TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Lehrstuhl für Rechnernetze

Master-Arbeit

ENTWICKLUNG AUTOMATISIERTER TESTS ZUR ÜBERWACHUNG DER INTEGRATION UND PERFORMANZ VON MOBILEN APPLIKATIONEN IM RAHMEN VON AMCS

Patrick Buchholz
Geboren am: 19. Februar 1991 in Prenzlau
Matrikelnummer: 3755074

zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

Betreuer
Dr.-Ing. Iris Braun
Dr.-Ing. Tenshi Hara
Betreuender Hochschullehrer
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

Eingereicht am: 30. November 2017

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur Professur Rechnernetze

AUFGABENSTELLUNG FÜR DIE MASTER-ARBEIT

THEMA: Entwicklung automatisierter Tests zur Überwachung der Integration und Performanz von mobilen Applikationen im Rahmen von AMCS

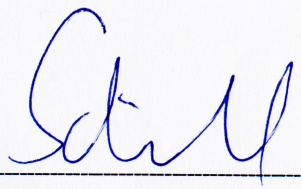
Name:	Buchholz, Patrick	Studiengang:	Master Informatik (PO 2010)
Matrikel-Nummer:	3755074	Projekt/Fokus:	AMCS
verantwortlicher Hochschullehrer:	Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill		
involvierte Mitarbeit	Dr.-Ing. Iris Braun, Dr.-Ing. Tenshi Hara		
Beginn am:	19. Mai 2017	einreichen bis:	27. Oktober 2017

ZIELSTELLUNG

Am Lehrstuhl Rechnernetze wurden in den vergangenen Jahren mehrere prototypische Untersuchungen zum Einsatz technischer Werkzeugen im Lehrbetrieb durchgeführt. Unter anderem wurden Werkzeuge in Vorlesungen und Übungen getestet; es wurden für den jeweiligen Einsatz valide Ergebnisse gewonnen. Am Ende der bisherigen Entwicklungen steht eine komplexe Anwendungsarchitektur und -plattform Auditorium Mobile Classroom Service (AMCS) mit zugehörigen mobilen Applikationen (Apps).

Da es sich um eine über mehrere Iterationen gewachsene Plattform handelt, gibt es hinreichenden Bedarf für Optimierungen. Beispielsweise gibt es derzeit nicht-deterministische Ladeverzögerungen und -fehler, die unbedingt untersucht werden müssen. Ein weiteres Problem stellt die Erweiterbarkeit dar. Mangels integrierter, automatischer Tests ist die parallele Weiterentwicklung der Apps schwierig.

Im Rahmen dieser Master-Arbeit sollen verschiedene Testszenarien entwickelt und umgesetzt werden. Schwerpunkte dabei sind Integrationstests, speziell Tests zur Anforderungsabdeckung und Kommunikationsvollständigkeit. Es sollen automatisierte Testszenarien für das Zusammenspiel der Apps mit dem Nutzer und dem Backend anhand vordefinierter Use-Cases erstellt werden. Für die jeweiligen Anwendungsfälle sind verschiedene Tools anhand zu definierender Kriterien miteinander zu vergleichen. Darauf basierend soll eine Auswahl für die Integration und Umsetzung getroffen werden.



Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
(verantwortlicher Hochschullehrer)

SCHWERPUNKTE

- Analyse und Auswahl bestehender Ansätze und Lösungen,
- Definieren von Anforderungen,
- Definieren von Bewertungskriterien für die Anforderungserfüllung,
- Konzept zur automatisierten Durchführung von Tests in AMCS Apps,
- prototypische Umsetzung des Konzepts,
- Evaluation und Auswertung der Ergebnisse.

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Prüfungsamt

Antrag auf Verlängerung der Bearbeitungszeit der Masterarbeit (ab PO 2010)

Studiengang:	Master Informatik (2010)
Name:	Buchholz
Vorname:	Patrick
Matrikelnummer:	3755074
Studienjahrgang:	2015
Betreuender HSL:	Prof. Alexander Schill
Beginn:	19. Mai 2017
Abgabe:	27. Oktober 2017
Dauer der Verlängerung ***(max. 13 Wochen)***:	5 Wochen
Neuer Abgabetermin:	1. Dezember 2017

Begründung der Verlängerung:

Auf Grund unverschuldeter technischer Probleme (extern bestimmt) war ein Evaluation der Ergebnisse bisher nicht im gewünschten Umfang möglich. Um eine belastbare Aussage über die Qualität der Ergebnisse treffen zu können, beantrage ich eine Verlängerung um 5 Wochen. Noch vor Beginn der Verlängerung werden die technischen Probleme durch den externen Dienstanbieter gelöst werden, wodurch ich dann 5 Wochen für die solide Evaluierung haben werde.
Das Vorgehen wurde mit dem Betreuer abgestimmt.

12.10.2017 | i.A.

Datum, Unterschrift Studierende/r | Zustimmung betreuender HSL

TECHNISCHE UNIVERSITÄT DRESDEN
Fakultät Informatik
Institut für Systemarchitektur
Professur für Rechnernetze
01062 Dresden

Dieser Antrag ist mit einer Kopie der Anmeldung zur Masterarbeit fristgerecht im Prüfungsamt vorzulegen.

12. Okt. 2017

Technische Universität Dresden
Fakultät Informatik
Prüfungsamt
01062 DRESDEN

Datum | Unterschrift Prüfungsamt

Entscheidung des Prüfungsausschusses:

Dem Antrag auf Verlängerung der Masterarbeit wird stattgegeben / ~~nicht stattgegeben~~.

Prof. Dr. Christel Baier
Vorsitzende des Prüfungsausschusses
Studiengang Informatik
Technische Universität Dresden
Fakultät Informatik

13.10.2017

Datum | Unterschrift Prüfungsausschuss

1

KURZZUSAMMENFASSUNG

Die Integration von mobilen Geräten und Anwendungen in Lehrveranstaltungen gewinnt immer mehr an Bedeutung. An der TU Dresden wurde ein Audience Response System namens Auditorium Mobile Classroom Service (AMCS) entwickelt. Ein wichtiger Bestandteil des Projektes ist die Entwicklung von mobilen Anwendungen. In dieser Arbeit wird die Steigerung der Qualität dieser durch eine automatisierte Testausführung untersucht. Die AMCS Apps sind während des Projektes über mehrere Iterationen herangewachsen. Der Bedarf an Tests und Testszenarien ist durch die beständige Weiterentwicklung gestiegen. Das Ziel der Arbeit war die Entwicklung eines Konzeptes zur automatisierten Ausführung von Tests zur Überwachung der Funktionalität, Integration und Performanz. Es wurden eine Reihe an unterschiedlichen Testframeworks, Tools sowie unterstützenden Architekturen für die Erfüllung der automatisierten Testausführung untersucht. Das Konzept nutzt die Möglichkeiten der iterativen Testerstellung, der Versionskontrolle und den Einsatz einer kontinuierlichen Integrationsumgebung. Die Implementierung beschränkt sich auf den Einsatz der Android Umgebung und setzt das Konzept an konkreten Beispielen um. Während der Erstellung dieser Arbeit wurden verschiedene Problematiken beim Umgang der testgetriebenen Entwicklung und beim Einsatz von Jenkins aufgedeckt. Die Ergebnisse wurden anhand einer Befragung evaluiert sowie Performanzbetrachtungen und Aufwandsuntersuchungen der Testabdeckung durchgeführt.

ABSTRACT

Nowadays there are usefull means to integrate mobile devices and applications into teaching. At the TU Dresden an Audience Response System named Auditorium Mobile Classroom Service was developed. Part of the project involves the development of mobile applications. In this thesis, automated test execution is used to imporve the quality of the apps. The AMCS apps have grown over several iterations during the project, increasing the need for tests and test scenarios. The aim of this work was to develop a concept for an automated execution of tests to monitor functionality, integration and performance. A number of different test frameworks, tools and supporting architectures have been studied. The concept uses the possibilities of iterative test creation, version control and continuous integration environments to achieve the objective. The implementation is limited to the utilization of the Android environment and implements the concept with concrete examples. During the creation of this work, various problems were identified for example the use of test-driven development or Jenkins. The results were evaluated on the basis of a survey, as well as performance observations and test coverage costs.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

QUELLTEXTVERZEICHNIS

1. EINLEITUNG

In diesem Kapitel wird ein kurzer Einstieg in die Thematik, sowie die Zielstellung vorgestellt. Verwendete Definitionen in dieser Masterarbeit werden ebenfalls erläutert.

1.1. MOTIVATION

Heutzutage spielen Smartphones eine wichtige Rolle im alltäglichen Leben. Im Play Store befinden sich bereits über 2 Millionen Android Applikationen und in Apples App Store sind es rund 1,5 Millionen Applikationen. [1] Im August 2015 gab es bereits mehr als 24.000 [2] verschiedene und im Mai 2017 gab es zwei Milliarden [3] monatlich aktive Android Geräte. Aufgrund dieser großen Fragmentierung an Geräten sind Tests für Funktionalität, Stabilität und Performanz wesentlich.
Bei einer solchen Masse an verschiedenen Geräten ist es notwendig, dass die Applikationen auf ihre Korrektheit getestet werden. Durch neue Bedien- und Entwicklungskonzepte sowie limitierter Ressourcen ist mobiles Testen eine aufwändige Arbeit.
Laut dem „World Quality Report 2014-15“ [4] nutzen ungefähr 87% der untersuchten Organisationen Tests in mobilen Anwendungen. Es wird beschrieben, dass die größte Herausforderung die fehlende Unterstützung von guten Testprozessen und Methoden ist. Des Weiteren fehlt die Zeit zum Testen als auch professionelle mobile Testumgebungen. Daher ist die Automatisierung der Testausführungen notwendig.
In einer aktuelleren Version des „World Quality Reports 2017-18“ wird ausgesagt, dass nur 16 % der befragten Organisationen und Unternehmen Testautomatisierung einsetzen. [5]
Zwei wesentliche Methoden beim Testen sind White-Box Tests und Black-Box Tests. Bei einem White-Box Test handelt es sich um einen Test mit Wissen über die innere Funktionsweise, d.h. der Zugriff auf den Quellcode muss gegeben sein. Es werden Teilkomponenten einer Applikation oder die interne Funktionsweise getestet. Ein wesentlicher Nachteil dieser Tests ist, dass die Spezifikation nicht überprüft wird. [6] Black-Box Tests sind Tests, die ohne Wissen über die innere Funktionalität einer Applikation geschrieben werden. Es werden nur die Anforderungen und nicht die Implementierung der Applikation genutzt, um einen Black-Box Test zu erstellen. Im Gegensatz zu den White-Box Tests wird hier die Spezifikation überprüft. [7]
Die bisherigen Applikationen von AMCS sind über mehrere Versionen gewachsen und aus der Sicht der Softwarearchitektur mangelhaft oder schlecht aufgestellt. Zum einen ist die Erweiterbarkeit der Applikationen schwer umsetzbar und zum anderen behindern fehlende Tests eine Weiterentwicklung.

1.2. ZIELSTELLUNG

Am Lehrstuhl für Rechnernetze der TU Dresden wurde ein System namens Auditorium Mobile Classroom Service (AMCS) entwickelt. Mit Hilfe des Tools sollen Studenten dazu motiviert werden, sich aktiv an Lehrveranstaltungen zu beteiligen.
Im Zuge der Entwicklung des Systems sind zwei Applikationen für die mobilen Betriebssysteme Android und iOS entstanden. Um eine Weiterentwicklung zu gewährleisten, werden in dieser Masterarbeit verschiedene Testszenarien entwickelt und umgesetzt.
Der Hauptanteil liegt dabei in Integrationstests und Tests, welche die Anforderungen und Kommunikation zwischen Endgerät und Backend abdecken. Ein weiterer Punkt ist die Automatisierung von Tests, um eine fortlaufende Sicherung der Qualität zu gewährleisten. Anhand der Anwendungsfälle sollen verschiedene Tools mit Hilfe von zu definierender Kriterien miteinander verglichen werden.

1.3. DEFINITIONEN

Apps	–	Apps sind die gekürzte Form von Applikationen.
Unit	–	Der Begriff „Unit“ wird im Umfang dieser Arbeit als kleine, isolierte Elemente einer Software betrachtet.
Activity	–	Eine Activity wird in Android für die Darstellung eines Fensters mit der Nutzeroberfläche und möglicher Interaktion verwendet.
Fragment	–	Repräsentiert unter Android einen Teil der Nutzeroberfläche in einer Activity.

2. GRUNDLAGEN UND STAND DER TECHNIK

Dieses Kapitel befasst sich mit den Grundlagen, die für das Testen von mobilen Applikationen notwendig sind. Dabei werden verschiedene Testarten erläutert und insbesondere auf die Besonderheiten von mobilen Geräten eingegangen.
Anschließend werden verschiedene Testframeworks vorgestellt, welche die Testerstellung vereinfachen sollen. Des Weiteren werden grundlegenden Architekturen für den Einsatz in mobilen Anwendungen untersucht. Werkzeuge für die Erstellung einer kontinuierlichen Integrationsumgebung wird zum Schluss beschrieben.

2.1. KONZEPTE

2.1.1. GRUNDLAGEN VON MOBILEN APPLIKATIONEN

Zunächst werden verschiedene Arten von mobilen Applikationen genauer untersucht. Diese können in drei verschiedene Kategorien eingeteilt werden. Dabei handelt es sich um native, hybride oder Web Applikationen. Das Buch „Hands-On Mobile App Testing" beschreibt die verschiedenen Arten im zweiten Kapitel, diese werden nachfolgend zusammengefasst. [8]

NATIVE APPLIKATIONEN

Unter nativen Applikationen versteht man Apps, die spezifisch für eine Plattform, wie z.B. Android oder iOS entwickelt werden. Die Vorteile liegen in dem kompletten Zugriff auf plattformspezifische Hard- und Softwarefeatures. Durch gezielte Optimierung für eine Plattform haben native Applikationen die beste Performanz von allen Applikationsarten. Wenn die Design- und Entwicklungsrichtlinien durch die Entwickler umgesetzt werden, resultiert dies in einer guten Nutzbarkeit und „Look and Feel". Beispiele, welche die Richtlinien unterstützen, sind die vorinstallierten Standard Applikationen, wie Kalender oder Kontakte. Durch den gegebenen Zugriff auf die Hardware können alle Touchgesten, Sensoren und die lokale Speicherung von Daten unterstützt werden. Ein weiterer Vorteil ist die einfache Verteilung entwickelter Applikationen über die vorinstallierten App Stores.
Nichtsdestotrotz gibt es auch einige Nachteile, wie z.B. den größeren Entwicklungsaufwand. Wenn mehrere Plattformen unterstützt werden sollen, müssen verschiedene Codebasen vorhanden sein, weil jede Plattform andere Tools und Programmiersprachen unterstützt. Auch bei der Bereitstellung von Updates kann es zu Verzögerungen kommen, da das Update zunächst von den App Store Betreibern überprüft werden muss. Des Weiteren werden rund

30% vom Gewinn von den Betreibern einbehalten.

HYBRIDE APPLIKATIONEN

Bei hybriden Applikationen handelt es sich um Apps, die sich eine Codebasis für verschiedene Plattformen teilen. Dies ist auch der entscheidende Vorteil, da der Entwicklungsaufwand deutlich geringer ist als bei nativen Apps. Der Zugriff auf Hardwarefeatures ist zwar eingeschränkter, aber durch die Verwendung von Frameworks kann trotzdem auf einen großen Teil der Hardware zugegriffen werden. Die Verteilung der App und nachfolgende Updates sind einfach umsetzbar, wenn die benötigten Daten über das Internet und nicht über die App Stores verteilt werden. Es gibt aber auch die Möglichkeit die Apps über die App Stores zu verteilen und upzudaten.
Nachteile bei hybriden Applikationen sind die fehlende Performanz, da die Inhalte und die Komponenten von einem Server heruntergeladen werden müssen. Außerdem werden weniger Features einer Plattform unterstützt. Durch die Abstraktion der Frameworks können nicht alle Design- oder Entwicklungsrichtlinien umgesetzt werden, was die Nutzbarkeit zusätzlich beeinträchtigen kann.

WEB APPLIKATIONEN

Die Einführung von Webstandards, wie HTML5, JavaScript oder Designansätze, wie Mobile First führte in den letzten Jahren dazu, dass immer mehr Webseiten auf mobilen Geräten bedienbar sind. Ein weiterer Schritt sind mobile Web Applikationen. Diese imitieren native Apps und benötigen dafür nur den vorinstallierten Web Browser. Dies hat den Vorteil, dass die Applikationen schnell und billig entwickelt werden können. Sie sind unabhängig von dem Betriebssystem, da nur der installierte Web Browser genutzt wird. Wie bei normalen Web Anwendungen sind Updates schnell und einfach umsetzbar, indem direkt der Quellcode verändert wird. Allerdings ist der Zugriff auf Hardwarefunktionen sehr eingeschränkt und die Performanz ist, wie bei den hybriden Apps, nicht optimal. Sowohl der Inhalt als auch der Code einer App (hier also eine Webseite) muss heruntergeladen werden. Die Performanz und das „Look and Feel" sind begrenzter als bei nativen Applikationen. Ein weiteres Problem sind die verschiedenen Web Standards, welche von den einzelnen Browser Herstellern unterschiedlich umgesetzt oder implementiert werden.

2.1.2. GRUNDLAGEN VON SOFTWARETESTS

Das Testen von Software ist ein wichtiger Aspekt in der Softwareentwicklung. Das Handbuch „Software Engineering Body of Knowledge" definiert diesen wie folgt: „Software testing consists of the dynamic verification that a program provides expected behaviours on a finite set of test cases, suitably selected from the usually infinite execution doman." [9] Da Programme schnell komplex werden und zu vielen Zeilen Code heran wachsen können, ist es wichtig zu planen, welche Teile einer Software wie getestet werden soll. Tests werden daher in verschiedene Kategorien unterteilt. Das Handbuch „Software Engineering Body of Knowledge" gliedert diese in die drei folgenden Arten: Unit Tests, Integrationstests und System Tests.

ARTEN VON TESTS

Unit Tests

Bei Unit Tests wird ein separates und isoliertes Element einer Software getestet. Es werden funktionale Einzelteile auf ihre korrekte Funktionalität überprüft. Daraus folgt, dass nicht die Korrektheit des ganzen Systems gewährleistet werden kann. Nicht-funktionale Kriterien, wie Performanz oder Nutzbarkeit, können deswegen nicht abgedeckt werden. Unit Tests oder

auch Modultests haben meistens einen einheitlichen Aufbau. Zuerst wird ein initialer Zustand hergestellt, darauf wird eine zu testende Aktion ausgeführt und anschließend das Ergebnis mit einem definierten Erwartungsergebnis verglichen. [10] Die Vorteile bei Modultests liegen in der schnellen und zuverlässigen Ausführung. Nur ein kleiner Teil des Codes wird ausgeführt und überprüft. Die Erstellung ist einfacher, weil weniger Abhängigkeiten im Vergleich zu Integrationstests vorhanden sind. Außerdem sind die Fehlermeldungen, wie z.B. „Funktion sollte 1 liefern, aber lieferte 2", verständlicher und schneller auffassbar. [11]

Integrationstests

Um die Interaktion von den verschiedenen Komponenten einer Software zu überprüfen, werden Integrationstests genutzt. Es wird ein Testszenario entwickelt, dass die einzelne Komponente sowie den Datenaustausch zwischen Komponenten überprüft. Daraus lässt sich ableiten, ob die gemeinsame Schnittstelle spezifikationsgemäß funktioniert. [12] Im Vergleich zu Unit Tests sind Integrationstests langsamer, da mehr Code ausgeführt und überprüft werden muss. Des Weiteren erfordert die Einrichtung mehr Aufwand, da mehr Abhängigkeiten beachtet werden müssen. [11]

System Tests

Die höchste Stufe dieser drei Testarten sind System Tests. Dabei wird das Verhalten des kompletten Systems verifiziert. Unit und Integrationstests dienen dazu, Fehler der Software aufzudecken. Bei System Tests werden die Anforderungen und Qualität kontrolliert. Nichtfunktionalen Aspekte, wie Performanz oder Zuverlässigkeit, z.B. Vermeidung von Abstürzen, werden ebenfalls geprüft. [9]

ZIELE VON TESTS

Tests können auf die Überprüfung von verschiedenen Eigenschaften eingeordnet werden. Zum einen gibt es Testfälle die funktionale Spezifikationen verifizieren. Zum anderen können auch nicht-funktionale Eigenschaften, wie Performanz, Nutzbarkeit, Akzeptanz oder Zuverlässigkeit getestet werden. Im folgenden Abschnitt werden einige Tests zur Überprüfung von nicht-funktionalen Eigenschaften des Handbuchs [9] aufgeführt und kurz beschrieben.

- Akzeptanz- oder Qualitätstests bestimmen, ob der Nutzer die Software akzeptiert und ob das System nach vorher festgelegten Merkmalen funktioniert.
- Installationstests dienen zur Überprüfung der Funktionalität einer Software nach der Installation auf einer bestimmten Zielumgebung.
- Alpha-/Betatests - Bei Alpha-/Betatests wird die Software von einer kleinen, selektierten Gruppe vor dem Release auf mögliche Fehler untersucht.
- Regressionstests werden bei der inkrementellen Softwareentwicklung genutzt. Hierbei werden die Änderungen gegen bereits bestandene Tests geprüft, um zu zeigen, dass die Modifizierungen das bisherige Verhalten nicht verändert haben.
- Performanztests prüfen die Verarbeitungszeit von bestimmten Anwendungsfällen. Bei mobilen Applikationen kann man z.B. die Startzeit einer Applikation messen, die Verzögerungen bei Nutzerinteraktionen oder die Ladezeit des Inhalts.

2.2. TESTFRAMEWORKS

Für die Entwicklung von verschiedenen Tests gibt es unterschiedliche Frameworks, welche den Entwickler bei der Erstellung und Umsetzung unterstützen. Im Folgenden werden die Eigenschaften erläutert und anschließend tabellarisch zusammengefasst.

XUNIT

xUnit sind, wie der Name schon suggeriert, Unit Testframeworks für verschiedene Programmiersprachen. JUnit wird für das Testen von Java Anwendungen genutzt und ist gut geeignet für das automatisierte Testen einzelner Elemente einer Software. Es gibt nur zwei Arten von Ergebnissen. Der Test wird entweder bestanden oder er misslingt. Beim Misslingen kann die Ursache ein Fehler (*Error*) oder ein falsches Ergebnis (*Failure*) sein. Der Unterschied liegt darin, dass falsche Ergebnisse erwartet werden, aber Fehler unerwartet auftreten. Unit Tests sind elementarer Teil der agilen Softwareentwicklung, wie z.B. der testgetriebenen Entwicklung (englisch *test-driven development*). Dabei werden zuerst die Tests und anschließend der zu testende Code geschrieben. Dies hat den Vorteil, dass bei späterer Änderung des Codes auch die schon vorhandenen Tests bestanden werden müssen. Dies führt dazu, Code mit weniger Fehlern zu entwickeln, da nichts implementiert wird, das nicht auch getestet wird. [13]

ESPRESSO

Espresso ist ein von Google entwickeltes Testframework für die Abdeckung von Benutzeroberflächentests. Es wurde speziell für Android entwickelt und bietet den großen Vorteil, dass extra auf die Synchronisierungsprobleme zwischen Test- und UI-Ausführung geachtet wird. Bei anderen Testframeworks, wie z.B. Robotium, müssen extra *wait()*-Methoden zu den UI-Tests hinzugefügt werden, um auf die eventuell noch nicht vorhandenen UI Elemente zu warten. Das kann zu unzuverlässigen Ergebnissen führen, wenn die Aktualisierung der Oberfläche etwas länger dauert als erwartet. [14] Espresso umgeht dies durch die Überwachung des internen Main-UI-Threads. Das Framework deckt eine breite Anzahl von API-Leveln ab, beginnend ab API Level 10 (Android 2.3.3) bis zur neusten Android Version. Dadurch werden laut Android Dashboard 100% der Android Versionen abgedeckt, welche zur Zeit in Nutzung sind.[1] [16] Es gibt außerdem einen Record-Playback Ansatz, um die Erstellung von UI Tests zu vereinfachen. Hierbei werden die Nutzerinteraktionen von einem Gerät aufgezeichnet und es können Überprüfungen, wie z.B. TextView sollte den Text „XY" haben, hinzugefügt werden. Anschließend wird die Aufzeichnung in Code, genauer in Form eines UI Tests, umgewandelt. Der Test kann später wieder ausgeführt und die Funktionalität überprüft werden. [17]

APPIUM

Appium ist ein Open Source Testautomatisierungswerkzeug für native, hybride und Web Applikationen. Der Vorteil von Appium ist, dass man nur ein Test Skript schreiben muss und damit sowohl Android als auch iOS Apps testen kann. Es basiert auf Selenium, was ein beliebtes Testframework für Web Anwendungen ist. Appium ist in Form einer Client/Server Architektur aufgebaut. Dabei wird ein Server (*WebDriver*) auf einem Testgerät oder Simulator ausgeführt. Dieser nimmt Befehle vom Computer entgegen und führt diese auf dem Gerät aus. Die Resultate werden via HTTP zurückgeliefert, um eine Auswertung der Tests zu ermöglichen. Durch die Architektur und Entwickler API können die Programmiersprachen Java, Ruby oder Python genutzt werden, um die Tests zu erstellen. Ein Nachteil ist, wie bei

[1] Stand 24.11.2017 – Verteilung der Android Versionen im Google Play Store [15]

Robotium, dass *wait()*-Methoden benutzt werden müssen, um z.B. UI Elemente mit der Testausführung zu synchronisieren. [18]

XCTEST FRAMEWORK

Mit dem XCTest Framework werden Tests für macOS und iOS erstellt. Es wurde erstmals in XCode 5.0 integriert, welches auf dem WWDC 2013 vorgestellt wurde. [19] Seit XCode 6 kamen einige Performanz Werkzeuge hinzu und seit XCode 7 ist es möglich UI Tests zu erstellen. Die Tests werden sowohl mit Objective-C als auch mit Swift erstellt. Des Weiteren werden die Tests auch per Befehlszeilen Skripte auf Bots des XCode Servers ausgeführt. Dadurch ist es möglich die Tests auch von einem anderen Rechner oder Server zu starten. Ein Vorteil des Frameworks ist, dass es direkt in XCode integriert wurde und so keine weiteren Abhängigkeiten bereitgestellt werden müssen. Es gibt, wie bei Espresso, auch einen Record-Playback Ansatz, um UI Tests zu erstellen. [20] Einfache Performanztests können mit Hilfe von *Measure* Blöcken umgesetzt werden, wobei diese zehn mal nacheinander ausgeführt, um eine zeitliche Basislinie zu ermitteln. Der Wert wird mit den anschließenden Testausführungen verglichen und als Fehler symbolisiert, wenn die Ausführungsdauer über einen definierten Toleranzwert hinaus geht. [21]

MOCKITO

Mockito ist ein Mocking Framework für Java Anwendungen, welches bei der Isolation von Komponenten bei einem Testlauf hilft. Es ersetzt klassische Objekte einer Klasse mit Platzhalter Objekten. Dadurch können eventuelle Abhängigkeiten umgangen werden. Das Framework wird hauptsächlich bei Unit Tests eingesetzt. Es können Klassen oder Interfaces durch das Framework in Mock-Objekte bzw. -Interfaces ersetzt werden. [22] Mockito ist ein beliebtes Framework und zählt laut einer Analyse zu den zehn am meisten genutzten Java Bibliotheken auf GitHub. [23]

ÜBERSICHT DER TESTFRAMEWORKS

Die vorgestellten fünf Frameworks und weitere Beispiele, wie Robolectric und Calabash, sind unter 2.1 tabellarisch zusammengefasst und die Vor- bzw. Nachteile gegenüber gestellt.

	Eigenschaften	Vorteile	Nachteile
Robolectric [24]	• Imitiert das Android SDK nach	• Tests können direkt in der Java VM ausgeführt werden • Sehr schnelle Tests • Mock Frameworks werden nicht benötigt	• Nicht alle Android API Level werden unterstützt • Nicht alle SDK Funktionen sind abgedeckt
Robotium [25]	• Gute API um UI Tests für Android zu erstellen • Unterstützung für native und hybride Apps • Record-Playback Funktion für UI Tests • Android Versionen ab API Level 8 werden unterstützt	• Breite Android API Unterstützung	• Benötigt wait-Methoden zur UI und Test Synchronisierung
Espresso	• Framework von Google für Android UI Tests • Entwickelt worden, um die UI Synchronisierungsprobleme abzudecken • Record-Playback Funktion für UI Tests • Android Versionen ab API Level 10 werden unterstützt	• Braucht keine wait-Methoden	• Nur für UI Tests
Appium	• Basiert auf Selenium • Nutzt Web Server im Hintergrund, um Befehle auf den Geräten auszuführen	• Unterstützung für native, hybride und Web Apps • Nur ein Skript, um auf iOS und Android zu testen	• Benötigt wait-Methoden zur UI und Test Synchronisierung
Calabash [26]	• Akzeptanz Tests für die UI • Nutzt eine natürliche Sprache (Cucumber) um Tests zu schreiben	• Unterstützung für native und hybride Apps • Natürliche Sprache für Nichtprogrammierer schnell erlernbar	• Benötigt wait-Methoden zur UI und Test Synchronisierung
JUnit	• Unit Testframework für Java	• Schnelle Ausführung	• Nur Funktionalität wird abgedeckt
XCTest	• Unit Testframework für iOS	• Bereits in XCode integriert • Schnelle Ausführung • Einfache Performanztests möglich	

Tabelle 2.1.: Übersicht der Testframeworks

2.3. ARCHITEKTUREN IN MOBILEN APPLIKATIONEN

Für das Testen spielt die zugrunde liegende Architektur eine wichtige Rolle. Ohne Trennung der einzelnen Komponenten und Abhängigkeiten ist zum einen die Weiterentwicklung und das Hinzufügen von neuen Funktionen schwierig und zum anderen ist die Erstellung von Tests aufwendiger. Deswegen werden im Folgenden grundlegende Architekturen untersucht, welche die Testerstellung vereinfachen.

