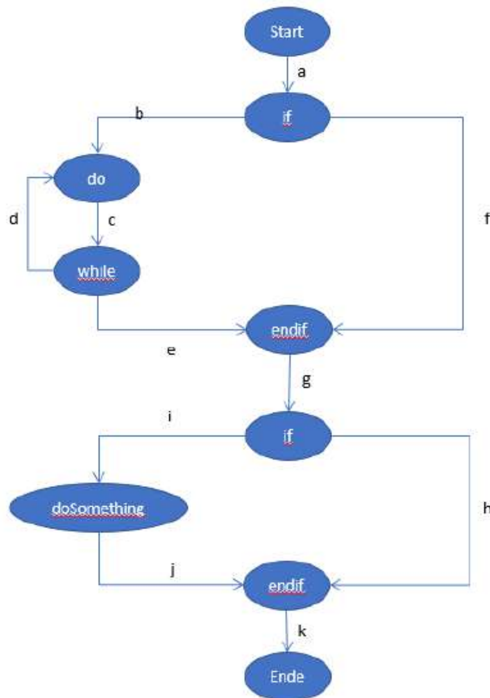

Abbildung 2: Beispiel eines Kontrollflussgraphen

Die Zweigabdeckung hingegen stellt das „Verhältnis von durchlaufenen Zweigen zur Gesamtzahl der Zweige“[31] dar. Für eine Abdeckung von einhundert Prozent müssen alle Kanten mindestens einmal durchlaufen werden. Bezogen auf die Abbildung 2 sind also zwei Testfälle „a, b, c, d, c, e, g, i, j, k“ und „a, f, g, h, k“ nötig. Die Mehrfachausführung von einigen Kanten ist nicht zu vermeiden.[32]

Eine noch striktere Metrik ist die Pfadabdeckung. Hierbei zählt nicht nur, dass jeder einzelne Zweig mindestens einmal durchlaufen wird. Zudem muss auch jeder mögliche Pfad, also Kombination von Zweigen, von dem Start bis zum Ziel durchlaufen werden. Zu den zwei eben genannten Pfaden in der Zweigabdeckung kommen nun noch die Pfade „a, b, c, d, c, e, g, h, k“ und „a, f, g, i, j, k“.

Bei der Bedingungsabdeckung zählt das Verhältnis von überprüften Termen zu der Gesamtanzahl der Terme. Für eine Bedingung wie zum Beispiel a < 6 AND b > 2 ergeben sich $2^2 = 4$ Testfälle. Jeder der beiden Terme kann entweder TRUE oder FALSE sein. Der erste Testfall prüft mit TRUE AND TRUE = TRUE

[31] Rätzmann, M. 2004: S.141

[32] Vgl. Aydin, G. ; Pohl, H. 2012,

den positiven Fall. Die nächsten drei Testfälle hingegen prüfen mit den unterschiedlichen Eingangsparametern TRUE AND FALSE, FALSE AND TRUE, FALSE AND FALSE denselben negativen Fall. Bei einer komplexeren Software kann die Anzahl an Testfällen aber auch exponentiell ansteigen, ohne einen Mehrwert zu bringen.

Um dem exponentiellen Anstieg entgegenzuwirken, ist es bei einer komplexeren Software sinnvoll, sich für die Abdeckung der Bedingungskombinationen zu entscheiden. Für diese Metrik sind nicht sämtliche verschiedenen Kombinationen der Eingangsparameter interessant, sondern das Endergebnis der Gesamtbedingung. In dem eben genannten Beispiel gäbe es dann anstatt 4 Testfällen, nur einen bei dem TRUE und einen bei dem FALSE das Endergebnis ist.

2.3 Verfahren zur Erstellung von Testfällen

Um bei der Erstellung der Testfälle die zielführendsten zu betrachten, stellt das nachfolgende Kapitel einige Verfahren vor, um gute Testfälle zu ermitteln.

2.3.1 Analyse fachlicher und technischer Konzepte

Fällt die Entscheidung, die Testfälle und die dazugehörigen Daten durch Analyse von fachlichen und technischen Konzepten zu erstellen, kann schon vor dem eigentlichen Entwicklungsprozess damit angefangen werden. Für die Analyse werden sämtliche Dokumente herangezogen, die Informationen zu dem Endprodukt enthalten wie zum Beispiel die Verträge, ein Geschäftsmodell, Ablaufdiagramme, das Lastenheft und sonstige fachliche, sowie technische Konzepte. Der Testdesigner muss bei dem Analysieren der Dokumente akribisch darauf achten ungenaue Beschreibungen nicht zu überlesen, sondern auf die entsprechenden Tilgungen, Generalisierungen und Verzerrungen aufmerksam zu machen. In dem eben beschriebenen Kontext ist die Bedeutung für eine Tilgung das Ausbleiben von wichtigen Informationen. Ein Beispiel wäre das Fehlen eines Akteurs in einer User Story. Bei jeder Beschreibung einer Funktion sollte

definiert sein wer die Funktionen ausführt. Generalisierungen hingegen sind undefinierte Verallgemeinerungen. In vielen Beschreibungen sind Wörter zu finden wie „alle“ oder „jeder“, allerdings sollten derartige Bezeichnungen in den meisten Fällen nicht wörtlich genommen werden. „Alle“ könnte für „alle registrierten Benutzer“ stehen, aber auch für „alle Menschen“. Außer den Tilgungen und Generalisierungen verbergen sich auch ungewollte falsche Darstellungen in den Dokumenten, die in dem Kontext als „Verzerrungen“ betitelt sind. Meistens besteht eine Verzerrung aus umgangssprachlichen und komplizierten Formulierungen. Alle analysierten Dokumente dienen als Testbasis und stellen somit die Quelle dar, aus der Anforderungen extrahiert werden.[33]

Eine allgemeine Vorgehensweise beim Extrahieren ist das Suchen nach Formulierungen, die ein gewünschtes Verhalten beschreiben. „Alle relevanten Stellen der Testbasis werden mit der Frage untersucht: ‚Was bedeutet das hier Beschriebene für das Testobjekt? Kann daraus ein notwendiges Verhalten des Testobjektes abgeleitet werden?‘“[34]. Wird so eine Anforderung spezifiziert, sollte sie mit dem Nennen des Testobjektes beginnen, gefolgt von dem Wort „muss“ nachdem dann die eigentliche Anforderung steht. Anforderungen in diesem Format können nun tabellarisch dargestellt und mit einem Verweis zur Textquelle versehen werden. Für spätere Änderungen an der Testbasis ist es auch wichtig, dass die entsprechende Textquelle einen Verweis zur Anforderung besitzt. Somit wird für ein einheitliches Dokument auch bei Änderungen gesorgt.

2.3.2 Datenkombinationstest

Die Testfallermittlung mit der Methode des Datenkombinationstest zielt sehr stark auf eine Testabdeckung mit der Metrik Bedingungsabdeckung ab. Bei der Ermittlung eines Testfalls mit Datenkombination wird analysiert, welche Eingabedaten und Werte den Programmablauf steuern. Es hilft, eine Tabelle mit allen möglichen Ergebnissen einer Komponente zu erstellen und im Nachhinein auf-

[33] Vgl. Rätzmann, M. 2004: S.234-237
[34] Rätzmann, M. 2004: S.138

zuschlüsseln, wie die Werte erreicht werden. Viele Entscheidungstabellen konzentrieren sich zu Beginn auf alle fehlerhaften Eingabewerte, damit es möglich ist, sich im Anschluss genauer mit den plausiblen Eingabewerten auseinanderzusetzen. Nicht alle plausiblen Eingabewerte sind jedoch relevante Testwerte, hierfür wird eine Äquivalenzklassenanalyse und gegebenenfalls zusätzlich eine Grenzwertanalyse zur Bestimmung angewendet.[35]

Eine Äquivalenzklasse bestimmt einen Wertebereich von Eingabewerten, deren Auswirkungen bei der Testausführung identisch sind. Auf die Weise muss nicht für sämtliche Werte ein Testfall entstehen, sondern lediglich ein Testwert als Repräsentant einer Äquivalenzklasse ausgewählt werden.[36] Soll eine Software zum Beispiel eine Altersidentifikation vornehmen und nur den Zugang ab 18 Jahren gewähren, reicht es eine Äquivalenzklasse von 0-17 Jahren und eine ab 18 zu definieren. Es entstehen also nur 2 Testfälle anstatt mehrerer Testfälle, um jedes erdenkliche Alter zu prüfen.

Mit der Auswahl idealer Testfälle steigt auch die Wahrscheinlichkeit einen Fehler im Code zu finden.[37] Die Äquivalenzklassen schränken die Anzahl der sinnvollen Tests schon erheblich ein, allerdings besteht weiterhin die Chance, dass einige Eingabewerte innerhalb einer Äquivalenzklasse wichtiger sind, um Fehler aufzudecken als andere. „Die Grenzwertanalyse geht von der Annahme aus, dass an Rändern von Definitionsbereichen mehr Fehler auftreten als innerhalb des Definitionsbereichs.“[38] Um die kompletten Ränder der Äquivalenzklassen abzuprüfen bezieht sich die Anzahl der Testfälle nicht mehr allein auf die Anzahl der Äquivalenzklassen, sondern auf die Grenzen. Für die meisten Wertebbereiche reicht es jede Grenze mit 3 Testfällen abzudecken. Hierfür wird einmal die Grenze selbst gewählt, zusätzlich der direkte Wert unter und über der Grenze.[39]

[35] Vgl. Rätzmann, M. 2004: S.241
[36] Vgl. Seidl, R. ; Baumgartner, M. ; Bucsics, T. 2012: S.27-28
[37] Vgl. Thaller, G. 2002: S.87
[38] Seidl, R. ; Baumgartner, M. ; Bucsics, T. 2012: S.29
[39] Vgl. Seidl, R. ; Baumgartner, M. ; Bucsics, T. 2012: S.29

2.3.3 Objektlebenszyklustest

Eine weitere Methode für die Ermittlung eines Testfalls ist der Objektlebenszyklustest. Anhand eines Objekts, welches auch in der Datenbank persistiert ist, werden alle Funktionen überprüft, die den Status des Objekts verändern. Der erste Schritt in der Methode ist das Erstellen eines Zustandsdiagramms für das Testobjekt. Eventuell wurde ein solches Diagramm sogar schon in den Dokumenten der Testbasis geplant und erstellt. Anhand der visuellen Darstellung werden im darauffolgenden Schritt Testfälle gesucht, die entweder das sogenannte Testmaß 1 oder Testmaß 2 erfüllen. Das Testmaß 1 besagt, dass alle Zustände mindestens einmal abgedeckt sein sollen. Bei dem Testmaß 2 sollen alle Zustandsübergänge berücksichtigt werden.[40] Um bei den bisherigen Begriffen der Arbeit zu bleiben, lässt sich der Objekt Lebenszyklustest auch mit den Metriken der Testabdeckung zusammenfassen. Das Zustandsdiagramm wird mit der Intention durchsucht, die Anweisungsabdeckung oder die Zweigabdeckung zu hundert Prozent zu erfüllen.

2.3.4 Datenzyklustest

Die Methode des Datenzyklustests ist eine sehr gute Erweiterung des Lebenszyklustest. Auch hier werden lediglich Objekte betrachtet, die unverändert in der Datenbank persistiert sind. Allerdings liegt der Schwerpunkt der Überprüfung auf den Datenänderungen selbst und auf den Änderungen bei verknüpften Objekten. Bei dem Lebenszyklustest wurden lediglich die Funktionen überprüft, die die Datenänderungen vollziehen sollten. Zunächst wird für einen Datenzyklustest eine **CRUD-Matrix** erstellt. CRUD steht für Create, Read, Update und Delete was übersetzt Erstellen, Lesen, Ändern und Löschen bedeutet. Eine solche **CRUD-Matrix** beinhaltet als Spalten das Testobjekt und die damit verknüpften anderen Objekte. Auf den Zeilen der Matrix sind dann die Funktionen zu finden, die Änderungen verursachen. Für jede aufgeführte Funktion wird nun die Art der Änderungen mit den Buchstaben C, R, U oder D in den Spalten der Objekte festgehalten. Anstatt für jede Funktion einen einzelnen Testfall zu kreieren, soll-

[40] Vgl. Rätzmann, M. 2004: S.244

ten sinnvolle Funktionsketten gebildet werden. Dennoch wird nach jeder einzelnen Funktion das Objekt über einen Lesemechanismus kontrolliert. Eine hundertprozentige Abdeckung erfolgt, wenn die gesamte Menge an Funktionen, die in der **CRUD-Matrix** aufgeführt ist, durch mindestens einen Testfall abgedeckt ist.[41]

2.3.5 Szenariotests

„Immer dann, wenn festgehalten werden soll, welche Schritte ein Anwender des Systems zum Erreichen eines bestimmten Ziels tun muss, entsteht eine Anwendungsfall-Beschreibung."[42] Aus den Anwendungsfall-Beschreibungen werden die Testfälle für die Szenariotests abgeleitet und so gewährleistet, dass die Testfälle möglichst nah an der alltäglichen Nutzung liegen. Wichtig hierbei ist, dass jeder Schritt des Ablaufs mindestens einmal durchgeführt wird. In vielen Projekten werden derartige Tests auf das Nötigste beschränkt, da selbst wenige manuelle Tests bereits einen hohen Aufwand mit sich bringen. Auch, wenn eine Umgebung geschaffen wird um Szenariotests automatisiert auszuführen, ist der einmalige Aufwand zum Erstellen dieser End-to-End Testumgebung und die Pflege umfangreich.

[41] Vgl. Rätzmann, M. 2004: S.248-250
[42] Rätzmann, M. 2004: S.252

3 Konzept

Eine gute Testabdeckung bezieht sich nicht darauf, dass jede Codezeile durch die Tests mindestens einmal ausgeführt wird. Vielmehr sollten alle wichtigen Codefragmente mit mehreren Kombinationen von Eingabeparametern durchlaufen werden.[43] Eine Wissensbasis für die Auswahl der verschiedenen Testarten und das Finden der passenden Testfälle wurde bereits in dem zweiten Kapitel erarbeitet. Folgendes Kapitel beinhaltet ein Konzept, welches beispielhaft anhand einer expliziten React *App* des Testgegenstands eLMS entwickelt wurde. Zunächst wird die Architektur der *App* in Bereiche geteilt, um für einzelne Teile der *App* sinnvoll zu entscheiden, welche Testart die ideale ist. Danach werden einige Merkmale von wichtigen Codestellen spezifiziert, um daran zu erkennen, welche Stellen bevorzugt getestet werden sollten.

[43] Vgl. Büchner F. 2010

3.1 Auswahl des konkreten Testgegenstands

Im Rahmen der vorliegenden Arbeit dienen die modernisierten, mit React entwickelten, Oberflächen des eLMS als Testgegenstand. Da jene bereits bestehen, sollte die Aufgliederung in verschiedene Testbereiche an dem Aufbau einer solchen React *App* erarbeitet und für die einzelnen Bereiche die entsprechende Testart definiert werden. Die React *App* „CompactCourseListOverview“ generiert die Oberfläche für eine der Hauptfunktionalitäten im eLMS (siehe Abbildung 3).