MODEL VIEW CONTROLLER (MVC)

Beim MVC handelt es sich um eine klassische Architektur, die von Apple für die Entwicklung von iOS Applikationen empfohlen wird. In Android wird es für die Entwicklung von Applikationen ebenfalls eingesetzt. Die Controller sind verhaltenssteuernd und können sich mehrere Views teilen. View und Model haben die Möglichkeit miteinander zu kommunizieren. Die Vorteile liegen darin, dass keine Anwendungslogik in den Views vorhanden ist und dadurch die Umsetzung von Tests einfacher ist. Probleme mit dieser Architektur sind die schlechte Skalierbarkeit und die nicht vollständige Trennung der Komponenten. Das Model ist immer noch abhängig vom Controller. Ein weiteres Problem ist, dass die Controller sehr schnell groß und unübersichtlich werden. [27]

MODEL VIEW PRESENTER (MVP)

Bei dem Model View Presenter wird auf eine striktere Trennung der einzelnen Komponenten geachtet. View und Model sind getrennt voneinander und haben nicht die Möglichkeit miteinander zu kommunizieren. Der Presenter spielt hier die Vermittlerrolle zwischen Model und View. Im Allgemeinen gibt es eine 1:1 Abbildung zwischen View und Presenter, es gibt aber auch die Möglichkeit mehrere Presenter für einen komplexen View zu nutzen. Die einzelnen Komponenten sind durch die drei Schichten gut voneinander getrennt. Der passive View ist für die Darstellung der Logik verantwortlich, der Presenter kümmert sich um die User Events und das Model enthält die Anwendungslogik. Von Vorteil ist, dass komplexe Aufgaben in kleinere unterteilt werden können. Durch weniger Code pro Klasse ist es übersichtlicher, Fehler können schneller entdeckt werden und das Testen ist einfacher umsetzbar. Bei dem Model View Presenter ist es von Nachteil, dass man mehr Verdrahtungscode schreiben muss, um die einzelnen Komponenten miteinander zu verbinden. Außerdem kann das Model nicht wiederverwendet werden, da es an einen speziellen Use Case gebunden ist.
So wie die Architektur, die drei Schichten vorgibt, kann man ebenfalls die Testfälle aufbauen. Wenn man Tests für den View schreibt, testet man die Darstellungslogik und die Interaktion mit dem Presenter. Dafür muss nur der Presenter durch ein Mock-Objekt ersetzt werden.
Mit dem Presenter wird überprüft, ob die Events des Views die entsprechenden Methoden des Models aufrufen. Hier müssen sowohl der View als auch das Model mit einem Mock-Objekt ersetzt werden.
Durch das Model kann die Anwendungslogik getestet werden. Die Abhängigkeiten sind die Quelle der Daten und der Presenter. Um die Anwendungslogik zu testen, müssen diese daher mit Mock-Objekten ersetzt werden. [27]

MODEL VIEW VIEWMODEL (MVVM)

Dieses Pattern bzw. Architektur wurde erstmals von Microsoft für die eigene Windows Presentation Foundation (kurz WPF) genutzt. Unter Android stellen Activities und Fragments die Ansicht einer Applikation dar. Der UI Code ist zumindest bei Android nicht mehr an Activities und Fragements gebunden und daher unabhängig von deren Lebenszyklen. Der View hat kein Wissen über das Model und beide Komponenten können daher nicht miteinander

kommunizieren. MVVM ist stark vom DataBinding abhängig, dabei werden Datenquelle und Verbraucher miteinander verbunden und synchronisiert. Die Vorteile sind, dass weniger Codezeilen geschrieben werden müssen und Tests einfach umgesetzt werden können. Ein weiterer großer Vorteil ist, dass das ViewModel unabhängig von der Plattform ist. So muss man z.B. nur ein ViewModel entwickeln und kann es sowohl für iOS als auch Android nutzen. Problematisch sind allerdings das bereits erwähnte DataBinding, was nicht immer angewendet werden kann und so genannten Spaghetti Code fördert, da z.B. Anwendungslogik in Android in XML-Dateien ausgelagert wird. Die Anwendungslogik sollte im Allgemeinen im Code geschrieben werden, da es sonst schnell unübersichtlich und das Finden von Fehlern komplizierter wird. [27]

CLEAN

Clean ist mehr eine Architektur von Architekturen, um eine noch bessere Trennung und Wiederverwendbarkeit der einzelnen Komponenten zu gewährleisten. Dazu werden zwei weitere Schichten zwischen den drei Hauptschichten Model, View und Controller oder Presenter eingeführt. Die Übersicht 2.1 veranschaulicht die Clean Architektur. Es gibt eine Präsentationsschicht die intern MVP nutzt, eine Domainschicht, welche aus regulären Objekten besteht und eine Datenschicht, die das Repository Pattern nutzt. Die Schichten zwischen den drei Hauptkomponenten dienen zur Kommunikation zwischen den einzelnen Hauptschichten. Die strikte Trennung der Schichten ist von Vorteil für die Umsetzung der Tests. Die komplexere Umsetzung durch die Vielzahl von Schichten ist jedoch von Nachteil. [27]

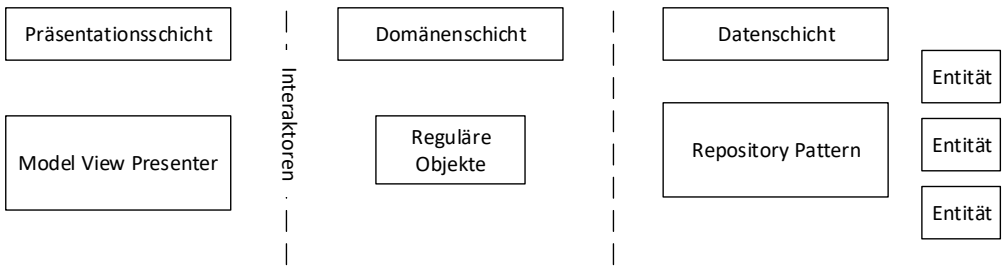


Abbildung 2.1.: Übersicht der Clean Architektur

REPOSITORY PATTERN

Repository Pattern ist ein Entwurfsmuster und bietet eine Schnittstelle zwischen der Domänen- und der Datenzugriffsschicht. Die Domäne beinhaltet die Geschäftslogik. Das Muster ist hilfreich bei unterschiedlichen Zugriffen auf Daten, es kann z.B. verschiedene Quellen für die Produktions- oder Testversion geben. Dabei werden die Datenobjekte aus der Datenzugriffsschicht abstrahiert, d.h. die Quelle, woher die Daten stammen ist nicht relevant. Sie können z.B. aus einer Datenbank, einem internen Speicher oder von einem Webservice stammen. [28]

ÜBERSICHT DER ARCHITEKTUREN

Die verschiedenen Architekturen werden in 2.2 zusammengefasst und die Vor- und Nachteile dargestellt.

	Eigenschaften	Vorteile	Nachteile
Klassisches Android	• keine feste Struktur, ansatzweise MVC	• Besser das Übel, dass man schon kennt • Schnelle Entwicklung	• Activities und Fragmente werden sehr groß • Schwer neue Funktionen und Änderungen umzusetzen • Keine klare Trennung der Komponenten • Tests nur sehr schwer umsetzbar
Model View Controller	• Die Controller steuern das Verhalten und können sich mehrere Views teilen • View kann direkt mit dem Model kommunizieren	• Keine Anwendungslogik in der UI • Einfacher Unit Tests durchzuführen	• Schlechte Skalierung • Ungenügende Trennung der Komponenten, Model ist abhängig vom Controller • Controller Klassen werden groß • Verstößt gegen einzelne Verantwortlichkeiten
Model View Presenter	• View und Model sind getrennt • Presenter dient als Vermittler zwischen View und Model • 1:1 Abbildung zwischen View und Presenter • Klare Trennung der Komponenten	• Komplexe Aufgaben werden in kleinere gespalten • Kleinere Objekte, bedeuten weniger Fehler und Debugging ist einfacher • Weniger Spaghetti Code • Sehr gut testbar	• Mehr Boilerplate Code, um alles miteinander zu verknüpfen • Model kann nicht wiederverwendet werden • Presenter und View sind an Datenobjekt gebunden (teilen das gleiche Model Objekt) • Mehr Klassen als MVVM
Model View ViewModel	• Entfernt UI Code von Activities und Fragmente • View hat kein Wissen über das Model • Nutzt DataBinding • 1:N Abbildung zwischen View und ViewModel	• Präsentationsschicht in XML • Weniger Code als MVP • Sehr gut testbar	• Benötigt unbedingt DataBinding • Skaliert nicht gut • Anwendungslogik gehört in Code und nicht in XML • Verursacht mehr Spaghetti Code bei komplexen Layouts
Clean	• Besseres Trennen und Wiederverwenden von Komponenten • Mehr eine Architektur von Architekturen • Drei Hauptschichten mit verbindenden Kommunikationsschichten	• Sehr gute Trennung der einzelnen Komponenten • Sehr gut testbar	• Erhöhter Aufwand durch die Mehrzahl an Schichten

Tabelle 2.2.: Übersicht der Architekturen

2.4. KONTINUIERLICHE INTEGRATION

Unter kontinuierliche Integration versteht man den Prozess des fortlaufenden Zusammenführens von Komponenten zu einer Anwendung. Es dient zur Steigerung der Software Qualität und arbeitet meistens mit einem Versionskontrollsystem zusammen. [29] Durch das ständige Bauen und Testen wird eine frühe Fehlererkennung und -beseitigung gefördert. Für die kontinuierliche Integration werden in der Regel Server genutzt, die Zugriff auf den Quellcode haben und die einzelnen Schritte abarbeiten. Einige Beispiele werden im Folgenden näher betrachtet.

BAMBOO

Bamboo ist ein kommerzieller Serveransatz für kontinuierliche Integration der Firma Atlassian. Es ist eine auf Java basierende Web Anwendung und ermöglicht die Konfiguration über eine Weboberfläche. Es werden viele gängige Buildwerkzeuge und dadurch viele Programmiersprachen unterstützt. Die allgemein genutzten Versionskontrollsysteme, wie SVN oder Git, werden ebenfalls unterstützt. Durch die Nutzung von Addons können z.B. XCode oder Tomcat verwendet werden. Mit anderen Produkten von Atlassian, wie z.B. Jira oder HipChat, ist eine einfache Integration möglich. Die Preisgestaltung ist von der Anzahl der Server zum Bauen, den sogenannten Remote Agents, abhängig. Es gibt weiterhin eine Software as a Service Lösung, bei der die Atlassian Cloud genutzt werden kann, um Anwendungen zu bauen, zu testen oder zu veröffentlichen. Nichtsdestotrotz sind Open Source, Non-Profit Organisationen oder Schulklassen Projekte kostenlos nutzbar, sie sind nur in der Anzahl der Remote Agents eingeschränkt. [30]

TRAVIS CI

Bei Travcis CI handelt es sich um einen Open Source Ansatz für das Erstellen und Testen von Projekten, die auf GitHub gehostet werden. Für private Projekte wird eine Gebühr erhoben, öffentliche Projekte können kostenlos genutzt werden. Für die Verwaltung und Integration wird YAML verwendet, indem verschiedene Parameter zur Konfiguration angegeben werden. Travis wird über aktuelle Änderungen von GitHub informiert. Anschließend werden die Anweisungen der Konfigurationsdatei, wie z.B. Bauen oder Testen, ausgeführt. [31]

JENKINS

Jenkins ist ein webbasiertes System und entstand ursprünglich aus einer Abzweigung (engl. „Fork“) von Hudson, welches von Sun bzw. Oracle entwickelt wird. Jenkins wurde in Java geschrieben und läuft in einer beliebigen Anzahl von Enterprise JavaBeans Containern. Wie bei Bamboo werden viele Buildwerkzeuge, Programmiersprachen und Versionskontrollsysteme unterstützt. Es gibt ein umfangreiches Plugin System, um die bereits vorhandenen Möglichkeiten zu erweitern. Über eine REST-basierte Schnittstelle ist eine Steuerung über andere Programme möglich. Es kann aber auch eine Weboberfläche genutzt werden. [32] Die Vorteile liegen in der einfachen Installation, Einrichtung und Konfiguration. Das umfangreiche Plugin System bietet einfache Erweiterungsmöglichkeiten, z.B. die Verteilung auf verschiedenen Plattformen, wie Linux, Windows oder Mac. [33]

ÜBERSICHT ÜBER KONTINUIERLICHE INTEGRATIONSWERKZEUGE

In der Tabelle 2.3 werden die Eigenschaften der vorgestellten Werkzeuge und zwei Weitere kurz zusammengefasst. Einige Vor- und Nachteile von kontinuierlichen Integrationsumgebungen werden gegenüber gestellt.

	Eigenschaften	Vorteile	Nachteile
Bamboo	• Webbasiert • Addon System • Integration in IDE und Benachrichtigungen • Open Source • Versionskontrollsysteme und Buildwerkzeuge Unterstützung	• Viele Programmiersprachen werden unterstützt • Erweiterbar durch Addons • Einfache Integration mit anderen Atlassian Produkten (Bitbucket)	• Nicht komplett kostenlos nutzbar • Keine native Unterstützung für iOS
Travis CI	• Open Source Werkzeug • Stark auf GitHub ausgerichtet	• Sehr gute Integration mit GitHub • Einfache Konfiguration per YAML	• Nur auf GitHub Projekte anwendbar • Nur für Open Source Projekte kostenlos
Jenkins	• Basierend auf Java mit Weboberfläche • Plugin System zur Erweiterung • REST-Schnittstelle zur Konfiguration • Unterstützung von VCS	• Einfache Installation • Unkomplizierte Einrichtung und Konfiguration • Sehr viele Plugins und dadurch gut erweiterbar • Problemlose Verteilung auf verschiedenen Plattformen	• Initial nicht so umfangreich wie Bamboo
Teamcity [34]	• Java basierte und von JetBrains entwickelt • Integration mit IDEs von JetBrain und anderen • REST-Schnittstelle zur Konfiguration • Unterstützung von VCS	• Cloud Unterstützung für Amazon, Azure oder vSphere • Gated Commits • Einfache Integration in IDEs (IntellJ, Visual Studio, Eclipse)	• Freie Version auf 20 Build Konfigurationen und 3 Build Agenten eingeschränkt
Robot Framework [35]	• Gehört eher zur Testautomatisierung als zur kontinuierlichen Integration • Tests können per Java oder Python geschrieben werden • Schlüsselwort basierter Ansatz mit tabellarischen Testdaten • Wird für Akzeptanztests genutzt	• Kann mit Selenium oder Appium genutzt werden	• Android und iOS Bibliotheken werden nicht mehr gepflegt [36, 37]

Tabelle 2.3.: Übersicht über kontinuierliche Integrationswerkzeuge

2.5. ZUSAMMENFASSUNG

In diesem Kapitel wurden die Grundlagen von mobilen Applikationen und Softwaretests genauer dargestellt. Tests können in verschiedene Kategorien unterteilt werden, welche unterschiedliche Bereiche abdecken. Dabei können funktionale und nicht-funktionale Eigenschaften überprüft werden. Durch die unterschiedlichen Bereiche von Tests gibt es verschiedene Testframeworks, z.B. für Unit, UI oder Performanz Tests. Diese wurden mit einigen Beispielen und mit Hilfe von Tabelle 2.1 anhand verschiedenen Testframeworks erläutert und miteinander verglichen. Ein weiterer Punkt für die Testerstellung sind die zugrunde liegenden Architekturen. Ohne eine klare Trennung der einzelnen Schichten wird die Testbarkeit einer Applikation erschwert. Einige unterstützende Architekturen wurden in Tabelle 2.2 zusammengefasst und Vor- bzw. Nachteile kurz beschrieben. Zum Schluss wurden Werkzeuge für die kontinuierliche Integration vorgestellt. Sie haben die Aufgabe auf einem Server den aktuellen

Quellcode zu bauen, zu testen und je nach Konfiguration auch zu veröffentlichen. Daher sind für die Testautomatisierung kontinuierliche Integrationswerkzeuge entscheidend. Im Unterkapitel 2.4 Kontinuierliche Integration wurden aktuelle Werkzeuge verglichen und mit Tabelle 2.3 ein Überblick gegeben.

3. ANALYSE DES AKTUELLEN STANDES

Dieses Kapitel analysiert den aktuellen Stand sowie die Auditorium Mobile Classroom Service Anwendungen. Außerdem werden Anforderungen an verschiedene Testwerkzeuge zur Automatisierung untersucht.

3.1. AUDITORIUM MOBILE CLASSROOM SERVICE („AMCS“)

In Kooperation der Professur für Rechnernetze der Fakultät Informatik und der Professur für die Psychologie des Lehrens und Lernens der Fachrichtung Psychologie entstand an der TU Dresden das Forschungsprojekt AMCS. Das System bietet die Möglichkeit zur bestimmten Interaktion zwischen Studenten und Dozenten während einer Lehrveranstaltung. [38]
Eine Form der Interaktion läuft über Fragen ab. Diese werden vom Vortragenden erstellt und können von den Studenten beantwortet werden.
Das System besteht aus mehreren Frontend Systemen für Web, Android, iOS und Präsentationsapplikationen sowie einer Backend Applikation.
Die Kommunikation zwischen Frontend und Backend erfolgt über eine REST-Schnittstelle und einer WebSocket Verbindung. Des Weiteren wird für Benachrichtigungen unter Android und iOS der „Firebase Cloud Messaging“ Dienst genutzt.
Die zwei mobilen Anwendungen sind separat entwickelte Projekte und haben daher unterschiedliche Codebasen und Strukturen. Außerdem ist die Unterstützung von Funktionen unterschiedlich. Aktuell bietet die Android Applikation mehr Features als die iOS Version. Der folgende Abschnitt befasst sich mit einigen Funktionen der AMCS Apps sowie den Unterschieden zwischen den beiden Versionen.
Weiterhin gibt es zwei verschiedene Versionen der Backends, die über unterschiedliche URLs aufrufbar sind. Dabei handelt es sich um das Test- und das Produktionssystem. Das Testsystem stellt die Vorstufe dar, indem die internen Entwickler neue Funktionen entwickeln und testen. Wenn die neuen Funktionen stabil und fehlerfrei laufen, werden die Änderungen in das Produktionssystem übertragen und stehen somit allen Nutzern zur Verfügung.
Die Abbildung 3.1 bietet einen kleinen Überblick über die Kommunikationsweise zwischen Apps und Backend.

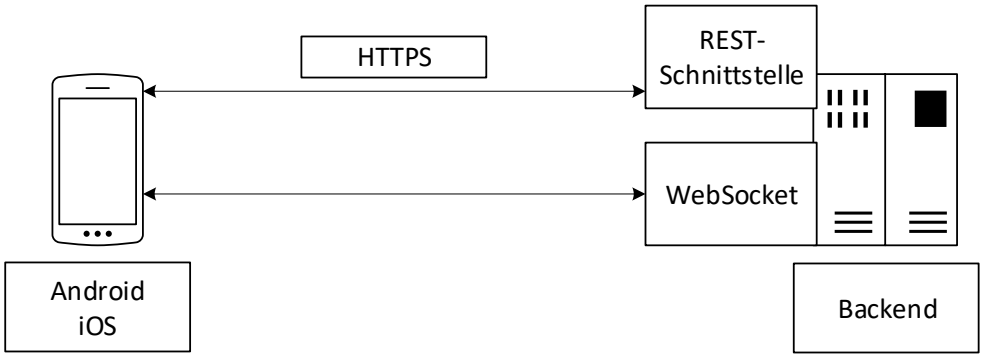


Abbildung 3.1.: Kommunikationsübersicht zwischen Apps und Backend

3.1.1. FUNKTIONSÜBERSICHT

LOGIN

Die Studenten können sich anhand eines Pseudonyms und Passworts im AMCS System anmelden bzw. registrieren. Es wird eine individuelle ID für jeden Nutzer vergeben, um mögliche Antworten eindeutig zuzuordnen. Dadurch können z.B. mehrere Antworten durch den gleichen Nutzer vermieden werden. Weiterhin wird die ID z.B. bei der Kurseintragung benötigt.

KURSE

Studenten können sich mit Hilfe einer PIN in einen Kurs einschreiben. Ist dies erfolgt, werden in der Übersichtsansicht die eingeschriebenen Kurse und deren aktive Vorlesungen angezeigt.

VORLESUNGEN

Ein Kurs kann mehrere Vorlesungen beinhalten, welche zu einem bestimmten Zeitpunkt gehalten werden. Es gibt vier verschiedene Zustände der Vorlesung: Offline, Vorbereitung, Online und Nachbereitung. Je nach Zustand können in der Veranstaltungsübersicht verschiedene Arten von Fragen beantwortet werden.

FRAGEN BEANTWORTEN

Die Hauptfunktion des Systems ist die Beantwortung von Fragen. Diese können in drei verschiedene Kategorien unterteilt werden. Dabei handelt es sich um Folien-, Vorlesungs-, und Kursfragen. Die Folienfragen sind an eine Folie der aktuellen Vorlesung gekoppelt. Der Folienwechsel wird in nahezu Echtzeit über eine WebSocket Verbindung vom Backend an die mobilen Geräte übertragen. Vorlesungsfragen sind unabhängig von den Folienfragen und werden während einer aktiven Vorlesung dargestellt. Vor- bzw Nachbereitungsfragen werden entsprechend der Vor- und Nachbereitungszeit angezeigt. Diese sind eine Untergliederung der Vorlesungsfragen. Die Kursfragen sind unabhängig von den Vorlesungen und können in der gesamten Zeit beantwortet werden.
Die einzelnen Fragen sind in verschiedene Typen unterteilt, diese werden im Folgenden kurz aufgelistet. Auf die Funktionalität der einzelnen Typen wird hier nicht näher eingegangen. Diese kann auf der Webseite https://amcs.website/manual#questionTypes [1] genauer betrachtet

[1] Letzter erfolgreicher Aufruf: 27.11.2017

werden.

- Fragen mit Einfachauswahl
- Fragen mit Mehrfachauswahl
- Lernfragen - bestehen aus Einfachauswahl oder Mehrfachauswahl, können aber zwei mal beantwortet werden
- Skalenfragen
- Freitextfragen
- Gruppierte Skalenfragen

BENACHRICHTIGUNGEN

Für Nachrichten wird in beiden App Versionen Googles „Firebase Cloud Messaging" Dienst genutzt. Wenn die Anwendungen im Hintergrund laufen oder die Geräte im Standby Modus sind, werden die Nachrichten in der Benachrichtigungsleiste dargestellt. Laufen die Applikationen im Vordergrund, werden die Nachrichten in einem Overlay angezeigt.

UNTERSCHIEDE ZWISCHEN ANDROID UND IOS

Mit der Entwicklung der iOS App wurde später begonnen, daher ergeben sich einige Funktionsunterschiede zwischen beiden Versionen. Die Android App bietet einen größeren Funktionsumfang als die iOS Version. Zum einen werden verschiedene Rollen unterstützt. Das AMCS System bietet drei verschiedene Rollen an: Student, Dozent und Admin. Dozent und Admin Ansichten werden bisher nur von Android unterstützt. Sie bieten die Möglichkeit, den Status einer Vorlesung zu ändern und die aktuell aktive Folie zu wechseln.
Zum anderen können Studenten die bereits beantworteten Lernfragen im Nachhinein nochmals betrachten. Dieses Feature ist bisher nur in der Android Version implementiert.
Der Student hat die Möglichkeit in den Einstellungen die Server URL zu ändern und sich aus mehreren Kursen auf einmal auszutragen.
Zuletzt werden mehrere Nachrichten in der Android App gesammelt und untereinander angezeigt, in iOS wird immer nur die aktuellste Nachricht dargestellt.
Der restliche Funktionsumfang ist bis auf einige visuelle Unterschiede, bedingt durch die unterschiedlichen Plattformen, identisch.

3.1.2. ARCHITEKTUR

In beiden Versionen wurde bisher die grundlegende Architektur vernachlässigt. Es sind nur ein paar allgemeine Softwaremuster, wie z.B. Singleton und Callbacks verwendet worden. Dadurch befindet sich viel Code in den Activities unter Android bzw. den Controllern in iOS. Dies hat zur Folge, dass Tests nur sehr schwer umsetzbar sind und bisher keine Tests umgesetzt wurden.

3.1.3. VERSIONSKONTROLLE UND KONTINUIERLICHE INTEGRATION

Für die Versionskontrolle wird das von Atlassian entwickelte BitBucket verwendet. Da beide Applikationen unterschiedliche Programmiersprachen und Projektstrukturen haben, befinden sich die Versionen in zwei getrennten Quellen. Da nur ein Entwickler mit den Projekten beschäftigt ist, werden nur zwei Branches (develop und master) genutzt. Diese werden einmal für die Entwicklung und Release Versionen verwendet.
Da keine Tests vorhanden sind und die Veröffentlichung der Apps per Hand vorgenommen wird, gibt es bisher keine kontinuierliche Integration. Allerdings gibt es für das Backend und die Web Version bereits einen Jenkins Server, der das Bauen und Testen der Web Version und des Backends übernimmt.

3.2. ANFORDERUNGEN

Bei der Erstellung der automatischen Ausführung von Tests sind verschiedene Auswahlkriterien hilfreich. Es gibt bereits einige Werkzeuge, welche die Ausführung unterstützen, aber jeweils verschiedene Eigenschaften besitzen. Für den Einsatzzweck im Rahmen dieser Arbeit müssen diese überprüft werden.

3.2.1. AUSWAHL DER WERKZEUGE

Für die Umsetzung dieser Arbeit können verschiedene vorgefertigte Werkzeuge genutzt werden, die den Entwicklern bei der Umsetzung von Tests und der automatischen Ausführung dieser unterstützen.
A1 – Als Erstes wird die Unterstützung von unterschiedlichen App Typen festgelegt, welche in 2.1.1 Grundlagen von mobilen Applikationen beschrieben wurde.
A2 – Für die Umsetzung von Tests spielt die Unterstützung der Plattform ebenfalls eine Rolle. Speziell muss geprüft werden, ob die Möglichkeit besteht einen Test plattformübergreifend zu erstellen oder für jede Plattform extra Tests geschrieben werden müssen.
A3 – Die Entscheidung der Ausführung von Tests oder Applikationen auf Simulator/Emulator und/oder echten Geräten.
A4 – Es muss festgestellt werden, ob das Werkzeug die Möglichkeit besitzt Tests auf mehreren Geräten parallel auszuführen oder nur eine sequentielle Ausführung möglich ist.
A5 – Die Ausführungsdauer der Tests bzw. das Starten der Umgebung für die Testausführung muss untersucht werden.
A6 – Die Betrachtung der Bedeutung des Aufweckens der Geräte aus dem StandBy:
Weiterhin muss geprüft werden, ob sowohl verschiedene Ausrichtungen (vertikal oder horizontal) als auch die Verwendung von Hardware Tasten genutzt werden können.
A7 – Betrachtung der Programmiersprache für die Testerstellung:
Gibt es die Möglichkeit, die gleiche Sprache, wie die Entwicklungssprache zu nutzen oder muss eine andere Sprache erlernt werden?
A8 – Die Integration der Entwicklungswerkzeuge in das kontinuierliche Integrationssystem muss betrachtet werden.
A9 – Eine ausführliche Dokumentation und ein öffentlicher Quellcode kann für die Einarbeitung und das Finden von Features von Nutzen sein.
A10 – Die Auswertung der Tests nach derer Ausführung:
Die Aufbereitung von Log Dateien, z.B. zur Anzeige der wichtigsten Informationen können von Vorteil sein. Bei UI-Tests ist das automatische Anlegen von Screenshots oder sogar Videos nützlich, um Fehler reproduzieren zu können.
A11 – Speziell für UI-Tests ist die Art der Elementidentifizierung zu betrachten. Capture-Replay Verfahren sind gut geeignet, um schnell Tests zu erstellen, decken aber eventuell nicht alles ab. Es ist problematisch, wenn mit Koordinaten in verschiedenen Auflösungen und Texterkennung gearbeitet wird, um Elemente zu identifizieren. Durch verschiedene Auflösungen oder Ausrichtungswechsel können Elemente schwer erkennbar sein. Ein besseres System ist die direkte Identifizierung über IDs z.B. über eine XML-Struktur. Dadurch ist die Erkennung unabhängig von der Auflösung, Orientierung, Sprache oder Werten, welche sich von Gerät zu Gerät unterscheiden können.
Um die Auswahl einzuschränken, werden die einzelnen Punkte in Tabelle 3.1 zusammengefasst und eine Priorität für diese Arbeit festgelegt.
Kein untersuchtes Werkzeug kann alle Anforderungen abbilden und daher werden im Rahmen dieser Arbeit verschiedene Werkzeuge miteinander gebündelt, um die gewünschten Anforderungen zu erfüllen.