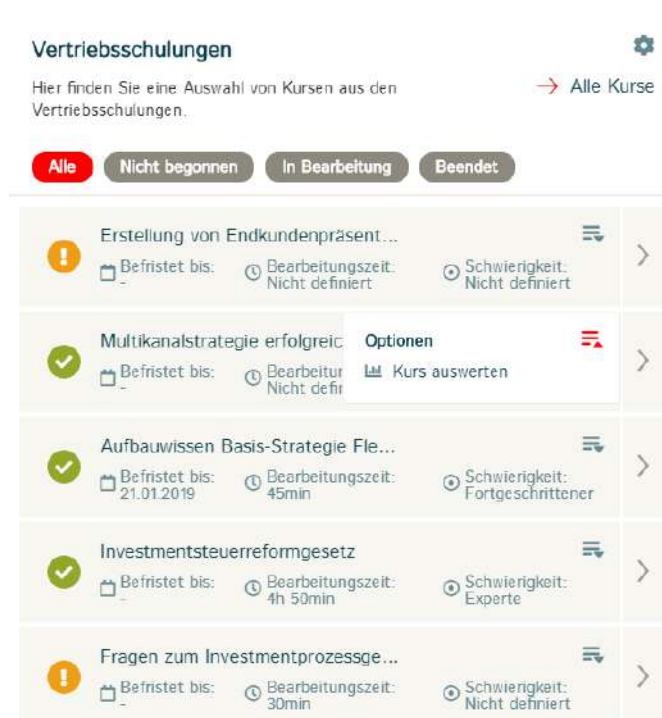

Abbildung 3 CompactCourseListOverview

Sie stellt einen Teil der Kurse dar, in denen der Benutzer eingeschrieben ist und bietet durch Menüs an jedem einzelnen Kurs Interaktionsmöglichkeiten. Des Weiteren können die Kurse mit den **Tabs** über der Liste nach einem bestimmten Status sortiert werden. Alle Funktionalitäten der React App sind den User-Stories im Anhang 3 zu entnehmen. Durch die einzelnen Komponenten der *App* zur Anzeige von Daten und der Interaktion mit den Daten sollte es realisierbar sein, sämtliche erforderlichen Testweisen zu planen und dabei das Konzept der Testpyramide zu erfüllen.

Der Aufbau und die einzelnen Teile der *App* lassen sich sehr gut an dem unidirektionalen Datenfluss der **Flux-Architektur** erklären, angefangen bei den **Actions**.

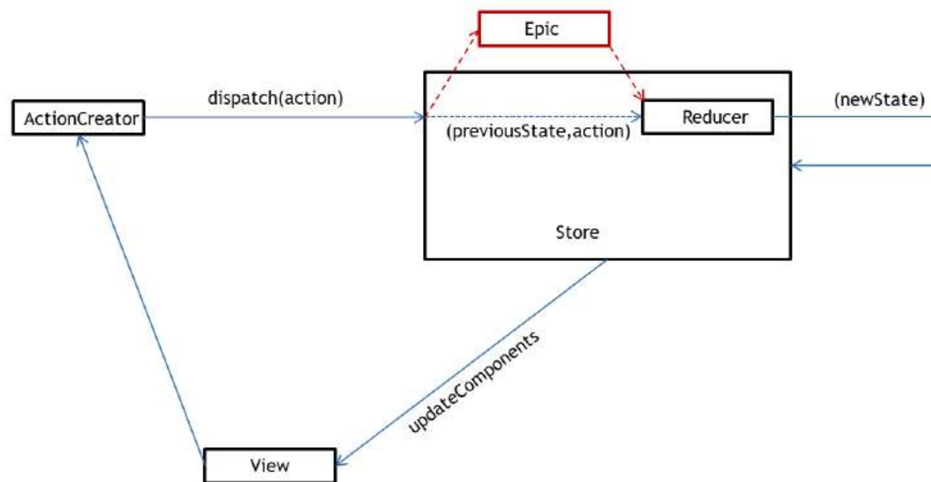

Abbildung 4 Flux/Redux Architektur[44]

Eine **Action** besteht aus einem Datensatz und einem **Actiontype**. Entsteht eine Datenänderung, wird sie per **Action** an den **Reducer** weitergegeben. Im **Reducer** wiederum ist für jeden Typ von **Actions** definiert, wie mit den mitgeschickten neuen Daten ein neuer **State** erzeugt wird. Der **State** einer React *App* definiert jederzeit den aktuellen Datenstand der *App*, der in dem Moment als HTML-Code visualisiert wird. Der neue **State** wird auf einem Stapel im **Store** abgelegt und das Ablegen stößt das generieren des HTML-Codes an. Für den HTML-Code sind die **Components** verantwortlich. Eine **Component** bezieht aus ihren **Properties** und dem aktuellen **State** alle nötigen Daten, um das gewünschte HTML-Markup zu generieren. Um eine **Component** übersichtlicher und wartbarer zu machen, werden einzelne HTML-Bausteine in **Presentationals** ausgelagert. Eine solche **Presentational** ist lediglich eine Funktion, die aus bestimmten Eingabewerten stets denselben HTML-Code generiert.[45] Für die Datenanbindung an das Backend existiert in dem beschriebenen Lebenszyklus eine Abzweigung vor dem **Reducer**, die die **Epics** in die Applikation einbindet. Ein **Epic** fängt definierte **Actions** ab und stellt mit den enthaltenen

[44] Vgl. Redux Api (o. J.)
[45] Vgl. Deutsch S. 2015

Daten eine **AJAX-Anfrage**. Das Ergebnis der Anfrage wird wieder in eine neue **Action** eingebunden, die von dem **Reducer** entgegengenommen wird.[46]

3.2 Erste Ebene: Unittests

Auf der ersten Ebene der Pyramide besteht die Aufgabe eine große Basis an Unit-Tests zu schaffen. Zunächst wären in dem Lebenszyklus der React *App* die **Actions** zu betrachten. Für diese ist keinerlei spezielle Logik abzuprüfen, stattdessen sollten die Testfälle eher durch Analyse der Konzepte entstehen. Dabei ist eine maßgebliche Überprüfung das Vorhandensein von jeglichen konzeptionierten Typen von **Actions** sowie funktionierender **Creator-Funktionen** für jeden Typus, die eine typsichere und valide Action erzeugen.

Der nächste Schritt in dem Aufbau einer React *App* ist der **Reducer**. Auch der ist sehr gut für Unittests geeignet und kann abgekapselt von dem Rest der Anwendung getestet werden. Durch die Analyse der **Actiontypes** ist bereits bekannt, welche Typen von **Actions** in der Anwendung vorhanden sind und welche Datenänderungen die jeweilige **Action** im **State** verursacht. Der generelle Aufbau des **Reducers** ist eher simpel, für jeden Typ existiert ein Zweig, der einen neuen **State** mit den neuen Daten erstellt. Letztendlich entsteht also pro **Actiontype** ein Unittest, der als Eingabeparameter eine **Action** erhält und ein Objekt des **States** zurückbekommt, welches anschließend auf Korrektheit geprüft wird. Ziel sollte hier eine hundertprozentige Zweigabdeckung sein.

Wie in dem Lebenszyklus beschrieben, löst das Speichern eines neuen **States** das Generieren des HTML-Codes aus. Den gesamten HTML-Code zu generieren erfordert allerdings das Zusammenspiel von **Actions**, dem **Reducer**, dem **State**, dem **Store** und letzten Endes der **Component**. Derartiges Zusammenspiel wird vorzugsweise in einem Integrationstest überprüft. Zunächst sollten nur die **Presentationals** als Unittest isoliert getestet werden. Hierfür sind zum einen **Snapshot-Tests** geeignet, die den generierten HTML-Code im JSON-Format speichern um bei weiteren Entwicklungszyklen ungewollte Änderungen

[46] Vgl. Salter K. 2016

wahrzunehmen. Zum anderen sollten aber auch signifikante Merkmale wie beispielsweise spezielle CSS-Klassen, Keys oder IDs überprüft werden. Die Testfälle an sich können sehr gut mit Hilfe eines Datenkombinationstest bestimmt werden. In Kombination mit der Äquivalenzklassenanalyse hätte Besagtes auch den Vorteil, dass alle möglichen HTML-Code-Varianten lediglich einmal überprüft werden müssen. Darüber hinaus sorgt es automatisch für eine hundertprozentige Abdeckung der Bedingungskombinationen.

Des Weiteren entstehen beim Entwickeln von komplexeren React **Components** auch häufig Hilfsfunktionen in denen Business-Logik isoliert ist. Derartige Funktionen liegen entweder in der **Component** selbst oder sind als öffentliche Funktion in Hilfsklassen ausgelagert. In beiden Fällen sind die Hilfsfunktionen optimal über Unittests abzudecken.

Zuletzt fehlen ein separater Test für den **Application-State** und ein Unittest für die Grundfunktionen des **Stores**. Der Unittest für den **Application-State** kann lediglich Instanzen des **States** anlegen und die Instanzen im Nachhinein auf Typ und Attribute testen. In vielen React *Apps* gibt es **States,** die auch optionale Attribute haben. Bei solchen Fällen ist es ratsam den **State** einmal mit der minimalen und einmal mit der maximalen Anzahl an Attributen zu bilden. Auch der Unittest für den **Store** beschränkt sich auf das Bilden einer Instanz und die Überprüfung, ob die Grundfunktionalitäten definiert sind. Weitere funktionale Tests für den **State** und den **Store** folgen auf der Integrationsebene.

3.3 Zweite Ebene: Integrationstest

In der zweiten Ebene der Pyramide entsteht schrittweise die Gewissheit, ob alle einzelnen Bausteine der React *App* auch zusammen als eine *App* funktionieren.

Wie bereits beschrieben, existiert innerhalb einer React Anwendung ein unidirektionaler Datenfluss in Form eines Kreises. Zunächst gilt es den Kreis mittels Abstraktion auf eine hierarchische Struktur abzubilden. Eine hierarchische Struktur eines Softwaresystems beginnt auf der untersten Ebene meist mit der **Persistenz** der Daten. Abstrahiert auf die gegebene Struktur einer React Anwendung, wäre der **Store** die **Persistenz** der Daten. Auf der nächsthöheren

Ebene folgen die **Reducer** mit den **Actions**. Darüber liegen die **Epics**, die für die Anbindung an einen externen Service sorgen. Als letzte und höchste Ebene folgt die Benutzerschnittstelle, also die **Components**. In der Reihenfolge, nach dem Bottom-up-Prinzip, lassen sich drei Integrationsschritte bis zur kompletten Anwendung durchführen. Der erste Testschritt legt einen **Store** mit einem initialen **State** an. Der **Store** versendet pro Actiontype eine **Action** und der Test verfolgt jene via **Spies** durch den **Reducer**. **Spies** können innerhalb des Tests deklariert werden und überwachen den Aufruf einer Funktion. Am Ende folgt eine Kontrolle, ob der **Reducer** einen **State** mit den korrekten Datenänderungen im **Store** hinterlegt hat. Der zweite Schritt konzentriert sich auf die **Actions**, die eine **AJAX-Anfrage** auslösen. Der Testaufbau des zweiten Schritts ist weitestgehend identisch, allerdings empfängt ein **Epic** die von dem **Store** versendeten **Actions**. In dem **Epic** wird die Kommunikation mit dem externen Service simuliert und eine weitere **Action** zum Setzen der empfangenen Daten an den **Reducer** versendet. Ab diesem Punkt ist der Testaufbau wieder identisch. Durch die Simulation des externen Services, auch **Stub** genannt, existiert eine volle Kontrolle über die Testdaten, die es ermöglicht besondere Zustände von Datenkombinationen und Fehlerfälle zu überprüfen. In dem letzten Schritt erfolgt die Integration der Benutzerschnittstelle. Ähnlich wie in den Snapshot-Unittests wird ein virtuelles DOM erzeugt, in das die **Component** geladen und mit einem initialen **Store** verknüpft. Die **Component** selbst übernimmt das Versenden der **Actions** und versucht die Daten anzufragen. Über **Spies** kann der genaue Ablauf der Funktionen verfolgt werden und durch Simulation des externen Datenservices werden Daten für verschiedene HTML-Varianten in die **Component** eingespeist.

3.4 Dritte Ebene: End-to-End-Test

Auf der Spitze der Pyramide, dem End-to-End-Testen, entstehen zunächst Klickstrecken die eine reale Benutzung eines Nutzers wiederspiegeln. Eine gute Basis dafür sind User Stories, die entweder am Anfang in der Planungsphase des Projektes entstanden sind oder an dieser Stelle vor den Klickstrecken geplant werden.

Für eine optimale Testabdeckung des ausgewählten Testgegenstands, wird als erstes überprüft, ob der Nutzer die React *App* auf seinem Dashboard (siehe Anhang A4) findet. Ist der Test positiv, findet eine Überprüfung aller Wechselwirkungen bei dem Bedienen der **Tabs**, dem Kursmenü und den Einstellungsmöglichkeiten statt.

3.5 Kriterien von wichtigen Codestellen

Ein sehr wichtiger Faktor in einem Projekt ist die Zeit. Gerade gegen Ende eines Projekts wird gerne deswegen auf das Schreiben von Tests verzichtet. Um dem entgegenzuwirken ist es von Bedeutung ein Vorgehen zu definieren, welches es ermöglicht zu entscheiden, ob eine gerade entwickelte Funktion direkt zu testen ist oder der Zeitpunkt des Testens hintenangestellt wird. Auf diese Weise kann Zeit für unwichtige Tests gespart werden und wichtige Stellen im Code sind trotzdem abgesichert. Im Anhang A2 befindet sich ein Fragebogen[47,48,49] der im Rahmen dieser Arbeit entstanden ist und mit deren Hilfe ein Entwickler die relevanten Stellen identifizieren kann. Sollte nur eine der Fragen mit „Ja“ beantwortet werden, verfügt der Testgegenstand über eine Komplexität, die sich zu testen lohnt. Die erste Frage konzentriert sich auf falsche oder fehlerhafte Eingabewerte. Grundlegend hierbei ist, dass der Entwickler sich nicht auf eine gute Typisierung verlässt, sondern überlegt, ob es Werte gibt, die einen Fehler verursachen, wie die Überschreitung eines Zahlenbereichs. Die zweite Frage beschäftigt sich mit der Art, wie ein Eingabewert weitergenutzt wird. Sollte einer der Eingabewerte zum Beispiel ein Objekt sein, könnte es bei dem Zugriff auf das Objekt zu einem **ReferenceError** kommen, in dem ein Attribut oder eine Funktion in dem Objekt fehlt oder das Objekt selbst nicht definiert ist. Als nächstes folgt eine Frage zur inneren Struktur der Funktion. Sollte die Funktion durch IF-Verzweigungen mehrere Abarbeitungswege beinhalten, muss sichergestellt sein, dass in jedem entsprechenden Szenario die passende Verzweigung gewählt wird. Auch die vierte Frage bezieht sich auf den Inhalt

---

[47] Vgl. Vorlesung Uni Bremen: Software-Technik von Prof. Dr. Rainer Koschke
[48] Vgl. Martin, R.; Ottinger, T.; Langr, J.; et. al. 2009: S.103-112
[49] Vgl. Rätzmann, M. 2004: S.161-164

und besagt, dass jede Funktion, die eine Berechnung beinhaltet auf Zuverlässigkeit und Richtigkeit zu prüfen ist. In der letzten und fünften Frage, die sich ein Entwickler stellen kann, geht es um die Umwandlung von Daten in ein anderes Datenformat. In solchen Fällen ist der Gedanke ähnlich wie bei der ersten Frage, gibt es bestimmte Werte, wie eine führende Null bei Zahlen, die Fehler auslösen.