	Beschreibung	Priorität
A1	Unterstützung von App Typen	niedrig
A2	Plattform Unterstützung	niedrig
A3	Ausführung auf Simulator/Emulator/Geräten	hoch
A4	Parallele/Sequentielle Ausführung	niedrig
A5	Ausführungsdauer	niedrig
A6	Aufwecken, Ausrichtung, Hardwaretasten	mittel
A7	Programmiersprache für Tests	mittel
A8	Integration in IDEs, kontinuierliche Integrationsysteme	mittel
A9	Dokumentation und Quellecode	niedrig
A10	Aufbereitung der Logs	mittel
A11	UI Elementidentifizierung	hoch

Tabelle 3.1.: Auswahl von Werkzeugen

3.2.2. TESTARTEN

Ein Teil der Arbeit ist die Umsetzung von Tests anhand der in 2.1.2 vorgestellten Testarten. Unit Tests decken den funktionalen Teil einer Applikation gut ab und können durch weitere Frameworks - zu einem gewissen Grad - die Interaktion zwischen verschiedenen Komponenten überprüfen. Unit Tests haben daher eine hohe Priorität.
Integrationstests sind wichtig für die Überprüfung der Interaktion zwischen Komponenten. Aber durch die Komplexität führt dies oft zu Problemen oder unterschiedlichen Testergebnissen bei Testläufen. Daher haben Integrationstests nur eine mittlere Priorität.
Systemtests dienen der Überprüfung der nicht-funktionalen Aspekte. In dieser Arbeit wird auf Performanztests eingegangen und im Umfang von Systemtests haben diese eine niedrige Priorität. Daher wird nur auf die Performanz gesondert eingegangen.

3.3. ZUSAMMENFASSUNG

In diesem Kapitel wurde das Auditorium Mobile Classroom Service System genauer untersucht. Es wurde die Struktur und Funktionsweise von Backend und Frontend Systemen erläutert. Des Weiteren wurden die Funktionen der mobilen Applikationen beschrieben und auf die Unterschiede zwischen der iOS und Android Versionen eingegangen. Für die Testautomatisierung sind eine grundlegende Architektur, Versionskontrolle und kontinuierliche Integrationssysteme wichtig. Des Weiteren müssen verschiedene Werkzeuge und Testarten miteinander gebündelt werden, um die Anforderungen zu erfüllen.

4. KONZEPT

In diesem Kapitel wird ein Konzept für die Automatisierung von Tests für die mobilen Betriebssysteme Android und iOS entwickelt. Durch die Entwicklung erfolgen Änderungen an dem bisherigen Stand der jeweiligen Applikationen. Weiterhin wird die verwendete Architektur untersucht. Dabei wird überprüft, wie diese bei der Testautomatisierung zum Einsatz kommen kann. Die Ansätze der Testautomatisierung sowie die Testarten werden betrachtet. Anschließend wird die Versionskontrolle und der Prozess der kontinuierlichen Integration vorgestellt.

4.1. ARCHITEKTUR

Vor Beginn dieser Untersuchung war in der App keine grundlegende Architektur vorhanden. Somit ist die Planung der neuen Architektur immens wichtig. Ohne diese ist die Umsetzung von Tests sowie die Automatisierung nur sehr schwer realisierbar. Die Activities unter Android und die Controller unter iOS beinhalten fast die komplette Anwendungslogik und sind dementsprechend groß. Dadurch ist eine klare Trennung der einzelnen Komponenten nicht gegeben. Das Testen, die Wartung oder das Hinzufügen von neuen Funktionen zu den Applikationen ist nur eingeschränkt möglich. Wie bereits in Tabelle 2.2 und im Abschnitt Architekturen in mobilen Applikationen beschrieben, ist das Model View Presenter Muster ein sehr guter Ansatz für das Testen. Es gibt eine klare Trennung der einzelnen Komponenten. Der leicht erhöhte Aufwand durch die zusätzliche Erstellung von Interfaces wird durch die Vorteile aufgehoben. Die Übersichtlichkeit wird gefördert, da es eine zentrale Stelle gibt, in welchem die wesentlichen Funktionen abgebildet werden. Die Abbildung 4.1 bietet einen Überblick über die Funktionsweise des Musters.

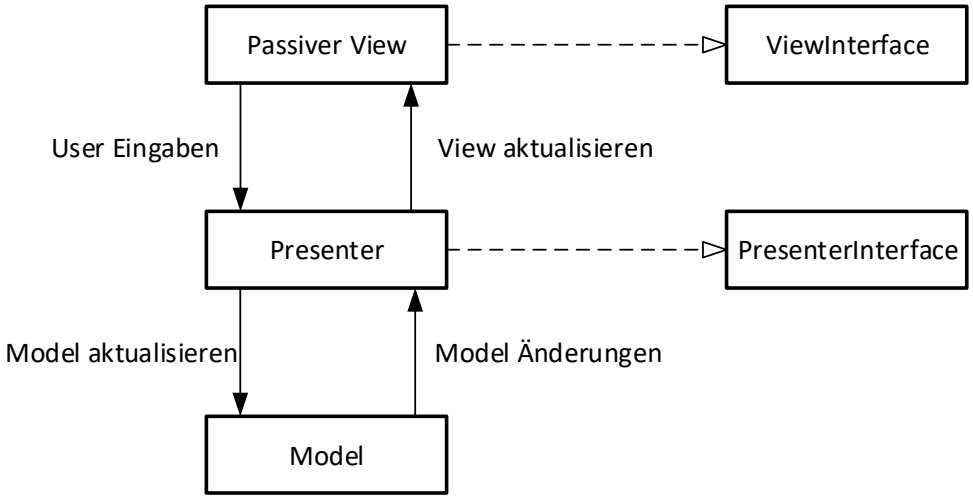


Abbildung 4.1.: Übersicht der Model View Presenter Architektur

Das **Repository Muster** bietet eine Abstraktionsschicht für das Model. Dadurch wird das Model von der eigentlichen Quelle der Daten unabhängig. Diese können über eine REST-Schnittstelle oder lokal aus einer Datenbank stammen. Vorteilhaft ist dabei, dass die Quelle ausgetauscht werden kann. Somit ist die Nutzung von Testdaten möglich, indem man eine andere Quelle nutzt. Durch die Verwendung von Testdaten lassen sich zum Beispiel Integrationstests umsetzen.
Für das eigentliche Testen der Anwendung können Mocking Frameworks genutzt werden, um teilweise unabhängig von Betriebssystem und Betriebsumgebung zu arbeiten. Die Tests können mit einfachen Unit Tests realisiert werden. Diese werden deutlich schneller ausgeführt als Tests, die Zugriffe auf das Betriebssystem bzw. bereitgestellte Methoden benötigen.
Für die Tests des **MVP** Musters müssen die entsprechenden anderen Schichten entweder über ein Mocking Framework oder durch eine eigene Implementierung ausgetauscht werden. Ein möglicher Testablauf für den Presenter wird in Abbildung 4.2 dargestellt. Dabei werden sowohl die Ansichtsschicht als auch die Modellschicht durch eine Mock Implementierung ersetzt. Hierbei wird vor allem überprüft, ob die Interaktionen des Views durch den Presenter die entsprechenden Methoden des Models aufruft.

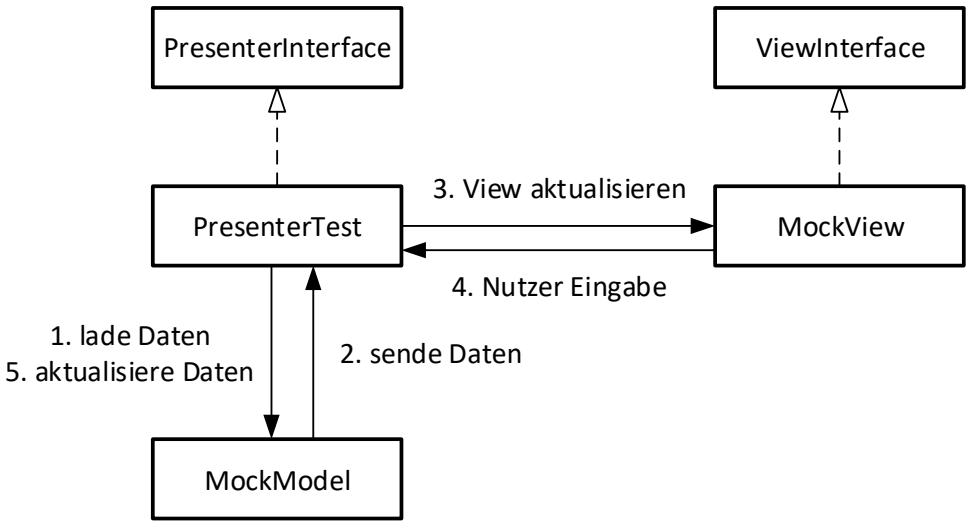


Abbildung 4.2.: Übersicht Presenter Test

Bei Tests für die Ansicht bzw. den View wird die Darstellungslogik und die Interaktion mit dem Presenter geprüft. Entsprechend muss nur der Presenter durch eine Mockimplementierung ersetzt werden.
Wenn das Model mitsamt der Anwendungslogik die Speicherung oder Beschaffung von Daten überprüft werden soll, muss ebenfalls der Presenter nachgeahmt werden und eine Datenquelle bereitgestellt werden.

4.2. ANSÄTZE FÜR TESTAUTOMATISIERUNG

Für die Erstellung von automatisierten Tests gibt es eine Reihe von Ansätzen, welche in Betracht kommen. Für einen Überblick werden im folgenden Abschnitt mehrere Ansätze beschrieben.

4.2.1. TESTSKRIPT

Die einfachste Form sind reine Skripte, welche später zum Testen einer Anwendung ausgeführt werden. Diese decken verschiedene Testfälle ab und überprüfen dabei den Code und die Funktionen einer Anwendung. Man kann diesen Ansatz zu einer modularen Vorgehensweise erweitern, wobei kleine und voneinander unabhängige Tests ausgeführt werden. Diese können hierarchisch kombiniert werden, um größere Tests zu erstellen und dadurch größere Testfälle abdecken. In dieser Form wird von einem modul-basierten Testskriptansatz gesprochen.

4.2.2. DATENGETRIEBENER ANSATZ

Bei dem datengetriebenen Ansatz werden Eingabe- und Ausgabewerte aus Skripten, Dateien oder Datenbanken geladen, um diese innerhalb der Software zu überprüfen. Dies kann z.B. bei einer Anmeldung oder dem Ausfüllen von Formularen eines oder mehrerer Nutzer innerhalb eines System hilfreich sein. [39]

4.2.3. SCHLÜSSELWORTGETRIEBENER ANSATZ

Bei dem schlüsselwortgetriebenen Ansatz handelt es sich um einen ähnlichen Ansatz wie der modul-basierte Testskriptansatz. Ziel ist es, die Abläufe in kleine Schritte zu zerlegen, um Tests flexibel erstellen zu können. Testskripte werden erstellt, indem man auf Schlüsselwörter einer Eingabetabelle zurückgreift. Die Tabelle beinhaltet kleinere Tests, die jeweils mit einem Schlüsselwort versehen sind, um bestimmte Funktionalitäten zu überprüfen. Durch diesen Ansatz wird eine übersichtliche Struktur der Tests gefördert und erhöht somit die Wiederverwendbarkeit. Nachteil ist der höhere Aufwand bei der initialen Erstellung der Tabelle und der Tests. [40]

4.2.4. HYBRIDER ANSATZ

Bei dem hybriden Ansatz handelt es sich um eine Kombination aus dem datengetriebenen und schlüsselwortgetriebenen Ansatz. Es werden die Vorteile von beiden Ansätzen kombiniert, um eine größere Testabdeckung zu erreichen. Die Testdaten und Schlüsselwörter müssen initial durch einen Programmierer erstellt werden. Anschließend können Tests auch ohne Programmierkenntnisse entwickelt werden.

4.2.5. CODE BASIERTES TESTEN

Das Code basierte Testen ist kein direkter Ansatz für die Testautomatisierung. Es ist der testgetriebenen Entwicklung (engl.: „Test-Driven Development“) zuzuordnen. Dieser Ansatz wird zur Vervollständigung dennoch aufgeführt. Hierbei werden Tests und Quellcode iterativ geschrieben, um Fehler zu entdecken. Je nach Anforderungen werden zuerst die Tests geschrieben und anschließend der Quellcode angepasst bis alle Tests bestanden werden. [41]

Im gesamten Umfang dieser Arbeit wird ein gemischter Ansatz verwendet. Zum einen ist durch die große Umstrukturierung der Applikationen ein Code basiertes Testen vorteilhaft, da eine einfache Refaktorisierung nicht ausreichend ist und viel Code neu erstellt werden muss. Daher ist die Erstellung von Testskripten vor der eigentlichen Codeerstellung durchaus sinnvoll, da so früh mögliche Fehler aufgedeckt werden.
Zum anderen sind einfache Testskripte im Umfang von AMCS ausreichend. Einzig der Login könnte anhand von einem datengetriebenen Ansatz überprüft werden. Aber die einfachen Unterscheidungen von korrekten und falschen Pseudonymen bzw. Passwörtern können auch innerhalb eines Testskripts kontrolliert werden.
Die Tests werden zunächst lokal beim Entwickler ausgeführt. Später durch die Übergabe an die Versionskontrolle automatisiert und innerhalb der kontinuierlichen Integration überprüft.

4.3. UNTERSUCHUNG VON TESTARTEN

Die Testarten spielen ein wichtige Rolle in der Dauer der Testausführung und in der Abdeckung von verschiedenen Zielen der Tests.

4.3.1. UNIT

Unit Tests haben den Vorteil, dass die Tests einfach, klein, schnell und vor allem unabhängig von anderen Komponenten sind. Soweit möglich sollen im Rahmen dieser Arbeit die meisten Tests mit Hilfe von Unit Tests umgesetzt werden. Unter Verwendung von Architekturen, Mustern und Mocking Frameworks können die einzelnen Komponenten einfach simuliert werden.

Dadurch lässt sich die einzelne Funktionalität leicht mit Unit Tests überprüfen. Im Grunde sollen die im Abschnitt 3.1.1 Funktionsübersicht untersuchten Funktionen durch Tests überprüft werden.

4.3.2. INTEGRATION

Integrationstests überprüfen die Interaktion von verschiedenen Komponenten einer Software. Sie haben den Nachteil, dass die Tests umfangreicher sind, mehr Abhängigkeiten besitzen und dies gegebenenfalls zu unsicheren Tests führt, welche manchmal erfolgreich sind oder fehlschlagen. Im Rahmen dieser Arbeit dienen sie der Überprüfung zwischen der REST Schnittstelle des Backends und der Modellschicht der Anwendungen.

4.3.3. PERFORMANZ

Die Performanz der Applikationen ist ein weiterer zu untersuchender Aspekt. Die Performanz kann auf verschiedene Weise betrachtet werden, z.B. die Startzeit, mögliche Verzögerungen bei Interaktionen, die Ladezeit des Inhalts oder die Zeit für das Ausführen von Code. Vorzugsweise sollten die Tests auf langsamen Geräten oder Emulatoren ausgeführt werden, um die Performanz zu untersuchen. Langsame Geräte geben einen besseren Aufschluss über Engpässe in der Ausführung der Apps.
Tests oder Testklassen sollten nach der Ausführungszeit klassifiziert werden und mit Hilfe von Anmerkungen, wie z.B. *@SmallTest < 100ms Ausführungszeit, @MediumTest < 2s Ausführungszeit, @LargeTest > 2s Ausführungszeit*, versehen werden.

4.4. ANFORDERUNGEN

Neue Funktionen oder Anforderungen sollten an einer zentralen Stelle festgehalten werden, da es wichtig ist einen Überblick über notwendige Aufgaben zu erhalten. Dazu kann ein Ticketsystem genutzt werden, um neue Funktionen oder Anforderungen festzuhalten. Anhand der einzelnen Tickets können Entwickler in einer testgetriebenen Entwicklungsumgebung zuerst die Tests ableiten und anschließend mit der eigentlichen Codeimplementierung beginnen. Die Definition von Tests anhand von User Stories sind ebenfalls wichtig, da somit eine Anforderung abdeckt werden kann.

4.5. VERSIONSKONTROLLE

Die Versionskontrolle ist ein wichtiges Werkzeug bei der Testautomatisierung. Sie bietet Zugriff auf den Quellcode und die Testskripte eines Projektes. Mit Hilfe von Tags und Zweigen können in der kontinuierlichen Integration vorher definierte Aktionen ausgeführt werden.
Grundsätzlich gibt es vier verschiedene Arten von Zweigen. Es gibt drei langlebige Zweige, dabei wird unter dem `master`-, dem `release`- und dem `develop`-Zweig unterschieden. Der `master`-Zweig beinhaltet immer eine lauffähige Version und stellt mit Hilfe von Tags, die die Versionsnummer darstellen, einen veröffentlichten Stand dar.
Der `develop`-Zweig dient als Integrations- und Grundlagenzweig für die Entwicklung von einzelnen Anforderungen. Mit Hilfe des `develop`-Zweigs werden die Integrationstests ausgeführt, um das Zusammenspiel der Funktionen zu überprüfen. Falls es zu Fehlern bei der Integration kommt, werden die Probleme im `develop`-Zweig behoben. Da der `master`-Zweig immer eine lauffähige Version beinhaltet, ist diese Vorgehensweise nicht problematisch. Nur wenn ein Entwickler von einem fehlerhaften `develop`-Zweig einen neuen `feature`-Zweig erstellt, sollte nach Behebung des Problems ein `rebase` ausgeführt werden, um die Änderun-

gen vom develop-Zweig in den aktuellen feature-Zweig zu integrieren.
Release Zweige bereiten eine Veröffentlichung vor und dienen zum Testen und Dokumentieren vor einer Veröffentlichung einer neuen Version.
Weiterhin werden die einzelnen Anforderungen vom develop-Zweig in feature-Zweige abgeleitet und nach Fertigstellung wieder zurück in den develop-Zweig integriert. Dies erfolgt über einen Pull-Request, wobei die Funktionalität der neuen Funktion innerhalb des gesamten Systems überprüft wird, um fehlerhafte Stände im develop-Zweig zu vermeiden. Wenn es im Pull-Request zu einem Fehler kommt, müssen diese entsprechend behoben werden. Entstehen keine Fehler, kann der feature-Zweig in den develop-Zweig integriert werden. Für die Testautomatisierung, zukünftige Entwicklungen und Erweiterungen ist diese Vorgehensweise vorteilhaft. [42]
Die Abbildung 4.3 dient zur Veranschaulichung des beschriebenen Prinzips.

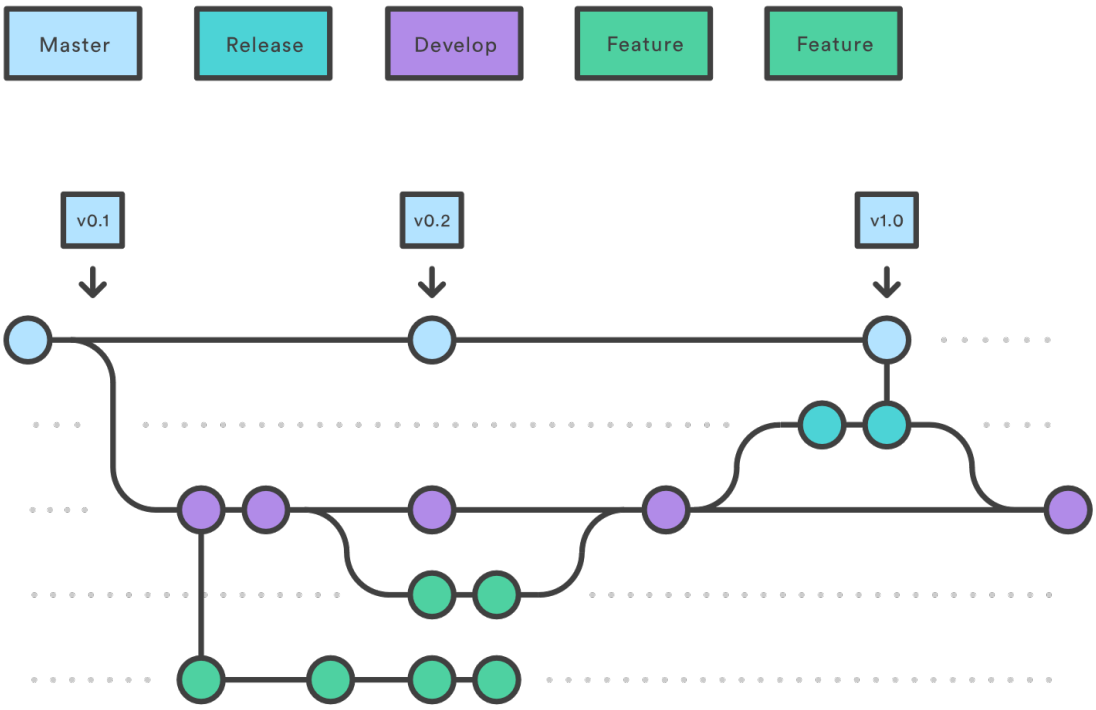


Abbildung 4.3.: Release- und Feature-Zweige Arbeitsablauf [43]

Bei Veröffentlichung einer neuen Version der App erfolgt nach Abarbeitung von Anforderungen oder anhand eines Stichtages, wann eine neue Version veröffentlicht werden soll. Die Tags orientieren sich an einer Versionsnummer, welche sich aus Haupt-, Neben- und Revisionsnummer zusammensetzt. Soll beispielsweise auf Grund von Kundenwünschen eine Funktion schnell veröffentlicht werden, kann man den Workflow Ansatz an die jeweilige Situation anpassen und eine Veröffentlichung auch nur mit einer neuen Funktion umsetzen.
Wenn eine neue Funkion umfangreicher ist oder in der Entwicklung länger dauert, als anfangs geplant, ist eine Spaltung des aktuellen Zweiges sinnvoll. Es sollte eine stabile Version des feature-Zweiges zurück in den develop-Zweig integriert, anschließend neue feature-Zweige erstellt und die Anforderung besser aufgeteilt werden. Eine stabile Version bedeutet in diesem Fall, dass die Anwendung gebaut werden kann und keine Unit Tests fehlschlagen. Zur Überprüfung der Funktionen sollten weiterhin für die feature-Zweige Tests erstellt werden und die Integration der Komponenten mit Integrationstests im develop-Zweig getestet werden.

4.6. KONTINUIERLICHE INTEGRATION

Der kontinuierliche Integrationsprozess ist im eigentlichen Sinne für die Automatisierung der Tests verantwortlich. Dafür ist der Zugriff auf den Quellcode bzw. auf die Versionskontrolle des Projektes notwendig. Er nutzt die verschiedenen Zweige der Versionskontrolle, um das automatische Testen in Gang zu setzen. Welche Tests dabei ausgeführt werden, kann unter Verwendung der Zweige konfiguriert werden. Es müssen nicht immer alle Tests ausgeführt werden, da dies einige Zeit in Anspruch nehmen kann. In einem `feature`-Zweig werden nur die Tests für die entsprechende Anforderung überprüft. Nach Integration einer neuen Funktion in den `develop`-Zweig sowie vor Veröffentlichung der App wird ein kompletter Testdurchlauf gestartet. So werden mögliche Integrationsprobleme aufgedeckt, welche bei den einfachen Tests der neuen Funktion nicht auftreten können, da hier ein kleinerer Funktionsumfang überprüft wird.
Die Herausforderung bei mobilen Applikationen besteht darin, die verschiedenen Betriebssystem Versionen, Auflösungen und Gerätevielfalt so gut wie möglich abzudecken.
Eine Testmatrix mit verschiedenen Eigenschaften ist in diesem Fall hilfreich. Somit können verschiedene Randbedingungen, wie die minimale und aktuelle Plattform API Unterstützung, verschiedene Auflösungen und Sprachen überprüft werden. Speziell für Android ist ein breites Testen der Betriebssystem Versionen (API Level) aufgrund der starken Fragmentierung wichtig. Aktuell [1] sind 98,7% aller Nutzer auf **neun** verschiedenen Android API Plattformen verteilt. Unter iOS wird sehr schnell auf die neueste Version des Betriebssystems aktualisiert. So genügt es, die neueste und vorherige Version zu testen. Aktuell [2] nutzen 52% die aktuelle iOS 11 Version und 38% iOS 10, wodurch 90% aller Nutzer abgedeckt sind.
Vorzugsweise sollten die Tests auf echten Geräten ausgeführt werden, um eine realitätsnahe Überprüfung zu erreichen. Auf Grund von fehlenden Ressourcen werden die Tests aber in Emulatoren bzw. unter iOS in Simulatoren durchgeführt. Für das AMCS Projekt sieht die Testmatrix wie folgt aus:

OS	Kategorie	Smartphone	Phablet	Tablet
Android	Neuste	Pixel 2	Pixel 2 XL	Pixel C
	Meist genutzt	Nexus 5X	Nexus 6	Nexus 10
	Älteste	Nexus 4	Samsung Note	Nexus 7
iOS	Neuste	iPhone 8	iPhone 8 Plus	iPad Pro
	Vorherige	iPhone 4S	iPhone 6 Plus	iPad 2

Tabelle 4.1.: Testmatrix für AMCS

Zur Simulation verschiedener Geräteklassen, werden eine Reihe von Emulatoren mit unterschiedlichen Auflösungen und API Versionen erstellt. Hierbei wurden die Daten der „Google Play Console“ untersucht, um die meist genutzte Android Version (Android 6.0, API Level 23) und Auflösungsdichten (hdpi und xhdpi) zu bestimmen. Die älteste Version ist Android 4.0 (API Level 15) und die neuste Version ist Android 7.1 (API Level 25). Unter iOS werden die neuesten und die ältesten unterstützten Modelle für eine breite Testabdeckung genutzt.

4.7. ÜBERSICHT DER KOMPONENTEN

In der folgenden Übersicht 4.4 wird das Zusammenspiel der einzelnen Komponenten dargestellt. Der komplette Ablauf kann in drei Hauptkomponenten zusammengefasst werden.

[1] Stand 24.11.2017 – Verteilung der Android Versionen im Google Play Store [15]
[2] Stand 24.11.2017 – Verteilung der iOS Versionen im App Store [44]

Zunächst werden aus den Anforderungen neue Aufgaben und Funktionen abgeleitet und in der ersten Komponente umgesetzt. Der erste Teil ist die testgetriebene Entwicklung, wobei iterativ Tests und Code erstellt werden. Die Versionskontrolle als zweite Komponente verwaltet den Quell- und Testcode und dient als Grundlage für die dritte Komponente. Die kontinuierliche Integration überprüft das Zusammenspiel der einzelnen entwickelten Anforderungen und Funktionen.