Ziel des Fragenbogens ist es, den Entwicklungsprozess zu optimieren und während des Prozesses den neu entstanden Code, als „wichtig zu testen“ oder „unwichtig zu testen“ zu klassifizieren. Zuerst benannte Codebestandteile sollten direkt über automatisierte Tests abgedeckt werden und zuletzt genannte sollten wegen Zeitersparnissen gegen Ende des Projektes folgen.

4 Testframeworks im React-Kontext

Es gibt etliche Javascript und Typescript **Testframeworks** die für React geeignet sind. Im folgenden Kapitel wird eine Auswahl der bekanntesten **Testframeworks** vorgestellt und entschieden, welches zur Umsetzung der in vorliegender Arbeit entstehenden Spezifikation und des Konzeptes genutzt wird.

4.1 Eigenschaften und Funktionalitäten von Testframeworks

Ein **Testframework** muss nicht nur aus einem **Testrunner**, der die Tests auf der Konsole oder in dem Browser ausführt, bestehen, sondern sollte auch eine Vergleichs-Bibliothek mitbringen, die Funktionen zur Verfügung stellt, um Testergebnisse mit erwarteten Ergebnissen zu vergleichen. Zu diesen beiden wichtigen Eigenschaften existieren noch einige weitere Methodiken wie **Spies**, **Stubs**, **Mocks** und das generieren von **Markup-Snapshots** die ein **Testframework** mitbringen sollte. **Spies** dienen zum Beispiel zur Überwachung von Funktionsaufrufen. Mit ihnen kann festgelegt werden, wie oft und mit welchen Parametern eine Funktion aufgerufen werden muss. **Stubs** hingegen, dienen dazu in einem Test einen externen Datenservice zu simulieren. Auf diese Weise ist es möglich, dass Absenden und Empfangen von Daten an eine Rest-API zu überprüfen, ohne dass der Test wirklich mit dem Service kommunizieren muss. Ein ähnliches Prinzip steckt auch hinter den **Mocks**. In einem Unittest wird eine einzelne Funktion getestet, sämtliche Abhängigkeiten zu anderen Funktionen und Klassen werden durch **Mocks** simuliert, damit der Unittest isoliert von dem Rest des Systems laufen kann. Der letzte entscheidende Punkt für das Testen von Frontend Applikationen ist das Snapshot testen. Ein Snapshot hält den genauen Aufbau einer grafischen Benutzeroberfläche fest. Sollte nach Wartungsarbeiten oder Weiterentwicklungen eine unbeabsichtigte Änderung vorliegen, wird sie durch den erstellten Snapshot erkannt.

4.2 Framework: Mocha

Eines der bekanntesten Javascript **Testframeworks** ist Mocha. Mocha ist in erster Linie ein **Testrunner**, der es ermöglicht Tests auf der Konsole und in einem Browser zu starten. Allerdings bringt Mocha keine von den oben genannten wichtigen Eigenschaften und Methodiken von Haus aus mit, sondern benötigt das Einbinden von weiteren Bibliotheken. Sehr häufig findet man Mocha in Kombination mit der Vergleichs-Bibliothek Chai und für die Szenarien in denen **Spies**, **Stubs** und **Mocks** gebraucht werden, wird meistens die Bibliothek Sinon eingebunden. Mocha selbst ist also spezialisiert darauf ein **Testrunner** zu sein und ermöglicht es beliebige weitere Bibliotheken einzubinden.[50]

4.3 Framework: Karma

Ähnlich aufgebaut wie Mocha, ist auch das **Framework** Karma. Karma wurde ursprünglich von dem AngularJs Team entwickelt, um speziell auf Angular zugeschnittene Tests laufen zu lassen. Wie auch das **Framework** Mocha, ist Karma allerdings nicht wirklich ein komplettes **Testframework**. Karma startet einen eigenen HTTP-Server und stellt einen **Testrunner** zur Verfügung, in dem es möglich ist, eine favorisierte Testbibliothek einzubinden und seinen Code in allen gängigen Desktop- und Mobile-Browsern zu testen. Da es möglich ist sämtliche Testbibliotheken einzubinden eignet sich Karma nicht nur für Angular Anwendungen, sondern kann auch für React und vielen anderen Javascript **Frameworks** verwendet werden.[51]

4.4 Framework: Jest

Das etwas neuere Javascript **Testframework** Jest stammt von Facebook und wurde 2014 an die Open-Source-Gemeinde übergeben. Jest eignet sich für das Ausführen von Tests über die Kommandozeile, aber auch das Bauen einer Oberfläche kann in einer virtuellen DOM-Implementierung getestet werden, oh-

[50] Vgl. MochaJs Api (o. J.)
[51] Vgl. Karma Api (o. J.)

ne dass ein Browser gebraucht wird. Neben den Funktionalitäten eines **Testrunners** stellt Jest auch eine Vergleichs-Bibliothek, ermöglicht **Spies** und bietet umfangreiche **Mock**- und **Stub**-Funktionalitäten. Doch das Alleinstel-Alleinstellungsmerkmal von Jest ist die Idee der **Markup-Snapshots**. Bei den Tests wird das HTML-Markup der Benutzeroberfläche in dem virtuellen DOM gebaut und abgespeichert. Ein erneutes Ausführen der Tests vergleicht automatisch das frisch erstellte HTML-Markup mit dem zuletzt abgespeicherten.[52] Andere **Frameworks** besitzen ähnliche Funktionalitäten, dabei werden allerdings Bilder von der Oberfläche gespeichert und miteinander verglichen. Die Nachteile der Bildvergleiche, gegenüber der neueren Methode von Jest, sind die Abhängigkeiten zu Browsern und der verwendeten Hardware. Schlägt ein solcher Bildvergleich fehl, ist es nicht sicher, ob der Fehler in der Komponente liegt oder ob der genutzte Browser durch eine Aktualisierung zum Beispiel ein neues Menu erhalten hat. Zudem können durch Änderungen an der Hardware minimale Farbänderungen auftreten, die mit dem menschlichen Auge nicht ersichtlich wären, aber den Bildvergleich fehlschlagen lassen.[53]

4.5 Testwerkzeug-Bibliotheken

Neben den Frameworks Mocha, Karma und Jest, sind auch noch die Testwerkzeug-Bibliotheken Enzyme, Sinon erwähnenswert, sowie auch Selenium und Puppeteer für die Automatisierung von Klickstrecken.

Enzyme ist ein speziell für React Anwendungen entwickeltes Werkzeug, um in den Tests auf drei verschiedene Arten das HTML-Markup zu erstellen. Die erste Art, das Shallow-Rendering, ist hilfreich um eine Komponente abgekapselt als alleinstehende Unit zu untersuchen. Das Full-DOM-Rendering wird benötigt, um den kompletten Lebenszyklus einer Komponente zu testen. Die dritte Variante ist das Static-Rendering. Bei der Variante kann optimal auf valide HTML-

[52] Vgl. Facebook Inc. Jest Api (o. J.)
[53] Vgl. Barlev, E. 2018

Struktur getestet werden. Alle drei Arten geben ein Wrapper-Objekt zurück, welches das Traversieren und Manipulieren der Komponente erleichtert.[54]

Sinon hingegen hilft eher bei den Tests, die sich nicht auf die grafische Oberfläche beziehen, sondern stellt Hilfsmittel für Unit Tests, bei denen Abhängigkeiten zu anderen Klassen oder externen Services bestehen. Genauer gesagt bietet Sinon eine leichte Herangehensweise für die bereits erklärten Methoden **Spies**, **Stubs** und **Mocks**. [55]

Selenium sowie auch Puppeteer sind für das automatisieren von End-to-End-Tests geeignet. Beide Bibliotheken bieten über CSS-Selektoren eine leichte Navigation auf der Oberfläche und können sämtliche Benutzerinteraktion, wie das Klicken von Links oder Befüllen von Eingabefeldern, übernehmen. [56]

4.6 Auswahl im Hinblick auf das Konzept

Um jede Ebene der Pyramide automatisiert umzusetzen, sollte das **Testframework** oder die Kombination von einem **Testframework** und den Werkzeug-Bibliotheken, jede in Kapitel 4.1 beschriebene Eigenschaft besitzen. Letztendlich fiel die Entscheidung auf das **Framework** Jest. Jest ist extra für React Anwendungen entwickelt und somit einfach zu integrieren. Allerdings wird auch hier die Einbindung aller drei Werkzeug-Bibliotheken benötigt, da die mit Jest mitgelieferten Werkzeuge lediglich minimalistisch sind. Der große Vorteil gegenüber den anderen beiden **Frameworks** ist allerdings, dass Jest für die jeweiligen Kombinationen mit den Werkzeug-Bibliotheken Zusatzpakete bereitstellt, um den manuellen Konfigurationsaufwand so gering wie möglich zu halten.[57] Durch die Auswahl kann ein Großteil der Unittest allein mit dem **Framework** Jest entwickelt werden. Für die **Snapshot-Tests** auf der Unittest-Ebene ist Enzyme eine perfekte Ergänzung um Komponenten abgekapselt von anderen zu überprüfen. Auf der Integrationsebene der Pyramide ist es durch

[54] Vgl. Airbnb Inc. Enzyme Api (o. J.)
[55] Vgl. SinonJs Api (o. J.)
[56] Vgl. Puppeteer Api (o. J.)
[57] Vgl. Facebook Inc. Jest Api (o. J.)

Sinon möglich, auch tiefer liegende Funktionalitäten, wie ausgelöste **AJAX-Anfragen** zu simulieren. Und durch die Kombination mit Puppeteer, kann mit demselben **Testrunner** und demselben Programmierstil eine Umgebung geschaffen werden, die es ermöglicht sich auf einem Testsystem einzuloggen und dort automatisiert Klickstrecken auszuführen. Für Selenium sind zwar auch Packages für eine leichte Integration in Jest vorhanden, allerdings empfiehlt Jest die Kombination mit Puppeteer. Die anderen beiden **Testframeworks** Mocha und Karma waren in der Erstkonfiguration viel aufwändiger, sodass nach einigen Tagen noch kein vielversprechendes Testen einer React *App* möglich war.

5 Praktische Umsetzung

Das Kapitel führt durch wichtige und schwierige Teile der Umsetzung des Konzeptes und beleuchtet das Anwenden von Methoden zur Testfallerstellung beispielhaft. Zudem werden die jeweiligen Schwierigkeiten und Probleme bei Implementierungen der Tests und Nutzung der Technologien erläutert.

5.1 Umsetzung Unittest

Die ersten Unittests ließen sich, wie in dem Konzept geplant, problemlos anhand der fachlichen und technischen Beschreibungen des Testgegenstands entwickeln. Für die nachfolgenden **Snapshot-Tests** der **Presentationals** diente zur Testfallermittlung die Methode „Datenkombinationstest“. Aufgrund mangelnder technischer Dokumentation der **Presentationals**, musste zunächst eine Codeanalyse stattfinden, um den Programmablauf zu verstehen und die verschiedenen HTML-Varianten zu identifizieren.

Das Ergebnis der Code-Analyse für die **Presentational** BlockHeader, die in der Abbilddung 5 rot eingerahmt ist, befindet sich in Tabelle 1.

Abbildung 5 Visuelle Darstellung der Presentational BlockHeader

Variante 1	Nur Headline
Variante 2	Nur SubHeadline
Variante 3	Nur SubHeadline mit Icon
Variante 4	Headline und SubHeadline
Variante 5	Nichts

Tabelle 1 Ergebnis Codeanalyse BlockHeader

Nach der Code-Analyse ergab der eigentliche Datenkombinationstest untenstehende Tabelle 2. Ein „true“ steht dafür, dass der Eingabewert gesetzt ist und ein

„false“ dafür, dass der Eingabewert nicht gesetzt ist. Die fünf verschiedenen Ausgabewerte sind auch gleichzeitig die Äquivalenzklassen. So entstand pro Äquivalenzklasse ein Test.

Headline	SubHeadline	Icon	Ausgabe
true	true	true	Variante 4
true	true	false	Variante 4
true	false	true	Variante 1
true	false	false	Variante 1
false	true	true	Variante 3
false	true	false	Variante 2
false	false	true	Variante 5
false	false	false	Variante 5

Tabelle 2 Ergebnis des Datenkombinationstest für den BlockHeader

Durch das Einhalten des Konzepts und die damit verbunden Strategien zur Testfallermittlung, sind für den Testgegenstand 64 Unittests entstanden. Das in Jest integrierte Code Coverage Tool, bietet Aussagen darüber wieviel Prozent aller Funktionen und wieviel Prozent aller Codezeilen abgedeckt sind. Durch die in der Bachelorarbeit erstanden Unittests existiert bereits eine 72.38% Funktions- und 79.09% Codezeilen-Abdeckung.

5.2 Umsetzung Integrationstest

Der Integrationstest konnte eins zu eins, wie im Konzept vorgenommen, umgesetzt werden. Dabei ermöglichte die Bibliothek Sinon das Verfolgen der **Actions** im **Reducer** und die Simulation der **AJAX-Anfrage** im **Epic**. Im dritten Integrationsschritt wurde das Laden des HTML-Codes in ein virtuelles DOM durch Enzyme sehr erleichtert.

Des Weiteren ist bei der Analyse der Code Coverage aufgefallen, dass zwei ausgelagerte **Epics** von Unterkomponenten nicht im Konzept berücksichtigt wurden. Für den Test der beiden **Epics**, diente Sinon sowohl zur Simulation eines **Stores**, als auch für die Simulation der **AJAX-Anfragen**.

Alle drei Integrationsschritte haben dafür gesorgt, dass auch die Verbindungspunkte der einzelnen Komponenten getestet wurden. Letztendlich stieg die Funktionsabdeckung dadurch auf 88.81% und die Codezeilen Abdeckung auf 93.63%.

5.3 Umsetzung End-to-End-Tests

Bei den End-to-End-Tests kann nicht alleine das Frontend als Testgegenstand bezeichnet werden, sondern auch das dazugehörige Backend sollte funktionieren. Für diese Problematik wurde ein komplettes Testsystem aufgesetzt, in dem die React *App* eingebunden ist. Laut der User-Stories aus Anhang A3 soll sich der CompactCourseListOverview auf dem individuellen Dashboard des Nutzers befinden und dort einen schnellen Zugang zu den für ihn sichtbaren Kursen liefern. Des Weiteren kann der Nutzer die Liste nach einem Status sortieren und die Listengröße festlegen. In der Tabelle 3 sind alle Klickstrecken aufgeführt, die ein Nutzer vornehmen muss, um sämtliche Funktionalitäten zu überprüfen.