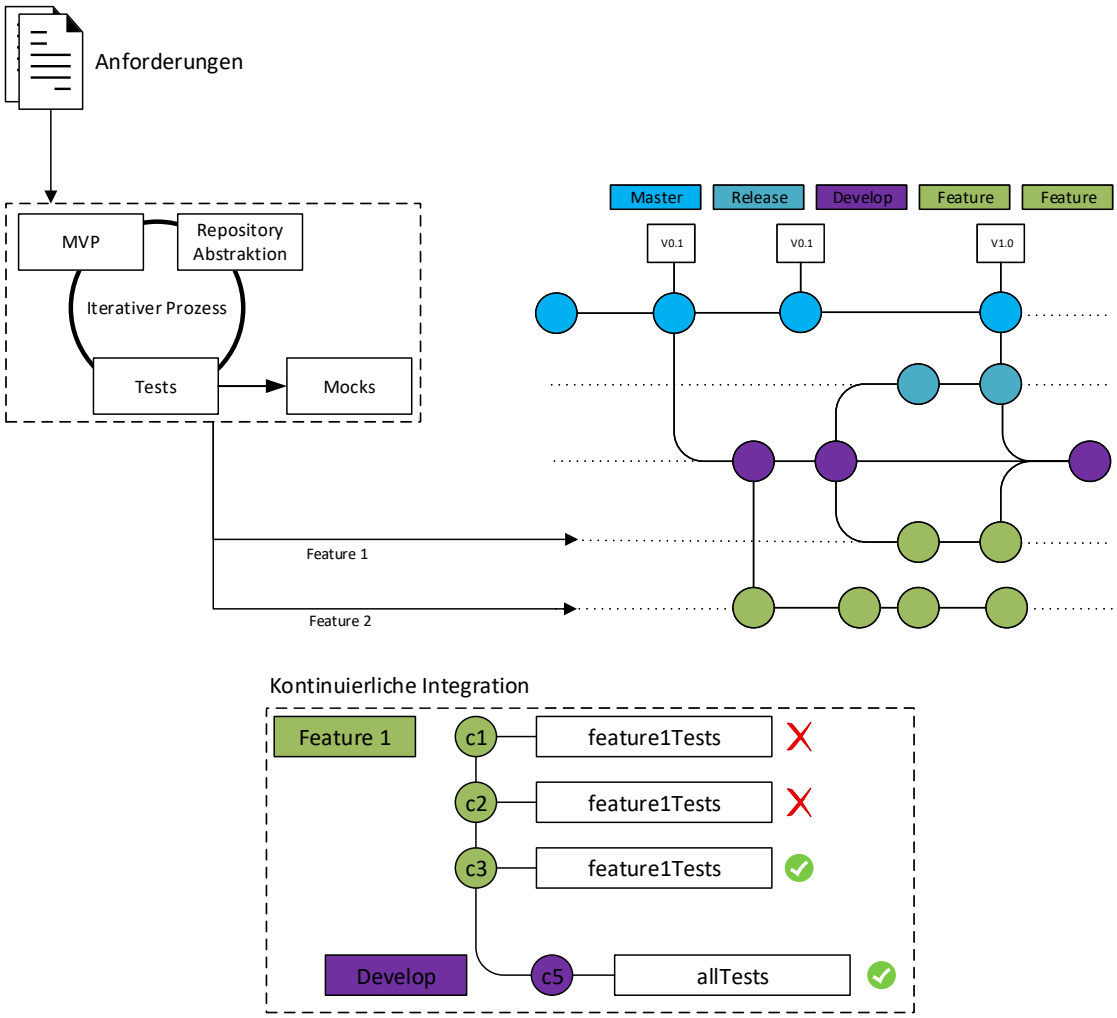


Abbildung 4.4.: Übersicht der Komponenten

4.8. ZUSAMMENFASSUNG

Dieses Kapitel beschreibt das Konzept für die Testautomatisierung. Eine wichtige Grundlage für das Testen ist das Model View Presenter Muster, um eine klare Trennung der einzelnen Komponenten zu ermöglichen. Das Repository Muster wird genutzt, um das Model von der eigentliche Quelle unabhängig zu machen. Dadurch können auch Testdaten genutzt werden und zur Überprüfung der Anwendungen beitragen. Es wurden verschiedene Ansätze für die Testautomatisierung betrachtet und sich für einen gemischten Ansatz entschieden. Da viel Code neu geschrieben werden muss, können die Tests iterativ während der Entwicklung geschrieben werden und dementsprechend den neuen Code überprüfen. Viele Tests sollen durch Mocking Frameworks unterstützt werden, um unabhängig von Plattform Frameworks

zu bleiben. Dadurch kann der Code durch einfache und schnelle Unit Tests getestet werden. Anhand von Integrationstests soll das Zusammenspiel zwischen den lokalen Datenbanken, der REST Schnittstelle und der eigentlichen Anwendung an sich überprüft werden.
Die Versionskontrolle verwaltet den Quellcode und Testskripte in verschiedenen Zweigen und mittels Tags können Aktionen in der kontinuierlichen Integration ausgeführt werden.
In der kontinuierlichen Integration erfolgt anschließend die eigentliche Testautomatisierung. Mit Hilfe einer Testmatrix kann ein breites Spektrum an unterschiedlichen Plattform APIs, Auflösungen und Geräten mit verschiedener Hardware abgedeckt werden.

5. IMPLEMENTIERUNG

In diesem Kapitel wird die Implementierung des Konzepts vorgestellt. Zum Ersten werden die Rahmenbedingungen bei der Umsetzung dieser Arbeit festgelegt. Zweitens wird auf die Architektur der Anwendung eingegangen und anhand der Testautomatisierung erklärt, wie die einzelnen Komponenten getestet werden. Darauffolgend wird die Versionskontrolle und deren Umgang genauer beschrieben. Zuletzt wird die kontinuierliche Integration mit Hilfe von Jenkins und einigen Plugins umgesetzt.

5.1. RAHMENBEDINGUNGEN

Im folgenden Abschnitt werden ausgewählte Rahmenbedingungen aufgelistet und definiert, die bei der Implementierung zum Einsatz kommen.
Bei der prototypischen Umsetzung des Konzeptes wurde zunächst die Implementierung einer Android Version fokussiert, da zum einen der OpenSource Gedanke unterstützt werden soll und zum anderem durch fehlende Ressourcen eine Umsetzung für iOS nicht möglich ist. Für die Automatisierung von iOS Tests ist eine Macintosh Umgebung notwendig, welche aktuell im Rahmen des AMCS Projektes nicht gegeben ist.

5.2. ARCHITEKTUR

Da Tests mit der aktuellen Codebasis nur schwer realisierbar sind, wurde die Architektur grundlegend überarbeitet und weite Teile des Codes neu implementiert. Wie bereits im Konzept im Abschnitt 4.1 angedacht, wurde das Model View Presenter Muster umgesetzt.
Hintergrundarbeiten, wie zum Beispiel das Laden von Daten, sollten unter Android immer unabhängig von der Ansichtsdarstellung erfolgen. Unter Android wird das MVP Muster genutzt, um unabhängig von Activities, Fragements, Views und vor allem von Events des Lebenszyklus einer Activity zu sein. Die einzelnen Schichten sind sauber getrennt, wodurch der Anwendungscode vereinfacht und Wartungsmöglichkeiten verbessert werden. [45]
Bei dem MVP Muster wird der View und der Presenter von jeweils zwei Interfaces abgeleitet. In der AMCS Applikation werden die zwei Interfaces des Views und Presenters innerhalb eines Vertrags (engl.: „Contract“) zusammengefasst. Dies führt zu einem leicht verständlichen Überblick über den Funktionsumfang des Views und Presenters, da die Funktionen innerhalb einer Datei deklariert werden. Anhand eines Beispiels für die Login Funktionalität eines Users wird die Umsetzung des MVP Musters in Abbildung 5.1 dargestellt.
Weiterhin wird für die Anmeldung eines Nutzers eine vereinfachte Form des Repository Musters genutzt, um eine Abstraktionsschicht für das Modell einzuführen. Für die Speicherung der

Login Daten werden die SharedPreferences genutzt. Unter Android wird dabei eine XML Datei verwendet, welche innerhalb des Applikations Ordners gespeichert wird. Für den Zugriff auf die Informationen wird eine Key/Value Struktur genutzt. [46] Da diese Daten durch andere Apps oder Nutzer mit „root"-Zugriff ausgelesen werden können, wird das Passwort des Nutzers **nicht** abgespeichert. Nichtsdestotrotz werden nach einer erfolgreichen Anmeldung verschiedene Felder gespeichert, z.B. die Zeitspanne des gültigen Authentifizierungstokens, die Benutzerrolle und Nutzer ID. Die Nutzer ID wird für die Benachrichtigungen via „Firebase Cloud Messaging" (FCM) benötigt, indem die von FCM zur Verfügung gestellte Geräte ID und Nutzer ID an den AMCS Server geschickt wird. Dadurch können Benachrichtigungen an einen Nutzer mit Hilfe des AMCS Servers und FCM versendet werden.

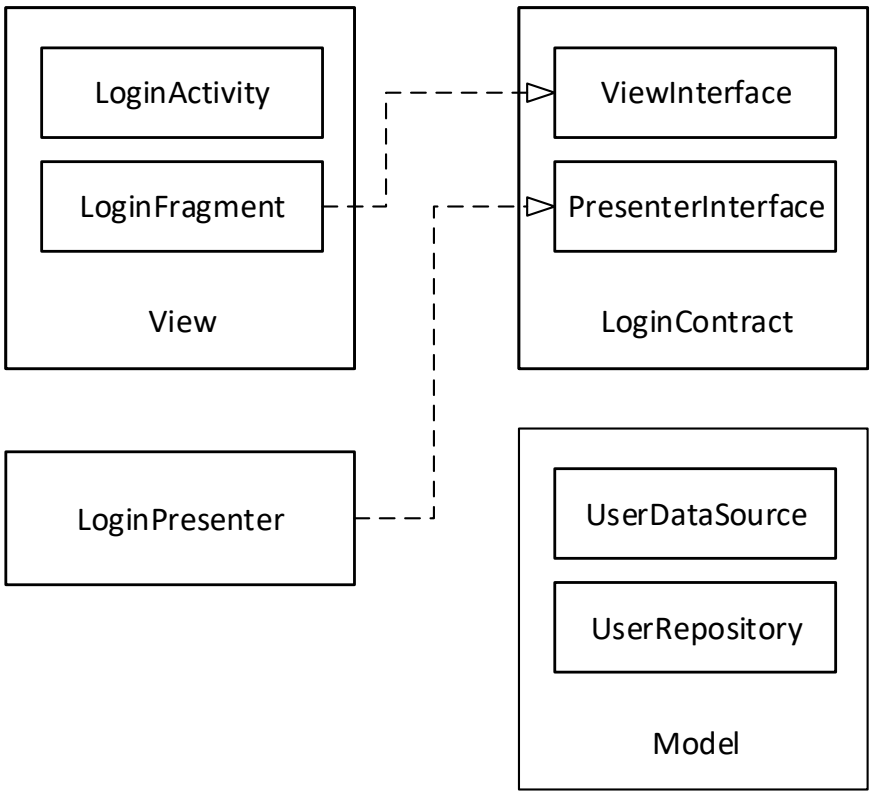


Abbildung 5.1.: Login Implementierung Model View Presenter und Repository Pattern

5.3. ANSÄTZE DER TESTAUTOMATISIERUNG

Für die Automatisierung der Testausführung ist es notwendig, zuerst Testklassen zu erstellen, welche später ausgeführt werden können. In Rahmen des Konzeptes wurde das Code basierte Testen bzw. „Test Driven Development"(TDD) vorgestellt.
Während der Implementierung der Tests und Restrukturierung bzw. Neuerstellung der Komponenten wurde das Konzept der testgetriebenen Entwicklung umgesetzt. Dadurch ergibt sich der Vorteil, dass weite Teile der Applikation durch Tests abgedeckt werden. Die Testklassen haben unterschiedliche Aufgabenbereiche. Zum einen gibt es einfache JUnit Tests und zum anderem gibt es UI Tests, welche das Verhalten der Oberfläche und Zusammenspiel mehrerer Komponenten überprüfen. Da bei den UI Tests meist komplexere Aufgaben getes-

tet werden und dabei mehrere Komponenten eine Rolle spielen, kann man UI Tests als eine Art von Integrationstests klassifizieren.

5.3.1. TESTEN DES PRESENTER

Anhand des Beispiels der Login Funktionalität wird die Testerstellung fortgeführt. Für das Testen des Presenters wird unter anderem das Mocking Framework Mockito verwendet. Dadurch ist es möglich, die Funktionalität des Presenters durch einfache JUnit Tests abzudecken. Mit Hilfe des Frameworks ist es überdies möglich, die `SharedPreferences` zu simulieren, wodurch ein Austausch der `UserRepository` Klasse möglich ist, um eigene Testdaten zu verwenden. Vor allem ist dies notwendig, um bereits vorhandene Einträge in den `SharedPreferences` zu löschen. Dies ist beim Testen des korrekten Anmeldeverhaltens notwendig, um gespeicherte Einträge zu löschen, welche das Verhalten beeinflussen können. Innerhalb des Testskripts können verschiedene Parameter - das Pseudonym und Passwort - definiert werden, um das korrekte Login Verhalten zu überprüfen. Die einzelnen Testfälle werden in folgender Tabelle 5.1 aufgelistet.

Fall	Beschreibung	Resultat
1	leeres Pseudonym, leeres Passwort	Hinweis: Pseudonym darf nicht leer sein
2	korrektes Pseudonym, leeres Passwort	Hinweis: Passwort darf nicht leer sein
3	zu kurzes Pseudonym, nicht leeres Passwort	Hinweis: Das Pseudonym ist zu kurz
4	zu langes Pseudonym, nicht leeres Passwort	Hinweis: Das Pseudonym ist zu lang
5	Pseudonym mit Leerzeichen, nicht leeres Passwort	Hinweis: Das Pseudonym darf keine Leerzeichen enthalten
6	korrektes Pseudonym, nicht korrektes Passwort	Hinweis: Die angegebenen Anmeldedaten sind nicht korrekt
7	korrektes Pseudonym, korrektes Passwort	Ausführung: Starten der Übersicht

Tabelle 5.1.: Fallübersicht der Anmeldung

Ein weiteres Beispiel zum Testen eines Presenters ist die `OverviewPresenterTest` Klasse. Es wird überprüft, ob eine Liste von Kursen oder das Anzeigen einer Information, dass keine Kurse vorhanden sind, dargestellt werden. Im folgenden Codebeispiel 5.1 wird die Logik des Presenters beim Laden von eingeschriebenen Kursen eines Nutzers und anschließender Darstellung aufgezeigt. Das Codebeispiel wurde aus Gründen der Übersichtlichkeit auf das Wesentliche gekürzt. Die möglichen Fälle, welche eintreten können, sind Antworten des Repositories in Form einer leeren Liste oder eine Liste mit Kursen. Zunächst werden in der `setupOverviewPresenter()`-Methode die Mocks initialisiert, der Presenter angelegt, zwei Testkurse erstellt und zu einer Liste hinzugefügt. Die zwei möglichen Fälle werden in eigenen Methoden `showEmptyCourses()` und `showCourses()` überprüft. Die bereitgestellten Testkurse werden über ein Callback vom Repository an den Presenter übergeben. Die Funktionen und das Callback für das Laden von Kursen wurde in dem Interface `CourseDataSource` definiert. Mit Hilfe von Mockito ist es möglich, einen `ArgumentCaptor` mit der entsprechend genutzten Callback Definition (`LoadCoursesCallback`) zu erstellen und mit Hilfe dessen Ergebnisse zurückzuliefern. In Zeile 31 und 41 des Codebeispiels wird überprüft, ob im Repository die Methode zum Laden von Kursen aufgerufen wird und übergibt den `ArgumentCaptor`

```
1  public class OverviewPresenterTest {
2
3    private static List<Course> COURSES;
4    @Mock
5    private OverviewContract.View mOverviewView;
6    @Mock
7    private CourseRepository mCourseRepository;
8    @Captor
9    private ArgumentCaptor<CourseDataSource.LoadCoursesCallback>
         mLoadCoursesCallbackCaptor;
10
11   private OverviewPresenter mOverviewPresenter;
12
13   @Before
14   public void setupOverviewPresenter(){
15     MockitoAnnotations.initMocks(this);
16
17     mOverviewPresenter = new OverviewPresenter(mCourseRepository, ...,
           mOverviewView);
18
19     Course course = new Course(); ...
20     Course course2 = new Course(); ...
21
22     COURSES = new ArrayList<>();
23     COURSES.add(course);
24     COURSES.add(course2);
25   }
26
27   @Test
28   public void showEmptyCourses(){
29     mOverviewPresenter.loadCourses();
30
31     verify(mCourseRepository).loadCourses(mLoadCoursesCallbackCaptor.
           capture());
32     mLoadCoursesCallbackCaptor.getValue().onCoursesLoaded(new ArrayList<
           Course>(0));
33
34     verify(mOverviewView).showEmptyCourses();
35   }
36
37   @Test
38   public void showCourses(){
39     mOverviewPresenter.loadCourses();
40
41     verify(mCourseRepository).loadCourses(mLoadCoursesCallbackCaptor.
           capture());
42     mLoadCoursesCallbackCaptor.getValue().onCoursesLoaded(COURSES);
43
44     verify(mOverviewView).showCourses(COURSES);
45   }
46 }
```

Quelltext 5.1: OverviewPresenterTest – Laden von Kursen

als Callback. Über den `ArgumentCaptor` in Zeile 32 wird eine leere Liste von Kursen übergeben. Anschließend wird überprüft, dass der Presenter dem View mitteilt die richtige Methode aufzurufen. In diesem Fall ist dies dem Nutzer zu symbolisieren, dass er in keine Kurse eingeschrieben ist. Dies geschieht über die `showEmptyCourses()`-Methode.
Der zweite Fall in Zeile 42 nutzt den `ArgumentCaptor`, um eine Liste mit Kursen zu übergeben. Die Kurse wurden in der `setupOverviewPrenseter()`-Methode definiert und der statischen Kursliste (`COURSES`) hinzugefügt. Anschließend wird wieder getestet, ob der Presenter die richtige Methode des Views aufruft. In diesem Fall ist das die *showCourses(COURSES)*-Methode, um die Liste von Kursen anzuzeigen.
Die Überprüfung der aktiven Vorlesungen erfolgt äquivalent zu den vorgestellten Tests zum Anzeigen einer Kursliste oder der Information, dass der Nutzer in keine Kurse eingeschrieben ist.
Eine Übersicht über alle Unit Tests befindet sich im Anhang.

5.3.2. TESTEN DES VIEWS

Durch die nur sehr passive Implementierung des Views und auf Grund der Steuerung des Views durch Presenter, ist die Überprüfung der Funktionalität bereits innerhalb des Presenters abgedeckt. Dabei wird getestet, ob die richtigen Funktionen aufgerufen werden. Genauer gesagt wird geprüft, ob über die Logik des Presenters dem Nutzer Daten und Hinweise angezeigt oder die Eingaben verarbeitet und dadurch entsprechende Aktionen ausgelöst werden. Innerhalb des Views ist also kaum Logik vorhanden. An dieser Stelle sind stattdessen UI-Tests zur Überprüfung der korrekten Initialisierung und des Verhaltens der Anzeigeelemente sinnvoll. In dem Anmeldebeispiel sind dies die Eingabefelder für Pseudonym und Passwort, der Hinweis zur Pseudonymerstellung und der Anmeldebutton, sowie die Kontrolle der Hinweistexte, welche bei einem falschen Anmeldeversuch angezeigt werden.
Für die Erstellung von UI-Tests wird das von Google entwickelte Espresso Framework genutzt, weil bei der Entwicklung implizit auf die Synchronisation zwischen Testausführung und Oberflächenanzeige geachtet wurde. Das hat den Vorteil, dass keine `wait()`-Methoden benötigt werden, welche bei verschiedenen Geräten zu lang oder auch zu kurz sein können. Dadurch werden unnötige falsche Fehlschläge bei der Testausführung vermieden, z.B. wenn die Oberfläche noch nicht korrekt dargestellt wird, aber bereits getestet wird, ob ein Element angezeigt wird. Für die Überprüfung der Nutzeroberfläche werden eigene Testklassen erstellt, um eine Trennung zwischen den schnellen Unit Tests und langsameren UI Tests zu erreichen.
Für die Ausführung von UI Tests unter Android werden ein paar zusätzliche Einstellungen bzw. Abhängigkeiten benötigt. Unter anderem werden die Abhängigkeiten für das bereits erwähnte Espresso Framework benötigt und weiterhin muss in der `.gradle`-Datei ein TestRunner definiert werden, um die UI Tests ausführen zu können. Gradle wird für das Bauen von Android Projekten verwendet. Die `.gradle`-Datei dient zur Konfiguration der Android App und dem Verwalten von Abhängigkeiten.
In der folgenden Auflistung 5.2 wird überprüft, ob verschiedene Ansichtselemente auf dem Gerät oder Emulator dargestellt werden.

```
1  @RunWith(AndroidJUnit4.class)
2  public class LoginUITest {
3    @Rule
4    public ActivityTestRule<LoginActivity> mActivityRule = new
         ActivityTestRule<LoginActivity>(LoginActivity.class){
5      ... };
6    @Test
7    public void testViewsAreBeingDisplayed(){
8      onView(withId(R.id.login_username_et))
9        .check(matches(isDisplayed()));
10     onView(withId(R.id.login_pwd_et))
11       .check(matches(isDisplayed()));
12     onView(withId(R.id.login_pseudoyn_note_tv))
13       .check(matches(isDisplayed()));
14     onView(withId(R.id.login_btn))
15       .check(matches(isDisplayed()));
16   }
17 }
```

Quelltext 5.2: LoginUITest Überprüfung, ob Ansichtselemente dargestellt werden

Durch die Annotation `@RunWith(AndroidJUnit4.class)` wird der entsprechende Teststil initialisiert. Dies hat zur Folge, dass die Ergebnisse des Testlaufs wie in JUnit 4 dargestellt werden. Für die Ausführung von einem UI Test wird eine sogenannte `ActivityTestRule` benötigt. Anhand der definierten `ActivityTestRule` übernimmt das Espresso Framework die Initialisierung des Testablaufs und der jeweiligen Activity. Dadurch muss der Entwickler weniger Initialisierungscode und Code zum Verbinden von Elementen schreiben. Weiterhin wird die Activity nach der Testausführung wieder entsprechend beendet. In früheren Versionen von Espresso unter der Verwendung von `ActivityInstrumentationTestCase2` musste das Starten und Beenden einer Activity selbst implementiert werden.
Das Finden von UI Elementen kann anhand von verschiedenen Parametern bestimmt werden. Die sicherste Methode ist die ID des Elements zu nutzen, welche in der Layout-Datei oder im Code festgelegt wurde. Die Layout-Datei ist in Form einer XML-Struktur aufgebaut und definiert die Anzeigeelemente der Nutzeroberfläche. Bei der Selektion von Elementen gibt es außerdem die Möglichkeit Elemente per Text, Kinderelemente, Tags oder Eingabetypen zu bestimmen.
Auf den gefundenen Elementen können zum Beispiel Überprüfungen oder Aktionen ausgeführt werden. Zur Überprüfung, ob ein Element angezeigt wird, kann `onView(withId(R.id.some_id).check(matches(isDisplayed()));` genutzt werden. Anhand von Aktionen können unterschiedliche Tätigkeiten ausgeführt werden, z.B. die Eingabe von Text oder das Drücken eines Elementes.
Zum Starten von anderen Activities werden in Android in der Regel `Intent` Objekte genutzt. Zum Testen, ob die Intents zum erfolgreichen Aufruf einer anderen Activity führen, werden `IntentsTestRule` verwendet. Dadurch können aufgerufene Intents verifiziert werden. Im Quellcode 5.3 wird das Starten der `OverviewActivity` nach einer erfolgreichen Anmeldung eines Nutzers überprüft. Aus Gründen der Übersichtlichkeit wurde das Codebeispiel gekürzt und nicht der komplette Methodeninhalt bzw. Klasseninhalt dargestellt.

```
@RunWith(AndroidJUnit4.class)
public class LoginUIIntentTest {
  @Rule
  public ActivityTestRule<LoginActivity> mActivityTestRule = new
      ActivityTestRule<LoginActivity>(LoginActivity.class){
  ...};

  @Rule
  public WireMockRule wireMockRule = new WireMockRule();

  //setup WireMock

  @Test
  public void testShowOverview(){
    //enter correct username and password
    onView(withId(R.id.login_username_et))
      .perform(typeText("uiTestUser"));
    onView(withId(R.id.login_pwd_et))
      .perform(typeText("abc"));

    Intents.init();

    //configure WireMock stubs

    //create and register an IdlingResource
    RequestIdlingResource requestIdlingResource = new RequestIdlingResource
        (context);
    IdlingRegistry.getInstance().register(requestIdlingResource);

    onView(withId(R.id.login_btn))
      .perform(click());

    //unregister the IdlingResource
    IdlingRegistry.getInstance().unregister(requestIdlingResource);

    //check for an Intent to start the OverviewActivity
    intended(hasComponent(OverviewActivity.class.getName()));

    Intents.release();
  }
}
```

Quelltext 5.3: LoginUIIntentTest - Aufruf der OverviewActivity testen

Wie bereits erwähnt, stellt Espresso die Funktionalität zur Überprüfung von Activity Aufrufen über die `IntentsTestRule` und die Funktion `intended()` bereit. In diesem Beispiel wird allerdings ein `Context` benötigt, weil eine Netzwerkanfrage durchgeführt werden muss. Der `Context` kann aber nur über eine `ActivityTestRule` und nicht über `IntentsTestRule` abgerufen werden. Die Activity wird nur gewechselt, wenn der Anmeldeversuch erfolgreich war. An dieser Stelle tritt ein weitreichendes Problem auf und zwar die Ausführung von Anfragen über ein Netzwerk. Zur Überprüfung der korrekten Funktionalität der Oberfläche werden verschiedene Ressourcen, wie z.B. Kurs- oder Vorlesungsdaten benötigt. Außerdem soll bei UI Tests die Darstellung einer korrekten Oberfläche überprüft werden, die Abhängigkeit von Anfragen über das Internet ist daher nicht optimal. Dieses Problem wurde gelöst, indem die Netzwerkanfragen nur auf dem lokalen Gerät verarbeitet werden und nicht auf fehleranfällige Anfragen über das Internet angewiesen ist. Dabei wurde auf das „WireMock“ Framework zurückgegriffen, dies ermöglicht ein einfaches Starten eines lokalen Web Servers sowie die Definition von Standardantworten, wenn auf bestimmte Adressen zugegriffen wird. Das Wi-

reMock Framework und die Funktionsweise im Bezug auf das Testen wird in Abschnitt 5.3.3 genauer beschrieben.
Weiterhin ist für Espresso nicht bekannt, dass eine Netzwerkanfrage durchgeführt wird und wie lange diese benötigt. Obwohl die Ausführung von Anfragen auf dem lokalen Gerät statt findet, benötigen diese eine gewisse Zeit. Espresso greift zur Synchronisation der Test und UI Ausführung auf den UI Thread zurück. Netzwerkanfragen laufen nicht auf dem UI-Thread ab, sondern werden in einem separaten Thread ausgeführt und verarbeitet. Durch die Verwendung des Espresso Frameworks kann eine eigene `IdlingResource` implementiert werden, welche Espresso mitteilt, dass gerade eine Aktivität im Hintergrund ausgeführt wird. [47] Um Espresso zu signalisieren, dass während der Testausführung ein separater Thread läuft, wurde eine `RequestIdlingResource` Klasse erstellt. Die `RequestIdlingResource` Klasse implementiert das Interface `IdlingResource` und nutzt intern das Volley Framework, um zu überprüfen, ob gerade eine Netzwerkanfrage durchgeführt wird. Volley wird innerhalb der Anwendung genutzt, um HTTP Anfragen an das Backend zu stellen. Die `RequestIdlingResource` wird in der `IdlingRegistry` registriert und dadurch wird Espresso mitgeteilt, dass die Testausführung solange pausieren soll bis die Anfrage ein Resultat liefert.
Im Prinzip handelt es sich bei der `RequestIdlingResource` um einen atomaren Zähler, der seinen Zustand thread-übergreifend beibehält und den Wert des Zählers erhöht bzw. herabsetzt, wenn eine Anfrage ausgeführt wird oder beendet wurde. Wenn der ursprünglicher Wert wieder erreicht wird, wird Espresso über ein Callback informiert, sodass die Testausführung weiterlaufen kann.
Zur Überprüfung des eigentlichen Tests, dem Wechsel der Activity, werden drei wesentliche Methodenaufrufe benötigt. Als erstes wird in Zeile 16 Espresso signalisiert, dass `Intents` genutzt werden. Mit Hilfe von `intended(...)` und dem Klassennamen der aufzurufenden Activity kann der Aufruf eines `Intents` getestet werden. Zuletzt wird in Zeile 34 Espresso signalisiert, dass der `Intent` Aufruf beendet worden ist. Dadurch kann überprüft werden, ob ein Wechsel der Activity mit Hilfe eines `Intents` stattgefunden hat.

Ein weiteres Problem beim Testen der Oberfläche ist die Verwendung eines Tokens, um den Nutzer zu authentifizieren. Im Zusammenspiel zwischen dem Front- und Backend des AMCS Systems wird ein REST Webservice genutzt. REST ist per Definition bei jeder Kommunikation zwischen Client und Server zustandslos, d.h. alle notwendigen Informationen zum Verstehen der Anfrage, werden mitgeliefert. [48] Der Ablauf des AMCS Systems sieht vor, dass der Nutzer sich zuerst im System anmelden muss. Anschließend bekommt die Applikation vom Backend einen JSON Web Token. Der Token dient als kompakte und sichere Form der Übertragung von Authentifizierungsinformationen. [49] Innerhalb des JSON Web Tokens befindet sich die Nutzerrolle, der Ablaufzeitstempel sowie ein Token, welcher den Nutzer im Backend authentifiziert. Dieser Token muss in allen nachfolgenden Backendanfragen mitgesendet werden. Die Zustandslosigkeit wird dabei missachtet, was ebenfalls zu Abhängigkeiten in der Testausführung führt.
Tests sollten in ihrer Ausführung unabhängig von anderen Tests sein, außerdem ist die Reihenfolge der Testausführung jedes mal unterschiedlich. Bis auf die Anmeldeansicht, der „Über"-Ansicht und den Einstellungen, müssen alle anderen UI Tests als erstes einen Login eines Testnutzers ausführen. Das Anmeldeproblem lässt sich lösen, indem die Anmeldung durch die Methode mit der Anmerkung `@Before` vor jedem Test ausgeführt wird. Das hat jedoch den Nachteil, dass für jeden UI Test die `LoginActivity` als erstes gestartet werden muss, die Anmeldeanfrage eines Testnutzers gestellt wird und erst im Anschluss zu der zu testenden Ansicht navigiert werden kann. Erst danach kann die eigentliche Testausführung beginnen, um die Darstellung oder Funktionalität der Oberfläche zu überprüfen. Die Abhängigkeit des Anmeldens ist nicht optimal und außerdem ist die Ausführungszeit der Tests deutlich höher. Wenn die Anmeldung umgangen werden würde, hätte dies andere Probleme zur Folge. Der

grundlegende Aufbau und Ablauf der Anwendung sowie die Vorgehensweise des kompletten AMCS Systems werden außer Acht gelassen. Daher muss bei der Erstellung der Tests auf die vorhandene Abhängigkeit eingegangen werden und Tests müssen entsprechend angepasst werden.

5.3.3. TESTEN DES MODELS

Für das Testen der dritten Schicht, also die Datenschicht bzw. das Model, können ebenfalls Mock Frameworks genutzt werden. Das Model kümmert sich um die Beschaffung oder Speicherung von Daten, welche von der Applikation benötigt werden. Wie bereits erwähnt, kann die Speicherung und das Auslesen von Daten über die `SharedPreferences` erfolgen. Die meisten Objekte werden im AMCS Projekt im Hauptspeicher gehalten. Es werden REST-Anfragen an das Backend von AMCS gestellt, um verschiedene Daten zu beschaffen oder zu versenden. Für die Umsetzung der Anfragen wird das HTTP-Protokoll genutzt und bietet entsprechende Operationen, welche nachfolgend kurz aufgelistet werden.

- GET – Lesen von Ressourcen
- POST – Erstellen von neuen Ressourcen
- PUT – Erstellen oder Bearbeiten von Ressourcen
- DELETE – Löschen von Ressourcen

Da das Netzwerk oft eine Fehlerquelle ist, aber die eigentliche Funktionalität des Models überprüft werden soll, kann das Netzwerk entfernt werden. Anfragen und Antworten vom Backend werden lokal eingespeist.
Die eigentlichen Tests sollen das Verwalten der Daten überprüfen. Das sind unter anderem die Auswertung und Umwandlung von JSON Daten in interne Objekte. Weiterhin die Speicherung von Daten, zum Beispiel in den `SharedPreferences` oder das Schicken von Antworten an den AMCS Server. Zunächst war der Einsatz von „REST-assured“ geplant, welches zum Testen und Validieren von REST Schnittstellen innerhalb von Java genutzt werden kann. [50] Allerdings lässt sich das Werkzeug nicht nutzen, da Android offiziell nicht unterstützt wird. Im GitHub Repository gibt es zwei Tickets bezüglich der Android Unterstützung, die jeweils mit dem Status „won't fix“ geschlossen wurden. [51, 52] Das heißt, dass eine offizielle Unterstützung seitens der Entwickler nicht vorhanden ist und somit dieses Problem nicht weiter untersucht wurde.
Stattdessen wird das „WireMock“ Framework genutzt. Mit Hilfe von WireMock ist es möglich, einen lokalen Webserver zu starten und über eine programmierbare Schnittstelle (kurz API) zu konfigurieren. Über die API können verschiedene URL Adressen im lokalen Webserver gesetzt werden und je nach Anfrage Typ (GET, PUT, POST oder DELETE) können vorgegebenen Antworten zurückgegeben werden. Die Antworten in Form von JSON Anweisungen werden in Dateien innerhalb der Android Test Assets gespeichert, dadurch werden die Dateien nur zur Test .apk Datei hinzugefügt. Die .apk Datei ist eine verpackte Form der Applikation, welche unter Android installiert werden kann. In der Veröffentlichungsversion werden die JSON Daten nicht hinzugefügt, wodurch die App nicht unnötig vergrößert wird.
Im folgenden Codebeispiel 5.4 wird ein fehlerhafter Anmeldeversuch simuliert. Das Beispiel wurde aus Gründen der Übersichtlichkeit gekürzt.

```
1  @RunWith(AndroidJUnit4.class)
2  public class LoginIntegrationTest {
3
4    @Rule
5    public ActivityTestRule<LoginActivity> mActivityRule = new
         ActivityTestRule<LoginActivity>(LoginActivity.class){
6      ...
7    }
8
9    @Rule
10   public WireMockRule wireMockRule = new WireMockRule();
11
12   @Before
13   public void setup(){
14     //set BaseUrl, init AssetUtil, init UserRepository
15   }
16
17   @Test
18   public void testInvalidCredentials(){
19     User fakeUser = new User(); ...
20
21     String json = mAssetUtil.readAsset("errors/invalidCredentialsProvided.
           json");
22
23     stubFor(post(urlMatching(url))
24       .willReturn(aResponse()
25       .withStatus(409)
26       .withBody(json)));
27     ...
28
29     mUserRepository.login(fakeUser, new UserDataSource.GetUserCallback() {
30       @Override
31       public void onUserLoaded(User user) {
32         assertFalse(!user.getUsername().isEmpty());
33       }
34
35       @Override
36       public void onFailure(int error) {
37         assertEquals(UserRepository.INVALID_CREDENTIALS, error);
38       }
39     });
40   }
41   ...
42 }
```

Quelltext 5.4: LoginIntegrationTest - Falsche Anmeldedaten

Zunächst wird über die `wireMockRule` die WireMock Umgebung mit WebServer gestartet und in der `mActivityRule` die SharedPrefernces zurückgesetzt. In der `setup()`-Methode wird die URL, der Anwendung auf die Adresse `http://127.0.0.1:8080` gesetzt. Diese wird standardmäßig von WireMock genutzt, kann bei Bedarf aber angepasst werden. An die angegebene Adresse laufen alle Anfragen der App. Für das Auslesen der JSON Dateien wird die `AssetUtil` Klasse genutzt und das zu testende Repository wird initialisiert. In der dargestellten `testInvalidCredentials()`-Methode findet der eigentliche Test statt. Zuerst wird ein neues User-Objekt erstellt und mit einem Pseudonym und Passwort initialisiert. Anschließend wird die JSON Datei, welche die Fehlerausgabe des Backends beinhaltet, ausgelesen und in einer Variable gespeichert. Mit der `stubFor(...)`-Methode wird der WireMock Webserver konfiguriert. In diesem Fall werden innerhalb der `UserRepository` Klasse die An-

meldedaten über ein POST an das Backend geschickt. Wenn die URL aufgerufen wird, antwortet der Webserver mit dem HTTP-Statuscode 409 - Conflict und den zuvor gespeicherten JSON Anweisungen. Die Auswertung der Anfrage erfolgt innerhalb der `UserRepository` Klasse, im Codebeispiel werden die Ergebnisse überprüft. In diesem Fall sollte kein `User` Objekt und stattdessen ein Fehlercode zurückgegeben werden. Wenn das `UserRepository` über das `GetUserCallback` ein valides `User` Objekt zurückgibt, ist ein Fehler aufgetreten, da dieser Fall nicht eintreten sollte. Wenn es durch einen internen Fehler trotzdem dazu kommen sollte, wird ein Misslingen des Tests verursacht. Dabei wird geprüft, ob der Nutzername eines `User` Objektes nicht leer ist. Wenn die Ausführung korrekt ist und die `onFailure()`-Methode aufgerufen wird, wird im Anschluss der Fehlerstatus überprüft. In diesem Fall sollte der Status: „Falsche Anmeldedaten“ zurückgegeben werden. Wodurch im Zusammenhang der kompletten Applikation sichergestellt wird, dass der View die richtige Fehlermeldung anzeigen kann.
Der Vorteil von WireMock ist, dass der Webserver einfach gestartet und über die API sehr leicht konfiguriert werden kann. Die Anwendungslogik des Models kann überprüft werden, indem verschiedene Fälle definiert werden. Diese können die Fehlerausgabe oder das Liefern der richtigen Ergebnisse testen. Die Fallerstellung richtet sich nach Funktionalität der Anwendung und vor allem anhand der AMCS Backend API Dokumentation. Die Dokumentation beschreibt die möglichen Routen, um Daten anzufragen oder zu verschicken. Weiterhin werden die entsprechenden Antworten und Fehlerausgaben beschrieben, die dabei auftreten können.
Ein Nachteil hat das Framework allerdings, durch verschiedene Abhängigkeiten werden ca. 70.000 Methoden benötigt. [53] In Android werden innerhalb der .apk Datei sogenannte „.dex“-Dateien erstellt, welche den ausführbaren Java Code beinhalten. Wenn die Anzahl der Methoden 64.000 übersteigt, werden mehrere .dex-Dateien erstellt. Dies ist beim Testen problematisch, da die Android Versionen unter 5.0 oder API Level 21 keine Tests mit mehreren .dex-Dateien ausführen können. Dadurch wird die Ausführung auf unterschiedlichen Android Versionen eingeschränkt. Aktuell bedeutet dies im Fall von AMCS, dass 21% der Android Versionen [1] nicht mit Integrationstests überprüft werden können. Es handelt sich ausschließlich um ältere Android Versionen mit einer fallenden Verwendungstendenz.

5.4. VERSIONSKONTROLLE

Für die Versionsverwaltung wird das von Atlassian entwickelte BitBucket genutzt. Es handelt sich dabei um ein Git-Versionsverwaltungssystem. Es wurde sich für BitBucket entschieden, da die Nutzung für Studenten kostenlos ist und innerhalb des AMCS Projektes schon verwendet wird.
Mit dem Beginn der Masterarbeit wurde das bisher entwickelte Android Projekt geforkt, um Änderungen der Masterarbeit unabhängig von der Entwicklung des AMCS Projektes zu ermöglichen. Dies ist notwendig, da die Implementierung im Rahmen dieser Arbeit einige Zeit in Anspruch nimmt, aber parallel eine lauffähige Android Version vorhanden sein muss. Die laufende Version wird weiterhin entwickelt und neue Funktionen werden integriert. Diese müssen nach Beendigung der Arbeit zusammengeführt werden.
Bei der Versionskontrolle wird der sogenannte Git Workflow Ansatz betrieben. Bei dem Git Workflow dient der `master`-Zweig als Veröffentlichungsverlauf, wobei Tags mit Versionsnummern zur Identifikation genutzt werden. Der `develop`-Zweig wird zur Integration der einzelnen `feature`-Zweige genutzt. Neue Anforderungen werden in `feature`-Zweige implementiert, welche als Basis den `develop`-Zweig nutzen und nach Abschluss der Implementierung wieder in diesen integriert werden. Der Git Workflow sieht vor, dass die `feature`-Zweige nur lokal erstellt und entwickelt werden. In dieser Arbeit werden die Stände der aktuellen Ent-

[1] Stand 24.11.2017 – Verteilung der Android Versionen im Google Play Store [15]

wicklung allerdings zu BitBucket geladen. Dies hat mehrere Vorteile, zum einen dient es als Backup und zum anderen ist dadurch eine einfache Verteilung möglich, um auch auf anderen Computern die Arbeit fortzusetzen. Außerdem können andere Entwickler den Stand verfolgen und weiterhin werden die Unit Tests in der kontinuierlichen Integration ausgeführt. Wenn in dem `develop`-Zweig genug neue Funktionen integriert wurden und meistens ein Datum zur Veröffentlichung vorangeschritten ist, wird ein neuer `release`-Zweig vom `develop`-Zweig erstellt. In dem `release`-Zweig werden keine neuen Funktionen mehr implementiert, sondern nur Fehlerbehebungen oder Dokumentation eingepflegt. Sobald der Zweig bereit zur Veröffentlichung ist, wird er in den `master`-Zweig integriert und mit einer Versionsnummer versehen. Des Weiteren wird der `release`-Zweig wieder zurück in den `develop`-Zweig integriert und damit beendet bzw. gelöscht.
Diese Vorgehensweise vereinfacht vor allem die Zusammenarbeit von mehreren Teams an einem Projekt. Beispielsweise bereitet ein Team die Veröffentlichung der Version 4.0 vor und ein anderes Team arbeitet an den nächsten Funktionen für die nächste Veröffentlichung.
Wenn nach einer Veröffentlichung ein schwerwiegender Fehler auffällt, wird ein `hotfix`-Zweig vom aktuellen `master` erstellt. Im `hotfix` wird eine entsprechende Lösung für das Problem erstellt und anschließend wieder mit dem `master` integriert und mit einer neuen Versionsnummer versehen. Außerdem wird der Stand des `master` in Stand des `develop`-Zweigs integriert. Durch dieses Vorgehen wird der Rest des Teams nicht in seiner Arbeit unterbrochen oder gar auf den nächsten Termin einer Veröffentlichung gewartet. [42] Die gesamte Vorgehensweise wird in folgendem Diagramm visualisiert.

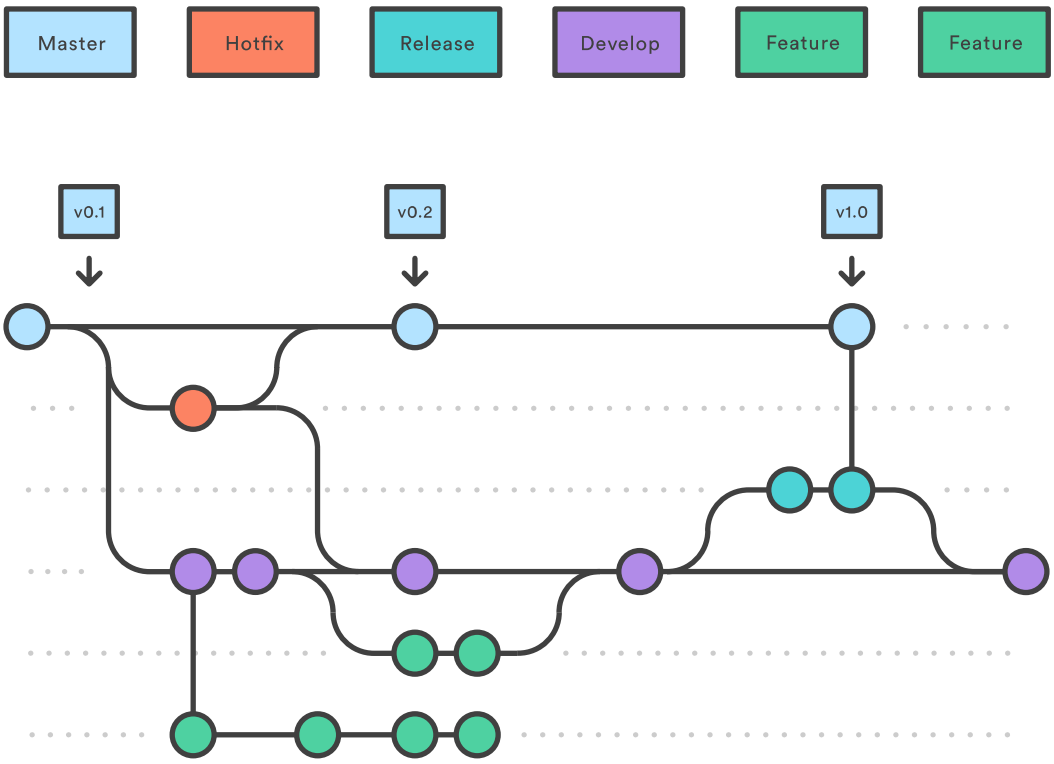


Abbildung 5.2.: Git Workflow mit Hotfix Zweig, Quelle: [54]

5.5. KONTINUIERLICHE INTEGRATION

In der kontinuierlichen Integration findet das automatische Testen der Anwendung statt. Dies wurde mit Hilfe des Automatisierungsservers Jenkins umgesetzt, da das System einfach einzurichten ist, viele Plugins bietet und vor allem keine Limitierungen bei der Anzahl an Jobs

oder Build-Agenten vorgibt.

5.5.1. PLUGINS

Für die Unterstützung von Android Projekten müssen initial einige Plugins installiert und Konfigurationen vorgenommen werden. Diese werden in folgender Auflistung dargestellt:

- Java JDK,
- Android Software Development Kit (SDK),
- Gradle Plugin,
- BitBucket Plugin,
- Android Emulator Plugin,
- Matrix Project Plugin und
- Pipeline Plugin.

Für die Installation von Jenkins sowie dem Bauen von Android Code wird das Java Development Kit benötigt. Jenkins wurde im Rahmen des AMCS Projektes schon für das Bauen und Testen des Backends genutzt. Jenkins läuft auf einer Ubuntu 14.04 LTS virtuellen Maschine (VM) innerhalb des Netzes der TU Dresden. Für das Bauen und Ausführen von Tests, welche das Android SDK benötigen, muss innerhalb der VM das SDK installiert werden. Der Pfad zu dem SDK muss in den Umgebungsvariablen hinzugefügt werden, wodurch Jenkins auf das SDK zugreifen kann. Für das Bauen von Android Projekten wird das Gradle Buildwerkzeug genutzt, welches durch ein Plugin zu Jenkins hinzugefügt werden kann [55]. Mit Hilfe von Gradle ist es möglich, über verschiedene Parameter das Projekt zu bauen oder die Testausführung zu starten. Ein Beispiel zum Löschen des aktuellen Baustandes, Bauen und anschließend der Testausführung in einem bestimmten Package sieht folgendermaßen aus:

```
1 ./gradlew clean assembleDebug :amcs:testDebugUnitTest --tests 'de.tudd.
    amcs.*'
```

Über den Aufruf `./gradlew` wird das Bauwerkzeug Gradle auf dem lokalen Computer initialisiert und gestartet. Der Parameter `clean` wird zum Löschen der aktuellen Dateien, welche beim Bauen benutzt werden, verwendet und `assembleDebug` startet das Bauen der Debug Version der Anwendung. Mit `:amcs:testDebugUnitTest – 'de.tudd.amcs.*'` werden alle definierten Unit Tests inner- und unterhalb des `de.tudd.amcs` Java Package im AMCS Modul ausgeführt. Das Android Projekt besteht aus zwei Modulen, zum einen das „amcs“ Modul in dem die normale Android Applikation entwickelt wird und zum anderen ein „profwatch“ Modul. Das „profwatch“ Modul beinhaltet die Android Wear Anwendung und hat im Rahmen dieser Arbeit keine Verwendung.
Um die Arbeit mit Jenkins und BitBucket zu vereinfachen, wird das BitBucket Plugin genutzt. [56] Beispiele dazu sind eine einfachere Verwendung der BitBucket Git Versionskontrolle und es gibt außerdem die Möglichkeit, Jenkins und BitBucket über Web-Hooks miteinander zu verbinden. Die Hooks können so konfiguriert werden, dass mit jedem Push in BitBucket ein neues Bauen und Testen des Projektes ausgelöst werden kann. Leider ist der Jenkins Server nur über das interne TU Dresden Netzwerk erreichbar, so dass die Push-Nachrichten von BitBucket nicht an Jenkins weitergereicht werden können. Stattdessen wird Jenkins so konfiguriert, dass alle 10 Minuten auf neue Commit Meldungen in BitBucket überprüft wird. Wenn Änderungen vorhanden sind, wird ein neuer Bau- und Testprozess gestartet.
Das Android Emulator Plugin wird benötigt, um innerhalb von Jenkins verschiedene Android Emulatoren starten zu lassen. [57] Die Fragmentierung der Android Versionen ist groß, daher ist es sinnvoll, die Android App auf verschiedenen Android Versionen zu testen. Dabei kommt auch das Matrix Project Plugin zum Einsatz. Es können verschiedene Jobs in Form

einer Matrix erstellt werden. Wie schon im Konzept Abschnitt Kontinuierliche Integration in der Tabelle 4.1 Testmatrix für AMCS definiert, kommen verschiedene Android Versionen mit unterschiedlichen Auflösungen zum Einsatz. Diese werden für Integrationstests und UI-Tests benötigt. Es werden Emulatoren mit der ältesten unterstützten, die laut Play Developer Console meist genutzte und die neuste Android Version mit verschiedenen Displayauflösungen erstellt und zum Testen der Nutzeroberfläche verwendet.

dpi	Android 21	Android 23	Android 26
160	android_21_160	android_23_160	android_26_160
240	android_21_240	android_23_240	android_26_240
320	android_21_320	android_23_320	android_26_320

Tabelle 5.2.: Jenkins Testmatrix der Android Emulatoren

Bei den Pipeline Plugins kommt das Pipeline:Multibranch Plugin zum Einsatz. [58] Mit Hilfe des Plugins ist es möglich, mit einer initialen Konfiguration alle Zweige des Projektes automatisch zu bauen und entsprechend zu testen, egal wann diese erstellt oder hinzugefügt wurden.

JENKINSFILE

Dazu wird eine Konfigurationsdatei mit in das Projekt und in die Versionskontrolle gelegt. Diese Datei muss als „Jenkinsfile“ benannt werden, damit Jenkins die Konfigurationen umsetzen kann. In der Jenkinsfile Datei wird eine Pipeline definiert, welche ausgeführt werden soll. Eine Pipeline besteht aus verschiedenen „stages“ und diese bestehen aus einzelnen Ausführungsschritten.

```
1  pipeline {
2    agent none
3    stages {
4      stage('Checkout') {
5        agent any
6        steps {
7          checkout scm
8        }
9      }
10
11     stage('Build') {
12       agent any
13       steps {
14         //clean it and build it
15         sh "./gradlew :amcs:clean :amcs:assembleDebug"
16       }
17     }
```

Quelltext 5.5: Jenkinsfile - Pipeline Checkout und Build Schritt

Die ersten zwei Schritte („steps“) dienen dazu, die aktuellen Änderungen aus der Versionsverwaltung zu laden, Baudateien zu löschen und den aktuellen Stand zu bauen.

```
18        stage('All Unit Tests') {
19          agent any
20          when {
21            expression {BRANCH_NAME ==~ /(develop|master|release|hotfix)/}
22          }
23          steps {
24            //run all unit tests
25            sh "./gradlew :amcs:testDebugUnitTest"
26
27            //publish test results in JUnit xml format
28            publishUnitTestResults()
29          }
30        }
```

Quelltext 5.6: Jenkinsfile - Pipeline Ausführung aller Unit Tests

Der nächste Schritt unterscheidet, ob der aktuelle Zweig einer der vier Hauptzweige ist und führt entsprechend alle Unit Tests aus, welche im Projekt definiert sind. Nach der Testausführung werden die Resultate aufbereitet und in der Jenkins Oberfläche dargestellt bzw. können darüber abgerufen werden.

```
31        stage ('Unit Tests on feature'){
32          agent any
33          when {
34            expression {BRANCH_NAME =~ "feature/(.*)"}
35          }
36          steps {
37            //run unit tests of feature branch
38            sh "./gradlew :amcs:testDebugUnitTest --tests 'de.tudd.amcs.${
                  removeFeature(env.BRANCH_NAME)}.*'"
39
40            publishUnitTestResults()
41          }
42        }
```

Quelltext 5.7: Jenkinsfile - Unit Tests des aktuellen feature/...-Zweiges

Wenn der Prozess der kontinuierlichen Integration innerhalb eines `feature/...`-Zweiges gestartet wird, dann werden die Unit Tests innerhalb des Java Package ausgeführt. Dabei wird der „feature/“ Teil des Zweig Namens entfernt. Es werden die Tests innerhalb des `de.tudd.amcs.X` Packages ausgeführt, wobei X der Name des Zweiges ohne „feature/“ entspricht. Im Anschluss werden die Testergebnisse aufbereitet und können in Jenkins abgerufen werden.

```
43        stage('UI and Integration Tests'){
44          agent any
45          when {
46            expression {BRANCH_NAME ==~ /(develop|master|release|hotfix)/}
47          }
48          steps {
49            build 'Android UI Integration'
50          }
51        }
52      }
53    }
```

Quelltext 5.8: Jenkinsfile - UI und Integrations Tests

Im Beispiel 5.8 werden die UI und Integrationstests ausgeführt, wenn der aktuelle Zweig einer der vier Hauptzweige ist. Die Ausführung geschieht in einem separaten Jenkins Jobs. Innerhalb des Jobs kümmert sich das Android Emulator Plugin um das Starten und Ausführen der Android Emulatoren. Per Shell-Befehl werden alle Tests ausgeführt, die die Android Umgebung benötigen. In diesem Fall sind dies die UI und Integrationstests.

```
56    //publish test results in JUnit xml format
57    def archiveUnitTestResults(){
58      step([$class: "JUnitResultArchiver", testResults: "amcs/build/test-
            results/testDebugUnitTest/*.xml"])
59    }
60    //remove the "feature/" from feature/...-branch name
61    def removeFeature(branchName) {
62      def matcher = (env.BRANCH_NAME =~ "feature/(.*)")
63      assert matcher.matches()
64      matcher[0][1]
65  }
```

Quelltext 5.9: Jenkinsfile - Hilfsmethoden

Zum Schluss folgt die Definition der zwei Hilfsmethoden. Die erste dient dazu, die generierten Testergebnisse aufzubereiten. Dadurch können die Ergebnisse innerhalb der Jenkins Oberfläche betrachtet werden. Die zweite Funktion hat die Aufgabe, den „feature/“ Teil der `feature/...`-Zweige zu entfernen. Dies geschieht über einen regulären Ausdruck, wobei der Zweigname anhand von „feature/“ gespalten wird und nur der letzte Teil zurück gegeben wird.
Die Jenkinsfile Datei wurde aus Gründen der Übersichtlichkeit gekürzt, die gesamte Datei kann innerhalb von Bitbucket betrachtet werden.

5.5.2. ABLAUF

Der Ablauf in Jenkins sieht nach der initialen Konfiguration der Plugins und anderen Einstellungen wie nachfolgend dargestellt aus. Die Testausführung sieht je nach Zweig anders aus, das Laden der Änderungen und Bauen des Projektes erfolgt in jedem Zweig gleich.

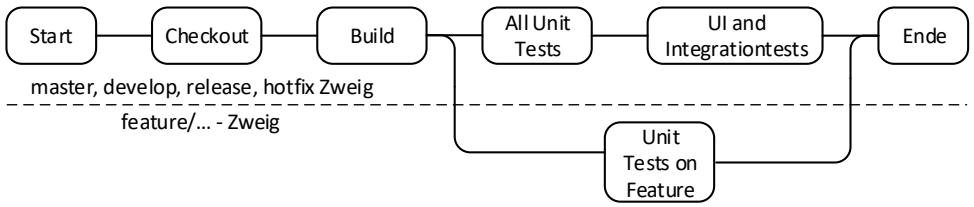


Abbildung 5.3.: Testausführungsablauf in Jenkins

Die automatische Überprüfung auf Änderungen alle 10 Minuten startet den Prozess des Bauens und Testens, wenn Änderungen vorhanden sind. Je nachdem, in welchem Zweig neuer Code hinzugefügt worden ist, werden verschiedene Schritte ausgeführt. Die Quell- und Testcode Änderungen treten in den verschiedenen `feature/...`-Zweigen auf. Die drei Hauptzweige dienen als Integrationszweige des aktuellen Standes sowie einen Zweig für Fehlerbehebungen in veröffentlichen Versionen. Wenn Änderungen in den Hauptzweigen auftreten, soll die komplette Testsuite mit Unit, Integrations und UI Tests ausgeführt werden.

Bei den `feature/...`-Zweigen treten Änderungen viel häufiger auf und deshalb sollen nur die Unit Tests des dazugehörigen Java Package ausgeführt werden.

5.6. ZUSAMMENFASSUNG

Dieses Kapitel untersucht die prototypische Implementierung des Konzeptes anhand des Android AMCS Projektes. Zunächst wurden die Änderungen der Architektur und die Implementierung des Model View Presenter Musters und Repository Musters umgesetzt. Anhand des Beispiels der Anmeldung eines Nutzers wurde die Umsetzung veranschaulicht. Der Ansatz des „Test Driven Developments“ wurde für die Umsetzung der Testautomatisierung genutzt und es wurde auf die verschiedenen Testmöglichkeiten des MVP Musters, der Nutzeroberfläche und Integration von verschiedenen Komponenten der Anwendung eingegangen. Des Weiteren wurde der zum Einsatz gekommene Git Workflow Ansatz beschrieben sowie die kontinuierliche Integration mit Hilfe von Jenkins. Dabei wurde eine Reihe von Plugins verwendet, um die Möglichkeiten von Jenkins zu erweitern. Dadurch ist es möglich, die Testausführung anhand der Struktur der Zweige in der Versionskontrolle zu beeinflussen. Während der Entwicklung neuer Funktionen sollen nur die schnellen Unit Tests zur Überpürfung des Codes dienen. Oberflächen- und Integrationstests dauern meist deutlich länger und werden daher nur bei der Verschmelzung von `feature`-Zweigen mit dem `release`-Zweig oder vor der Veröffentlichung ausgeführt.