Nr.	Klickstrecke	Ergebnis
1.	Login → klicke auf einen Kurs	System navigiert zu dem Kurs
2.	Login → klicke auf Tab „Nicht begonnen“	Die Kursliste zeigt nur noch Kurse die den Status „Nicht begonnen“ haben
3.	Login → klicke auf Menu-Icon bei einem Kurs → klicke auf „Kurs auswerten“	System navigiert zu der entsprechenden Kursauswertung
4.	Login → klicke auf das Steuerrad → wähle Kurslistenlänge gleich 3	Die Kursliste zeigt nur noch 3 Einträge
5.	Login → klicke auf das Steuerrad → wähle Kursstatus „ Nicht begonnen“ → Lade die Seite neu	In der Kursliste ist direkt der Tab Nicht begonnen aktiv

Tabelle 3 Klickstrecken für den End-to-End-Test

Der Befehl „Login“ in den Klickstrecken beinhaltet den kompletten Anmeldeprozess an dem Testsystem. Nach dem erfolgreichen Anmelden befindet der Nutzer sich auf seinem Dashboard, wo die React *App* eingebunden ist. Die eigentliche Implementation der Klickstrecken erfolgte ohne Probleme mit der Bibliothek Puppeteer. Allerdings wurde nicht bedacht, dass einige Klickstrecken

wichtige Daten in der Datenbank verändern. Die Änderungen führten bei erneutem Ausführen der Tests zu Fehlern. Damit die Tests in Zukunft automatisiert laufen, wurden die Klickstrecken 4. und 5. um das Zurückstellen auf die Startwerte erweitert.

6 Zusammenfassung und Ausblick

Ziel der Bachelorarbeit war es, eine mit React entwickelte Benutzeroberfläche, nach dem Konzept der Testpyramide, automatisiert auf ihre Korrektheit zu überprüfen. In der Vergangenheit wurden bei der engram GmbH die Benutzeroberflächen vor der Auslieferung entweder manuell von dem Entwicklerteam überprüft oder es wurden mit Selenium automatisierte Klickstrecken angelegt. Beide Wege erzeugen erheblichen Aufwand und eventuelle Fehler werden erst gegen Ende des Projekts entdeckt. Der in der Bachelorarbeit durchgeführte Weg der Testpyramide fordert, dass 70 Prozent aller Tests Unittests sein müssen. Solche Tests sind kostengünstig und können bereits rechtzeitig während der Entwicklungsphase mit implementiert werden. Selbst eine Vorgehensweise nach dem Prinzip „TDD“ wäre denkbar. Zu Begin der Arbeit wurde die Testpyramide erläutert und eine theoretische Wissensbasis über Testarten, Testabdeckungsmetriken und Verfahren zur Erstellung von Testfällen geschaffen. Basierend auf dem Wissen entstand ein ebenfalls theoretisches Konzept für eine React Anwendung. Trotz der konkreten Auswahl des Testgegenstandes, ist das Konzept lediglich an dem Lebenszyklus einer React Anwendung angelehnt und sollte aus dem Grund auch für weitere mit React entwickelte Benutzeroberflächen anwendbar sein.

Bei der Auswahl der **Testframeworks** konzentriert sich die Arbeit darauf, ein **Framework** zu finden, welches die wesentlichen Technologien mitbringt, die laut dem Konzept benötigt werden. Für die Unittests war dies ein **Testrunner** mit einer Vergleichsbibliothek, für die Integrationstests kamen **Spies**, **Stubs** und **Mocks** hinzu und für die wenigen abschließenden End-to-End-Tests brauchte es Technologien, um automatisierte Klickstrecken im Browser auszuführen. Letztendlich kam in der praktischen Umsetzung eine Kombination von Jest, Enzyme, Sinon und Puppeteer zum Einsatz. Die Implementierung der Tests mit den **Frameworks** und Bibliotheken war erfolgreich und die von Jest mitgelieferte Technologie zur Analyse von Testabdeckung zeigt, dass bereits die Unit und Integrationstests ein hohes Maß an Sicherheit zur fehlerfreien Anforderungsabdeckung liefern. Es ist dementsprechend ausreichend, dass nur

10 Prozent aller Tests, aufwändige End-to-End-Tests sind, um eine hohe Qualität vor der Auslieferung zuzusichern.

Für die Zukunft kann das Entwicklungsteam bei der engram GmbH nicht nur im Backend nach dem Konzept der Testpyramide arbeiten, sondern auch im Frontend, bereits während der Entwicklung auf gute Unittest wertlegen. Das Ziel der Änderung im Entwicklungsprozess ist, dass fachliche Anforderungen früher mit den verantwortlichen Personen diskutiert werden und aus den Gesprächen bereits erste Testfälle entstehen. Durch solch ein Vorgehen ist es leichter die gewünschte Produktqualität zu erreichen und früh bei möglichen Risiken zu handeln.

Um bei der Weiterentwicklung von bestehenden und neuen React Oberflächen sicher zu sein, dass die Änderungen keine Nebeneffekte mit sich bringen, ist es ratsam schnellstmöglich für alle bereits bestehenden React-Anwendungen, das Testkonzept der Arbeit umzusetzen.

7 Literaturverzeichnis

Airbnb Inc. Enzyme Api (o. J.)

https://airbnb.io/enzyme/ , Stand 30.01.2019

Aydin, G.; Pohl, H. (2012),

Code Coverage-Tools

https://www.softscheck.com/pdf/120725_Code_Coverage.pdf , Stand 04.01.2019.

Barlev, E. (2018)

Why Screenshot Image Comparison Tools Fail With Dynamic Visual Content

https://applitools.com/blog/why-screenshot-image-comparison-tools-fail , Stand 13.02.2018

Bindick, S. (2018)

Test-driven Development: Best Practices für beste Qualität

https://entwickler.de/online/development/methoden-best-practice-test-driven-development-579831664.html , Stand 12.02.2018

Büchner, F. (2010)

Fünf Irrtümer über Code Coverage

https://www.elektronikpraxis.vogel.de/fuenf-irrtuemer-ueber-code-coverage-a-252993/ , Stand 11.02.2019

Deutsch, S. (2015)

Die Flux-Architektur und React

https://reactjs.de/artikel/react-flux-architektur/ , Stand 11.02.2019

Facebook Inc. Jest Api (o. J.)

https://jestjs.io/docs/en/api , Stand 30.01.2019

Fowler, M. (2012),
TestPyramid
https://martinfowler.com/bliki/TestPyramid.html, Stand 02.12.2018.

Grechenig, T.; Bernhart, M.; Breiteneder, R.; Kappel, K. (2010),
Softwaretechnik, München.

Hansbauer, G. (o. J.)
Testautomatisierung von UI-Tests mit Selenium und Appium
https://www.testbirds.com/wp-content/uploads/Testbirds_Whitepaper_Testautomatisierung_DE.pdf , Stand 11.02.2019

Karma Api (o. J.)
https://karma-runner.github.io/latest/index.html , Stand 01.02.2019

Martin, R.; Ottinger, T.; Langr, J.; et. al. (2009)
Clean Code, Upper Saddle River, NJ.

MochaJs Api (o. J.)
https://mochajs.org/ , Stand 30.01.2019

Osherove, R. (2015),
The Art of Unit Testing, Deutsche Ausgabe, 2. Auflage, Frechen.

Pezzè, M.; Young, M. (2009),
Software testen und analysieren, München.

Platz, W. (2015),
End-to-End-Testing, Die Inversion der Testpyramide
https://www.sigs-datacom.de/uploads/tx_dmjournals/Platz_OTS_Testing_15.pdf , Stand 06.12.2018.