6. EVALUATION

Das folgende Kapitel evaluiert die Ergebnisse dieser Arbeit. Zunächst wurde die problematische Ausgangssituation der Jenkins Umgebung untersucht. Anschließend wurden die Vor- und Nachteile der testgetriebenen Entwicklung genauer betrachtet. Im dritten Abschnitt wurde die Befragung von 11 Teilnehmern ausgewertet und die Ergebnisse untersucht. Die Testabdeckung gibt Aufschluss für die Qualitätssicherung einer Software, diese wurde anhand der Unit Tests im vierten Abschnitt beschrieben. Zuletzt wurde die Performanz der neuen und alten AMCS App verglichen und die jeweiligen Unterschiede und Ergebnisse aufgeführt.

6.1. JENKINS

Im Rahmen des AMCS Projektes ist bereits eine Jenkins Instanz innerhalb des TU Dresden Netzwerkes vorhanden. Zunächst sollte das bereits vorhandene Jenkins für die kontinuierliche Integrationsumgebung genutzt und erweitert werden. Dazu mussten das Android SDK, Gradle und verschiedene Plugins installiert werden. Die Testausführung erfolgt innerhalb von sogenannten Jobs. Davon ist einer für die Ausführung der UI Tests zuständig. Für die unterschiedliche Darstellung und Versionen von Android wurde eine Testmatrix erstellt. Drei Android Versionen wurden ausgewählt, das heißt es wurden ebenfalls drei Android Versionsabbilddateien benötigt. Die Abbilddateien werden vom Emulator als Grundlage genutzt, um eine Android Umgebung zu erstellen. Die Abbilddateien werden an einem zentralen Ort des Android SDK Pfades hinterlegt, um nicht für jeden Emulator eine Datei zu benötigen. Die Dateien haben auf einem Ubuntu System eine Größe von: 12,8 GB. Da sich aus der Testmatrix insgesamt neun Emulatoren ergeben und diese separat erstellt werden, ergibt sich daraus eine Größe von 34,5 GB. Außerdem wird für die Ausführung eines Elementes der Testmatrix ein einzelner Ordner mit dem Projekt erstellt. Dadurch wird ebenfalls das Android AMCS Projekt neun Mal erstellt und gebaut, wodurch noch zusätzlicher Speicherplatz benötigt wird. Der komplette Projektordner mit allen Dateien, die beim Bauen der Anwendung erstellt werden, benötigt ca. 200 MB an Speicherplatz. Dies ergibt einen benötigten Speicherplatz von 49,1 GB.
Zusätzlich wird ein Job für die Ausführung der Unit Tests benötigt, hierbei werden jeweils ca. 200 MB Speicherplatz pro Zweig verwendet. Es gibt drei feste Hauptzweige, `master`, `release`, `develop` sowie einen eventuellen `hotfix`-Zweig. Außerdem eine variable Anzahl von `feature`-Zweigen. Insgesamt ergibt dies einen ungefähren Speicherplatz von **49,9 GB** plus 200 MB je nach Anzahl der aktuellen `feature`-Zweigen.

Job Name	Beschreibung	Größe
Android UI	System Abbilddateien	12,8 GB
	Android Emulatoren	3,83 GB * 9 = 34,5 GB
	Projektordner	ca. 200MB * 9 = 1,8 GB
Android	`master-`, `release-`, `develop-` und `hotfix`-Zweige	ca. 200 MB * 4 = 800 MB
	`feature`-Zweige	ca. 200 MB * X
Insgesamt		49,9 GB + X

Tabelle 6.1.: Speicherplatzbedarf der Testumgebung innerhalb Jenkins unter Ubuntu 14.04

Jenkins läuft innerhalb einer virtuellen Maschine, welche auf Ubuntu 14.04 basiert. Initial war ein freier Speicherplatz von 15 GB gegeben, welcher ohne größeren Aufwand um 15 GB auf 30 GB erweitert werden konnte. Wie oben aufgeführt, reicht dies nicht für die Umsetzung des Testkonzepts aus.
Des Weiteren konnte die Hardwarevirtualisierung der Android Emulatoren nicht genutzt werden. Jenkins läuft bereits innerhalb einer virtuellen Maschine, deshalb konnte unter Linux das „kvm“ (Englisch für „Kernel-based Virtual Maschine“) nicht genutzt werden. Dadurch verlangsamt sich die Ausführung der UI Tests und Tests, welche die Android Umgebung benötigen. Weiterhin benötigt das Starten der Android Emulatoren länger.
Ein weiteres Problem ist bei den physischen Laufwerken des Hostsystems vorhanden, wodurch die Ausführung initial sehr langsam war. Die Antwortzeiten waren sehr hoch und zum Teil wurden mehrere Timeouts (30s) ausgelöst. Dies trat vor allem auf, wenn das System eine Weile nicht genutzt wurde. Jenkins ist so eingestellt, dass alle zehn Minuten auf Änderungen in der Versionskontrolle geprüft wird und bei Änderungen der Bau- und Testprozess gestartet wird. Während der Erstellung der Umgebung wurde ein Zwischenstand genutzt, wobei das Android Projekt nur gebaut und alle vorhandenen Unit Tests ausgeführt wurden. Je nach Zustand der virtuellen Maschine ergaben sich unterschiedliche Ausführungszeiten der Testumgebung. Zum einen wurde der Prozess innerhalb von 30 Sekunden durchlaufen, wenn das System vorher aktiv war. Zum anderen wurden 14 Minuten benötigt, um den Prozess einmal zu durchlaufen, wenn das System eine längere Zeit nicht benutzt wurde. Die Zeit konnte verkürzt werden, indem der Entwickler vor dem Hochladen der aktuellen Änderungen in die Versionskontrolle auf Jenkins zugegriffen hat. Diese Vorgehensweise ist aber kein praktikabler Einsatz, da die kontinuierliche Integration im Hintergrund laufen soll und nicht dauerhaft die Aufmerksamkeit der Entwickler beanspruchen soll.
Letztendlich wurde aus den genannten Zeit- und Platzproblemen Jenkins erneut auf einem Laptop mit Ubuntu 16.04 und ausreichend Speicherplatz aufgesetzt. Dadurch ließ sich die Umsetzung des Konzeptes realisieren. Ein Nachteil ist, dass Jenkins nur innerhalb des lokalen Netzwerks erreichbar ist und von außerhalb nicht mehr genutzt werden kann, da eine öffentliche IP-Adresse nicht gegeben ist.

6.2. TEST DRIVEN DEVELOPMENT

In diesem Abschnitt werden Aussagen zur Nutzung der testgetriebenen Entwicklung getroffen. Normalerweise wird die Rentabilität (engl. „Return on Investment“) des Ansatzes betrachtet, da die AMCS App allerdings kostenlos, ohne Werbung oder sonstige Einnahmequellen zur Verfügung steht, können hier keine Aussagen abgeleitet werden. Weiterhin ist es innerhalb der zeitlichen Begrenzung dieser Arbeit nicht möglich, die Vermeidung von Fehlern oder Abstürzen zu untersuchen, da die umgeschriebene bzw. teilweise neu entwickelte Applikation nicht offiziell heruntergeladen werden kann. Daher können keine Aussagen zur

Nutzerzufriedenheit getroffen werden.
Nichtsdestotrotz können die Vorteile und Nachteile des Ansatzes ausgewertet werden. Die meist positiven Aussagen aus mehreren Blog Einträgen werden im nächsten Abschnitt zusammengefasst. [59, 60, 61]
Durch die Nutzung der testgetriebenen Entwicklung wird der Code strukturierter, da sich schon bei der Testerstellung der grundlegende Aufbau der Applikation definiert wird. Regressionstests werden seit Beginn der Entwicklung ausgeführt und decken Fehler bei Änderungen am bisherigen Code auf. Dadurch ist weiterhin die Refaktorisierung sicherer, da die vorher implementierten Testfälle die Funktionalität überprüfen. Bei Unterstützung durch ein kontinuierliches Integrationssystem können früh Fehler entdeckt werden. Der Ansatz bietet eine stabile Grundlage für das Projekt und eine spätere Weiterentwicklung. Durch die zusätzliche Verwendung von Mocking Frameworks wird die Entwicklung erleichtert, da eventuelle Abhängigkeiten abgedeckt werden können und der Fokus auf dem entsprechenden Testfall liegt. Die Tests können ebenfalls der Dokumentation dienen, da Fälle abgebildet werden, welche in der reinen Implementierung nicht gleich verständlich sind.
Die zusammengefassten Vorteile decken sich mit den eigenen Vorteilen, welche bei der Erstellung dieser Arbeit aufgetreten sind. Trotz der vielen Vorteile traten innerhalb dieser Arbeit außerdem einige Nachteile bei der testgetriebenen Entwicklung auf. Die Umsetzung des Ansatzes ist vor allem schwierig, wenn die Ziele der Umsetzung oder Architektur Muster, wie z.B. das Model View Presenter Muster nicht bekannt sind. Dies führt schnell dazu, eine Reihe von Tests zu erstellen, welche zunächst nutzlos sind und später geändert werden müssen. Dadurch wird nur der Entwicklungsaufwand erhöht und bietet keine Vorteile. Weiterhin ist der Aufwand bei einer Refaktorisierung der Anwendung höher, welche Änderungen an der Funktionalität betrifft. In diesem Fall muss sowohl der Quellcode als auch der Testcode angepasst werden.

6.3. BEFRAGUNG

Im Rahmen dieser Arbeit wurde eine Befragung durchgeführt. Ziel war eine Ableitung zur Einschätzung des Konzeptes im Kapitel 4. Weiterhin sollten mögliche Schwachstellen und Weiterentwicklungen aufgezeigt werden. Dies wurde mit einem passiven Ansatz mit Hilfe eines Fragebogens durchgeführt. Es wurden Fragen bezüglich des Testverhaltens, des Einsatzes von Versionskontrollsystemen und kontinuierlichen Integrationswerkzeugen gestellt. Die Antworten stammen teilweise von Mitarbeitern innerhalb des AMCS Projektes sowie Kommilitonen, die teilweise in der Wirtschaft tätig sind. Die Befragung wurde Online mit Hilfe des AMCS Tools durchgeführt. Es wurden unterschiedliche Fragetypen verwendet, so kann es vorkommen, dass mehr Antworten als Teilnehmende vorhanden sind. Der für die Befragung entwickelte Fragebogen mit teilweise vorgegebenen Antworten ist dem Anhang beigefügt. Es bestand bei einigen Fragen die Möglichkeit, Antworten in Form von Text einzugeben.
Insgesamt haben 11 Teilnehmende an der Befragung teilgenommen. Die ersten Fragen beziehen sich auf den generellen Einsatz von Tests und deren Umgang in Softwareprojekten. Später folgen Fragen zum Einsatz des Git Workflows, zur testgetriebenen Entwicklung, sowie zur kontinuierlichen Integration von Tests.
Der Großteil der Befragten, genauer gesagt zehn von elf, haben Softwaretests schon einmal verwendet. Die häufigsten Tests, welche dabei zum Einsatz kamen, sind Unit- und Integrationstests. Etwa die Hälfte haben Akzeptanztests und ca. ein Drittel haben Systemtests genutzt. Außerdem wurden Modul- und Regressionstests erwähnt. Die Ergebnisse spiegeln sich in den Testarten wieder, dabei haben ca. 91 % der Befragten funktionale Tests zur Überprüfung von Software eingesetzt. Komplexere Testarten, wie Performanz- oder Stresstests kamen deutlich weniger zum Einsatz, nur 3 von 11 Teilnehmende habe diese genutzt.

Die Erstellung von Testdefinitionen wurde bei den meisten Teilnehmenden durch User Stories (ca. 55 %) und aus Tickets (ca. 36 %) erarbeitet. Außerdem wurden Testtabellen und Akzeptanzkriterien bei Tests am Nutzer verwendet. Der Zeitpunkt der Testerstellung zeigt ein unterschiedliches Ergebnis. Am häufigsten werden Tests während der Codeerstellung geschrieben, gefolgt von der Codeerstellung. Zuletzt werden Tests vor der Codeerstellung implementiert. Daraus lässt sich ableiten, dass die testgetriebene Entwicklung seltener zum Einsatz kommt.
Der Zeitpunkt, wann die Tests ausgeführt werden, ist sehr unterschiedlich. Am häufigsten wurde geantwortet, dass Tests nach einer Integration von Zweigen per Pull-Request oder Merge ausgeführt werden. Vier Antworten der an der Befragung Teilnehmenden sagten aus, dass es keinen festen Plan für die Ausführung gibt und die Testausführung daher im Ermessen des Entwicklers liegt. Drei Antworten zeigen, dass eine zeitliche Ausführung, wie z.B. mittags oder abends, genutzt wird. Außerdem gibt es drei Antworten, wo Tests vor bzw. nach einem Push ausgeführt werden. Die knappe Mehrzahl (ca. 55 %) hat angegeben, dass Komponenten unterschiedlich getestet werden, weil sowohl die Komponenten unterschiedlich sein können als auch einige Teile einer Software häufiger genutzt werden. Ein Teilnehmer antwortete, dass Tests für den Hauptanwendungsfall weniger wichtig sind, wenn die Tests z.B. andere Betriebssysteme umfassen. Die anderen 45 % haben sich für gleichmäßiges Testen entschieden.
Die Durchführung der Tests spielt ebenfalls eine wichtige Rolle. Hier gab es sieben Antworten für die Ausführung von Tests per Hand, also über eine Konsole oder integriert in eine Entwickleroberfläche. Acht Antworten gab es für den Einsatz einer kontinuierlichen Integrationsumgebung. Außerdem gab es eine besondere Form der Testautomatisierung mit Hilfe von einem Bot, welcher Patch-E-Mails auf ein lokales Repository anwendet und Fehlerausgaben über den Bot ausgibt. Dies ist eine besondere Form, da auf ein lokales Repository zurückgegriffen wird. Der Unterschied liegt darin, dass bei der kontinuierlichen Integration ein zentrales Repository für die Testausführung genutzt wird. Auf die Testergebnisse wird meistens über die während der Testausführung generierten HTML- oder XML-Seiten zugegriffen. Weiterhin gibt es jeweils zwei Antworten zum Auslesen der Ergebnisse über die Konsole bzw. Log Dateien vom Buildsystem sowie die Benachrichtigung per E-Mail. Für die Testauswertung kommt ein IRC-Bot zum Einsatz.
Fünf Teilnehmer der Befragung gaben an, eine Form der automatischen Testausführung zu nutzen. Dabei kommen Jenkins oder Tools innerhalb des Microsoft Team Foundation Servers zum Einsatz. Die anderen sechs Teilnehmenden nutzen keine Form der Automatisierung. Bei Betrachtung der Testmaße wird vor allem auf die Anforderungsabdeckung (sechs Antworten) geachtet, drei Antworten gab es auf die Codeabdeckung und weiterhin werden Regressionstests bei der Beseitigung von Fehlern genutzt. Drei Antworten gab es bei keiner Berücksichtigung der Testmaße.
Bei der nächsten Frage handelte es sich, um die Abbildung von Tests auf den Zustand des Systems. Der überwiegende Teil antwortete, dass Tests ein guter Indikator für den Zustand des Systems sind, aber bei weitem nicht alles abbilden können. Zehn Antworten der Teilnehmenden liegen zwischen 50 und 80 %, wobei 100 % den Wert „Tests bilden den Zustand komplett ab“ darstellen. Der Hauptteil der Antworten liegt bei 60 %. Es gibt allerdings eine Antwort für nur 20 %, d.h. die Tests haben nur einen sehr geringen Einfluss auf den Zustand der Software.
Die folgenden Fragen bezogen sich auf das Verhalten im Umgang mit Versionskontrollsoftware, wie z.B. git. Als Erstes ging es um das Commit Verhalten. Der Hauptteil der Befragten entschied sich für die Commits so sehr modular bis so modular wie möglich auszuführen. Es gab nur eine Antwort, welche so selten wie möglich geantwortet hat.
Bei der Verwendung von Push Befehlen ist die Nutzung weitaus zurückhaltender. Die Tendenz geht in Richtung seltenere Ausführung von Push Befehlen anstatt diese so modular wie

möglich durchzuführen.
Die Befragten waren geteilter Meinung Commits auszuführen, bevor die Unit Tests bestanden wurden. Sechs Teilnehmende haben mit „nein“ geantwortet, weil fehlerhafter Code nicht committed werden sollte, da das Feature noch nicht lauffähig ist. Die anderen fünf Teilnehmende haben kein Problem Commits mit fehlschlagenden Unit Tests durchzuführen, da lokale Fehlschläge nicht so negativ, wie Fehlschläge auf dem Buildserver sind. Weiterhin kann der Zwischenstand von Code gespeichert werden, ohne den entsprechenden Testcode anzupassen. Es kann zeitlich nicht immer möglich sein, die Tests laufen zu lassen und außerdem besteht die Möglichkeit Commits im Nachhinein lokal anzupassen. Bei der Frage nach der Ausführung von Push Befehlen bevor Unit Tests bestanden wurden, sind die Antworten einheitlicher. Der Großteil der Befragten (ca. 82 %) ist gegen die Ausführung eines Pushes, wenn ein Unit Test fehlschlägt. Dies wird damit begründet, dass nur valide, ausführbare Zustände und vor allem fehlerfreier Code ins zentrale Repository gelangen sollte. Die anderen ca. 18 % geben an, dass dadurch Arbeitsstände gesichert werden können, wenn z.B. die Arbeitsumgebung gewechselt werden muss.
Die folgenden drei Fragen beziehen sich auf den Einsatz des Git Flow Ansatzes. Dieser kam bei etwas mehr als der Hälfte (6 von 11 Teilnehmenden) schon einmal zum Einsatz. Demzufolge haben fünf Befragte den Ansatz noch nicht benutzt. Nach den Vor- und Nachteilen des Ansatzes wurde ebenfalls gefragt. Die Teilnehmenden sehen die Vorteile darin, dass Anforderungen unabhängig voneinander entwickelt werden können. Außerdem ist ein sinnvolles Modell für verschiedene Nutzung von Zweigen gegeben und die Aufgaben der Zweige ist klar definiert. Weiterhin wird eine strukturierte Arbeitsweise gefördert und gibt einen Überblick über die Abarbeitung von Anforderungen. Nachteilig wurde gesehen, dass es bei Abhängigkeiten unter den Anforderungen zu Problemen kommen kann. Dadurch wird ebenfalls das Risiko von Merge Konflikten erhöht. Des Weiteren wird die Komplexität für kleinere Teams als zu hoch eingestuft. Bei der Frage, ob die Vor- oder Nachteile überwiegen, wurde angegeben, dass die Vorteile des Ansatzes überwiegen. Dabei wurde die Möglichkeit in größeren Gruppen unabhängig voneinander Code zu entwickeln und das Vorhanden sein eines so gut wie immer funktionierenden Zweiges hervorgehoben. Drei Teilnehmende meinten, dass die Nachteile überwiegen, haben aber keine aussagekräftige Begründung dafür abgegeben.
Die nächsten drei Fragen bezogen sich auf den Einsatz der testgetriebenen Entwicklung von Software. Die ersten zwei Fragen wurde nur von zehn statt elf Teilnehmenden beantwortet. Die Frage, ob die testgetriebene Entwicklung schon einmal verwendet wurde, ist ausgeglichen, fünf der Befragten haben den Ansatz bereits genutzt und die anderen fünf noch nicht. Bei der Frage, ob durch den Ansatz Fehler vermieden werden haben acht Teilnehmende mit „Ja“ geantwortet. Als Begründung wurde angegeben, dass von Beginn der Entwicklung an die Anforderungen überprüft werden. Die Testabdeckung wird erhöht und im Allgemeinen wird öfter getestet. Weiterhin geht man bei der Implementierung von Funktionalität innerhalb des Codes strukturierter vor. Die Grenzen des Systems werden direkt beim Programmieren deutlich. Zwei Teilnehmende sagten aus, dass dadurch nicht mehr Fehler vermieden werden. Eine Begründung war, dass durch den Ansatz bei der Codeerstellung zu sehr auf die Erfüllung der vorher geschriebenen Tests geachtet wird, anstatt auf die Anforderungen zu achten. Außerdem können die Tests nicht alle Anwendungsfälle abdecken. Statt der testgetriebenen Entwicklung wurde eine andere Vorgehensweise beschrieben, bei der Code und Tests unabhängig voneinander geschrieben werden. Anschließend werden die Tests ausgeführt und entsprechende Anpassungen durchgeführt. Dadurch erhält man einen allgemeingültigen Code, welcher nicht spezifisch für die Erfüllung eines Tests geschrieben wurde. Bei der Frage, ob für die testgetriebene Entwicklung der Mehraufwand gerechtfertigt ist, haben wieder elf Teilnehmende geantwortet. Die knappe Mehrheit der Befragten meinte TDD ist den Mehraufwand wert, weil ein gutes Bild vom System vermittelt wird und die Tests zeigen, wie das System funktionieren soll. Die Erstellung von Tests vor oder nach der Implementierung hat

einen ähnlich hohen Aufwand. Tests gar nicht zu nutzen, wurde als nachteilig gewertet. Von Vorteil ist, dass theoretisch jeder Code durch einen Test abgedeckt wird. Es wird Nacharbeit vermieden und man gewinnt zeitig ein Feedback, ob die Implementierung anhand der Anforderungen korrekt ist. Fünf Teilnehmende meinten es ist den Mehraufwand nicht wert und gaben an, dass der Einsatz von TDD von der Projektgröße abhängig ist. In einigen Fällen kommt dafür „Rapid Prototyping" zum Einsatz. Es müssen oft Funktionen unabhängig von Tests implementiert werden, wodurch keine Zeit zur Erstellung der Tests vorhanden ist. Die Verwendung von Rapid Prototyping ist kein direkter Nachteil des TDD Ansatzes, es zeigt eher, dass in der Wirtschaft Applikationen schnell entwickelt werden müssen.Der höhere Zeitaufwand für die Verwendung von TDD wird als nicht gerechtfertigt gesehen.
Die letzten drei Fragen beziehen sich auf die Arbeit mit kontinuierlichen Integrationsumgebungen. Der Großteil der Befragten hat allerdings noch nicht mit kontinuierlichen Integrationswerkzeugen gearbeitet. Nur drei Teilnehmende haben bereits mit Travis CI oder Microsoft Team Foundation Server gearbeitet. Die nachfolgende Frage bezieht sich auf die Granularität der Testausführung anhand verschiedener Zweige. Auf diese Frage haben zehn Teilnehmende geantwortet. Außerdem lässt sich die Granularität schwer bestimmen, deshalb gab es die Möglichkeit, die Konfiguration durch Texteingabe genauer zu beschreiben. Auch wenn nur drei Teilnehmende mit kontinuierlichen Integrationswerkzeugen gearbeitet haben, zeigen die Antworten der Frage über die Granularität ein generelles Verständnis für die kontinuierliche Integration. Zwei von zehn Befragten haben geantwortet, dass alle Tests auf jedem Zweig ausgeführt werden sollen. Die anderen Antworten bewegen sich zwischen gemischtem Testen und dem Ausführen von allen Tests auf den Hauptzweigen und den spezifischen Unit Tests auf den jeweiligen `feature`-Zweigen. Anhand der Textantworten lässt sich aber ein einheitlicheres Bild ableiten. Die stabilen Hauptzweige führen alle Tests aus und die anderen Zweige werden vom jeweiligen Verantwortlichen getestet, wo es im Ermessen des Verantwortlichen liegt, welche Tests ausgeführt werden sollen. Eine andere Antwort war, dass Systemtests auf dem `master`-Zweig und Unit Tests auf dem jeweiligen Zweig ausgeführt werden sollen. Weiterhin sollen alle Tests in den Integrationszweigen, was den Hauptzweigen entspricht, ausgeführt werden. Auf den `feature`-Zweigen werden nicht alle Tests ausgeführt. Eine andere Antwort sagt aus, dass nur der Haupt- und der aktuelle `feature`-Zweig getestet werden soll. Im Großen und Ganzen kann man die Antworten so vereinheitlichen, dass der Hauptteil der Tests auf den Hauptzweigen ausgeführt werden soll. Auf den aktuellen `feature`-Zweigen sollen nur Tests für die Funktionalität der Anforderung und weitere wichtige Überprüfungen durchgeführt werden.
Die letzte Frage bezog sich auf den Zeitpunkt der Ausführung von fünf unterschiedlichen Tests. **Unit Tests** sollten nach jeder Veränderung des Codes und während der Entwicklung ausgeführt werden. Weiterhin sollten vor jedem Push in die Versionskontrolle und bei jedem Merge mit einem anderen Zweig die Unit Tests durchgeführt werden. Für die Ausführung von **Integrationstests** gab es jeweils drei Antworten bei jedem Merge sowie nach jeder Fertigstellung einer Anforderung, was einem Merge in einen Hauptzweig entspricht. Außerdem sagte eine Antwort aus, dass vor jedem Push in die Versionskontrolle Integrationstests ausgeführt werden sollten. **Akzeptanztests** sollen vor jeder Veröffentlichung, gegen Ende des Projektes und sollten die entsprechende Ressourcen und Zeit für die Durchführung zur Verfügung stehen, durchgeführt werden. **Stresstests** sollten vor allem vor einer Veröffentlichung zum Einsatz kommen. Weiterhin gab es drei Antworten, dass Stresstests bei relevanten Komponenten kontinuierlich verwendet werden sollten, um Veränderungen zu bemerken. Eine Antwort sagte aus, dass Stresstests bei einem fertigen Produktionssystem in der Evaluationsphase zum Einsatz kommen sollten. Zuletzt sollten **Systemtests** vor jeder Veröffentlichung verwendet werden oder kontinuierlich auf den Hauptzweigen sowie im fertigen Produktionssystem in der Evaluationsphase.

Zusammenfassend lässt sich feststellen, dass die Ansätze, wie Git Flow, die testgetriebene Entwicklung und der Einsatz von kontinuierlichen Werkzeugen nicht so verbreitet sind, wie erwartet wurde. Nichtsdestotrotz erkennt man aus der Beantwortung der Fragen, dass die Konzepte zum größten Teil bekannt und vertraut sind. Die Antworten zeigten außerdem, dass das Konzept, welches in dieser Arbeit vorgestellt wurde nur kleine Anpassungen und klarere Definitionen benötigt. Explizit muss erwähnt werden, dass die Unit Tests der einzelnen `feature`-Zweige vor einem Commit und vor allem vor einem Push in die Versionskontrolle lokal auf dem Entwicklercomputer durchgeführt werden sollte. Außerdem sollte dem Entwickler einer Anforderung die Freiheit überlassen werden andere wichtige Tests für den `feature`-Zweig ausführen zu können. Die nötigen Anpassungen können in die Jenkinsfile-Datei für den entsprechenden Zweig integriert werden, um eine Ausführung in der kontinuierlichen Integration zu ermöglichen. Es muss nur beim Löschen es Zweiges darauf geachtet werden, dass die Änderungen an der Jenkinsfile-Datei auch entfernt werden. Die Aussage, dass alle Tests in den Hauptzweigen durchgeführt werden sollten, stimmt mit den Antworten der Umfrage zum größten Teil überein.

6.4. METRIKEN

Die Testabdeckung ist ein wichtiger Indikator für die Qualitätssicherung einer Software. Mit Hilfe der Abdeckung erhält man einen guten Überblick zu den getesteten Methoden und Klassen der Anwendung. Des Weiteren lässt sich hiermit die Vollständigkeit von Software Tests im Bezug auf den Quellcode ermitteln.
In der prototypischen Umsetzung wurden neun Testklassen mit insgesamt 77 Testmethoden in Form von Unit Tests umgesetzt. Zusätzlich wurden zwei Klassen für Integrationstests des Logins und der Übersicht erstellt. Weiterhin wurden einige Teile der Oberfläche der Applikation durch vier UI Test Klassen überprüft. Speziell wurden die UI-Elemente der Anmeldeansicht, die Vorlesungsansicht und die Übersicht der Kurse und aktive Vorlesungen anhand verschiedener Methoden überprüft.
Die Applikation bzw. der Code ist zu einem großen Anteil abhängig vom Android Framework. Bei der Umsetzung der Unit Tests wurden alle Klassen getestet, die die Android Umgebung für die Ausführung nicht benötigen. Das Android Framework kann nicht einfach über Mocks durch das Mockito Framework abgedeckt werden. Ein möglicher Ansatz wäre zusätzlich „Robolectric“ zu nutzen. Dies ist ein Unit Test Framework, wodurch das Android SDK in Form eines Mocks zu Verfügung steht und somit innerhalb der lokalen Java Virtual Machine auf dem Entwicklergerät ausgeführt werden kann. [62]
Die nachfolgende Tabelle gibt einen Überblick über die Testabdeckung anhand der Klassen, Methoden und Zeilen in der AMCS App.

Art	Anzahl	Prozent
Klassen	39/148	26,35 %
Methoden	248/745	33,29 %
Zeilen	803/3203	25,07 %

Tabelle 6.2.: Testabdeckung Übersicht

In dem Projekt gibt es 12 Java Packages mit insgesamt 148 Klassen, wovon 39 durch Unit Tests abgedeckt werden. Das entspricht einer Abdeckung von 26,35 %. Bei der Testabdeckung der Methoden werden etwa 1/3 aller Methoden überprüft, 248 von 745. Die Abdeckung der Codezeilen ist etwas geringer und umfasst 803 von 3203 Zeilen, dementsprechend werden ca. 1/4 der Anwendungszeilen mit Hilfe der Unit Tests abgedeckt.

Zur Erhebung der Quell- und Testcodezeilen wurde das Plugin „Statistic“ verwendet, welches in Android Studio integriert werden kann. Die Zählweise der Zeilen ist anders aufgebaut als bei der Testabdeckung. Statistic zählt alle Annotationen, Imports, den Package Pfad und alle Klammern, aber keine Leer- oder Kommentarzeilen. Die Anzahl der Testzeilen und Codezeilen haben daher andere Werte als durch die Testabdeckung ermittelten abgedeckten Codezeilen. Die folgende Tabelle 6.3 gibt Aufschluss über Abdeckung durch die verschiedenen Testklassen.

Klasse	Codezeilen	Tests	Methoden	Zeilen
JsonParserTest	251	6	6/6 (100 %)	125/125 (100 %)
LectureViewPresenterTest	126	10	12/13 (92,31 %)	40/42 (95,24 %)
LoginPresenterTest	92	10	12/14 (85,71 %)	42/52 (80,77 %)
OverviewPresenterTest	142	12	16/18 (88,89 %)	41/43 (95,35 %)
CourseRepositoryTest	90	4	16/18 (88,89 %)	40/51 (78,43 %)
LectureRepositoryTest	72	3	18/21 (85,71 %)	44/49 (89,90 %)
QuestionRepositoryTest	153	6	27/31 (87,10 %)	105/161 (65,22 %)
UserRepositoryTest	46	1	6/15 (40,00 %)	17/145 (11,72 %)
SlideViewPresenterTest	322	25	30/32 (93,75 %)	132/147 (89,80 %)

Tabelle 6.3.: Testabdeckung der Unit Tests

In der nächsten Tabelle 6.4 sind die Codezeilen der abgedeckten Klassen und das Verhältnis zwischen Quellcode und Testcodezeilen dargestellt.

Klasse	Codezeilen	Verhältnis
JsonParser	156	160,90 %
LectureViewPresenter	83	151,81 %
LoginPresenter	91	101,10 %
OverviewPresenter	88	161,36 %
CourseRepository	88	102,27 %
LectureRepository	92	83,70 %
QuestionRepository	244	62,70 %
UserRepository	216	21,30 %
SlideViewPresenter	247	130,36 %

Tabelle 6.4.: Codezeilen und Verhätlnis zwischen Quellcode zu Testcodezeilen

Zunächst lässt sich erkennen, dass durch die `UserRepositoryTest` Klasse eine schlechte Abdeckung erreicht wird. Nur 11,72 % der Zeilen und 40 % der Methoden der `UserRepository` Klasse werden während der Tests ausgeführt, was deutlich unter den Werten der anderen Klassen liegt. Die Ursache dafür liegt vor allem darin, dass die `UserRepository` Klasse keine Abstraktion zwischen lokalen und Daten aus dem Backend besitzt. Dadurch sind in der Klasse weiterhin viele Abhängigkeiten zum Android Framework gegeben, wodurch eine höhere Abdeckung nicht erreicht werden kann.
Der zweite extreme Fall ist die 100 %-ige Abdeckung der `JsonParser` Klasse. Dies lässt sich dadurch erklären, dass die Klasse genutzt wird, um JSON Daten in lokalen Objekte umzuwandeln. Die Testdaten für die Umwandlung sind vollständig, d.h. es werden alle Methoden zum Setzen der Objekteigenschaften ausgeführt. Wodurch eine Abdeckung von 100 % erreicht wird. Die Vollständigkeit beruht auf der gegebenen Dokumentation der Backend Schnittstel-

le, welche unter dem Link https://amcs-staging.inf.tu-dresden.de/apidoc/ [1] aufrufbar ist.
Die Presenter Tests erreichen eine Abdeckung von ca. 90 % und mehr, haben aber einen hohen Mehraufwand an Testcodezeilen gegenüber den Quellcodezeilen. Im Durchschnitt liegt der Mehraufwand bei ca. 36 %, dadurch wird allerdings eine hohe Abdeckung erreicht. Der Grund für die höheren Testzeilenanzahl liegt unter anderem an verschiedenen Fehlerüberprüfungen. Über `Callbacks` vom jeweiligen Repository werden Daten oder Fehlercodes an den Presenter übergeben. Innerhalb des Presenters wird der Fehlercode in einer If-Abfrage überprüft und eine entsprechende Methode des Views aufgerufen. Für das Testen, muss jedes Mal die zu überprüfende Methode aufgerufen werden, ein `CallbackCaptor` initialisiert werden, um mit dessen Hilfe einen Fehler hervorzurufen. Anschließend kann verifiziert werden, ob der Presenter die entsprechende Methode des Views aufruft. Es müssen in den Testklassen zunächst Testdaten initialisiert werden, wodurch einige Zeilen zu den Testklassen hinzu kommen. In der `SlideViewPresenterTest` Klasse werden zum Beispiel 48 Zeilen benötigt, um Testobjekte zu initialisieren. Wenn man diese Anzahl an Zeilen vernachlässigen würde, wäre der Aufwand nur ca. 10 % höher.
Bei den restlichen Repository Testklassen ist die Abdeckung eher durchwachsen, sie liegen zwischen ca. 65 % und ca. 90 %. Dies liegt unter anderem an unvollständigen Fehlerüberprüfungen, welche durch Antworten des Backends ausgelöst werden können. Außerdem sind nicht alle Testanforderungen vollständig implementiert. Im speziellen ist die Methoden Abdeckung des `QuestionRepository` mit 87 % zwar hoch, aber die Zeilenabdeckung in Höhe von ca. 65 % ist noch ausbaufähig. Des Weiteren ist die Umsetzung der neuen Version nicht abgeschlossen und vor allem in der `SlideView`, welche für die Darstellung der Fragen zuständig ist, fehlen einige Implementierungen. Dies führt dazu, dass auch die `QuestionRepository` Klasse nicht vollständig implementiert ist.
Weitere neun Klassen wurden indirekt über die implementierten Tests abgedeckt. Dies sind hauptsächlich einfache Objekt Klassen, welche als interne Repräsentation der Daten genutzt werden. Hier wurden keine Unit Tests geschrieben, da die Klassen zum größten Teil aus Getter und Setter Methoden bestehen, wo extra Testmethoden zur Überprüfung, ob die Parameter gesetzt oder zurückgegeben werden, nicht sinnvoll sind. Aber sie bieten einen guten Überblick, inwieweit die einzelnen Parameter innerhalb der Anwendung genutzt werden. So kann man z.B. aus der Abdeckung des `Course` Objektes mit 93,55 % schließen, dass so gut wie alle Parameter und Methoden verwendet werden. Im Gegensatz dazu hat das `Slide` Objekt eine Abdeckung von 72,41 %, was auf unvollständige Testabdeckung schließen lässt oder sogar Parameter entfernt werden müssen, da sie innerhalb der Applikation keine Verwendung haben. Dies kann auftreten, wenn Parameter nur für die Rollen „Lectuer“ oder „Admin“ in bestimmten Anwendungsfällen benötigt werden.
Aus den Daten der Integrationstests und UI Tests lassen sich keine direkten Testabdeckungen generieren. Sie werden hauptsächlich für die Abdeckung von Anwendungsfällen genutzt. Nichtsdestotrotz sind sie für ein stabiles und fehlerfreies Gesamtsystem sehr wichtig.
Schlussfolgernd ist die Testabdeckung ein wichtiger Indikator für die Qualität einer Software und sollte daher in das Konzept aufgenommen werden. Die bisherigen Daten stammen alle aus dem lokalen „Code Coverage“ Werkzeug, welches in Android Studio integriert ist. Um weitere Testabdeckung verfolgen zu können und dies nicht immer händisch machen zu müssen, wurde das „Cobertura“ Plugin zu Jenkins hinzugefügt [63]. Das Plugin ermöglicht eine automatische Veröffentlichung der Testabdeckung mit jedem Testlauf innerhalb von Jenkins. Dadurch lassen sich weiterhin Tendenzen zur Abdeckung des Codes während der Entwicklung feststellen und wenn nötig Änderungen an der Entwicklung vorzunehmen.

[1] Letzter erfolgreicher Aufruf: 26.11.2017 - im Netzwerk der TU Dresden

6.5. PERFORMANZ TESTS

Ein Teil der Aufgabenstellung umfasst die Betrachtung der Performanz. Zur Untersuchung der Performanz der neuen Applikation wurden einfache Messungen durchgeführt. Dabei wurden zwei Fälle betrachtet, um eine Bewertung abgeben zu können. Es wurde untersucht, inwieweit sich Zeiten innerhalb der vorherigen und neuen Version der Applikation unterscheiden. Die zwei Fälle setzen sich aus Zeitmessungen zusammen, welche in der folgenden Auflistung dargestellt werden.

- erste Anzeige der Übersichtseite für Kurse und aktive Vorlesungen
- erste Anzeige von geladenen Daten

Vorweg muss erwähnt werden, dass die Nutzung von Zeitangaben nicht optimal ist, da die Applikationen unterschiedlich optimiert sein können. Dies kann durch verschiedene Bedingungen, wie das Android Framework, die Android Runtime, der Quellcode oder der Compiler verursacht werden. Die Ladezeiten der Applikation kann z.B. durch unterschiedliche Dateigrößen beeinflusst werden, aber im Fall von AMCS ist hier kaum ein Unterschied gegeben. Die Debug Variante der alten Version hat eine Größe von 3,2 MB und die neue Version eine Größe von 3,4 MB. Außerdem wurde bei der Aufzeichnung der Testarten darauf geachtet, dass die Applikationen von derselben Entwicklungsumgebung gebaut worden sind. Speziell war dies ein MacBook Pro 13 mit folgenden Spezifikationen: Modell: Ende 2013, CPU: 2,4 GHz Intel Core i5, RAM: 8 GB 1600 MHz DDR3.
Weiterhin wurde nach Alternativen zu den Zeitmessungen gesucht. Eine Möglichkeit ist die Nutzung eines integrierten Profiler, welcher in die neue „Android Studio 3.0“ Version integriert wurde. Hier besteht die Möglichkeit, die CPU Auslastung auszulesen. Der CPU Takt muss mit anderen Mitteln ausgelesen werden, da dieser in den meisten Android Geräten nach Bedarf angepasst wird. Des Weiteren stellt der Profiler im Endeffekt wieder nur Zeitangaben in Form der Ausführungsdauer von verschiedenen Methoden zur Verfügung.
Eine andere Option ist die Nutzung der `perf`-Tools innerhalb des Linux Kernels, welche seit der Kernel Version 2.6.31 genutzt werden können. [64] In Android wurde die Kernel Version 2.6.32 mit Android 2.2 Froyo eingeführt. [65] Mit Hilfe der Tools können noch weitere Daten aufgezeichnet werden. Die Option, die in den Linux Kernel eingebauten `perf`-Tools zu nutzen, war keine Zielstellung der vorliegenden Arbeit.
Schlussendlich wurden doch Zeitangaben für die Betrachtung verwendet. Für die Zeiterfassung wurden Möglichkeiten des Android Frameworks genutzt. In der Entwicklungsumgebung „Android Studio“ gibt es eine Log Ausgabe des angeschlossenen Gerätes oder Emulators. Innerhalb des Logs werden verschiedene Events und Ausgaben dargestellt. Unter Android gibt es einen „ActivityManager“, welcher das Starten einer Activity und die Zeit bis zur ersten Darstellung der Applikation in das Log schreibt. Seit Android API Level 19 oder Android 4.4 gibt es außerdem eine Methode, um einen zusätzlichen Zeiteintrag über den „ActivityManager“ ausgeben zu lassen. Die Methode muss dem Quellcode hinzugefügt werden und wird verwendet, um die erste fertige Anzeige von geladenen Daten zu erfassen. Bei der Methode handelt es sich um `reportFullyDrawn()` und muss über eine `Activity` aufgerufen werden. [66] Im folgenden Beispiel wird die Log Ausgabe der ersten Darstellung einer Activity und die Ausgabe per `reportFullyDrawn()`-Methode dargestellt, welche aus Gründen der Übersichtlichkeit gekürzt wurde.

```
1 16:22:54.434 I/AM: Displayed ...OverviewActivity: +243ms (total +774ms)
2 16:22:54.466 I/AM: Fully drawn ...OverviewActivity: +274ms (total +805ms)
```

Bei der Messung der Daten wurde ein sogenannter „Kaltstart“ der Applikation durchgeführt. Das bedeutet, dass die Applikation vor dem Starten komplett beendet sein muss. Eine sche-

matische Darstellung mit einem Ablauf des Kaltstarts der AMCS Apps wird in der folgenden Abbildung aufgezeigt.

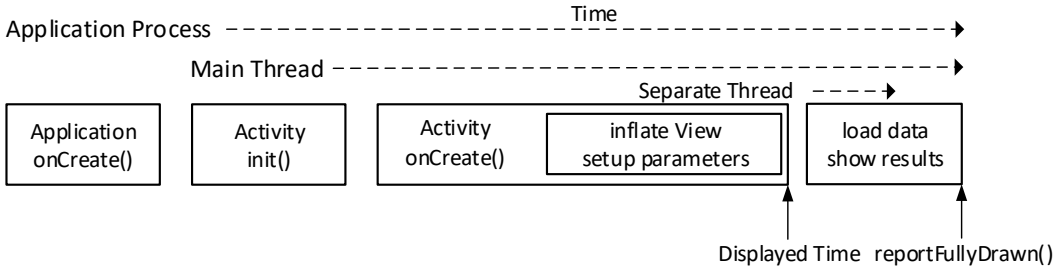


Abbildung 6.1.: Kaltstart der AMCS App

Um einen breiten Überblick zu bekommen, kamen bei der Erfassung der Daten verschiedene Android Geräte und unterschiedliche Orte zum Einsatz. Die Testgeräte mit den spezifischen Eigenschaften werden in der nachfolgenden Tabelle vorgestellt.

Gerät	Android Version	Hardware
Moto E (2nd)	Android 6.0	4,5"qHD(960x540p) Display, 1,2 GHz Quadcore CPU (Snapdragon 200), 1GB RAM, Veröffentlichung: 02.2015
Nexus 4	Android 6.0.1	4,7"HD(1280x768p) Display, 1,5 GHz Quadcore CPU (Snapdragon S4Pro), 2GB RAM, Veröffentlichung: 11.2012
LG GPad 8.3	Android 4.4.3	8,3"FullHD(1920x1200p) Display, 1,7 GHz Quadcore CPU (Snapdragon 600), 2GB RAM, Veröffentlichung: 10.2013
OnePlus 3	Android 7.1.1	5,5"FullHD(1920x1080p) Display, (2x2,15GHz und 2x1,6 GHz) Quadcore CPU (Snapdragon 820), 6 GB RAM, Veröffentlichung: 06.2016

Tabelle 6.5.: Testgeräte zur Performanzbewertung

Die `reportFullyDrawn()`-Methode wird erst nach einer Netzwerkanfrage aufgerufen und daher ist es wichtig, die Tests zu unterschiedlichen Zeiten, Orten und Netzwerkbedingungen durchzuführen. Bei der Aufzeichnung wurden zwei Anwendungsfälle simuliert. Zum einen wurde die Sächsische Landesbibliothek – Staats- und Universitätsbibliothek Dresden genutzt, um einen Testlauf in einem öffentlichen WLAN Netzwerk zu simulieren und zum anderen wurde ein Testlauf vom eigenen Wohnort durchgeführt. Für die Versuchsausführung wurde ein neuer Nutzer erstellt, welcher in genau einem Kurs mit einer aktiven Veranstaltung eingeschrieben ist.

- Nutzer: testPerf, Passwort: abc
- Kurs: Testkurs, PIN: pin1234, Veranstaltung: TestVeranstaltung (LIVE)

Dieser Nutzer wurde in jedem Versuch verwendet, außerdem wurde die Backendversion von AMCS genutzt, welche produktiv im Einsatz ist. Auf den Testgeräten wurde der vorherige Stand AMCS Applikation gelöscht, neu installiert und der Nutzer eingeloggt. Weiterhin wurde darauf geachtet, dass keine Hintergundapplikationen ausgeführt wurden und die AMCS

App jedes Mal über den TaskManager beendet wurde.
Der Aufbau der neuen und der alten AMCS Applikation unterscheidet sich in einigen Punkten. Die Darstellung der Daten und Oberfläche ist prinzipiell gleich. Die alte Version nutzt nur eine einzige `Activity` und verwendet mehrere `Fragments` für die Darstellung der Oberfläche. Ein Nachteil ist die erhöhte Komplexität innerhalb der einzelnen Activity, da diese durch die Verwaltung der verschiedenen `Fragments` schnell heranwächst und somit die Übertragung von Daten zwischen verschiedenen `Fragments` komplex ist. Ein Vorteil ist jedoch der schnellere Austausch von `Fragments` im Vergleich zum Löschen und Erstellen einer neuen `Activity`. In der neuen App gibt es für jede Ansicht jeweils eine `Activity` und ein `Fragment`. So erfolgt die Trennung der Komponenten, die Testbarkeit sowie der interne Austausch von Daten besser. Ursprünglich war geplant einen dritten Fall zu untersuchen, nämlich die Dauer des Startvorgangs. Durch den unterschiedlichen Aufbau der Applikationen unterscheidet sich die Messung der Startzeit und wurde daher nicht genauer untersucht. Die anderen zwei Fälle können als identisch betrachtet werden, da die Ausgabe auf demselben Ereignis beruht. In der neuen App wird in der `OverviewActivity` die `Displayed` Ausgabe im Log erzeugt und in der alten Version sorgt das `CourseOverviewFragment` für dieselbe Ausgabe. Die Ausgabe von `Fully drawn` erfolgt äquivalent, wenn die ersten Daten geladen und angezeigt werden, dies kann zuerst der Kurs oder die Veranstaltung sein.

Gerät	Testlauf	t Displayed	s Displayed	t FullyDrawn	s FullyDrawn
Moto E	1	2373,9 ms	13,98 ms	2418,5 ms	15,38 ms
	2	2416,2 ms	22,7 ms	2524,6 ms	22,03 ms
Nexus 4	1	3165,8 ms	217,69 ms	3307,5 ms	227,25 ms
	2	3055,3 ms	42,01 ms	3195,2 ms	61,75 ms
LG GPad 8.3	1	835,2 ms	93,76 ms	983,8 ms	101,86 ms
	2	878,7 ms	58,44 ms	1100,8 ms	70,6 ms
OnePlus 3	1	1096,9 ms	20,35 ms	1200,4 ms	69,34 ms
	2	1347,4 ms	20,24 ms	1526,9 ms	16,23 ms

Tabelle 6.6.: Messungen der neuen Applikation

Gerät	Testlauf	t Displayed	s Displayed	t FullyDrawn	s FullyDrawn
Moto E	1	2183,5 ms	15,67 ms	2236 ms	36,58 ms
	2	2276 ms	20,74 ms	2361 ms	27,44 ms
Nexus 4	1	3164,4 ms	50,32 ms	3373 ms	336,08 ms
	2	2743,3 ms	36,61 ms	2831,9 ms	34,2 ms
LG GPad 8.3	1	720,7 ms	89 ms	906 ms	264,94 ms
	2	685,8 ms	121,3 ms	869,4 ms	120,8 ms
OnePlus 3	1	859,8 ms	18,55 ms	949,6 ms	20,27 ms
	2	1083 ms	13,14 ms	1278,3 ms	17,58 ms

Tabelle 6.7.: Messungen der alten Applikation

App	Testlauf	t Displayed	s Displayed	t FullyDrawn	s FullyDrawn
Neue App	1	1733,725 ms	43,39 ms	1868,2 ms	164,47 ms
	2	1924,4 ms	15,53 ms	2086,87 ms	23,82 ms
Alte App	1	1701,58 ms	43,7 ms	1821,1 ms	169,45 ms
	2	1693,2 ms	43,19 ms	1828,78 ms	41,3 ms

	App	t Displayed	s Displayed	t FullyDrawn	s FullyDrawn
Kombination	neu	1896,18 ms	50,98 ms	2032,21 ms	63,64 ms
	alt	1713,46 ms	43,29 ms	1848,49 ms	102,88 ms

Tabelle 6.8.: Mittelwerte aus allen Messungen im Vergleich

Die ersten beiden Tabellen stellen die zwei Fälle mit den jeweiligen Durchschnittszeiten (t) und Standardabweichungen (s) von zehn Ausführungen dar. Die Ergebnisse werden nach Gerät und Testlauf unterschieden. Die erste Tabelle zeigt die Messungen der neuen Applikation und die zweite stellt die Messungen der alten Applikation dar. Die dritte Tabelle kombiniert die Werte aller Geräte und stellt diese sowohl für die Testläufe als auch die Kombination aus beiden Läufen dar. Die Werte der Darstellung der ersten Überischtseite und der Anzeige der ersten Daten zeigen auf, dass die neue Anwendung im Durschschnitt etwa **10 %** langsamer ist als die alte Applikation. Die Standardabweichung „s Displayed“ zeigt, dass die Messungen bzw. die Ausführungsdauer keine große Streuung besitzen. Es gibt nur zwei Läufe, wo die Standardabweichung größer als 100ms ist. Die zweite Standardabweichung „s FullyDrawn“ zeigt ein andere Bild, hier spielt nicht nur das Gerät sondern auch die Netzwerkverbindung eine Rolle. Es müssen Daten übertragen werden und auf Grund der Fehleranfälligkeit von WLAN und Netzwerken im Allgemeinen ist die Streuung höher.
Zuletzt wurde die durchschnittliche Zeit bis zum Anzeigen der ersten Daten ermittelt. In der neuen Applikation sind das ca. **2032 ms** und in der alten ca. **1848 ms**. Laut einem Blog-Eintrag von „Apteligent“ [67] sind Nutzer mit einer Ladezeit unter 2000 ms zufrieden. Dieser Wert wird im Durchschnitt von der neuen Applikation unterboten. Eine Umfrage von „CA Technolgoies“ [68] ergab, dass 60 % der befragten Nutzer eine Ladezeit von 3000 ms akzeptieren, darüber hinaus sinkt die Akzeptanz. Die drei Sekunden Grenze wird nur von einem ca. fünf Jahre alten Nexus 4 leicht überschritten. Die Ladezeiten der neuen Applikation haben laut den genannten Quellen kaum bis gar keinen negativen Einfluss auf die Akzeptanz und Zufriedenheit der Nutzer.
Die längere Dauer bis zur Anzeige der ersten Activity und der Daten lässt sich durch den bereits beschriebenen unterschiedlichen Aufbau der Applikationen erklären. In der neuen Applikation wird nach dem Start der Anwendung zuerst die `LoginActivity` gestartet, diese überprüft, ob das AMCS Token valide ist und startet anschließend die `OverviewActivity`. In der vorherigen App wurde nur die `StartActivity` gestartet und das `LoginFragment` hinzugefügt, welches überprüft, ob das AMCS Token valide ist. Anschließend wurde das `LoginFragment` mit dem `CoursesStudentFragment` ausgetauscht, was deutlich schneller geschieht als das löschen und neu erstellen einer Activity.
An dieser Stelle kann in Zukunft untersucht werden, ob sich in der neuen Applikation die Initialisierung der `LoginActivity` vermeiden lässt, um bei einem gültigen Token sofort die `OverviewActivity` zu starten. Theoretisch kann somit die Performanz erhöht werden und die neue Anwendung schneller als die alte Version starten. Dies muss nachträglich noch untersucht und gemessen werden.
Die manuelle Erfassung der Daten ist nicht optimal und nimmt einige Zeit in Anspruch. Mit Hilfe von Jenkins kann diese Aufgabe automatisiert werden. Man kann z.B. einen extra Job für die Performanzbewertung einrichten. Das Log eines Emulator kann über die Konsole kom-

plett ausgelesen werden. Dadurch ist es möglich die nötigen Angaben auszulesen und in eine XML-Struktur zu übertragen. Diese lässt sich innerhalb Jenkins so darstellen, dass auf die Ergebnisse zugegriffen werden kann. Damit die Performanzbetrachtung nicht mit jedem Hinzufügen von Code in die Versionsverwaltung überprüft wird, kann man definieren, dass sie nur bei Änderungen im `develop`-Zweig ausgeführt werden.

6.6. ZUSAMMENFASSUNG

Die Evaluation umfasst die Speicher- und Performanzprobleme der zuerst bereitgestellten virtuellen Maschine auf welcher eine Instanz von Jenkins ausgeführt wird. Der fehlende Speicherplatz begründete die Suche nach einer Alternative. Dies geschah in Form einer neuen Erstellung von Jenkins auf einem von der TU Dresden zur Verfügung gestellten Laptop. So ließen sich die langsamen Antworten der virtuellen Maschine umgehen. Weiterhin erfolgte die Betrachtung der testgetriebenen Entwicklung mit ihren Vorteilen, wie die bessere Strukturierung des Codes, die Überprüfung der Funktionalität und die gewonnene Sicherheit bei Refaktorisierungen. Von Nachteil ist die Umsetzung des Ansatzes, wenn die Architektur der Anwendung nicht bekannt ist. Ein weiterer Punkt in diesem Kapitel war die Befragung, um das Konzept dieser Arbeit beurteilen zu können. Es mussten nur kleinere Änderungen am Konzept vorgenommen werden. Dies war vor allem die lokale Ausführung der Unit Tests vor einem Push in die Versionskontrolle. Des Weiteren wurde die Testabdeckung der Anwendung untersucht. Teilweise wurden sehr hohe Abdeckungen erreicht, bedingt durch die Umsetzung eines höheren Aufwandes an Testcodezeilen. Die Gründe für den Mehraufwand an Zeilen sind untersucht worden. Es wurde festgestellt, dass die Testabdeckung in Jenkins integriert werden muss. Mit Hilfe des Plugins „Cobertura“ wurde dies erreicht.
Zuletzt wurde die Performanz zwischen der alten AMCS Version und der neuen verglichen. Es muss festgestellt werden, dass sich die Performanz geringfügig verschlechtert hat. Positiv ist aber eine viel bessere Testbarkeit, Wartbarkeit sowie die Möglichkeiten für zukünftigen Weiterentwicklungen.

7. ZUSAMMENFASSUNG UND AUSBLICK

Ziel dieser Arbeit war die Entwicklung eines Konzeptes zur automatisierten Ausführung von Tests innerhalb einer kontinuierlichen Integrationsumgebung.

7.1. ZUSAMMENFASSUNG

Das erste Kapitel sensibilisiert für den Einsatz von Tests in der mobilen Softwareentwicklung. Außerdem wird die Zielstellung dieser Arbeit näher untersucht.

Im zweiten Kapitel geht es zunächst um die Grundlagen von mobilen Applikationen und Softwaretests im Allgemeinen. Mit Hilfe von Testframeworks kann die Entwicklung von Tests unterstützt werden und auf Besonderheiten bei mobilen Applikationen geachtet werden. Um eine hohe Testbarkeit erreichen zu können, muss die Architektur der Anwendungen geändert werden. Es wurden gängige Architekturen von mobilen Applikationen untersucht und miteinander verglichen. Zuletzt wurden die für die Automatisierung notwendigen kontinuierlichen Integrationsumgebungen betrachtet und Vor- und Nachteile hervorgehoben.

Das dritte Kapitel zeigt den aktuellen Stand der AMCS Applikationen und bietet eine Funktionsübersicht der Android und iOS Versionen. Es wurden Anforderungen an die Auswahl der Werkzeuge gestellt, um die Umsetzung des Konzeptes zu ermöglichen. Beschrieben werden die verschiedenen Tests, welche mit dieser Arbeit abgedeckt werden.

Die Beschreibung des Konzeptes wird im vierten Kapitel dargestellt. Zunächst musste die Architektur der Applikationen geändert werden. Es wurde sich auf Grund von einigen Vorteilen und der nicht zu komplexen Umsetzung für die Model View Presenter Architektur entschieden. Des Weiteren wurden verschiedene Ansätze und Testarten untersucht, welche zur Automatisierung verwendet werden können. Im Anschluss wurden die Anforderungen an neue Funktionen, die Versionskontrolle und die kontinuierliche Integration beschrieben. Die Versionskontrolle und kontinuierliche Integration bilden hierbei die Hauptkomponenten in der automatischen Ausführung von Tests. Anhand der Zweige wird innerhalb der kontinuierlichen Integration der Umfang an auszuführenden Tests unterschieden.

Das fünfte Kapitel beschreibt die Implementierung des Konzeptes. Es wurde festgelegt, dass die Implementierung unter Android prototypisch umgesetzt und dementsprechend die MVP Architektur innerhalb des Android Frameworks genutzt wird. Anhand der Architektur wurde

erklärt, wie die einzelnen Komponenten getestet werden. Bei der Überprüfung der Ansichten wurden die erstellten UI Tests und deren Funktionsweise beschrieben. Die Integrationstests werden im Abschnitt der Kontrolle des Models erläutert. Anschließend wurde die Versionskontrolle im Rahmen der Implementierung durch Bitbucket umgesetzt und der Ansatz um einen zusätzlichen Zweig erweitert. Die kontinuierliche Integration wurde mit Hilfe von Jenkins umgesetzt. Jenkins ist eine Automatisierungsserver Software, welche einfach eingerichtet und durch zahlreiche Plugins erweitert werden kann. Durch die Plugins ist unter anderem die Verwendung der Android Emulatoren möglich. So kann ein breites Spektrum an Android Versionen mit unterschiedlichen Auflösungen abgedeckt werden. Für die Automatisierung wird eine Pipeline konfiguriert, in der, abhängig vom aktuellen Zweig, verschiedene Schritte ausgeführt werden.