Puppeteer Api (o. J.)

https://pptr.dev/ , Stand 01.02.2019

Rätzmann, M. (2004),

Software-Testing & Internationalisierung, 2. Auflage, Bonn.

Redux Api (o. J.)

Data-Flow

https://redux.js.org/basics/data-flow, Stand 14.02.2019

Salter K. (2016)

Epic Middleware in Redux

https://medium.com/kevin-salters-blog/epic-middleware-in-redux-e4385b6ff7c6 , Stand 11.02.2019

Seidl, R.; Baumgartner, M.; Bucsics, T. (2012),

Basiswissen Testautomatisierung, 1. Auflage, Heidelberg.

SinonJs Api (o. J.)

https://sinonjs.org/releases/v7.2.3/ Stand 01.02.2019

Stäudtner, J. (2016),

Definition von end-to-end Tests

http://www.cridon.de/definition-von-end-to-end-tests , Stand 06.12.2018.

Thaller, G. (2002),

Software-Test, 2. Auflage, Hannover.

Johner, C. (2016),

Code Coverage: Vollständigkeit von Software-Tests bestimmen

https://www.johner-institut.de/blog/iec-62304-medizinische-software/code-coverage/ , Stand 04.01.2019.

VI Anhangsverzeichnis

A1 Glossar A-1
A2 Hilfs-Fragebogen A-4
A3 User Stories zur Kursliste A-5
A4 Screenshot des Benutzerdashboards A-6
A5 Sourcecode A-7

A1 Glossar

Framework	Ein Framework bietet dem Programmierer ein Gerüst, in dem er eine Anwendung bauen kann.
Testframework	Siehe Glossareintrag „Framework“.
High-Level-Tests	Ein Test der in einer späteren Teststufe erfolgt, wie zum Beispiel Integrationstests.
Persistenz	In der Softwareentwicklung steht die Persistenz der Daten für Befähigung, Daten über lange Zeit in einem Speicher festzuhalten.
Scrum	Eine Projektmanagement Methode für ein agiles Vorgehen in der Softwareentwicklung.
CRUD-Matrix	Eine tabellarische Ansicht, aus der für alle Funktionen die jeweiligen Rechte ersichtlich sind. Eine Funktion kann das Recht haben Daten zu erstellen, zu lesen, zu bearbeiten oder zu löschen.
Action	Ein Objekt mit dem im React Framework der Datenfluss gesteuert wird.
Actiontype	Anhand des Typus einer Action, entscheidet sich der weitere Datenfluss in der React App.
Reducer	Ein Bestandteil einer React App der für sämtliche Actions die Datenänderungen im State vornimmt.
State	Der State oder auch Application-State definiert einen Datensatz der React App.
Application-State	Siehe Glossareintrag „State“.
Store	Der Speicherort in dem alle States abgelegt werden.

Component	Der Part einer React App, die für das generieren des HTML-Codes verantwortlich ist.
Properties	Daten die beim Aufrufen einer Component, an die Component übergeben werden.
Presentational	Eine Funktion in der ein Teil des HTML-Codes ausgelagert ist. Meistens wird eine Presentational von mehreren Components verwendet.
Epic	Ein Epic reagiert auf einen bestimmten Actiontype und stellt eine Verbindung zu dem entsprechenden Services im Backend her.
AJAX-Anfrage	AJAX steht für „Asynchronous Javascript and XML" und ist ein Konzept für das Empfangen und Senden von Daten zwischen einer Webanwendung dem Datenserver.
Spies	Objekte die eine bestimmte Funktion überwachen und sämtliche Aufrufe der Funktion protokollieren.
Tabs	Eine Art von Knöpfen, die den darunterliegende Inhalt Steuern.
ReferenceError	Der Fehler tritt zur Laufzeit des Programms auf, wenn z.B. ein Objekt nicht richtig referenziert ist. In Javascript ist eine typische ReferenceError-Meldung: „‘x‘ is not defined"
Testrunner	Eine Bibliothek die aus einer Ordnerstruktur alle Tests identifiziert und mit einer Reihe an Einstellungen ausführt. Die Ergebnisse werden in die Konsole oder eine Logdatei geschrieben.

Stubs	Objekte die bei einem Aufruf immer dieselbe vorher definierte Ausgabe liefern.
Mocks	Ein Objekt, welches ein eigentlich benötigtes Objekt ersetzt. Dadurch können Abhängigkeiten aufgelöst werden und somit einzelne Units isoliert von anderen getestet werden.
Markup-Snapshots	Ein HTML-Markup, welches für spätere Vergleiche im JSON-Format gespeichert wird.
Snapshot-Tests	Tests die ein Markup-Snapshot speichern und dieses mit vorherigen Testdurchläufen vergleichen.
Flux-Architektur	Eine Architekturbeschreibung die einen unidirektionalen Datenfluss vorsieht und den aktuellen State der Applikation in Stores speichert.
Creator-Funktion	Eine Funktion die eine typsichere und valide Action erzeugt.

A2 Hilfs-Fragebogen

1. Kann die Funktion falsche oder fehlerhafte Eingabewerte entgegenneh- men?

2. Wird innerhalb der Funktion, auf weitere Funktionen von einem Eingabewert zugegriffen?

3. Beinhaltet die Funktion IF-Verzweigungen oder andere Konstrukte die den Programmablauf auf splitten?

4. Findet in der Funktion eine Berechnung statt?

5. Werden innerhalb der Funktion Datentypen in andere Datentypen umgewandelt?

A3 User Stories zur Kursliste

1 Kursliste auf dem Dashboard

Als Lerner möchte ich auf meinem Dashboard zu jedem Themengebiet eine Liste mit Kursen in denen ich eingeschrieben bin, um bei der Navigation zu den Kursen Zeit zu sparen.

2 Anzeige der Kursliste

Als Lerner möchte ich zu jedem Kurs in der Kursliste folgende Eigenschaften sehen: Name des Kurses, Frist, Bearbeitungszeit, Schwierigkeit, Status (Alle, In Bearbeitung, Beendet, Nicht Begonnen), um einen besseren Überblick über meine Kurse zu bekommen.

3 Filtern der Kursliste

Als Lerner möchte ich die Kursliste nachdem Status (Alle, In Bearbeitung, Beendet, Nicht Begonnen) filtern können, um zum Beispiel alle noch nicht begonnenen Kurse auf einem Blick zu haben.

4 Einstellungen der Kursliste

Als Lerner möchte ich die Länge der Kursliste (Alle, 3, 5 oder 10) und den vorausgewählten Status-Filter einstellen können, um bei dem nächsten Aufruf des Dashboards die mir wichtigen Informationen direkt zu sehen.

5 Kursauswerten in der Kursliste

Als Lernbeauftragter möchte ich über ein Kursmenü an jedem Kurs in der Liste zu einer Auswertung des Kurses navigieren können, um bei den Kontrollen der Kurse Zeit zu sparen.

A4 Screenshot des Benutzerdashboards

A5 Sourcecode

Sämtliche in der Arbeit entstandenen Tests, sind auf dem beiliegenden USB-Stick in dem Ordner „\react-ui\src__tests__“ zu finden. Des Weiteren befindet sich auf dem USB-Stick eine Readme-Datei in der beschrieben ist, wie die Tests auszuführen sind.

ERKLÄRUNG

„Ich versichere hiermit, dass ich die Arbeit selbstständig verfasst, keine anderen Quellen und Hilfsmittel als die angegebenen benutzt und die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe. Das Gleiche gilt auch für eingefügte Zeichnungen, Kartenskizzen und Darstellungen."

Bremen, den 25.02.2019

Ort, Datum **Unterschrift**