Das Kapitel zur Evaluation dieser Arbeit zeigt verschiedene Probleme auf. Problematisch war vor allem die initiale Nutzung von Jenkins, da der Speicherplatz nicht ausreichend und die Verbindung bzw. der initiale Zugriff langsam war. Daher wurde Jenkins auf einem Laptop neu aufgesetzt, wodurch sich die Probleme beheben ließen. Außerdem wurden die Vor- und Nachteile der testgetriebenen Entwicklung betrachtet. In dem Kapitel wurde eine Befragung ausgewertet, um damit Ableitungen zum vorgestellten Konzept treffen zu können. Anhand der Testabdeckung wurde die Qualitätssicherung der Applikation untersucht. Zur weiteren Untersuchung und zum Bilden von Tendenzen wurde in Jenkins ein Plugin zur automatischen Erstellung von Berichten über die Testabdeckung hinzugefügt. Zuletzt wurde die Performanz zwischen der alten Android App und der neu entwickelten Version miteinander verglichen. Das Ergebnis zeigt, dass die neue Version durch einen unterschiedlichen Aufbau zwar etwas langsamer ist, aber dadurch die Testbarkeit, die Wartbarkeit und das Hinzufügen von neuen Funktionen viel einfacher umzusetzen ist.

7.2. AUSBLICK

Eine interessante Erweiterung der bisherigen UI Tests über die Android Emulatoren könnten Cloud Dienste sein. Es gibt unter anderem von „Firebase“, „Amazaon Web Services“ oder der „GenyMotion Cloud“ die Möglichkeit, UI Tests auf realen Geräten ausführen zu lassen. Der Blog Eintrag [69] bietet dazu eine gute Einstiegsmöglichkeit. Es wird gezeigt, dass für die genannten Dienste Jenkins Plugins vorhanden sind und wie diese integriert werden. Ein Nachteil ist die begrenzte Anzahl oder Umfänge der kostenlosen Testmöglichkeiten. An dieser Stelle müssten die Vorteile und benötigten Leistungen genauer untersucht werden, ob sich ein Einsatz für AMCS lohnen würde. Die Untersuchung der Dienste und das Herausarbeiten der Vor- und Nachteile hat in etwa einen Umfang einer Seminararbeit. Die Implementation anhand eines Anbieters oder der Vergleich von verschiedenen Diensten hat eher den Umfang einer Bachelorarbeit.
Ein weiterer Punkt ist die Optimierung der Durchführung von Integrationstests. Die möglichen Antworten des Backends werden aktuell per Hand in JSON-Dateien abgespeichert und über WireMock als Antwort auf eine Anfrage zurückgegeben. An dieser Stelle wäre es sinnvoll die Dokumentation der Backend Schnittstelle zu nutzen und die JSON-Dateien automatisch generieren zu lassen.
Der Prozess der kontinuierlichen Integration wird aktuell über eine zeitliche Überprüfung des in der Versionskontrolle gespeicherten Quellcodes gestartet, wenn Änderungen vorhanden sind. Zur Optimierung des Prozesses könnte die Nutzung von sogenannten „WebHooks“ zu Bitbucket beitragen. Dafür muss eine bisher nicht gegebene öffentliche Schnittstelle zwischen Jenkins und Bitbucket gegeben sein. Durch die Verwendung von WebHooks kann Bitbucket Jenkins über Änderungen am Quellcode benachrichtigen und somit den kontinuierlichen Integrationsprozess starten.

Die UI Testausführung könnte in Jenkins vereinfacht werden, indem man die Quellen an einem einheitlichen Ort ablegt. Jenkins führt jede Emulator Instanz in einem eigenen Job aus und jeder Job hat seinen eigenen Workspace. In dem Workspace wird jedes Mal der Quellcode heruntergeladen, die Applikation gebaut und anschließend die Tests ausgeführt werden. Wenn die Quellcode und das Bauen der App nur einmal durchgeführt wird, erspart dies einiges an Mehraufwand und würde die Ausführungszeit deutlich beschleunigen. Eventuell lässt sich hier das Jenkinsfile zur Konfiguration und Starten der Emulatoren nutzen, wodurch das Plugin für den Android Emulator nicht mehr verwendet werden muss. Der Aufwand der Untersuchung lässt sich aus momentaner Sicht nur schwer abschätzen, da eine Dokumentation zu Jenkinsfile in Kombination mit Android Projekten nur spärlich vorhanden ist.
Wie bereits im Abschnitt 6.4 der Auswertung zur Testabdeckung erwähnt, könnte der Einsatz des „Robolectric“ Frameworks [62] für eine breitere Testabdeckung und Verwendung von Unit Tests nützlich sein. Mit Hilfe des Frameworks lassen sich weite Teile der Android Abhängigkeiten durch Mocks ersetzen, wodurch sich z.B. Activities oder Buttons auf deren Funktionalität überprüfen lassen. Durch die Verwendung von Unit Tests lässt sich die Ausführungszeit der Tests stark beschleunigen, da keine Android Umgebung in Form eines Emulators oder angeschlossenen Gerätes benötigt wird.
Zu guter Letzt soll noch die fehlende Implementierung unter iOS erwähnt werden. Die vorgestellte MVP Architektur kann unter iOS mit wenigen an das System angepassten Änderungen ebenfalls umgesetzt werden. Die Unit und UI Tests lassen sich z.B. mit dem XCTest Framework umsetzen. Außerdem können die in das Framework integrierten „measure“-Blöcke genutzt werden, um einfache Performanzbetrachtungen durchzuführen. Ob es einen ähnlichen Ansatz wie „WireMock“ unter Android gibt, wurde nicht untersucht. Bei einem ähnlichen Ansatz könnten die möglichen Antworten in Form von JSON-Dateien wiederverwendet werden.
Die Versionskontrolle durch BitBucket und der GitFlow Ansatz kann auch für Swift verwendet werden. Bei Jenkins sind Änderungen nötig, da die iOS Applikation nur auf einem Computer mit dem Mac Betriebssystem und installierten Entwicklerwerkzeugen gebaut und getestet werden kann. Hier hat man die Wahl zwischen einer komplett neuen Jenkins Instanz auf einem macOS Computer, welche nur für das iOS Projekt zuständig ist oder man nutzt einen Build-Agenten. Der Build-Agent wird ebenfalls auf einem macOS Computer ausgeführt, dieser wird aber über die Haupt Jenkins Instanz gesteuert und mit der Ausführung von Jobs beauftragt. Die Jobs umfassen das Bauen und Testen der Anwendung sowie die Veröffentlichung von Testergebnissen und Testabdeckungen.

A. ANHANG

A.1. BEFRAGUNG

Die Befragung wurde mit dem AMCS Tool online durchgeführt. Im Nachfolgendem wird eine Übersicht der Fragen und vorgegebenen Antworten dargestellt. Bei einigen Fragen gab es die Möglichkeit, diese per Texteingaben zu beantworten und wurden mit „Freitextantwort" gekennzeichnet.

1. Haben Sie Softwaretests in irgendeiner Form schon mal verwendet?
    - Ja
    - Nein
2. Welche Ebenen von Tests werden verwendet? (mehrere Antworten möglich)
    - Unit Tests
    - Integrationstests
    - Akzeptanztests
    - Systemtests
    - Weitere (bitte angeben) + Freitextantwort
3. Welche Arten von Tests wurden verwendet?
    - Funktionale Tests
    - Performanztests
    - Stresstests
    - Weitere (bitte angeben) + Freitextantwort
4. Wie oder wo erfolgt die Erstellung von Testdefinitionen?
    - Tickets
    - User Stories
    - Weitere (bitte angeben) + Freitextantwort
5. Zu welchem Zeitpunkt werden die Tests erstellt?
    - Vor der Codeimplementierung
    - Nach der Codeimplementierung
    - Nebenbei der Codeimplementierung
6. Zu welchem Zeitpunkt wird getestet? (Im Bezug auf Git Befehle, wie commit, push, ...)
    - Nach einem Commit
    - Nach push
    - Nach einer Integration per Pull-Request oder Merge
    - Zeitlich
    - Wenn es gerade passt
    - Weitere (bitte Angeben) + Freitextantwort
7. Wird der Code gleichmäßig getestet oder werden Komponenten unterschiedlich betrachtet?
    - Gleichmäßig
    - Unterschiedlich
8. Was muss getan werden, um Tests auszuführen?
    - Tests werden per Hand ausgeführt
    - Tests werden automatisch in einer kontinuierlichen Integrationsumgebung ausgeführt
    - Weitere (bitte angeben) + Freitextantwort

9. Wie wird auf Testergebnisse zugegriffen?
    - Freitextantwort
10. Wird eine Art von automatischer Testausführung genutzt?
    - Ja
    - Nein
11. Werden Testmaße berücksichtig?
    - Code Abdeckung
    - Anforderungsabdeckung
    - Weitere (bitte angeben) + Freitextantwort
    - Nein
12. Bilden die Tests den Zustand des Systems gut ab? (Test fehlerfrei = System fehlerfrei?)
    - Skalenfrage – 0 % - 100 %
13. Wie häufig sollte ein Commit ausgeführt werden (1 so modular/kleinteilig wie möglich, 6 so selten wie möglich)
    - Skalenfrage – 1 - 6
14. Wie häufig sollte ein Push ausgeführt werden (1 so früh wie möglich, 6 so selten wie möglich)
    - Skalenfrage – 1 - 6
15. Sollte ein Commit ausgeführt werden, bevor ein Unit Test bestanden wurde?
    - Nein, weil + Freitextantwort
    - Ja, weil + Freitextantwort
16. Sollte ein Push ausgeführt werden, bevor ein Unit Test bestanden wurde?
    - Ja, weil + Freitextantwort
    - Nein, weil + Freitextantwort
17. Ist der Git Flow Ansatz schon einmal verwendet worden?
    - Ja
    - Nein
18. Welche Vor- und/oder Nachteile ergeben sich daraus?
    - Freitextantwort
19. Überwiegen dabei die Vor- oder die Nachteile?
    - Vorteile überwiegen, weil + Freitextantwort
    - Nachteile überwiegen, weil + Freitextantwort
20. Wurde der Test Driven Development Ansatz schon einmal verwendet?
    - Ja
    - Nein
21. Werden durch TDD mehr Fehler vermieden?
    - Ja, weil + Freitextantwort
    - Nein, weil + Freitextantwort
22. Ist TDD den Mehraufwand wert?
    - Ja, weil + Freitextantwort
    - Nein, weil + Freitextantwort

23. Wurde eine Art der kontinuierlichen Integration schon einmal benutzt?
    - Ja, in Form von + Freitextantwort
    - Nein
24. Wie würden Sie die Granularität zwischen Zweigen und Tests einteilen? (1 - alle Tests in jedem Zweig ausführen, 6 - nur im Hauptzweig testen)
    - Skalenfrage – 1 - 6
25. Wann wäre der beste Zeitpunkt um die jeweilige Testart auszuführen?
    - Unit Tests + Freitextantwort
    - Integrationstests + Freitextantwort
    - Akzeptanztests + Freitextantwort
    - Stresstests + Freitextantwort
    - Systemtests + Freitextantwort

A.2. ÜBERSICHT DER UNIT TESTS

package	Klasse/Methoden	Beschreibung
communication	JsonParserTest.java	Überprüfung: Umwandlung von JSON in interne Objekte
	testParseCourse()	
	testParseLectur()	
	testParseSlide()	
	testParseQuestion()	
	testParseGroupedQuestion()	
lectureView.student	LectureViewPresenterTest.java	Überprüfung: Verhalten des LectureViewPresenters
	testCreatePresenter_setsPresenterToView()	
	testShowCourseInfo()	
	testShowCourseNotSetHint()	
	testShowEmptyLectures()	
	testShowLectures()	
	testShowUnsubscribeDialog()	
	testShowSuccessfulUnsubscription()	
	testShowCourseWithIdNotFoundError()	
	testShowAuthorizationError()	
	testShowSlideView()	
login	LoginViewPresenterTest.java	Überprüfung: Verhalten des LoginViewPresenters
	testCreatePresenter_setsPresenterToView()	
	testPresenterStart()	
	testLoadUserFromRepositoryAndLoadIntoView()	
	testLoadUserFromRepositoryAndShowOverview()	
	testShowUsernameEmptyError()	
	testShowUsernameTooLongError()	
	testShowUsernameTooShortError()	
	testShowUsernameContainsWhitespace()	
	testShowPasswordEmptyError()	
	testCheckForCorrectUsernameAndPassword()	
overview	OverviewPresenterTest.java	Überprüfung: Verhalten des OverviewPresenters
	testCreatePresenter_setsPresenterToView()	
	testPresenterStart()	
	testShowEmptyCourses()	
	testShowEmptyCourses()	
	testShowEmptyActiveLectures()	
	testShowCourses()	
	testShowActiveLectures()	
	testShowPINDialog()	
	testAddCourse()	
	testShowCourseAlreadyAdded()	
	testShowNoCourseWithPINFound()	
	testShowLectureView()	
	testShowSlideVie	

repository.course	CourseRepositoryTest.java	Überprüfung: Kurse laden, hinzufügen, löschen
	testLoadCourses()	
	testDeleteCourse()	
	testAddCourse()	
	testDeleteCache()	
repository.lecture	LectureRepositoryTest.java	Überprüfung: Vorlesungen laden
	testLoadLecture()	
	testLoadLectures()	
	testLoadActiveLectures()	
repository.question	QuestionRepositoryTest.java	Überprüfung: Fragen laden und löschen, Antworten senden, Antworten ändern
	testLoadSlideQuestions()	
	testLoadLectureQuestions()	
	testLoadCourseQuestions()	
	testSendAnswer()	
	testChangeChoice()	
	testClearQuestions()	
repository.user	UserRepositoryTest.java	Überprüfung: Laden eines Users
	testGetUser()	
slideView.student	SlideViewPresenterTest.java	Überprüfung: Verhalten des SlideViewPresenters
	testCreatePresenter_setsPresenterToView()	
	testShowLectureInfo()	
	testShowLectureNotSetHint()	
	testSetSlideNumber()	
	testLectureStateChange()	
	testShowLectureNotFoundError()	
	testLoadLectureStateDuring_startsLoadQuestions()	
	testLoadLectureStateBefore_startsLoadQuestions()	
	testLoadLectureStateAfter_startsLoadQuestions()	
	testLoadLectureStateOffline_startsLoadQuestions()	
	testShowSlideQuestionsEmpty()	
	testShowLectureQuestionsEmpty()	
	testShowCourseQuestionsEmpty()	
	testShowSlideQuestionsAnswered()	
	testShowLectureQuestionsAnswered()	
	testShowCourseQuestionsAnswered()	
	testSlideShowQuestions()	
	testLectureShowQuestions()	
	testShowCourseQuestions()	
	testSendAnswer()	
	testChangeChoice()	
	testLoadSlideQuestionsCallsOnFailure()	
	testLoadLectureQuestionsCallsOnFailure()	
	testLoadCourseQuestionsCallsOnFailure()	
	testWebSocketFailure()	

B. LITERATURVERZEICHNIS

[1] Statista. *App stores: number of apps in leading app stores 2017*. URL: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/ (besucht am 03. 10. 2017).

[2] OpenSignal.com. *Android Fragmentation Report August 2015*. URL: http://opensignal.com/reports/2015/08/android-fragmentation/ (besucht am 03. 10. 2017).

[3] Google. *Thanks to developers and our partners around the world, there are now more than 2 billion monthly active Android devices. #io17pic.twitter.com/tMH6aHsAIP*. @Google. URL: https://twitter.com/Google/status/864890655906070529 (besucht am 26. 11. 2017).

[4] *World Quality Report 2014-2015*. Sogeti. URL: https://www.sogeti.com/explore/reports/world-quality-report-2014-2015/ (besucht am 27. 11. 2017).

[5] Ericka Chickowski. *World Quality Report 2017-18: The state of QA and testing*. TechBeacon. URL: https://techbeacon.com/world-quality-report-2017-18-state-qa-testing (besucht am 27. 11. 2017).

[6] *White-Box-Test*. In: *Wikipedia*. Page Version ID: 162837307. 20. Feb. 2017. URL: https://de.wikipedia.org/w/index.php?title=White-Box-Test&oldid=162837307 (besucht am 27. 11. 2017).

[7] *Black-Box-Test*. In: *Wikipedia*. Page Version ID: 156846981. 9. Aug. 2016. URL: https://de.wikipedia.org/w/index.php?title=Black-Box-Test&oldid=156846981 (besucht am 27. 11. 2017).

[8] Daniel Knott. *Hands-On Mobile App Testing: A Guide for Mobile Testers and Anyone Involved in the Mobile App Business*. Addison-Wesley Professional, 18. Mai 2015. Kap. 2. 256 S. ISBN: 978-0-13-419182-9. URL: http://proquest.tech.safaribooksonline.de/9780134191829.

[9] *SWEBOK V3 • IEEE Computer Society*. URL: https://www.computer.org/web/swebok/v3 (besucht am 16. 07. 2017).

[10] *Modultest*. de. Page Version ID: 166498423. Juni 2017. URL: https://de.wikipedia.org/w/index.php?title=Modultest&oldid=166498423.

[11] Li Haoyi. *Principles of Automated Testing*.
URL: http://www.lihaoyi.com/post/PrinciplesofAutomatedTesting.html (besucht am 22. 07. 2017).

[12] *Integrationstest*. de. Page Version ID: 148753628. Dez. 2015. URL: https://de.wikipedia.org/w/index.php?title=Integrationstest&oldid=148753628.

[13] *JUnit*. de. Page Version ID: 156614793. Juli 2016.
URL: https://de.wikipedia.org/w/index.php?title=JUnit&oldid=156614793.

[14] *android - Google Espresso or Robotium - Stack Overflow*.
URL: https://stackoverflow.com/questions/20046021/google-espresso-or-robotium (besucht am 23. 07. 2017).

[15] *Dashboards | Android Developers*.
URL: https://developer.android.com/about/dashboards/index.html (besucht am 24. 11. 2017).

[16] *Testing UI for a Single App | Android Developers*.
URL: https://developer.android.com/training/testing/ui-testing/espresso-testing.html (besucht am 23. 07. 2017).

[17] *Create UI Tests with Espresso Test Recorder | Android Studio*. URL: https://developer.android.com/studio/test/espresso-test-recorder.html (besucht am 23. 07. 2017).

[18] Cigniti. *Cross-Platform Mobile Test Automation Using Appium*. März 2016.
URL: http://www.cigniti.com/blog/cross-platform-mobile-test-automation-using-appium/ (besucht am 24. 05. 2017).

[19] *Xcode*. de. Page Version ID: 165630142. Mai 2017.
URL: https://de.wikipedia.org/w/index.php?title=Xcode&oldid=165630142.

[20] *About Testing with Xcode*. URL: https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/01-introduction.html (besucht am 23. 07. 2017).

[21] Shashikant Jagtap.
XCBlog-Continuous Performance Testing of an iOS Apps using XCTest. Juli 2017.
URL: http://shashikantjagtap.net/continuous-performance-testing-ios-apps-using-xctest/ (besucht am 23. 07. 2017).

[22] *Mockito*. de. Page Version ID: 160883719. Dez. 2016.
URL: https://de.wikipedia.org/w/index.php?title=Mockito&oldid=160883719.

[23] Tal Weiss. *We Analyzed 30,000 GitHub Projects - Here Are The Top 100 Libraries in Java, JS and Ruby*. Nov. 2013.
URL: http://blog.takipi.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby/ (besucht am 23. 07. 2017).

[24] *robolectric: Android Unit Testing Framework*. original-date: 2010-08-28T00:28:25Z. Juli 2017. URL: https://github.com/robolectric/robolectric.

[25] *robotium: Android UI Testing*. original-date: 2009-12-10T12:51:14Z. Juli 2017.
URL: https://github.com/RobotiumTech/robotium.

[26] *Calaba.sh - Automated Acceptance Testing for iOS and Android Apps*.
URL: http://calaba.sh/ (besucht am 16. 07. 2017).

[27] AnDevCon: The Android Developer Conference.
MVC, MVP, MVVM Design Patterns with Godfrey Nolan.
URL: https://www.youtube.com/watch?v=JV63czrUpbI (besucht am 24. 07. 2017).

[28] *Repository (Entwurfsmuster)*. Page Version ID: 165836478. 26. Mai 2017. URL: https://de.wikipedia.org/w/index.php?title=Repository_(Entwurfsmuster)&oldid=165836478.

[29] *Kontinuierliche Integration*. de. Page Version ID: 166295191. Juni 2017. URL: https://de.wikipedia.org/w/index.php?title=Kontinuierliche_Integration&oldid=166295191.

[30] *Bamboo (Software)*. de. Page Version ID: 156681616. Aug. 2016. URL: https://de.wikipedia.org/w/index.php?title=Bamboo_(Software)&oldid=156681616.

[31] *Travis CI*. de. Page Version ID: 167068794. Juli 2017. URL: https://de.wikipedia.org/w/index.php?title=Travis_CI&oldid=167068794.

[32] *Jenkins (Software)*. de. Page Version ID: 166332047. Juni 2017. URL: https://de.wikipedia.org/w/index.php?title=Jenkins_(Software)&oldid=166332047.

[33] *Jenkins*. URL: /index.html (besucht am 15. 07. 2017).

[34] *TeamCity: Hassle-free CI and CD Server by JetBrains*. JetBrains. URL: https://www.jetbrains.com/teamcity/ (besucht am 26. 11. 2017).

[35] *Robot Framework*. URL: http://robotframework.org/ (besucht am 15. 07. 2017).

[36] *robotframework-androidlibrary: Robot Framework Automation Library for Android*. original-date: 2012-07-30T08:26:27Z. März 2017. URL: https://github.com/lovelysystems/robotframework-androidlibrary.

[37] *robotframework-ioslibrary: Robot Framework Automation Library for iOS*. original-date: 2012-07-25T11:08:56Z. März 2017. URL: https://github.com/lovelysystems/robotframework-ioslibrary.

[38] *AMCS - Auditorium Mobile Classroom Service*. URL: https://amcs.website/about (besucht am 03. 10. 2017).

[39] *Introduction to Data Driven Testing*. URL: https://smartbear.com/learn/automated-testing/introduction-to-data-driven-testing/ (besucht am 03. 10. 2017).

[40] *Keyword-driven testing*. Page Version ID: 801193553. 18. Sep. 2017. URL: https://en.wikipedia.org/w/index.php?title=Keyword-driven_testing&oldid=801193553.

[41] HackingGig. *What is code Driven testing in software testing?* URL: http://hackingig.com/what-is-code-driven-testing-in-software-testing/ (besucht am 28. 08. 2017).

[42] Atlassian. *Git Workflows and Tutorials | Atlassian Git Tutorial*. Atlassian. URL: https://www.atlassian.com/git/tutorials/comparing-workflows (besucht am 11. 09. 2017).

[43] *Release- and featurebranch workflow - Bild*. Comparing Workflows. URL: https://wac-cdn.atlassian.com/dam/jcr:3555a856-675e-453a-b49d-ba60667809e1/04.svg?cdnVersion=gg (besucht am 11. 09. 2017).

[44] *App Store - Support - Apple Developer*. URL: https://developer.apple.com/support/app-store/ (besucht am 24. 11. 2017).

[45] *Introduction to Model View Presenter on Android - Konstantin Mikheev's programming blog*. URL: http://konmik.com/post/introduction_to_model_view_presenter_on_android/ (besucht am 30. 09. 2017).

[46] *SharedPreferences | Android Developers*. URL: https://developer.android.com/reference/android/content/SharedPreferences.html (besucht am 24. 09. 2017).

[47] *IdlingResource | Android Developers*. URL: https://developer.android.com/reference/android/support/test/espresso/IdlingResource.html (besucht am 08. 11. 2017).

[48] Roy Thomas Fielding. „Architectural Styles and the Design of Network-based Software Architectures". AAI9980887. Diss. 2000. ISBN: 0-599-87118-0.

[49] auth0.com. *JWT.IO - JSON Web Tokens Introduction*. URL: http://jwt.io/ (besucht am 08. 11. 2017).

[50] *rest-assured: Java DSL for easy testing of REST services*. original-date: 2010-10-21T08:35:43Z. 10. Okt. 2017. URL: https://github.com/rest-assured/rest-assured (besucht am 10. 10. 2017).

[51] *Usint with Android #547*. original-date: 2010-10-21T08:35:43Z. 10. Okt. 2017. URL: https://github.com/rest-assured/rest-assured/issues/547 (besucht am 10. 10. 2017).

[52] *Android support? #583*. original-date: 2010-10-21T08:35:43Z. 10. Okt. 2017. URL: https://github.com/rest-assured/rest-assured/issues/583 (besucht am 10. 10. 2017).

[53] *Methods Count - Your solution for a perfectly fit APK*. URL: http://www.methodscount.com/?lib=com.github.tomakehurst%3Awiremock%3A2.8.0 (besucht am 25. 10. 2017).

[54] *Git workflow with maintenance - Bild*. 1. Okt. 2017. URL: https://wac-cdn.atlassian.com/dam/jcr:21cf772d-2ba5-4686-8259-fcd6fd2311df/05.svg?cdnVersion=hm (besucht am 01. 10. 2017).

[55] *Gradle Plugin - Jenkins - Jenkins Wiki*. 1. Okt. 2017. URL: https://wiki.jenkins.io/display/JENKINS/Gradle+Plugin (besucht am 01. 10. 2017).

[56] *BitBucket Plugin - Jenkins - Jenkins Wiki*. URL: https://wiki.jenkins.io/display/JENKINS/BitBucket+Plugin (besucht am 15. 07. 2017).

[57] *Android Emulator Plugin - Jenkins - Jenkins Wiki*. 1. Okt. 2017. URL: https://wiki.jenkins.io/display/JENKINS/Android+Emulator+Plugin (besucht am 01. 10. 2017).

[58] *Pipeline Multibranch Plugin - Jenkins - Jenkins Wiki*. 1. Okt. 2017. URL: https://wiki.jenkins.io/display/JENKINS/Pipeline+Multibranch+Plugin (besucht am 01. 10. 2017).

[59] *Advantages of Test Driven Development*. Apiumhub. 13. Dez. 2016. URL: https://apiumhub.com/tech-blog-barcelona/advantages-of-test-driven-development/ (besucht am 19. 11. 2017).

[60] *agile - TDD - what are the short term gains/benefits? - Software Engineering Stack Exchange*. URL: https://softwareengineering.stackexchange.com/questions/87248/tdd-what-are-the-short-term-gains-benefits (besucht am 19. 11. 2017).

[61] *9 Benefits of Test Driven Development - Made Tech Blog*. URL: https://www.madetech.com/blog/9-benefits-of-test-driven-development (besucht am 19. 11. 2017).

[62] *Robolectric*. URL: http://robolectric.org/ (besucht am 16. 07. 2017).

[63] *Cobertura Plugin - Jenkins - Jenkins Wiki*. URL: https://wiki.jenkins.io/display/JENKINS/Cobertura+Plugin (besucht am 26. 11. 2017).

[64] *perf (Linux)*. In: *Wikipedia*. Page Version ID: 793237075. 31. Juli 2017. URL: https://en.wikipedia.org/w/index.php?title=Perf_(Linux)&oldid=793237075 (besucht am 17. 11. 2017).

[65] *Which Android runs which Linux kernel? - Android Enthusiasts Stack Exchange*. URL: https://android.stackexchange.com/questions/51651/which-android-runs-which-linux-kernel (besucht am 17. 11. 2017).

[66] *Launch-Time Performance | Android Developers*. URL: https://developer.android.com/topic/performance/launch-time.html (besucht am 13. 11. 2017).

[67] *Best Practice #3: Reduce App Load Time to Two seconds to Increase Engagement*. Apteligent. URL: https://www.apteligent.com/technical-resource/best-practice-3-reduce-app-load-time-to-two-seconds-to-increase-engagement/ (besucht am 22. 11. 2017).

[68] *Software: The New Battleground for Brand Loyalty*. URL: https://www.ca.com/us/modern-software-factory/content/software-the-new-battleground-for-brand-loyalty.html (besucht am 22. 11. 2017).

[69] Pablo A. Martínez. *Running Android Tests on cloud devices using a Jenkins CI server (Firebase Test Lab — Amazon Device...* Pablo A. Martínez Andrés. 19. Jan. 2017. URL: https://pamartinezandres.com/running-android-tests-on-cloud-devices-using-a-jenkins-ci-server-firebase-test-lab-amazon-device-b67cb4b16c40 (besucht am 20. 11. 2017).

SELBSTSTÄNDIGKEITSERKLÄRUNG

Ich versichere, dass ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Dresden, 30.11.2017 Patrick Buchholz